ANNAMACHARYA
INSTITUTE OF TECHNOLOGY AND SCIENCES
(AUTONOMOUS)
Approved by AICTE, New Delhi & Permanent Affiliation to JNTUA, Anantapur.
Three B. Tech Programmes (CSE , ECE & CE) are accredited by NBA, New Delhi,Accredited by NAAC with 'A' Grade , Bangalore.
A-grade awarded by AP Knowledge Mission. Recognized under sections 2(f) & 12(B) of UGC Act 1956.
Venkatapuram Village, Renigunta Mandal, Tirupati, Andhra Pradesh-517520.

Department of Computer Science and Engineering

ANNAMACHARYA EDUCATIONAL TRUST ★ VIDWAN SARVATRA PUJYATHE ★

Academic Year 2023-24

II. B.Tech I Semester

Basics of Python Programming

(20APC0526)

Prepared By

Mr. T Sreenivasula Reddy., M.Tech, (Ph.D.).
Assistant Professor
Department of CSE, AITS
seenu4linux@gmail.com

Course Code	Basics of Python Programming (common to CSE,CIC)	L	T	P	C
20APC0526		3	0	0	3
Pre-requisite	**NILL**	**Semester**	**II-I**		

Course Objectives:

- To learn the fundamentals of Python
- To elucidate problem-solving using a Python programming language
- To introduce a function-oriented programming paradigm through python
- To get training in the development of solutions using modular concepts
- To introduce the programming constructs of python

Course Outcomes (CO):

CO1: Understanding the syntax and semantics of Python programming.
CO2: Apply modularity to programs.
CO3: Select appropriate data structure of Python for solving a problem.
CO4: Implement Mutable and Immutable data types
CO5: Interpret the concepts of object oriented programming as used in Python

UNIT - I 9Hrs

Introduction: What is a program, Running python, Arithmetic operators, Value and Types. **Variables, Assignments and Statements**: Assignment statements, Script mode, Order of operations, string operations, comments. **Functions**: Function calls, Math functions, Composition, Adding new Functions, Definitions and Uses, Flow of Execution, Parameters and Arguments, Variables and Parameters are local, Stack diagrams, Fruitful Functions and Void Functions, Why Functions.

UNIT - II 9 Hrs

Case study: The turtle module, Simple Repetition, Encapsulation, Generalization, Interface design, Refactoring, docstring. **Conditionals and Recursion**: floor division and modulus, Boolean expressions, Logical operators, Conditional execution, Alternative execution, Chained conditionals, Nested conditionals, Recursion, Infinite Recursion, Keyboard input. **Fruitful Functions**: Return values, Incremental development, Composition, Boolean functions, more recursion, Leap of Faith, Checking types

UNIT - III 9 Hrs

Iteration: Reassignment, Updating variables, The while statement, Break, Square roots, Algorithms. **Strings**: A string is a sequence, len, Traversal with a for loop, String slices, Strings are immutable, Searching, Looping and Counting, String methods, The in operator, String comparison. **Case Study**: Reading word lists, Search, Looping with indices. **Lists**: List is a sequence, Lists are mutable, Traversing a list, List operations, List slices, List methods, Map filter and reduce, Deleting elements, Lists and Strings, Objects and values, Aliasing, List arguments.

UNIT - IV 8 Hrs

Dictionaries: A dictionary is a mapping, Dictionary as a collection of counters, Looping and dictionaries, Reverse Lookup, Dictionaries and lists, Memos, Global Variables. **Tuples:** Tuples are immutable, Tuple Assignment, Tuple as Return values, Variable-length argument tuples, Lists and tuples, Dictionaries and tuples, Sequences of sequences. **Files:** Persistence, Reading and writing, Format operator, Filename and paths, Catching exceptions, Databases, Pickling, Pipes, Writing modules. **Classes and Objects**: Programmer-defined types, Attributes, Instances as Return values, Objects are mutable, Copying.

UNIT - V 10Hrs

Classes and Functions: Time, Pure functions, Modifiers, Prototyping versus Planning **Classes and Methods**: Object oriented features, Printing objects, 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, Data encapsulation. **The Goodies:** Conditional expressions, List comprehensions, Generator expressions, any and all, Sets, Counters, default dict, Named tuples, Gathering keyword Args

Textbooks:

1. Allen B. Downey, “Think Python”, 2nd edition, SPD/O’Reilly, 2016.

Reference Books:

1. Martin C.Brown, “The Complete Reference: Python”, McGraw-Hill, 2018.
2. Kenneth A. Lambert, B.L. Juneja, “Fundamentals of Python”, CENGAGE, 2015.
3. R. Nageswara Rao, “Core Python Programming”, 2nd edition, Dreamtech Press, 2019

Mapping of course outcomes with program outcomes

	PO1	PO2	PO3	PO4	PO5	PO6	PO7	PO8	PO9	PO10	PO11	PO12	PSO1	PSO2
CO1	3		2		2									
CO2	2			2									2	1
CO3	2	2	2	2									2	1
CO4	2		3		2								2	1
CO5	2	2	3		3				2				2	1

(Levels of Correlation, viz., 1-Low, 2-Moderate, 3 High)

Regulation: AK20	**Subject Code:CSE/CIC** 20APS0526/20APC3605	**Subject Name :** Basics of Python Programming	**AY:** 2023-2024
UNIT-1 (Assignments & Statements, Functions)			

UNIT – I

INTRODUCTION: What is a program, running python, Arithmetic operators, Value and Types?

VARIABLES, ASSIGNMENTS AND STATEMENTS: Assignment statements, Script mode, Order of operations, string operations, comments.

FUNCTIONS: Function calls, Math functions, Composition, Adding new Functions, Definitions and Uses, Flow of Execution, Parameters and Arguments, Variables and Parameters are local, Stack diagrams, Fruitful Functions and Void Functions, Why Functions.

Introduction

- Python is a simple, general purpose, high level, and object-oriented programming language.
- Python is an interpreted scripting language also. **Guido Van Rossum** is known as the founder of Python programming.

What is Python?

- Python is a general purpose, dynamic, high-level, and interpreted programming language. It supports Object Oriented programming approach to develop applications.
- It is simple and easy to learn and provides lots of high-level data structures.
- Python is easy to learn yet powerful and versatile scripting language, which makes it attractive for Application Development.
- Python's syntax and dynamic typing with its interpreted nature make it an ideal language for scripting and rapid application development.
- Python supports multiple programming pattern, including object-oriented, imperative, and functional or procedural programming styles.
- Python is not intended to work in a particular area, such as web programming. That is why it is known as multipurpose programming language because it can be used with web, enterprise, 3D CAD, etc.
- We don't need to use data types to declare variable because it is dynamically typed so we can write a=10 to assign an integer value in an integer variable.
- Python makes the development and debugging fast because there is no compilation step included in Python development, and edit-test-debug cycle is very fast.

Why is it called Python?

- When he began implementing Python, **Guido van Rossum** was also reading the published scripts from **"Monty Python's Flying Circus"**, a BBC comedy series from the 1970s. **Van Rossum** thought he needed a name that was short, unique, and slightly mysterious, so he decided to call the language Python.

Python History and Versions

- Python laid its foundation in the late 1980s.
- The implementation of Python was started in December 1989 by Guido Van Rossum at CWI in Netherland.
- In February 1991, **Guido Van Rossum** published the code (labeled version 0.9.0) to alt.sources.
- In 1994, Python 1.0 was released with new features like lambda, map, filter, and reduce.
- Python 2.0 added new features such as list comprehensions, garbage collection systems.
- On December 3, 2008, Python 3.0 (also called "Py3K") was released. It was designed to rectify the fundamental flaw of the language.
- ABC programming language is said to be the predecessor of Python language, which was capable of Exception Handling and interfacing with the Amoeba Operating System.

The following programming languages influence Python:

- ABC language.
- Modula-3

Python Version List

Python programming language is being updated regularly with new features and supports. There are lots of update in Python versions, started from 1994 to current release.

Python Version	**Released Date**
Python 1.0	January 1994
Python 1.5	December 31, 1997
Python 1.6	September 5, 2000
Python 2.0	October 16, 2000
Python 2.1	April 17, 2001

Python 2.2	December 21, 2001
Python 2.3	July 29, 2003
Python 2.4	November 30, 2004
Python 2.5	September 19, 2006
Python 2.6	October 1, 2008
Python 2.7	July 3, 2010
Python 3.0	December 3, 2008
Python 3.1	June 27, 2009
Python 3.2	February 20, 2011
Python 3.3	September 29, 2012
Python 3.4	March 16, 2014
Python 3.5	September 13, 2015
Python 3.6	December 23, 2016
Python 3.7	June 27, 2018
Python 3.8	October 14, 2019

Python Features

Python provides many useful features which make it popular and valuable from the other programming languages. It supports object-oriented programming, procedural programming approaches and provides dynamic memory allocation. We have listed below a few essential features.

1) Easy to Learn and Use

2) Expressive Language

3) Interpreted Language

4) Cross-platform Language

5) Free and Open Source

6) Object-Oriented Language

7) Extensible

8) Large Standard Library

9) GUI Programming Support

10) Integrated

11. Embeddable

12. Dynamic Memory Allocation

1) Easy to Learn and Use

Python is easy to learn as compared to other programming languages. Its syntax is straightforward and much the same as the English language. There is no use of the semicolon or curly-bracket, the indentation defines the code block. It is the recommended programming language for beginners.

2) Expressive Language

Python can perform complex tasks using a few lines of code. A simple example, the hello world program you simply type print ("Hello World"). It will take only one line to execute, while Java or C takes multiple lines.

3) Interpreted Language

Python is an interpreted language; it means the Python program is executed one line at a time. The advantage of being interpreted language, it makes debugging easy and portable.

4) Cross-platform Language

Python can run equally on different platforms such as Windows, Linux, UNIX, and Macintosh, etc. So, we can say that Python is a portable language. It enables programmers to develop the software for several competing platforms by writing a program only once.

5) Free and Open Source

Python is freely available for everyone. It is freely available on its official website www.python.org. It has a large community across the world that is dedicatedly working towards make new python modules and functions. Anyone can contribute to the Python community. The open-source means, "Anyone can download its source code without paying any penny."

6) Object-Oriented Language

Python supports object-oriented language and concepts of classes and objects come into existence. It supports inheritance, polymorphism, and encapsulation, etc. The object-oriented procedure helps to programmer to write reusable code and develop applications in less code.

7) Extensible

It implies that other languages such as C/C++ can be used to compile the code and thus it can be used further in our Python code. It converts the program into byte code, and any platform can use that byte code.

8) Large Standard Library

It provides a vast range of libraries for the various fields such as machine learning, web developer, and also for the scripting. There are various machine learning libraries, such as Tensor flow, Pandas, Numpy, Keras, and Pytorch, etc. Django, flask, pyramids are the popular framework for Python web development.

9) GUI Programming Support

Graphical User Interface is used for the developing Desktop application. PyQT5, Tkinter, Kivy are the libraries which are used for developing the web application.

10) Integrated

It can be easily integrated with languages like C, C++, and JAVA, etc. Python runs code line by line like C,C++ Java. It makes easy to debug the code.

11. Embeddable

The code of the other programming language can use in the Python source code. We can use Python source code in another programming language as well. It can embed other language into our code.

12. Dynamic Memory Allocation

In Python, we don't need to specify the data-type of the variable. When we assign some value to the variable, it automatically allocates the memory to the variable at run time. Suppose we are assigned integer value 15 to x, then we don't need to write int x = 15. Just write x = 15.

Python Applications

Python is known for its general-purpose nature that makes it applicable in almost every domain of software development. Python makes its presence in every emerging field. It is the fastest-growing programming language and can develop any application.

1) Web Applications

2) Desktop GUI Applications

3) Console-based Application

4) Software Development

5) Scientific and Numeric

6) Business Applications

7) Audio or Video-based Applications

8) 3D CAD Applications

9) Enterprise Applications

10) Image Processing Application

1) Web Applications

We can use Python to develop web applications. It provides libraries to handle internet protocols such as HTML and XML, JSON, Email processing, request, beautifulSoup, Feedparser, etc. One of Python web-framework named Django is used on Instagram. Python provides many useful frameworks.

- Django and Pyramid framework(Use for heavy applications)
- Flask and Bottle (Micro-framework)
- Plone and Django CMS (Advance Content management)

2) Desktop GUI Applications

The GUI stands for the Graphical User Interface, which provides a smooth interaction to any application. Python provides a Tk GUI library to develop a user interface. Some popular GUI libraries are

- Tkinter or Tk
- wxWidgetM
- Kivy (used for writing multitouch applications)
- PyQt or Pyside

3) Console-based Application

Console-based applications run from the command-line or shell. These applications are computer program which are used commands to execute. This kind of application was more popular in the old generation of computers. Python can develop this kind of application very effectively. It is famous for having REPL, which means the Read-Eval-Print Loop that makes it the most suitable language for the command-line applications.

Python provides many free library or module which helps to build the command-line apps. The necessary IO libraries are used to read and write. It helps to parse argument and create console help text out-of-the-box. There are also advance libraries that can develop independent console apps.

4) Software Development

Python is useful for the software development process. It works as a support language and can be used to build control and management, testing, etc.

- SCons is used to build control.
- Buildbot and Apache Gumps are used for automated continuous compilation and testing.
- Round or Trac for bug tracking and project management.

5) Scientific and Numeric

This is the era of Artificial intelligence where the machine can perform the task the same as the human. Python language is the most suitable language for Artificial intelligence or machine learning. It consists of many scientific and mathematical libraries, which makes easy to solve complex calculations.

Implementing machine learning algorithms require complex mathematical calculation. Python has many libraries for scientific and numeric such as Numpy, Pandas, Scipy, Scikit-learn, etc. If you have some basic knowledge of Python, you need to import libraries on the top of the code.

- SciPy
- Scikit-learn
- NumPy
- Pandas
- Matplotlib

6) Business Applications

Business Applications differ from standard applications. E-commerce and ERP are an example of a business application. This kind of application requires extensively, scalability and readability, and Python provides all these features.

Oddo is an example of the all-in-one Python-based application which offers a range of business applications. Python provides a Tryton platform which is used to develop the business application.

7) Audio or Video-based Applications

Python is flexible to perform multiple tasks and can be used to create multimedia applications. Some multimedia applications which are made by using Python are TimPlayer, cplay, etc. The few multimedia libraries are given below.

- Gstreamer
- Pyglet
- QT Phonon

8) 3D CAD Applications

The CAD (Computer-aided design) is used to design engineering related architecture. It is used to develop the 3D representation of a part of a system. Python can create a 3D CAD application by using below functionalities.

- Fandango (Popular)
- CAMVOX
- HeeksCNC
- AnyCAD

- RCAM

9) Enterprise Applications

Python can be used to create applications that can be used within an Enterprise or an Organization. Some real-time applications are OpenERP, Tryton, Picalo, etc.

10) Image Processing Application

Python contains many libraries that are used to work with the image. The image can be manipulated according to our requirements. Some libraries of image processing areiu.

- OpenCV
- Pillow
- SimpleITK

PYTHON INITIALIZATION: To Download the Python open the www.Pyhton.Org Click on Download and Download the Latest Version of Python Shell.

BASIC TERMINOLOGY

PROBLEM SOLVING: The process of formulating a problem, finding a solution, and expressing it.

HIGH-LEVEL LANGUAGE: A programming language like Python that is designed to be easy for humans to read and write.

LOW-LEVEL LANGUAGE: A programming language that is designed to be easy for a computer

to run; also called “machine language” or “assembly language”.

PORTABILITY: A property of a program that can run on more than one kind of computer.

INTERPRETER: A program that reads another program and executes it

PROMPT: Characters displayed by the interpreter to indicate that it is ready to take input from the user.

PROGRAM: A set of instructions that specifies a computation.

PRINT STATEMENT: An instruction that causes the Python interpreter to display a value on the screen.

OPERATOR: A special symbol that represents a simple computation like addition, multiplication, or string concatenation.

VALUE: One of the basic units of data, like a number or string, that a program manipulates.

TYPE: A category of values. The types we have seen so far are integers (type int), floating point numbers (type float), and strings (type str).

INTEGER: A type that represents whole numbers.

FLOATING-POINT: A type that represents numbers with fractional parts.

STRING: A type that represents sequences of characters.

NATURAL LANGUAGE: Any one of the languages that people speak that evolved naturally.

FORMAL LANGUAGE: Any one of the languages that people have designed for specific purposes, such as representing mathematical ideas or computer programs; all programming languages are formal languages.

TOKEN: One of the basic elements of the syntactic structure of a program, analogous to a word in a natural language.

SYNTAX: The rules that govern the structure of a program.

PARSE: To examine a program and analyse the syntactic structure.

BUG: An error in a program.

DEBUGGING: The process of finding and correcting bugs.

WHAT IS A PROGRAM?

A program is a sequence of instructions that specifies how to perform a computation.

The computation might be something mathematical, such as solving a system of equations or finding the roots of a polynomial, but it can also be a symbolic computation, such as searching and replacing text in a document or something graphical, like processing an image or playing a video.

INPUT: Get data from the keyboard, a file, the network, or some other device.

OUTPUT: Display data on the screen, save it in a file, send it over the network, etc.

MATH: Perform basic mathematical operations like addition and multiplication.

CONDITIONAL EXECUTION: Check for certain conditions and run the appropriate code.

REPETITION: Perform some action repeatedly, usually with some variation.

Every program you've ever used, no matter how complicated, is made up of instructions that look pretty much like these. So you can think of programming as the process of breaking a large, complex task into smaller and smaller subtasks until the subtasks are simple enough to be performed with one of these basic instructions.

WHAT IS INPUT?

Input is reading the data from the keyboard or any input devices, then we can call it as input.

WHAT IS OUTPUT?

Output is a displaying the data on the monitor or any output device, then we can call it as output.

OUTPUT:

- ✓ New print format is introduced in Python 3.x version
- ✓ "print" function can be used for displaying the data on monitor or console
- ✓ The difference between version 2.x and version 3.x is, parenthesis is not used in version 2.x, but parenthesis is used in version 3.x

PYTHON IS A TOP DOWN APPROACH LANGUAGE

We use the symbol >>> is called REPL (READ EVALUATE PRINT LOOP)

BELOW ARE THE DIFFERENT WAYS WE CAN USE PRINT FUNCTION IN VERSION 3.X

The basic syntax for print function is “print (‘statements’). Print syntax is used to print the every statement in new line.

```
>>> print('string in single quotes verion3')
string in single quotes verion3
>>> print("string in double quotes verion3")
string in single quotes verion3
>>> print('''string in triple quotes verion3''')
string in single triple verion3
```

Below is the syntax to print the multiple statements in single line. To print the multiple statements, we need to separate each statement with comma ‘,’

```
>>>print('Hello','inpython3.x','in single quotes','multiple satatements')

Hello inpython3.x in single quotes multiple satatements

>>>print("Hello","inpython3.x","in double quotes","multiple satatements")

Hello inpython3.x in double quotes multiple satatements

>>>print('''Hello''','''inpython3.x''','''in triple quotes''','''multiple satatements''')

Hello inpython3.x in triple quotes multiple satatements
```

PYTHON 3.X VERSION SUPPORTS INPUT () FUNCTION

INPUT ():

input() function is also used to read the input from the keyboard. But, the difference between raw_input() and input() functions is, variable will be considered as string by default even if it’s not type casted in raw_input(). But, in input(), it will be type casted automatically with respect to input.

```
>>> num=input("enter the integer num:")
enter the interger number:10

>>> dec=input("enter the float num:")
enter the float num:24.37
```

THE FIRST PROGRAM

```
print("hello world")
```

Python Indenta

- Python indentation uses to define the block of the code.
- C, C++, and Java use curly braces {}.
- Whereas Python uses an indentation. Whitespaces are used as indentation in Python.
- Indentation uses at the beginning of the code and ends with the unintended line.
- That same line indentation defines the block of the code (body of a function, loop, etc.)

```
for i in range(5):
   print(i)
   if(i == 3):
      break
```

- To indicate a block of code we indented each line of the block by the same whitespaces.

Example:

```
dn = int(input("Enter the number:"))
if(n%2 == 0):
   print("Even Number")
else:
   print("Odd Number")

print("Task Complete")
```

Output:

```
Enter the number: 10

Even Number

Task Complete
```

- The above code, if and else are two separate code blocks.

- Both code blocks are indented four spaces.
- The print ("Task Complete") statement is not indented four whitespaces and it is out of the if-else block.
- If the indentation is not used properly, then that will result in Indentation Error.

DATA TYPES

- Variables can hold values, and every value has a data-type.
- Python is a dynamically typed language; hence we do not need to define the type of the variable while declaring it.
- The interpreter implicitly binds the value with its type.
- Python enables us to check the type of the variable used in the program.
- Python provides us the type() function, which returns the type of the variable passed.

The following example to define the values of different data types and checking its type.

```
a=10
b="Hi Python"
c = 10.5
print(type(a))
print(type(b))
print(type(c))
```

Output:

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

Standard data types

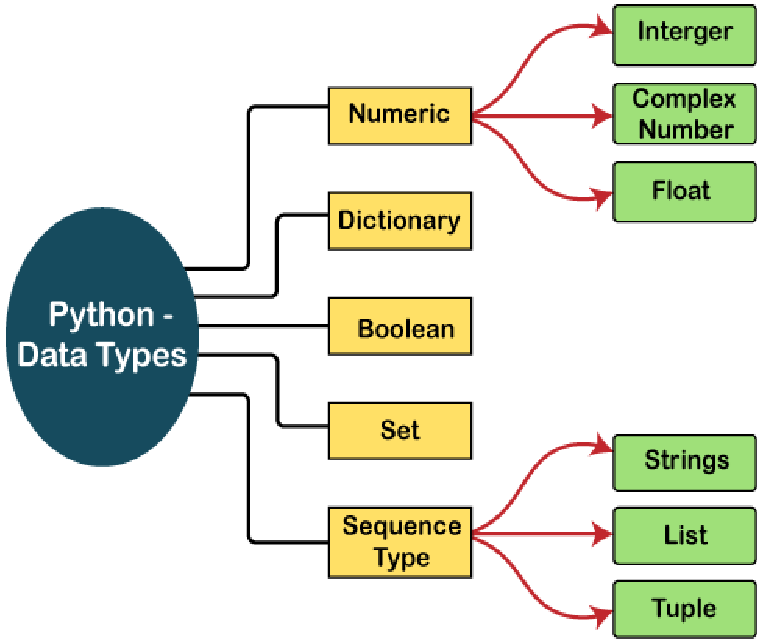
- A variable can hold different types of values.

For example, a person's name must be stored as a string whereas its id must be stored as an integer.

- Python provides various standard data types that define the storage method on each of them.

  - ✓ Numbers
  - ✓ Sequence Type
  - ✓ Boolean

- ✓ Set
- ✓ Dictionary



Numbers

- Number stores numeric values.
- The integer, float, and complex values belong to a Python Numbers data-type.
- Python provides the type() function to know the data-type of the variable.
- Similarly, the isinstance() function is used to check an object belongs to a particular class.
- Python creates Number objects when a number is assigned to a variable.

For example;

```
a = 5
print("The type of a", type(a))
b = 40.5
print("The type of b", type(b))
c = 1+3j
print("The type of c", type(c))
print(" c is a complex number", isinstance(1+3j,complex))
```

Output:

```
The type of a <class 'int'>
The type of b <class 'float'>
The type of c <class 'complex'>
```

```
c is complex number: True
```

Python supports three types of numeric data.

- **Int -** Integer value can be any length such as integers 10, 2, 29, -20, -150 etc. Python has no restriction on the length of an integer. Its value belongs to int
- **Float -** Float is used to store floating-point numbers like 1.9, 9.902, 15.2, etc. It is accurate upto 15 decimal points.
- **complex -** A complex number contains an ordered pair, i.e., x + iy where x and y denote the real and imaginary parts, respectively. The complex numbers like 2.14j, 2.0 + 2.3j, etc.

Sequence Type

String

- The string can be defined as the sequence of characters represented in the quotation marks. In Python, we can use single, double, or triple quotes to define a string.
- String handling in Python is a straightforward task since Python provides built-in functions and operators to perform operations in the string.
- In the case of string handling, the operator + is used to concatenate two strings as the operation **"hello"+" python" returns "hello python".**
- The operator * is known as a repetition operator as the operation **"Python" *2 returns 'Python Python'.**

Example - 1

```
str = "string using double quotes"
print(str)
s = """A multiline
string'''
print(s)
```

Output:

```
string using double quotes
A multiline
string
```

Example - 2

```
str1 = 'hello javatpoint' #string str1
str2 = ' how are you' #string str2
print (str1[0:2]) #printing first two character using slice operator
print (str1[4]) #printing 4th character of the string
print (str1*2) #printing the string twice
print (str1 + str2) #printing the concatenation of str1 and str2
```

Output:

```
he
o
hello javatpointhello javatpoint
hello javatpoint how are you
```

List

- Python Lists are similar to arrays in C.
- However, the list can contain data of different types.
- The items stored in the list are separated with a comma (,) and enclosed within square brackets [].
- We can use slice [:] operators to access the data of the list.
- The concatenation operator (+) and repetition operator (*) works with the list in the same way as they were working with the strings.

Example.

```
list1  = [1, "hi", "Python", 2]
#Checking type of given list
print(type(list1))
#Printing the list1
print (list1)
# List slicing
print (list1[3:])
# List slicing
print (list1[0:2])
# List Concatenation using + operator
print (list1 + list1)
# List repetation using * operator
print (list1 * 3)
```

Output:

```
[1, 'hi', 'Python', 2]
[2]
[1, 'hi']
[1, 'hi', 'Python', 2, 1, 'hi', 'Python', 2]
[1, 'hi', 'Python', 2, 1, 'hi', 'Python', 2, 1, 'hi', 'Python', 2]
```

Tuple

- A tuple is similar to the list in many ways. Like lists, tuples also contain the collection of the items of different data types.
- The items of the tuple are separated with a comma (,) and enclosed in parentheses ().
- A tuple is a read-only data structure as we can't modify the size and value of the items of a tuple.

Let's see a simple example of the tuple.

```
tup  = ("hi", "Python", 2)
# Checking type of tup
print (type(tup))
 #Printing the tuple
print (tup)
# Tuple slicing
print (tup[1:])
print (tup[0:1])
# Tuple concatenation using + operator
print (tup + tup)
# Tuple repatation using * operator
print (tup * 3)
 # Adding value to tup. It will throw an error.
t[2] = "hi"
```

Output:

```
<class 'tuple'>
('hi', 'Python', 2)
('Python', 2)
('hi',)
('hi', 'Python', 2, 'hi', 'Python', 2)
('hi', 'Python', 2, 'hi', 'Python', 2, 'hi', 'Python', 2)
```

```
Traceback (most recent call last):
  File "main.py", line 14, in <module>
    t[2] = "hi";
TypeError: 'tuple' object does not support item assignment
```

Dictionary

- Dictionary is an unordered set of a key-value pair of items. It is like an associative array or a hash table where each key stores a specific value.
- Key can hold any primitive data type, whereas value is an arbitrary Python object.
- The items in the dictionary are separated with the comma (,) and enclosed in the curly braces {}.

Consider the following example.

```
d = {1:'Jimmy', 2:'Alex', 3:'john', 4:'mike'}
  # Printing dictionary
print (d)
  # Accesing value using keys
print("1st name is "+d[1])
print("2nd name is "+ d[4])
print (d.keys())
print (d.values())
```

Output:

```
1st name is Jimmy
2nd name is mike
{1: 'Jimmy', 2: 'Alex', 3: 'john', 4: 'mike'}
dict_keys([1, 2, 3, 4])
dict_values(['Jimmy', 'Alex', 'john', 'mike'])
```

Boolean

- Boolean type provides two built-in values, True and False.
- These values are used to determine the given statement true or false.
- It denotes by the class bool. True can be represented by any non-zero value or 'T' whereas false can be represented by the 0 or 'F'.

# Python program to check the boolean type

```
print(type(True))
print(type(False))
print(false)
```

Output:

```
<class 'bool'>
<class 'bool'>
NameError: name 'false' is not defined
```

Set

- Python Set is the unordered collection of the data type.
- It is iterable, mutable(can modify after creation), and has unique elements.
- In set, the order of the elements is undefined; it may return the changed sequence of the element.
- The set is created by using a built-in function set(), or a sequence of elements is passed in the curly braces and separated by the comma.
- It can contain various types of values.

```
# Creating Empty set
set1 = set()
  set2 = {'James', 2, 3,'Python'}
  #Printing Set value
print(set2)
  # Adding element to the set
  set2.add(10)
print(set2)
  #Removing element from the set
set2.remove(2)
print(set2)
```

Output:

```
{3, 'Python', 'James', 2}
{'Python', 'James', 3, 2, 10}
{'Python', 'James', 3, 10}
```

OPERATORS

- The operator can be defined as a symbol which is responsible for a particular operation between two operands.
- Operators are the pillars of a program on which the logic is built in a specific programming language.

OPERATOR CLASSIFICATION:

- Operators can be classified based on the operations that they perform or based on the number of operands it operates on.
  1. Unary Operator
  2. Ternary Operator
  3. Binary Operator

Regulation: AK20	**Subject Code:CSE/CIC** 20APS0526/20APC3605	**Subject Name :** Basics of Python Programming	**AY:** 2023-2024
UNIT-1 (Assignments & Statements, Functions)			

1. UNARY OPERATORS

The unary operators operate on a single operand. The addition, subtraction and multiplication operators are used as unary operator in python.

2. TERNARY OPERATORS

The ternary operator is an operator that operates on three operands. The first argument is an expression, If expression is true then second argument will execute else third argument will execute.

3. BINARY OPERATORS

A binary operator is an operator that operates on two operands.

1. Arithmetic Operators
2. Assignment Operators
3. Relational Operators
4. Logical Operators
5. Membership operators
6. Identity operators
7. Bitwise operators

1. ARITHMETIC OPERATORS:

Arithmetic operators are used to perform arithmetic operations between two operands. It includes + (addition), - (subtraction), *(multiplication), /(divide), %(reminder), //(floor division), and exponent (**) operators.

Operator	**Description**
+ (Addition)	It is used to add two operands. For example, if a = 20, b = 10 => a+b = 30

- (Subtraction)	It is used to subtract the second operand from the first operand. If the first operand is less than the second operand, the value results negative. For example, if a = 20, b = 10 => a - b = 10
/ (divide)	It returns the quotient after dividing the first operand by the second operand. For example, if a = 20, b = 10 => a/b = 2.0
*** (Multiplication)**	It is used to multiply one operand with the other. For example, if a = 20, b = 10 => a * b = 200
% (reminder)	It returns the reminder after dividing the first operand by the second operand. For example, if a = 20, b = 10 => a%b = 0
**** (Exponent)**	It is an exponent operator represented as it calculates the first operand power to the second operand.
// (Floor division)	It gives the floor value of the quotient produced by dividing the two operands.

EXAMPLE:

```
x = 15
y = 4
print('x + y =',x+y)
print('x - y =',x-y)
print('x * y =',x*y)
print('x / y =',x/y)
print('x // y =',x//y)
print('x ** y =',x**y)
print('x % y=',x%y)
```

OUTPUT:

```
x + y = 19
x - y = 11
x * y = 60
x / y = 3.75
x // y = 3
x ** y = 50625
x % y= 3
```

2. ASSIGNMENT OPERATORS

- Assignment operator is used to assign value to a variable (memory location). It evaluates expression on right side of = symbol and assigns evaluated value to left side the variable.
- The RHS of assignment operator must be a constant, expression or variable. Whereas LHS must be a variable (valid memory location).

Operator	**Description**
=	It assigns the value of the right expression to the left operand.
+=	It increases the value of the left operand by the value of the right operand and assigns the modified value back to left operand. For example, if a = 10, b = 20 => a+ = b will be equal to a = a+ b and therefore, a = 30.
-=	It decreases the value of the left operand by the value of the right operand and assigns the modified value back to left operand. For example, if a = 20, b = 10 => a- = b will be equal to a = a- b and therefore, a = 10.
*=	It multiplies the value of the left operand by the value of the right operand and assigns the modified value back to then the left operand. For example, if a = 10, b = 20 => a* = b will be equal to a = a* b and therefore, a = 200.
%=	It divides the value of the left operand by the value of the right operand and assigns the reminder back to the left operand. For example, if a = 20, b = 10 => a % = b will be equal to a = a % b and therefore, a = 0.
**=	a**=b will be equal to a=a**b, for example, if a = 4, b =2, a**=b will assign 4**2 = 16 to a.
//=	A//=b will be equal to a = a// b, for example, if a = 4, b = 3, a//=b will assign 4//3 = 1 to a.

EXAMPLE:

```
a = 21
b = 10
c = 0
c = a + b
print ("Line 1 - Value of c is ", c)
```

```
c += a
print ("Line 2 - Value of c is ", c)
c *= a
print ("Line 3 - Value of c is ", c)
c /= a
print ("Line 4 - Value of c is ", c)
c  = 2
c %= a
print ("Line 5 - Value of c is ", c)
c **= a
print ("Line 6 - Value of c is ", c)
c //= a
print ("Line 7 - Value of c is ", c)
```

OUTPUT

```
Line 1 - Value of c is   31
Line 2 - Value of c is   52
Line 3 - Value of c is   1092
Line 4 - Value of c is   52.0
Line 5 - Value of c is   2
Line 6 - Value of c is   2097152
Line 7 - Value of c is   99864
```

3. RELATIONAL OPERATORS:

Relational operator is an operator which defines some kind of relationship between two entities or two objects or two variables.

Operator	**Description**
==	If the value of two operands is equal, then the condition becomes true.
!=, < >	If the value of two operands is not equal, then the condition becomes true.
<=	If the first operand is less than or equal to the second operand, then the condition becomes true.
>=	If the first operand is greater than or equal to the second operand, then the condition becomes true.

>	If the first operand is greater than the second operand, then the condition becomes true.
<	If the first operand is less than the second operand, then the condition becomes true.

EXAMPLE:

```
x = 10
y = 12
print('x > y is',x>y)
print('x < y is',x<y)
print('x == y is',x==y)
print('x != y is',x!=y)
print('x >= y is',x>=y)
print('x <= y is',x<=y)
```

OUT PUT:

```
x > y is False
x < y is True
x == y is False
x != y is True
x >= y is False
x <= y is True
```

4. LOGICAL OPERATORS:

- Logical operators are mainly used to control program flow
- The concept of logical operators is simple.
- In logical AND operator, if any one operand is false then result is false else result is true
- In logical OR operator, if any one operand is true then result is true else result is false
- In logical NOT operator, it will work on single operand, if operand is true result is false e lse vice

  Versa.

Operator	**Description**
And	If both the expression are true, then the condition will be true. If a and b are the two expressions, a → true, b → true => a and b → true.

Or	If one of the expressions is true, then the condition will be true. If a and b are the two expressions, a → true, b → false => a or b → true.
Not	If an expression **a** is true, then not (a) will be false and vice versa.

TRUTH TABLE FOR LOGICAL AND

Operand1 / Operand2	**Output**
False / False	False
False / True	False
True / False	False
True / True	True

TRUTH TABLE FOR LOGICAL OR

Operand1 / Operand2	**Output**
False / False	True
False / True	True
True / False	True
True / True	False

TRUTH TABLE FOR LOGICAL NOT

Operand1	**Output**
False	True

EXAMPLE:

```
x = True
y = False
print('x and y is',x and y)
print('x or y is',x or y)
print('not x is',not x)
```

OUTPUT:

```
x and y is False
x or y is True
not x is False
```

5. MEMBERSHIP OPERATOR:

- The membership operator is a special operator introduced in python programming language.

- It is used to validate the membership of value.
- It is used to test the membership in sequence data types like list, tuples, and strings.

Operator	**Meaning**
In	The 'in' operator is used to check if a value exists in a sequence or not. Evaluates to true if it finds a variable in the specified sequence and false otherwise. (List, tuple, or dictionary).
not in	This operand is used to find the variable which are not existing other than the values given in the sequence. (List, tuple, or dictionary).

EXAMPLE:

```
x = 'Hello world'
y = {1:'a',2:'b'}
print('H' in x)
print('hello' not in x)
print(1 in y)
print('a' in y)
```

OUTPUT:

```
True
True
True
False
```

6. IDENTITY OPERATORS:

This operator is used identify type of the object, variable etc. There are different identity operators such as,

Operator	**Meaning**
Is	Evaluate true if identity of two operands are same, else false
is not	Evaluate false if identity of two operands are same, else true

EXAMPLE:

```
 x1 = 5
y1 = 5
x2 = 'Hello'
y2 = 'Hello'
x3 = [1,2,3]
```

```
y3 = [1,2,3]
print(x1 is not y1)
print(x2 is y2)
print(x3 is y3)
```

OUTPUT:

```
False
True
False
```

EXAMPLE 2:

```
x=42
y=42
if ( x is y):
        print ("both are  same identity")
else:
        print ("both are different  identity")
y=35
if ( x is not y):
        print ("both are different  identity")
else:
        print ("both are  same identity")
```

OUTPUT:

```
both are  same identity
both are different  identity
```

7. BITWISE OPERATORS:

Bitwise operator is working on 0 and 1's and manipulate the bits, it will give the performance wise very well. Bitwise operation scripting is very helpful in hardware level programming.

Operator	Description
& (binary and)	If both the bits at the same place in two operands are 1, then 1 is copied to the result. Otherwise, 0 is copied.
| (binary or)	The resulting bit will be 0 if both the bits are zero; otherwise, the resulting bit will be 1.
^ (binary	The resulting bit will be 1 if both the bits are different; otherwise, the

xor)	resulting bit will be 0.
~ (negation)	It calculates the negation of each bit of the operand, i.e., if the bit is 0, the resulting bit will be 1 and vice versa.
<< (left shift)	The left operand value is moved left by the number of bits present in the right operand.
>> (right shift)	The left operand is moved right by the number of bits present in the right operand.

EXAMPLE

```
a = 9
b = 65
print("Bitwise AND Operator On 9 and 65 is = ", a & b)
print("Bitwise OR Operator On 9 and 65 is = ", a | b)
print("Bitwise EXCLUSIVE OR Operator On 9 and 65 is = ", a ^ b)
print("Bitwise NOT Operator On 9 is = ", ~a)
print("Bitwise LEFT SHIFT Operator On 9 is = ", a << 1)
print("Bitwise RIGHT SHIFT Operator On 65 is = ", b >> 1)
```

OUTPUT:

```
Bitwise AND Operator On 9 and 65 is =  1
Bitwise OR Operator On 9 and 65 is =  73
Bitwise EXCLUSIVE OR Operator On 9 and 65 is =  72
Bitwise NOT Operator On 9 is =  -10
Bitwise LEFT SHIFT Operator On 9 is =  18
Bitwise RIGHT SHIFT Operator On 65 is =  32
```

LITERALS

Python Literals can be defined as data that is given in a variable or constant.

1. String literals:

String literals can be formed by enclosing a text in the quotes. We can use both single as well as double quotes to create a string.

Example:

```
"Aman" , '12345'
```

Types of Strings:

There are two types of Strings supported in Python:

a) Single-line String- Strings that are terminated within a single-line are known as Single line Strings.

Example:

```
text1='hello'
```

b) Multi-line String - A piece of text that is written in multiple lines is known as multiple lines string.

There are two ways to create multiline strings:

i) Adding black slash at the end of each line.

Example:

```
text1='hello\
user'
print(text1)
```

Output:

```
'hellouser'
```

ii) Using triple quotation marks:-

Example:

```
str2="""welcome
to
VEMU'''
print str2
```

Output:

```
welcome
to
VEMU
```

2. Numeric literals:

Numeric Literals are immutable. Numeric literals can belong to following four different numerical types.

Int (signed integers)

Numbers (can be both positive and negative) with no fractional part.eg: 100

Long (long integers)

Integers of unlimited size followed by lowercase or uppercase L eg: 87032845L

Float (floating point)

Real numbers with both integer and fractional part eg: -26.2

Complex (complex)

In the form of a+bj where a forms the real part and b forms the imaginary part of the complex number. eg: 3.14j

Example - Numeric Literals

```
x = 0b10100 #Binary Literals
y = 100 #Decimal Literal
z = 0o215 #Octal Literal
u = 0x12d #Hexadecimal Literal

#Float Literal
float_1 = 100.5
float_2 = 1.5e2
  #Complex Literal
a = 5+3.14j
  print(x, y, z, u)
print(float_1, float_2)
print(a, a.imag, a.real)
```

Output:

```
20 100 141 301
100.5 150.0
(5+3.14j) 3.14 5.0
```

3. Boolean literals:

A Boolean literal can have any of the two values: True or False.

Example - Boolean Literals

```
x = (1 == True)
y = (2 == False)
z = (3 == True)
a = True + 10
b = False + 10
print("x is", x)
print("y is", y)
print("z is", z)
print("a:", a)
print("b:", b)
```

Output:

```
x is True
y is False
z is False
a: 11
b: 10
```

4. Special literals.

Python contains one special literal i.e., None.

None is used to specify to that field that is not created. It is also used for the end of lists in Python.

Example - Special Literals

```
val1=10
val2=None
print(val1)
print(val2)
```

Output:

```
10
None
```

5. Literal Collections.

Python provides the four types of literal collection such as List literals, Tuple literals, Dict literals, and Set literals.

List:

- List contains items of different data types. Lists are mutable i.e., modifiable.
- The values stored in List are separated by comma(,) and enclosed within square brackets([]). We can store different types of data in a List.

Example - List literals

```
list=['John',678,20.4,'Peter']
list1=[456,'Andrew']
print(list)
print(list + list1)
```

Output:

```
['John', 678, 20.4, 'Peter']
['John', 678, 20.4, 'Peter', 456, 'Andrew']
```

Dictionary:

- Python dictionary stores the data in the key-value pair.
- It is enclosed by curly-braces {} and each pair is separated by the commas (,).

Example

```
dict = {'name': 'Pater', 'Age':18,'Roll_nu':101}
print (dict)
```

Output:

```
{'name': 'Pater', 'Age': 18, 'Roll_nu': 101}
```

Tuple:

- Python tuple is a collection of different data-type. It is immutable which means it cannot be modified after creation.
- It is enclosed by the parentheses () and each element is separated by the comma(,).

Example

```
tup = (10,20,"Dev",[2,3,4])
print (tup)
```

Output: (10, 20, 'Dev', [2, 3, 4])

Set:

- Python set is the collection of the unordered dataset.
- It is enclosed by the {} and each element is separated by the comma(,).

Example: - Set Literals

```
set = {'apple','grapes','guava','papaya'}
print(set)
```

Output:

```
{'guava', 'apple', 'papaya', 'grapes'}
```

TOKENS:

- ✓ The tokens can be defined as a punctuator mark, reserved words, and each word in a statement.
- ✓ The token is the smallest unit inside the given program.
  - ➢ Keywords.
  - ➢ Identifiers.
  - ➢ Literals.
  - ➢ Operators.

FORMAL AND NATURAL LANGUAGES

Natural languages are the languages people speak, such as English,Telugu, Hindi, Spanish, and French.
They were not designed by people (although people try to impose some order on them); they evolved naturally.

Formal languages are languages that are designed by people for specific applications. For example, the notation that mathematicians use is a formal language that is particularly good at denoting relationships among numbers and symbols. Chemists use a formal language to represent the chemical structure of molecules. And most importantly:

Programming languages are formal languages that have been designed to express computations.

Formal languages tend to have strict syntax rules that govern the structure of statements. For example, in mathematics the statement 3 + 3 = 6 has correct syntax, but 3+ = 3$6 does not.

Syntax rules come in two flavors, pertaining to tokens and structure. Tokens are the basic elements of the language, such as words, numbers, and chemical elements. One of the problems with 3+ = 3$6 is that $ is not a legal token in mathematics.

The second type of syntax rule pertains to the way tokens are combined. The equation 3 + /3 is illegal because even though + and / are legal tokens, you can’t have one right after the other.

This is @ well-structured Engli$h sentence with invalid t*kens in it. This sentence all valid tokens has, but invalid structure with. When you read a sentence in English or a statement in a formal language, you have to figure out the structure (although in a natural language you do this subconsciously). This
process is called parsing.

Although formal and natural languages have many features in common—tokens, structure, and syntax—there are some differences:

Ambiguity: Natural languages are full of ambiguity, which people deal with by using contextual clues and other information. Formal languages are designed to be nearly or completely unambiguous, which means that any statement has exactly one meaning, regardless of context.

Redundancy: In order to make up for ambiguity and reduce misunderstandings, natural languages employ lots of redundancy. As a result, they are often verbose. Formal languages are less redundant and more concise.

Literalness: Natural languages are full of idiom and metaphor. If I say, “The penny dropped”, there is probably no penny and nothing dropping (this idiom means that someone understood something after a period of confusion). Formal languages mean exactly what they say.

Because we all grow up speaking natural languages, it is sometimes hard to adjust to formal languages. The difference between formal and natural language is like the difference between poetry and prose, but more so:

Poetry: Words are used for their sounds as well as for their meaning, and the whole poem together creates an effect or emotional response. Ambiguity is not only common but often deliberate.

Prose: The literal meaning of words is more important, and the structure contributes more meaning. Prose is more amenable to analysis than poetry but still often ambiguous.

Programs: The meaning of a computer program is unambiguous and literal, and can be understood entirely by analysis of the tokens and structure.

Formal languages are more dense than natural languages, so it takes longer to read them. Also, the structure is important, so it is not always best to read from top to bottom, left to right.

Instead, learn to parse the program in your head, identifying the tokens and interpreting the structure. Finally, the details matter. Small errors in spelling and punctuation, which you can get away with in natural languages, can make a big difference in a formal language.

KEYWORDS

They are 33 keywords in python.

- Python Keywords are special reserved words that convey a special meaning to the compiler/interpreter.
- Each keyword has a special meaning and a specific operation. These keywords can't be used as a variable.

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

COMMENTS

- Python Comment is an essential tool for the programmers.
- Comments are generally used to explain the code.
- We can easily understand the code if it has a proper explanation.
- A good programmer must use the comments because in the future anyone wants to modify the code as well as implement the new module; then, it can be done easily.
- In the other programming language such as C++, It provides the // for single-lined comment and /*.... */ for multiple-lined comment.
- But Python provides the single-lined Python comment. To apply the comment in the code we use the hash(#) at the beginning of the statement or code.

# This is the print statement

```
print("Hello Python")
```

Here we have written comment over the print statement using the hash(#). It will not affect our print statement.

Multiline Python Comment

We must use the hash(#) at the beginning of every line of code to apply the multiline Python comment.

```
# First line of the comment
# Second line of the comment
# Third line of the comment
```

Example:

```
# Variable a holds value 5
# Variable b holds value 10
# Variable c holds sum of a and b
# Print the result
a = 5
b = 10
c = a+b
print("The sum is:", c)
```

Output:

```
The sum is: 15
```

- The above code is very readable even the absolute beginners can under that what is happening in each line of the code. This is the advantage of using comments in code.
- We can also use the triple quotes (""") for multiline comment. The triple quotes are also used to string formatting.

Docstrings Python Comment

- The docstring comment is mostly used in the module, function, class or method.
- It is a documentation Python string.

Example:

```
def intro():
  """
  This function prints Hello Joseph
  """
  print("Hi Joseph")
intro()
```

Output:

```
Hello Joseph
```

We can check a function's docstring by using the __doc__ attribute.

Generally, four whitespaces are used as the indentation.
The amount of indentation depends on user, but it must be consistent throughout that block.

Example:

```
def intro():
  """
  This function prints Hello Joseph
  """
  print("Hello Joseph")
intro.__doc__
```

Output:

```
'\n  This function prints Hello Joseph\n  '
```

<u>Note:</u> The docstring must be the first thing in the function; otherwise, Python interpreter cannot get the docstring.

VARIABLES

Definition: - Variable is a name that is used to refer to memory location. Python variable is also known as an identifier and used to hold value.

- In Python, we don't need to specify the type of variable because Python is a infer language and smart enough to get variable type.
- Variable names can be a group of both the letters and digits, but they have to begin with a letter or an underscore.
- It is recommended to use lowercase letters for the variable name. Rahul and rahul both are two different variables.

Identifier Naming

- Variables are the example of identifiers.
- An Identifier is used to identify the literals used in the program.

The rules to name an identifier are.

- The first character of the variable must be an alphabet or underscore (_).
- All the characters except the first character may be an alphabet of lower-case (a-z), upper-case (A-Z), underscore, or digit (0-9).

- Identifier name must not contain any white-space, or special character (! @, #, %, ^, &, *).
- Identifier name must not be similar to any keyword defined in the language.
- Identifier names are case sensitive; for example, my name, and MyName is not the same.
  - ✓ Examples of valid identifiers: a123, _n, n_9, etc.
  - ✓ Examples of invalid identifiers: 1a, n%4, n 9, etc.

Declaring Variable and Assigning Values

- Python does not bind us to declare a variable before using it in the application.
- It allows us to create a variable at the required time.
- We don't need to declare explicitly variable in Python.
- When we assign any value to the variable, that variable is declared automatically.
- The equal (=) operator is used to assign value to a variable.

Object References

- It is necessary to understand how the Python interpreter works when we declare a variable.
- The process of treating variables is somewhat different from many other programming languages.
- Python is the highly object-oriented programming language; that's why every data item belongs to a specific type of class.

Example

```
print("John")
```

Output:

```
John
```

The Python object creates an integer object and displays it to the console. In the above print statement, we have created a string object. Let's check the type of it using the Python built-in type () function.

Example

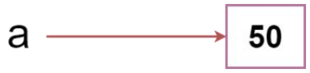
```
type ("John")
```

Output:

```
<class 'str'>
```

In Python, variables are a symbolic name that is a reference or pointer to an object. The variables are used to denote objects by that name.

a = 50

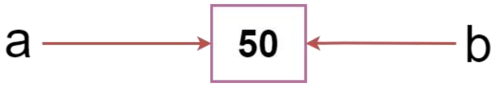


The variable a refers to an integer object.

Suppose we assign the integer value 50 to a new variable b.

a = 50

b = a

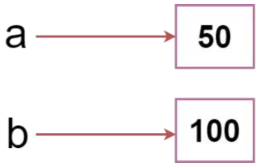


The variable b refers to the same object that a points to because Python does not create another object.

Let's assign the new value to b. Now both variables will refer to the different objects.

a = 50

b =100



Python manages memory efficiently if we assign the same variable to two different values.

Object Identity

In Python, every created object identifies uniquely in Python. Python provides the guaranteed that no two objects will have the same identifier. The built-in id() function, is used to identify the object identifier.

Example

```
a = 50
b = a
print(id(a))
print(id(b))
# Reassigned variable a
a = 500
print(id(a))
```

Output:

```
140734982691168
140734982691168
2822056960944
```

We assigned the b = a, a and b both point to the same object. When we checked by the id() function it returned the same number. We reassign a to 500; then it referred to the new object identifier.

Variable Names

We have already discussed how to declare the valid variable. Variable names can be any length can have uppercase, lowercase (A to Z, a to z), the digit (0-9), and underscore character (_).

Valid Variables Names Example.

```
name = "Krishna"
age = 20
marks = 80.50
 print(name)
print(age)
print(marks)
```

Output:

```
Krishna
20
80.5
```

Valid Variables Name.

```
          name = "A"
          Name = "B"
          naMe = "C"
          NAME = "D"
          n_a_m_e = "E"
          _name = "F"
          name_ = "G"
          _name_ = "H"
          na56me = "I"
 print(name,Name,naMe,NAME,n_a_m_e, NAME, n_a_m_e, _name, name_,_name, na56me)
```

Output:

```
A B C D E D E F G F I
```

In the above example, we have declared a few valid variable names such as name, _name_ , etc. But it is not recommended because when we try to read code, it may create confusion. The variable name should be descriptive to make code more readable.

The multi-word keywords can be created by the following method.

Camel Case - In the camel case, each word or abbreviation in the middle of begins with a capital letter. There is no intervention of whitespace. For example - nameOfStudent, valueOfVaraible, etc.

Pascal Case - It is the same as the Camel Case, but here the first word is also capital. For example - NameOfStudent, etc.

Snake Case - In the snake case, Words are separated by the underscore. For example - name_of_student, etc.

Multiple Assignment

Python allows us to assign a value to multiple variables in a single statement, which is also known as multiple assignments.

We can apply multiple assignments in two ways, either by **assigning a single value to multiple variables** or **assigning multiple values to multiple variables**.

1. Assigning single value to multiple variables

Eg:

```
x=y=z=50
print(x)
print(y)
print(z)
```

Output:

```
50
50
50
```

2. Assigning multiple values to multiple variables:

Eg:

```
a,b,c=5,10,15
print a
print b
print c
```

Output: 5 10 15

The values will be assigned in the order in which variables appear.

ASSIGNMENT STATEMENTS

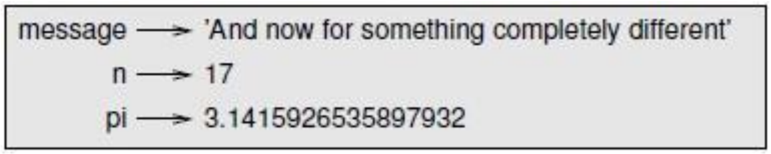
An assignment statement creates a new variable and gives it a value:

```
>>> message = 'And now for something completely different'
>>> n = 17
>>> pi = 3.1415926535897932
```

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

A common way to represent variables on paper is to write the name with an arrow pointing
to its value. This kind of figure is called a state diagram because it shows what state each

of the variables is in (think of it as the variable's state of mind).



VARIABLE NAMES

- Programmers generally choose names for their variables that are meaningful—they document what the variable is used for.
- Variable names can be as long as you like. They can contain both letters and numbers, but they can't begin with a number. It is legal to use uppercase letters, but it is conventional to use only lower case for variables names.
- The underscore character, _, can appear in a name. It is often used in names with multiple words, such as your_name or airspeed_of_unladen_swallow.

BASIC TERMINOLOGY

ASSIGNMENT: A statement that assigns a value to a variable.

OPERAND: One of the values on which an operator operates.

EXPRESSION: A combination of variables, operators, and values that represents a single result.

EVALUATE: To simplify an expression by performing the operations in order to yield a single value.

EXECUTE: To run a statement and do what it says.

ORDER OF OPERATIONS: Rules governing the order in which expressions involving multiple operators and operands are evaluated.

SCRIPT: A program stored in a file.

SCRIPT MODE: A way of using the Python interpreter to read code from a script and run it.

INTERACTIVE MODE: A way of using the Python interpreter by typing code at the prompt.

CONCATENATE: To join two operands end-to-end.

EXCEPTION: An error that is detected while the program is running.

SYNTAX ERROR: An error in a program that makes it impossible to parse (and therefore impossible
to interpret).

SEMANTICS: The meaning of a program.

SEMANTIC ERROR: An error in a program that makes it do something other than what the programmer intended.

EXPRESSIONS AND STATEMENTS

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

```
>>> 42
42
>>> n= 17
>>> n + 25
42
```

When you type an expression at the prompt, the interpreter evaluates it, which means that it finds the value of the expression. In this example, n has the value 17 and n + 25 has the A statement is a unit of code that has an effect, like creating a variable or displaying a value.

```
>>> n = 17
>>> print(n)
```

EXPLANITION

The first line is an assignment statement that gives a value to n. The second line is a print statement that displays the value of n.

When you type a statement, the interpreter executes it, which means that it does whatever the statement says. In general, statements don’t have values.

SCRIPT MODE

So far we have run Python in interactive mode, which means that you interact directly with the interpreter. Interactive mode is a good way to get started, but if you are working with more than a few lines of code, it can be clumsy.

The alternative is to save code in a file called a script and then run the interpreter in script mode to execute the script. By convention, Python scripts have names that end with .py.

Because Python provides both modes, you can test bits of code in interactive mode before you put them in a script. But there are differences between interactive mode and script mode that can be confusing.

For example, if you are using Python as a calculator, you might type

```
>>> miles = 26.2
>>> miles * 1.61
42.182
```

The first line assigns a value to miles, but it has no visible effect. The second line is an expression, so the interpreter evaluates it and displays the result. It turns out that a marathon is about 42 kilometers.

But if you type the same code into a script and run it, you get no output at all. In script mode an expression, all by itself, has no visible effect. Python evaluates the expression, but it doesn't display the result. To display the result, you need a print statement like this:

```
miles = 26.2
print(miles * 1.61)
```

This behavior can be confusing at first. To check your understanding, type the following statements in the Python interpreter and see what they do:

```
5
x = 5
x + 1
```

Now put the same statements in a script and run it. What is the output? Modify the script by transforming each expression into a print statement and then run it again.

ORDER OF OPERATIONS

When an expression contains more than one operator, the order of evaluation depends on the order of operations. For mathematical operators, Python follows mathematical convention. The acronym PEMDAS is a useful way to remember the rules.

Parentheses have the highest precedence and can be used to force an expression to evaluate in the order you want. Since expressions in parentheses are evaluated first, 2 * (3-1) is 4, and (1+1)**(5-2) is 8. You can also use parentheses to make an expression easier to read, as in (minute * 100) / 60, even if it doesn't change the result.

Exponentiation has the next highest precedence, so 1 + 2**3 is 9, not 27, and 2 * 3**2 is 18, not 36.

Multiplication and Division have higher precedence than Addition and Subtraction. So 2*3-1 is 5, not 4, and 6+4/2 is 8, not 5.

Operators with the same precedence are evaluated from left to right (except exponentiation). So in the expression degrees / 2 * pi, the division happens first and the result is multiplied by pi. To divide by 2p, you can use parentheses or write degrees / 2 / pi.

STRING OPERATIONS

In general, you can't perform mathematical operations on strings, even if the strings look like numbers, so the following are illegal:

'chinese'-'food' 'eggs'/'easy' 'third'*'a charm'

But there are two exceptions, **+ and *.**

The + operator performs string concatenation, which means it joins the strings by linking them end-to-end. **For example:**

```
>>> first = 'throat'
>>> second = 'warbler'
>>> first + second
throatwarbler
```

The * operator also works on strings; it performs repetition. For example, 'Spam'*3 is 'SpamSpamSpam'. If one of the values is a string, the other has to be an integer.

This use of + and * makes sense by analogy with addition and multiplication. Just as 4*3 is equivalent to 4+4+4, we expect 'Spam'*3 to be the same as 'Spam'+'Spam'+'Spam', and it is.

On the other hand, there is a significant way in which string concatenation and repetition are different from integer addition and multiplication. Can you think of a property that addition has that string concatenation does not?

TYPES OF ERRORS

Three kinds of errors can occur in a program:

- syntax errors,
- runtime errors, and
- Semantic errors.

It is useful to distinguish between them in order to track them down more quickly.

Syntax error: “Syntax” refers to the structure of a program and the rules about that structure. For example, parentheses have to come in matching pairs, so (1 + 2) is legal, but 8) is a syntax error.

If there is a syntax error anywhere in your program, Python displays an error message and quits, and you will not be able to run the program. During the first few weeks of your programming career, you might spend a lot of time tracking down syntax errors. As you gain experience, you will make fewer errors and find them faster.

Runtime error: The second type of error is a runtime error, so called because the error does not appear until after the program has started running. These errors are also called exceptions because they usually indicate that something exceptional (and bad) has happened. Runtime errors are rare in the simple programs you will see in the first few chapters, so it might be a while before you encounter one.

Semantic error: The third type of error is “semantic”, which means related to meaning. If there is a semantic error in your program, it will run without generating error messages, but it will not do the right thing. It will do something else. Specifically, it will do what you told it to do. Identifying semantic errors can be tricky because it requires you to work backward by looking at the output of the program and trying to figure out what it is doing.

FUNCTIONS

- Functions are the most important aspect of an application.

- A function can be defined as the organized block of reusable code, which can be called whenever required.
- Python allows us to divide a large program into the basic building blocks known as a function.
- The function contains the set of programming statements enclosed by {}.
- A function can be called multiple times to provide reusability and modularity to the Python program.
- The Function helps to programmer to break the program into the smaller part.
- It organizes the code very effectively and avoids the repetition of the code. As the program grows, function makes the program more organized.
- Python provide us various inbuilt functions like range() or print(). Although, the user can create its functions, which can be called user-defined functions.

Advantage of Functions in Python

- Using functions, we can avoid rewriting the same logic/code again and again in a program.
- We can call Python functions multiple times in a program and anywhere in a program.
- We can track a large Python program easily when it is divided into multiple functions.
- Reusability is the main achievement of Python functions.
- However, Function calling is always overhead in a Python program.

Creating a Function

Python provides the def keyword to define the function.

Syntax:

```
def my_function(parameters):
      function_block
return expression
```

Let's understand the syntax of functions definition.

- The **def keyword**, along with the function name is used to define the function.
- The identifier rule must follow the function name.
- A function accepts the parameter (argument), and they can be optional.

- The function block is started with the colon (:), and block statements must be at the same indentation.
- The return statement is used to return the value. A function can have only one return.

Function Calling

- In Python, after the function is created, we can call it from another function.
- A function must be defined before the function call; otherwise, the Python interpreter gives an error.
- To call the function, use the function name followed by the parentheses.

Consider the following example of a simple example that prints the message "Hello World".

```
#function definition
def hello_world():
    print("hello world")
# function calling
hello_world()
```

Output:

```
hello world
```

There are mainly two types of functions.

1. User-define functions - The user-defined functions are those define by the user to perform the specific task.

2. Built-in functions - The built-in functions are those functions that are pre-defined in Python.

In this tutorial, we will discuss the user define functions.

1. USER DEFINED FUNCTION WITHOUT ARGUMENTS:

EXAMPLE:

```
def calculation():#this is fuction defination
    a=int(input("Enter a number:"))
    b=int(input("Enter a number:"))
    print("addition:",a+b)
    print("substraction:",a-b)
    print("multiplication:",a*b)
    print("division:",a/b)
    print("Modulus:",a%b)
```

```
calculation()#this is function call
```

INPUT:

```
Enter a number:3
Enter a number:2
```

OUTPUT:

```
addition: 5
substraction: 1
multiplication: 6
division: 1.5
Modulus: 1
```

2. USER DEFINED FUNCTION WITH ARGUMENTS:

EXAMPLE:

```
def calculation(a,b):#this is fuction defination
        print("addition:",a+b)
        print("substraction:",a-b)
        print("multiplication:",a*b)
        print("division:",a/b)
        print("Modulus:",a%b)
a=int(input("Enter a number:"))
b=int(input("Enter a number:"))
calculation(a,b)#this is function call
```

INPUT:

```
Enter a number:3
Enter a number:2
```

OUTPUT:

```
addition: 5
substraction: 1
multiplication: 6
division: 1.5
Modulus: 1
```

The return statement

- The return statement is used at the end of the function and returns the result of the function.
- It terminates the function execution and transfers the result where the function is called.

- The return statement cannot be used outside of the function.
- It can contain the expression which gets evaluated and value is returned to the caller function.
- If the return statement has no expression or does not exist itself in the function then it returns the None object.

Syntax

```
return [expression_list]
```

Consider the following example:

Example 1

```
# Defining function
def sum():
    a = 10
    b = 20
    c = a+b
    return c
# calling sum() function in print statement
print("The sum is:",sum())
```

Output:

```
The sum is: 30
```

In the above code, we have defined the function named sum, and it has a statement c = a+b, which computes the given values, and the result is returned by the return statement to the caller function.

Example 2 Creating function without return statement

```
# Defining function
def sum():
    a = 10
    b = 20
    c = a+b
# calling sum() function in print statement
print(sum())
```

Output:

```
None
```

In the above code, we have defined the same function without the return statement as we can see that the sum() function returned the None object to the caller function.

ARGUMENTS IN FUNCTION

The arguments are types of information which can be passed into the function. The arguments are specified in the parentheses. We can pass any number of arguments, but they must be separate them with a comma.

Consider the following example, which contains a function that accepts a string as the argument.

Example 1

```
#defining the function
def func (name):
    print("Hi ",name)
#calling the function
func("Dev")
```

Output:

```
Hi Dev
```

Example 2

```
#Python function to calculate the sum of two variables
#defining the function
def sum (a,b):
    return a+b;
#taking values from the user
a = int(input("Enter a: "))
b = int(input("Enter b: "))
#printing the sum of a and b
print("Sum = ",sum(a,b))
```

Output:

```
Enter a: 10
Enter b: 20
Sum =  30
```

CALL BY REFERENCE

In Python, call by reference means passing the actual value as an argument in the function. All the functions are called by reference, i.e., all the changes made to the reference inside the function revert back to the original value referred by the reference.

Example 1 Passing Immutable Object (List)

```
#defining the function
def change_list(list1):
    list1.append(20)
    list1.append(30)
    print("list inside function = ",list1)
#defining the list
list1 = [10,30,40,50]
#calling the function
change_list(list1)
print("list outside function = ",list1)
```

Output:

```
list inside function =  [10, 30, 40, 50, 20, 30]
list outside function =  [10, 30, 40, 50, 20, 30]
```

Example 2 Passing Mutable Object (String)

```
#defining the function
def change_string (str):
    str = str + " Hows you "
    print("printing the string inside function :",str)
string1 = "Hi I am there"
#calling the function
change_string(string1)
print("printing the string outside function :",string1)
```

Output:

```
printing the string inside function : Hi I am there Hows you
printing the string outside function : Hi I am there
```

TYPES OF ARGUMENTS

There may be several types of arguments which can be passed at the time of function call.

- Required arguments
- Keyword arguments
- Default arguments
- Variable-length arguments

Required Arguments

Till now, we have learned about function calling in Python. However, we can provide the arguments at the time of the function call. As far as the required arguments are concerned, these are the arguments which are required to be passed at the time of function calling with the exact match of their positions in the function call and function definition. If either of the arguments is not provided in the function call, or the position of the arguments is changed, the Python interpreter will show the error.

Example 1

```
def func(name):
    message = "Hi "+name
    return message
name = input("Enter the name:")
print(func(name))
```

Output:

```
Enter the name: John
Hi John
```

Example 2

```
#the function simple_interest accepts three arguments and returns the simple interest accordingly
def simple_interest(p,t,r):
    return (p*t*r)/100
p = float(input("Enter the principle amount? "))
```

```
r = float(input("Enter the rate of interest? "))
t = float(input("Enter the time in years? "))
print("Simple Interest: ",simple_interest(p,r,t))
```

Output:

```
Enter the principle amount: 5000
Enter the rate of interest: 5
Enter the time in years: 3
Simple Interest:  750.0
```

DEFAULT ARGUMENTS

Python allows us to initialize the arguments at the function definition. If the value of any of the arguments is not provided at the time of function call, then that argument can be initialized with the value given in the definition even if the argument is not specified at the function call.

Example 1

```
def printme(name,age=22):
    print("My name is",name,"and age is",age)
printme(name = "john")
```

Output:

```
My name is John and age is 22
```

Example 2

```
def printme(name,age=22):
    print("My name is",name,"and age is",age)
printme(name = "john") #the variable age is not passed into the function however the default value of age is considered in the function
printme(age = 10,name="David") #the value of age is overwritten here, 10 will be printed as age
```

Output:

```
My name is john and age is 22
My name is David and age is 10
```

VARIABLE-LENGTH ARGUMENTS (*ARGS)

In large projects, sometimes we may not know the number of arguments to be passed in advance. In such cases, Python provides us the flexibility to offer the comma-separated values which are internally treated as tuples at the function call. By using the variable-length arguments, we can pass any number of arguments.

However, at the function definition, we define the variable-length argument using the *args (star) as *<variable - name >.

Example

```
def printme(*names):
    print("type of passed argument is ",type(names))
    print("printing the passed arguments...")
    for name in names:
        print(name)
printme("john","David","smith","nick")
```

Output:

```
type of passed argument is  <class 'tuple'>
printing the passed arguments...
john
David
smith
nick
```

In the above code, we passed *names as variable-length argument. We called the function and passed values which are treated as tuple internally. The tuple is an iterable sequence the same as the list. To print the given values, we iterated *arg names using for loop.

KEYWORD ARGUMENTS(KWARGS)**

Python allows us to call the function with the keyword arguments. This kind of function call will enable us to pass the arguments in the random order.

The name of the arguments is treated as the keywords and matched in the function calling and definition. If the same match is found, the values of the arguments are copied in the function definition.

Example 1

```
#function func is called with the name and message as the keyword arguments
def func(name,message):
    print("printing the message with",name,"and ",message)


    #name and message is copied with the values John and hello respectively
    func(name = "John",message="hello")
```

Output:

```
                    printing the message with John and  hello
```

Example 2 providing the values in different order at the calling

```
#The function simple_interest(p, t, r) is called with the keyword arguments the order of arguments doesn't matter in this case
def simple_interest(p,t,r):
    return (p*t*r)/100
print("Simple Interest: ",simple_interest(t=10,r=10,p=1900))
```

Output:

```
                    Simple Interest:  1900.0
```

If we provide the different name of arguments at the time of function call, an error will be thrown.

SCOPE OF VARIABLES

The scopes of the variables depend upon the location where the variable is being declared. The variable declared in one part of the program may not be accessible to the other parts.

In python, the variables are defined with the two types of scopes.

- Global variables
- Local variables

The variable defined outside any function is known to have a global scope, whereas the variable defined inside a function is known to have a local scope.

Example 1 Local Variable

```
def print_message():
   message = "hello !! I am going to print a message." # the variable message is local to the function itself
   print(message)
print_message()
print(message) # this will cause an error since a local variable cannot be accessible here.
```

Output:

```
hello !! I am going to print a message.
  File "/root/PycharmProjects/PythonTest/Test1.py", line 5, in
    print(message)
NameError: name 'message' is not defined
```

Example 2 Global Variable

```
def calculate(*args):
   sum=0
   for arg in args:
       sum = sum +arg
   print("The sum is",sum)
sum=0
calculate(10,20,30) #60 will be printed as the sum
print("Value of sum outside the function:",sum) # 0 will be printed  Output:
```

Output:

```
The sum is 60
Value of sum outside the function: 0
```

EXAMPLE

```
glo_var=50
def fun():
  loc_var=40
  print ("Inside Fun:Global variable is:",glo_var)
  print ("Inside Fun Local variable is:",loc_var)
fun()
print ("Outside Fun:Global variable is:",glo_var)
print ("Outside Fun Local variable is:",loc_var)
```

OUTPUT:

```
Inside Fun:Global variable is: 50
Inside Fun Local variable is: 40
Outside Fun:Global variable is: 50
        TRACEBACK (MOST RECENT CALL LAST):
          File “fun5.py”, line 8, in <module>
            print (“Outside Fun Local variable is:”,loc_var)
        NameError: name ‘loc_var’ is not defined
```

Note: If we observe above example, i am trying to access the local variable outside of the function,after execute got is not defined ,because of it is a local variable .

EXAMPLE

```
var="Global"
def fun():
  var="Local"
  print("Inside Fun:Value of the variable is :",var)
fun()
print("Outside Fun:Value of the variable is :",var)
```

OUTPUT:

```
Inside Fun:Value of the variable is : Local
Outside Fun:Value of the variable is : Global
```

DEFAULT ARGUMENT OR DEFAULT PARAMETERS:

Default arguments is nothing but whenever we are defining the function at the time only we can initialize the variable to that function definition and whenever we invoke the function without any arguments that time it will take from the already assigned values to arguments inside the definition

SYNTAX:

```
def fun_name(arg=value1,arg2=value2……..argn=valuen):
```

EXAMPLE:

```
def default_arg_fun(a=10,b=20,c=30):
        print("Value a is:",a)
        print("Value b is:",b)
        print("Value c is:",c)
x=1
y=2
z=3
print("Calling without Args")
default_arg_fun()
print("Calling with single Arg")
default_arg_fun(x)
print("Calling with two Args")
default_arg_fun(x,y)
print("Calling with two Args")
default_arg_fun(x,z)
print("Calling with three Args")
default_arg_fun(x,y,z)
```

OUTPUT:

```
Calling without Args
Value a is: 10
Value b is: 20
Value c is: 30
Calling with single Arg
Value a is: 1
Value b is: 20
Value c is: 30
Calling with two Args
Value a is: 1
Value b is: 2
Value c is: 30
Calling with two Args
Value a is: 1
Value b is: 3
Value c is: 30
Calling with three Args
Value a is: 1
Value b is: 2
Value c is: 3
```

FUNCTION CALLS

We have already seen one example of a function call:

```
>>> type(42)
<class 'int'>
```

The name of the function is type. The expression in parentheses is called the argument of the function. The result, for this function, is the type of the argument.

It is common to say that a function “takes” an argument and “returns” a result. The result is also called the return value.

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

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

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

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

float converts integers and strings to floating-point numbers:

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

Finally, str converts its argument to a string:

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

MATH FUNCTIONS

Python has a math module that provides most of the familiar mathematical functions. A module is a file that contains a collection of related functions.

Before we can use the functions in a module, we have to import it with an import statement:

```
>>> import math
```

This statement creates a module object named math. If you display the module object, you get some information about it:

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

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

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

The first example uses math.log10 to compute a signal-to-noise ratio in decibels (assuming that signal_power and noise_power are defined). The math module also provides log, which computes logarithms base e.

The second example finds the sine of radians. The variable name radians is a hint that sin and the other trigonometric functions (cos, tan, etc.) take arguments in radians. To convert from degrees to radians, divide by 180 and multiply by p:

```
>>> degrees = 45
>>> radians = degrees / 180.0 * math.pi
>>> math.sin(radians)
0.707106781187
```

The expression math.pi gets the variable pi from the math module. Its value is a floating point approximation of p, accurate to about 15 digits.

If you know trigonometry, you can check the previous result by comparing it to the square root of two divided by two:

```
>>> math.sqrt(2) / 2.0
0.707106781187
```

ADDING NEW FUNCTIONS

we have only been using the functions that come with Python, but it is also possible to add new functions. A function definition specifies the name of a new function and the sequence of statements that run when the function is called.

Here is an example:

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

def is a keyword that indicates that this is a function definition. The name of the function is print_lyrics. The rules for function names are the same as for variable names: letters, numbers and underscore are legal, but the first character can’t be a number. You can’t use a keyword as the name of a function, and you should avoid having a variable and a function with the same name.

The empty parentheses after the name indicate that this function doesn’t take any arguments.The first line of the function definition is called the **header**; the rest is called the **body**. The **header** has to **end** with a **colon:** and the body has to be indented.

By convention, indentation is always four spaces. The body can contain any number of statements.

The strings in the print statements are enclosed in double quotes. Single quotes and double quotes do the same thing; most people use single quotes except in cases like this where a single quote (which is also an apostrophe) appears in the string.

All quotation marks (single and double) must be “straight quotes”, usually located next to Enter on the keyboard. “Curly quotes”, like the ones in this sentence, are not legal in Python.

If you type a function definition in interactive mode, the interpreter prints dots (...) to let you know that the definition isn’t complete.

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

To end the function, you have to enter an empty line.

Defining a function creates a function object, which has type function:

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

The syntax for calling the new function is the same as for built-in functions:

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

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

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

And then call repeat_lyrics:

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

But that’s not really how the song goes.

DEFINITIONS AND USES

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

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

This program contains two function definitions: print_lyrics and repeat_lyrics. Function definitions get executed just like other statements, but the effect is to create function objects. The statements inside the function do not run until the function is called, and the function definition generates no output.

- As you might expect, you have to create a function before you can run it. In other words, the function definition has to run before the function gets called.
- As an exercise, move the last line of this program to the top, so the function call appears before the definitions. Run the program and see what error message you get.
- Now move the function call back to the bottom and move the definition of print_lyrics after the definition of repeat_lyrics. What happens when you run this program?

FLOW OF EXECUTION

- To ensure that a function is defined before its first use, you have to know the order statements run in, which is called the flow of execution.
- Execution always begins at the first statement of the program. Statements are run one at a time, in order from top to bottom.
- Function definitions do not alter the flow of execution of the program, but remember that statements inside the function don’t run until the function is called.
- A function call is like a detour in the flow of execution. Instead of going to the next statement, the flow jumps to the body of the function, runs the statements there, and then comes back to pick up where it left off.
- That sounds simple enough, until you remember that one function can call another. While in the middle of one function, the program might have to run the statements in another

function. Then, while running that new function, the program might have to run yet another function!

- Fortunately, Python is good at keeping track of where it is, so each time a function completes, the program picks up where it left off in the function that called it. When it gets to the end of the program, it terminates.
- In summary, when you read a program, you don’t always want to read from top to bottom. Sometimes it makes more sense if you follow the flow of execution.

PARAMETERS AND ARGUMENTS

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

Inside the function, the arguments are assigned to variables called parameters. Here is a definition for a function that takes an argument:

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

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

This function works with any value that can be printed.

```
>>> print_twice('Spam')
Spam
Spam
>>> print_twice(42)
42
42
>>> print_twice(math.pi)
3.14159265359
3.14159265359
```

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

```
>>> print_twice('Spam '*4)
```

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

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

You can also use a variable as an argument:

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

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

VARIABLES AND PARAMETERS ARE LOCAL

When you create a variable inside a function, it is local, which means that it only exists inside the function.

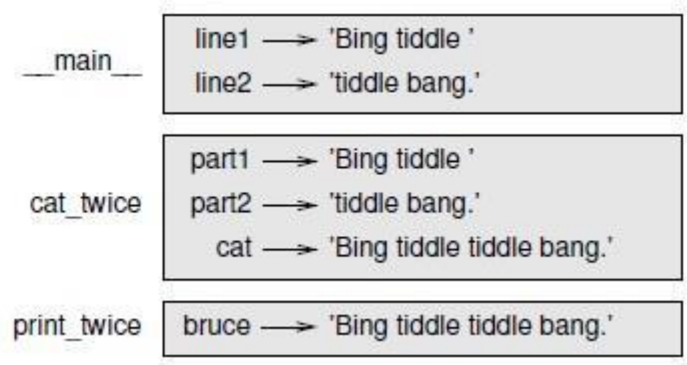
For example:

```
def cat_twice(part1, part2):
cat = part1 + part2
print_twice(cat)
```

This function takes two arguments, concatenates them, and prints the result twice. Here is an example that uses it:

```
>>> line1 = 'Bing tiddle '
>>> line2 = 'tiddle bang.'
>>> cat_twice(line1, line2)
Bing tiddle tiddle bang.
Bing tiddle tiddle bang.
```

When cat_twice terminates, the variable cat is destroyed. If we try to print it, we get an exception.



```
>>> print(cat)
NameError: name 'cat' is not defined
```

Parameters are also local. For example, outside print_twice, there is no such thing as bruce.

STACK DIAGRAMS

- To keep track of which variables can be used where, it is sometimes useful to draw a stack diagram. Like state diagrams, stack diagrams show the value of each variable, but they also show the function each variable belongs to.
- Each function is represented by a frame. A frame is a box with the name of a function beside it and the parameters and variables of the function inside it. The stack diagram for the previous example is shown in above figure.
- The frames are arranged in a stack that indicates which function called which, and so on. In this example, print_twice was called by cat_twice, and cat_twice was called by __main__, which is a special name for the topmost frame. When you create a variable outside of any function, it belongs to __main__.
- Each parameter refers to the same value as its corresponding argument. So, part1 has the same value as line1, part2 has the same value as line2, and bruce has the same value as cat.
- If an error occurs during a function call, Python prints the name of the function, the name of the function that called it, and the name of the function that called that, all the way back to __main__.

For example, if you try to access cat from within print_twice, you get a NameError:

```
Traceback (innermost last):
File "test.py", line 13, in __main__
cat_twice(line1, line2)
File "test.py", line 5, in cat_twice
print_twice(cat)
File "test.py", line 9, in print_twice
print(cat)
NameError: name 'cat' is not defined
```

- This list of functions is called a traceback. It tells you what program file the error occurred in, and what line, and what functions were executing at the time. It also shows the line of code that caused the error.
- The order of the functions in the traceback is the same as the order of the frames in the stack diagram. The function that is currently running is at the bottom.

FRUITFUL FUNCTIONS AND VOID FUNCTIONS

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

When you call a fruitful function, you almost always want to do something with the result;

For example, you might assign it to a variable or use it as part of an expression:

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

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

```
>>> math.sqrt(5)
2.2360679774997898
```

But in a script, if you call a fruitful function all by itself, the return value is lost forever!

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

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

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

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

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

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

The functions we have written so far are all void. We will start writing fruitful functions in a few chapters.

WHY FUNCTIONS?

- It may not be clear why it is worth the trouble to divide a program into functions. There are several reasons.
- Creating a new function gives you an opportunity to name a group of statements, which makes your program easier to read and debug.
- Functions can make a program smaller by eliminating repetitive code. Later, if you make a change, you only have to make it in one place.
- Dividing a long program into functions allows you to debug the parts one at a time and then assemble them into a working whole.
- Well-designed functions are often useful for many programs. Once you write and debug one, you can reuse it.

************* END OF FIRST UNIT *************

UNIT – II

Unit – II

Case study: The turtle module, Simple Repetition, Encapsulation, Generalization, Interface design, Refactoring, docstring.

Conditionals and Recursion: floor division and modulus, Boolean expressions, Logical operators, Conditional execution, Alternative execution, Chained conditionals, Nested conditionals, Recursion, Infinite Recursion, Keyboard input.

Fruitful Functions: Return values, Incremental development, Composition, Boolean functions, More recursion, Leap of Faith, Checking types

Unit Outcomes:

Student should be able to

- **Apply the conditional execution of the program.**
- **Apply the principle of recursion to solve the problems.**

Case Study: Interface Design

This chapter presents a case study that demonstrates a process for designing functions that work together.

It introduces the turtle module, which allows you to create images using turtle graphics. The turtle module is included in most Python installations, but if you are running Python using Python Anywhere, you won’t be able to run the turtle examples.

The turtle Module

To check whether you have the turtle module, open the Python interpreter and type:

```
>>> import turtle

>>> bob = turtle.Turtle()
```

When you run this code, it should create a new window with a small arrow that represents the turtle. Close the window.

Overview of available Turtle and Screen methods

Turtle methods

Turtle motion

Move and draw

forward() | fd()

backward() | bk() | back()

right() | rt()

left() | lt()

goto() | setpos() | setposition()

setx()

sety()

setheading() | seth()

home()

circle()

dot()

stamp()

clearstamp()

clearstamps()

undo()

speed()

Tell Turtle’s state

position() | pos()

towards()

xcor()

ycor()

heading()

distance()

Setting and measurement

degrees()

radians()

Pen control

Drawing state

pendown() | pd() | down()

penup() | pu() | up()

pensize() | width()

pen()

isdown()

Color control

color()

pencolor()

fillcolor()

Filling

filling()

begin_fill()

end_fill()

More drawing control

reset()

clear()

write()

Turtle state

Visibility

showturtle() | st()

hideturtle() | ht()

isvisible()

Appearance

shape()

resizemode()

shapesize() | turtlesize()

shearfactor()

settiltangle()

tiltangle()

tilt()

shapetransform()

get_shapepoly()

Using events

onclick()

onrelease()

ondrag()

Special Turtle methods

begin_poly()

end_poly()

get_poly()

clone()

getturtle() | getpen()

getscreen()

setundobuffer()

undobufferentries()

Methods of TurtleScreen/Screen

Window control

bgcolor()

bgpic()

clear() | clearscreen()

reset() | resetscreen()

screensize()

setworldcoordinates()

Animation control

delay()

tracer()

update()

Using screen events

listen()

onkey() | onkeyrelease()

onkeypress()

onclick() | onscreenclick()

ontimer()

mainloop() | done()

Settings and special methods

mode()

colormode()

getcanvas()

getshapes()

register_shape() | addshape()

turtles()

window_height()

window_width()

Input methods

textinput()

numinput()

Methods specific to Screen

bye()

exitonclick()

setup()

title()

Create a file named mypolygon.py and type in the following code:

```
import turtle

bob = turtle.Turtle()

print(bob)

turtle.mainloop()
```

OUTPUT

```
<turtle.Turtle object at 0x03041FB0>
```

Code Explanation

The turtle module (with a lowercase t) provides a function called Turtle (with an uppercase T) that creates a Turtle object, which we assign to a variable named bob. Printing bob displays something like:

```
<turtle.Turtle object at 0xb7bfbf4c>
```

- This means that bob refers to an object with type Turtle as defined in module turtle.
- mainloop tells the window to wait for the user to do something, although in this case there’s not much for the user to do except close the window.

Once you create a Turtle, you can call a method to move it around the window. A method is similar to a function, but it uses slightly different syntax. For example, to move the turtle forward:

```
bob.fd(100)
```

The method, fd to move forward, is associated with the turtle object we're calling bob. Calling a method is like making a request: you are asking bob to move forward.

The argument of fd is a distance in pixels, so the actual size depends on your display. Other methods you can call on a Turtle are bk to move backward, lt for left turn, and

rt right turn. The argument for lt and rt is an angle in degrees.

Also, each Turtle is holding a pen, which is either down or up; if the pen is down, the Turtle leaves a trail when it moves. The methods pu and pd stand for "pen up" and "pen down".

```
def polyline(t, n, length, angle):
    """Draws n line segments.

    t: Turtle object
    n: number of line segments
    length: length of each segment
    angle: degrees between segments
    """
    for i in range(n):
        fd(t, length)
        lt(t, angle)
```

To draw a right angle, add these lines to the program (after creating bob and before calling mainloop):

```
bob.fd(100)

bob.lt(90)

bob.fd(100)
```

OUTPUT

When you run this program, you should see bob move east and then north, leaving two line segments behind.Now modify the program to draw a square. Don’t go on until you’ve got it working!

Create a file named rectangle.py and type in the following code:

```
import turtle

bob=turtle.Turtle()

turtle.mainloop()

bob.color("red")

bob.fd(100)

bob.color("green")

bob.fd(100)

bob.lt(90)

bob.color("blue")

bob.fd(100)

bob.lt(90)

bob.color("red")

bob.fd(100)

bob.color("green")

bob.fd(100)

bob.lt(90)

bob.color("black")

bob.fd(100)
```

OUTPUT

Simple Repetition

```
bob.fd(100)
bob.lt(90)
bob.fd(100)
bob.lt(90)
bob.fd(100)
bob.lt(90)
bob.fd(100)
```

We can do the same thing more concisely with a for statement. Add this example to mypolygon.py and run it again:

```
for i in range(4):
    print('Hello!')
```

You should see something like this:

```
Hello!
Hello!
Hello!
Hello!
```

This is the simplest use of the for statement; we will see more later. But that should be enough to let you rewrite your square-drawing program. Don't go on until you do.

Here is a for statement that draws a square:

```
for i in range(4):
  bob.fd(100)
  bob.lt(90)
```

OUTPUT

Encapsulation

The first exercise asks you to put your square-drawing code into a function definition and then call the function, passing the turtle as a parameter. Here is a solution:

```
import turtle
bob=turtle.Turtle()
def square(t):
  for i in range(4):
    t.fd(100)
    t.lt(90)
square(bob)
```

OUTPUT

The innermost statements, fd and lt, are indented twice to show that they are inside the for loop, which is inside the function definition. The next line, square(bob), is flush with the left margin, which indicates the end of both the for loop and the function definition.

Inside the function, t refers to the same turtle bob, so t.lt(90) has the same effect as bob.lt(90). In that case, why not call the parameter bob? The idea is that t can be any turtle, not just bob, so you could create a second turtle and pass it as an argument to square:

```
import turtle

bob=turtle.Turtle()

def square(t):

  for i in range(4):

    t.fd(100)

    t.lt(90)

square(bob)

alice = bob #alice copies the properties of bob object

square(alice) # Square function is invoked by using alice (bob’s) properties (alice encapsulates bob’s properties)
```

OUTPUT (two times the square is drawn)

Wrapping a piece of code up in a function is called encapsulation. One of the benefits of encapsulation is that it attaches a name to the code, which serves as a kind of documentation. Another advantage is that if you reuse the code, it is more concise to call a function twice than to copy and paste the body!

Generalization

The next step is to add a length parameter to square. Here is a solution:

```
import turtle
bob=turtle.Turtle()
def square(t, length):
  for i in range(4):
    t.fd(length)
    t.lt(90)
square(bob, 50) #Square function takes any object (bob) as the first argument and any value  as the second argument (parameter passing or Generalization)
```

Adding a parameter to a function is called generalization because it makes the function more general: in the previous version, the square is always the same size; in this version it can be any size.

The next step is also a generalization. Instead of drawing squares, polygon draws regular polygons with any number of sides. Here is a solution:

```
def polygon(t, n, length):
    """Draws a polygon with n sides.
    t: Turtle
    n: number of sides
    length: length of each side.
    """
    angle = 360.0/n
    polyline(t, n, length, angle)


import turtle
bob=turtle.Turtle()
def polygon(t, sides, length):
  angle = 360/ sides
  for i in range(sides):
    t.fd(length)
    t.lt(angle)
bob.color("black")
polygon(bob, sides=3,length=100)#Triangle as keyword arguments
bob.color("violet")
polygon(bob, 4, 100)#Square
bob.color("blue")
polygon(bob, 5, 100)#Pentagon
bob.color("red")
polygon(bob, 6, 100)#Hexagon
bob.color("green")
```

```
polygon(bob, 7, 100)#Septagon

bob.color("brown")

polygon(bob, 8, 100)#Octagon          import turtle

bob=turtle.Turtle()

def polygon(t, sides, length):

  angle = 360.0 / sides #Float value at Numerator

  for i in range(sides):

    t.fd(length)

    t.lt(angle)

bob.color("black")

polygon(bob, sides=3, length=100)#Triangle as keyword arguments

bob.color("violet")

polygon(bob, 4, 100)#Square

bob.color("blue")

polygon(bob, 5, 100)#Pentagon

bob.color("red")

polygon(bob, 6, 100)#Hexagon

bob.color("green")

polygon(bob, 7, 100)#Septagon

bob.color("brown")

polygon(bob, 8, 100)#Octagon
```

OUTPUT

This example draws from 3 till 8-sided polygon with different sides and angles (interior).

If you are using Python 2, the value of angle might be off because of integer division. A simple solution is to compute angle = 360.0 / n. Because the numerator is a floating-point number, the result is floating point. When a function has more than a few numeric arguments, it is easy to forget what they are, or what order they should be in. In that case it is often a good idea to include the names of the parameters in the argument list:

```
polygon(bob, n=7, length=70)
```

These are called keyword arguments because they include the parameter names as “keywords” (not to be confused with Python keywords like while and def).

This syntax makes the program more readable. It is also a reminder about how arguments and parameters work: when you call a function, the arguments are assigned to the parameters.

Interface Design

The next step is to write circle, which takes a radius, r, as a parameter. Here is a simple solution that uses polygon to draw a 50-sided polygon:

```
def circle(t, r):
    circumference = 2 * math.pi * r
    n = 50
    length = circumference / n
    polygon(t, n, length)
```

```
import turtle

bob=turtle.Turtle()

bob.color('green', 'cyan')

bob.begin_fill()

def polygon(t,length, n):

    for i in range(n):

        bob.forward(length)

        bob.left(360.0/n)


import math

def circle(t, r):

        circumference= 2*math.pi*r

        n= 50

        length= circumference/n

        polygon(t,length, n)


circle(bob, 100)

bob.end_fill()

turtle.done()
```

OUTPUT

The first line computes the circumference of a circle with radius r using the formula 2 π r. Since we use math.pi, we have to import math. By convention, import statements are usually at the beginning of the script. n is the number of line segments in our approximation of a circle, so length is the length of each segment. Thus, polygon draws a 50-sides polygon that approximates a circle with radius r.

One limitation of this solution is that n is a constant, which means that for very big circles, the line segments are too long, and for small circles, we waste time drawing very small segments. One solution would be to generalize the function by taking n as a parameter. This would give the user (whoever calls circle) more control, but the interface would be less clean.

The interface of a function is a summary of how it is used: what are the parameters? What does the function do? And what is the return value? An interface is “clean” if it allows the caller to do what they want without dealing with unnecessary details.

In this example, r belongs in the interface because it specifies the circle to be drawn. n is less appropriate because it pertains to the details of how the circle should be rendered. Rather than clutter up the interface, it is better to choose an appropriate value of n depending on circumference:

```
def circle(t, r):

    circumference = 2 * math.pi * r

    n = int(circumference / 3) + 1

    length = circumference / n

    polygon(t, n, length)


import turtle

bob=turtle.Turtle()

bob.color('green', 'cyan')

bob.begin_fill()
```

```
def polygon(t,length, n):
    for i in range(n):
        bob.forward(length)
        bob.left(360.0/n)


import math
def circle(t, r):
        circumference= 2*math.pi*r
        n= int(circumference/10)+1
        length= circumference/n
        polygon(t,length, n)


circle(bob, 100)
bob.end_fill()
turtle.done()
```

OUTPUT

Now the number of segments is (approximately) circumference/3, so the length of each segment is (approximately) 3, which is small enough that the circles look good, but big enough to be efficient, and appropriate for any size circle.

```
import turtle

"""Draws a polygon with n sides.

    t: Turtle

    n: number of sides

    length: length of each side.

    """

# creating turtle pen

t = turtle.Turtle()

# taking input for the no of the sides of the polygon

n = int(input("Enter the no of the sides of the polygon : "))

# taking input for the length of the sides of the polygon

l = int(input("Enter the length of the sides of the polygon : "))

for _ in range(n):

    turtle.forward(l)

    turtle.right(360.0 / n)
```

OUTPUT

Refactoring

When I wrote circle, I was able to reuse polygon because a many-sided polygon is a good approximation of a circle. But arc is not as cooperative; we can’t use polygon or circle to draw an arc.

One alternative is to start with a copy of polygon and transform it into arc. The result might look like this:

```
def arc(t, r, angle):
arc_length = 2 * math.pi * r * angle / 360
n = int(arc_length / 3) + 1
step_length = arc_length / n
step_angle = angle / n
for i in range(n):
t.fd(step_length)
t.lt(step_angle)


import math
import turtle


bob = turtle.Turtle()


def polyline(t, n, length, angle):
    """Draws n line segments.

    t: Turtle object
    n: number of line segments
    length: length of each segment
    angle: degrees between segments
    """
    for i in range(n):
        t.fd(length)
        t.lt(angle)


def polygon(t, n, length):
    """Draws a polygon with n sides.

    t: Turtle
    n: number of sides
    length: length of each side.
    """
    angle = 360.0/n
    polyline(t, n, length, angle)


def arc(t, r, angle):
    """Draws an arc with the given radius and angle.

    t: Turtle
    r: radius
    angle: angle subtended by the arc, in degrees
    """
    arc_length = 2 * math.pi * r * abs(angle) / 360
    n = int(arc_length / 4) + 3
    step_length = arc_length / n
    step_angle = float(angle) / n

    # making a slight left turn before starting reduces
    # the error caused by the linear approximation of the arc
    t.lt(step_angle/2)
    polyline(t, n, step_length, step_angle)
    t.rt(step_angle/2)

def circle(t, r):
    """Draws a circle with the given radius.

    t: Turtle
    r: radius
    """
    arc(t, r, 90) #    arc(t, r, 180) – Half Circle  #    arc(t, r, 360) – Full Circle

circle(bob, 100)
```

OUTPUT

The second half of this function looks like polygon, but we can’t reuse polygon without changing the interface. We could generalize polygon to take an angle as a third argument, but then polygon would no longer be an appropriate name! Instead, let’s call the more general function polyline:

```
def polyline(t, n, length, angle):
for i in range(n):
t.fd(length)
 t.lt(angle)
```

Now we can rewrite polygon and arc to use polyline:

```
def polygon(t, n, length):

angle = 360.0 / n
polyline(t, n, length, angle)

def arc(t, r, angle):
arc_length = 2 * math.pi * r * angle / 360
n = int(arc_length / 3) + 1
step_length = arc_length / n
step_angle = float(angle) / n
polyline(t, n, step_length, step_angle)
```

Finally, we can rewrite circle to use arc:

```
def circle(t, r):

arc(t, r, 360)
```

This process—rearranging a program to improve interfaces and facilitate code reuse is called refactoring. In this case, we noticed that there was similar code in arc and polygon, so we “factored it out” into polyline.

If we had planned ahead, we might have written polyline first and avoided refactoring, but often you don’t know enough at the beginning of a project to design all the interfaces. Once you start coding, you understand the problem better. Sometimes refactoring is a sign that you have learned something.

A Development Plan

A development plan is a process for writing programs. The process we used in this case study is “encapsulation and generalization”. The steps of this process are:

1. Start by writing a small program with no function definitions.

2. Once you get the program working, identify a coherent piece of it, encapsulate the piece in a function and give it a name.

3. Generalize the function by adding appropriate parameters.

4. Repeat steps 1–3 until you have a set of working functions. Copy and paste working code to avoid retyping (and re-debugging).

5. Look for opportunities to improve the program by refactoring. For example, if you have similar code in several places, consider factoring it into an appropriately general function.

This process has some drawbacks, we will see alternatives later but it can be useful if you don’t know ahead of time how to divide the program into functions. This approach lets you design as you go along.

docstring

A docstring is a string at the beginning of a function that explains the interface ("doc" is short for "documentation"). Here is an example:

```
def polyline(t, n, length, angle):

"""Draws n line segments with the given length and angle (in degrees) between them. t is a turtle.
"""

for i in range(n):

t.fd(length)

t.lt(angle)
```

By convention, all docstrings are triple-quoted strings, also known as multiline strings because the triple quotes allow the string to span more than one line.

It is terse, but it contains the essential information someone would need to use this function. It explains concisely what the function does (without getting into the details of how it does it). It explains what effect each parameter has on the behavior of the function and what type each parameter should be (if it is not obvious).

Writing this kind of documentation is an important part of interface design. A well- designed interface should be simple to explain; if you have a hard time explaining one of your functions, maybe the interface could be improved.

Debugging

An interface is like a contract between a function and a caller. The caller agrees to provide certain parameters and the function agrees to do certain work.

For example, polyline requires four arguments: t has to be a Turtle; n has to be an integer; length should be a positive number; and angle has to be a number, which is understood to be in degrees.

These requirements are called pre conditions because they are supposed to be true before the function starts executing. Conversely, conditions at the end of the function are post conditions. Post conditions include the intended effect of the function (like drawing line segments) and any side effects (like moving the Turtle or making other changes).

Preconditions are the responsibility of the caller. If the caller violates a (properly documented!) precondition and the function doesn't work correctly, the bug is in the caller, not the function.

If the preconditions are satisfied and the post conditions are not, the bug is in the function. If your pre- and post conditions are clear, they can help with debugging.

Conditionals and Recursion

The main topic of this chapter is the if statement, which executes different code depending on the state of the program. Two new operators: floor division and modulus.

Floor Division and Modulus

The floor division operator, //, divides two numbers and rounds down to an integer. For example, suppose the run time of a movie is 105 minutes. You might want to know how long that is in hours. Conventional division returns a floating-point number:

```
>>> minutes = 105

>>> minutes / 60

1.75
```

But we don't normally write hours with decimal points. Floor division returns the integer number of hours, dropping the fraction part:

```
>>> minutes = 105

>>> hours = minutes // 60

>>> hours
```

1

To get the remainder, you could subtract off one hour in minutes:

```
>>> remainder = minutes - hours * 60

>>> remainder

45
```

An alternative is to use the modulus operator, %, which divides two numbers and returns the remainder:

```
 >>> remainder = minutes % 60

>>> remainder

45
```

The modulus operator is more useful than it seems. For example, you can check whether one number is divisible by another—if x % y is zero, then x is divisible by y.

Also, you can extract the right-most digit or digits from a number. For example, x % 10 yields the right-most digit of x (in base 10). Similarly x % 100 yields the last two digits.

If you are using Python 2, division works differently. The division operator, /, per- forms floor division if both operands are integers, and floating-point division if either operand is a float.

Boolean Expressions

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

```
>>> 5 == 5
True
>>> 5 == 6
False
```

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

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

The == operator is one of the relational operators; the others are:

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

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

Logical Operators

There are three logical operators: and, or, and not. The semantics (meaning) of these operators is similar to their meaning in English. For example, x > 0 and x < 10 is true only if x is greater than 0 and less than 10.

n%2 == 0 or n%3 == 0 is true if either or both of the conditions is true, that is, if the number is divisible by 2 or 3.

Finally, the not operator negates a boolean expression, so not (x > y) is true if x > y is false, that is, if x is less than or equal to y.

Strictly speaking, the operands of the logical operators should be boolean expressions, but Python is not very strict. Any nonzero number is interpreted as True:

```
>>> 42 and True

True
```

This flexibility can be useful, but there are some subtleties to it that might be confusing. You might want to avoid it (unless you know what you are doing).

Conditional Execution

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

```
if x > 0:

print('x is positive')
```

The boolean expression after if is called the condition. If it is true, the indented statement runs. If not, nothing happens.

if statements have the same structure as function definitions: a header followed by an indented body. Statements like this are called compound statements.

There is no limit on the number of statements that can appear in the body, but there has to be at least one. Occasionally, it is useful to have a body with no statements (usually as a place keeper for code you haven’t written yet). In that case, you can use the pass statement, which does nothing.

```
if x < 0:

pass     # TODO: need to handle negative values!
```

Alternative Execution

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

```
if x % 2 == 0:

print('x is even')

else:

print('x is odd')
```

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

Chained Conditionals

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

```
if x < y:

print('x is less than y')

elif x > y:

print('x is greater than y')

else:

print('x and y are equal')
```

elif is an abbreviation of “else if ”. Again, exactly one branch will run. There is no limit on the number of elif statements. If there is an else clause, it has to be at the end, but there doesn’t have to be one.

```
if choice == 'a':

draw_a()

elif choice == 'b':
```

```
draw_b()

elif choice == 'c':

draw_c()
```

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

Nested Conditionals

One conditional can also be nested within another. We could have written the exam- ple in the previous section like this:

```
if x == y:

print('x and y are equal')

else:

if x < y:

print('x is less than y')

else:

print('x is greater than y')
```

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

Although the indentation of the statements makes the structure apparent, nested conditionals become difficult to read very quickly. It is a good idea to avoid them when you can.

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

```
if 0 < x:

if x < 10:

print('x is a positive single-digit number.')
```

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

```
if 0 < x and x < 10:

print('x is a positive single-digit number.')
```

For this kind of condition, Python provides a more concise option:

```
if 0 < x < 10:

print('x is a positive single-digit number.')
```

Recursion

It is legal for one function to call another; it is also legal for a function to call itself. It may not be obvious why that is a good thing, but it turns out to be one of the most magical things a program can do. For example, look at the following function:

```
def countdown(n):
if n <= 0:
print('Blastoff!')
else:
print(n)
countdown(n-1)
```

If n is 0 or negative, it outputs the word, “Blastoff!” Otherwise, it outputs n and then calls a function named countdown—itself—passing n-1 as an argument.

What happens if we call this function like this?

```
>>> countdown(3)
```

The execution of countdown begins with n=3, and since n is greater than 0, it outputs the value 3, and then calls itself...

The execution of countdown begins with n=2, and since n is greater than 0, it outputs the value 2, and then calls itself...

The execution of countdown begins with n=1, and since n is greater than 0, it outputs the value 1, and then calls itself...

The execution of countdown begins with n=0, and since n is not greater than 0, it outputs the word, “Blastoff!” and then returns.

The countdown that got n=1 returns.

The countdown that got n=2 returns.

The countdown that got n=3 returns.

And then you’re back in main . So, the total output looks like this:

```
3

2

1

Blastoff!
```

A function that calls itself is recursive; the process of executing it is called recursion. As another example, we can write a function that prints a string n times:

```
def print_n(s, n):

if n <= 0:

return

print(s)

print_n(s, n-1)
```

If n <= 0 the return statement exits the function. The flow of execution immediately returns to the caller, and the remaining lines of the function don’t run.

The rest of the function is similar to countdown: it displays s and then calls itself to display s n-1 additional times. So the number of lines of output is 1 + (n - 1), which adds up to n.

Stack Diagrams for Recursive Functions

Previously, the stack diagram to represent the state of a program during a function call is discussed. The same kind of diagram can help interpret a recursive function.

Every time a function gets called, Python creates a frame to contain the function's local variables and parameters. For a recursive function, there might be more than one frame on the stack at the same time.

Figure shows a stack diagram for countdown called with n = 3.

As usual, the top of the stack is the frame for main . It is empty because we did not create any variables in main or pass any arguments to it.

The four countdown frames have different values for the parameter n. The bottom of the stack, where n=0, is called the base case. It does not make a recursive call, so there are no more frames.

As an exercise, draw a stack diagram for print_n called with s = 'Hello' and n=2. Then write a function called do_n that takes a function object and a number, n, as arguments, and that calls the given function n times.

Infinite Recursion

If a recursion never reaches a base case, it goes on making recursive calls forever, and the program never terminates. This is known as infinite recursion, and it is generally not a good idea. Here is a minimal program with an infinite recursion:

```
def recurse():

recurse()
```

In most programming environments, a program with infinite recursion does not really run forever. Python reports an error message when the maximum recursion depth is reached:

```
File "<stdin>", line 2, in recurse

File "<stdin>", line 2, in recurse

File "<stdin>", line 2, in recurse

.

.

File "<stdin>", line 2, in recurse

RuntimeError: Maximum recursion depth exceeded
```

This traceback is a little bigger than the one we saw in the previous chapter. When the error occurs, there are 1,000 recurse frames on the stack!

If you write an infinite recursion by accident, review your function to confirm that there is a base case that does not make a recursive call. And if there is a base case, check whether you are guaranteed to reach it.

Keyboard Input

The programs we have written so far accept no input from the user. They just do the same thing every time.

Python provides a built-in function called input that stops the program and waits for the user to type something. When the user presses Return or Enter, the program resumes and input returns what the user typed as a string. In Python 2, the same function is called raw_input.

```
>>> text = input()

What are you waiting for?
```

```
>>> text

What are you waiting for?
```

Before getting input from the user, it is a good idea to print a prompt telling the user what to type. input can take a prompt as an argument:

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

What...is your name?

Arthur, King of the Britons!

>>> name

Arthur, King of the Britons!
```

The sequence \n at the end of the prompt represents a newline, which is a special character that causes a line break. That’s why the user’s input appears below the prompt.

If you expect the user to type an integer, you can try to convert the return value to int:

```
>>> prompt = 'What...is the airspeed velocity of an unladen swallow?\n'

>>> speed = input(prompt)

What...is the airspeed velocity of an unladen swallow? 42

>>> int(speed)

42
```

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

```
>>> speed = input(prompt)

What...is the airspeed velocity of an unladen swallow?

What do you mean, an African or a European swallow?

>>> int(speed)

ValueError: invalid literal for int() with base 10
```

Fruitful Functions

Many of the Python functions we have used, such as the math functions, produce return values. But the functions we’ve written are all void: they have an effect, like printing a value or moving a turtle, but they don’t have a return value. In this chapter you will learn to write fruitful functions.

Return Values

Calling the function generates a return value, which we usually assign to a variable or use as part of an expression.

```
e = math.exp(1.0)

height = radius * math.sin(radians)
```

The functions we have written so far are void. Speaking casually, they have no return value; more precisely, their return value is None.

In this chapter, we are (finally) going to write fruitful functions. The first example is

area, which returns the area of a circle with the given radius:

```
def area(radius):

a = math.pi * radius**2

return a
```

We have seen the return statement before, but in a fruitful function the return statement includes an expression. This statement means: “Return immediately from this function and use the following expression as a return value.” The expression can be arbitrarily complicated, so we could have written this function more concisely:

```
def area(radius):

return math.pi * radius**2
```

On the other hand, temporary variables like a can make debugging easier. Sometimes it is useful to have multiple return statements, one in each branch of a conditional:

```
def absolute_value(x):

if x < 0:

return -x

else:

return x
```

Since these return statements are in an alternative conditional, only one runs.

As soon as a return statement runs, the function terminates without executing any subsequent statements. Code that appears after a return statement, or any other place the flow of execution can never reach, is called dead code.

In a fruitful function, it is a good idea to ensure that every possible path through the program hits a return statement. For example:

```
def absolute_value(x):

if x < 0:

return -x

if x > 0:

return x
```

This function is incorrect because if x happens to be 0, neither condition is true, and the function ends without hitting a return statement. If the flow of execution gets to the end of a function, the return value is None, which is not the absolute value of 0:

```
>>> absolute_value(0)

None
```

By the way, Python provides a built-in function called abs that computes absolute values.

As an exercise, write a compare function takes two values, x and y, and returns 1 if x
> y, 0 if x == y, and -1 if x < y.

Incremental Development

As you write larger functions, you might find yourself spending more time debugging.

To deal with increasingly complex programs, you might want to try a process called incremental development. The goal of incremental development is to avoid long debugging sessions by adding and testing only a small amount of code at a time.

As an example, suppose you want to find the distance between two points, given by the coordinates (x1, y1) and (x2, y2) . By the Pythagorean theorem, the distance is:

distance =

The first step is to consider what a distance function should look like in Python. In other words, what are the inputs (parameters) and what is the output (return value)?

In this case, the inputs are two points, which you can represent using four numbers. The return value is the distance represented by a floating-point value.

Immediately you can write an outline of the function:

```
def distance(x1, y1, x2, y2):

return 0.0
```

Obviously, this version doesn’t compute distances; it always returns zero. But it is syntactically correct, and it runs, which means that you can test it before you make it more complicated.

To test the new function, call it with sample arguments:

```
>>> distance(1, 2, 4, 6)

0.0
```

I chose these values so that the horizontal distance is 3 and the vertical distance is 4; that way, the result is 5, the hypotenuse of a 3-4-5 triangle. When testing a function, it is useful to know the right answer.

At this point we have confirmed that the function is syntactically correct, and we can start adding code to the body. A reasonable next step is to find the differences x2 − x1 and y2 − y1. The next version stores those values in temporary variables and prints them:

```
def distance(x1, y1, x2, y2):
dx = x2 - x1
dy = y2 - y1
print('dx is', dx)
print('dy is', dy)
return 0.0
```

If the function is working, it should display dx is 3 and dy is 4. If so, we know that the function is getting the right arguments and performing the first computation correctly. If not, there are only a few lines to check.

Next we compute the sum of squares of dx and dy:

```
def distance(x1, y1, x2, y2):
dx = x2 - x1
dy = y2 - y1
dsquared = dx**2 + dy**2
print('dsquared is: ', dsquared)
return 0.0
```

Again, you would run the program at this stage and check the output (which should be 25). Finally, you can use math.sqrt to compute and return the result:

```
def distance(x1, y1, x2, y2):

dx = x2 - x1

dy = y2 - y1

dsquared = dx**2 + dy**2

result = math.sqrt(dsquared)

return result
```

If that works correctly, you are done. Otherwise, you might want to print the value of

result before the return statement.

The final version of the function doesn’t display anything when it runs; it only returns a value. The print statements we wrote are useful for debugging, but once you get the function working, you should remove them. Code like that is called scaffolding because it is helpful for building the program but is not part of the final product.

When you start out, you should add only a line or two of code at a time. As you gain more experience, you might find yourself writing and debugging bigger chunks. Either way, incremental development can save you a lot of debugging time.

The key aspects of the process are:

1. Start with a working program and make small incremental changes. At any point, if there is an error, you should have a good idea where it is.

2. Use variables to hold intermediate values so you can display and check them.

3. Once the program is working, you might want to remove some of the scaffolding or consolidate multiple statements into compound expressions, but only if it does not make the program difficult to read.

As an exercise, use incremental development to write a function called hypotenuse that returns the length of the hypotenuse of a right triangle given the lengths of the other two legs as arguments. Record each stage of the development process as you go.

Composition

As you should expect by now, you can call one function from within another. As an example, we’ll write a function that takes two points, the center of the circle and a point on the perimeter, and computes the area of the circle.

Assume that the center point is stored in the variables xc and yc, and the perimeter point is in xp and yp. The first step is to find the radius of the circle, which is the distance between the two points. We just wrote a function, distance, that does that:

```
radius = distance(xc, yc, xp, yp)
```

The next step is to find the area of a circle with that radius; we just wrote that, too:

```
result = area(radius)
```

Encapsulating these steps in a function, we get:

```
def circle_area(xc, yc, xp, yp):

radius = distance(xc, yc, xp, yp)

result = area(radius)

return result
```

The temporary variables radius and result are useful for development and debugging, but once the program is working, we can make it more concise by composing the function calls:

```
def circle_area(xc, yc, xp, yp):

return area(distance(xc, yc, xp, yp))
```

Boolean Functions

Functions can return booleans, which is often convenient for hiding complicated tests inside functions. For example:

```
def is_divisible(x, y):

if x % y == 0:

return True

else:

return False
```

It is common to give boolean functions names that sound like yes/no questions;

is_divisible returns either True or False to indicate whether x is divisible by y. Here is an example:

```
>>> is_divisible(6, 4)

False

>>> is_divisible(6, 3)

True
```

The result of the == operator is a boolean, so we can write the function more concisely by returning it directly:

```
def is_divisible(x, y):
 return x % y == 0
```

Boolean functions are often used in conditional statements:

```
if is_divisible(x, y):
print('x is divisible by y')
```

It might be tempting to write something like:

```
if is_divisible(x, y) == True:
print('x is divisible by y')
```

But the extra comparison is unnecessary.

As an exercise, write a function is_between(x, y, z) that returns True if x ≤ y ≤ z

or False otherwise.

More Recursion

A recursive definition is similar to a circular definition, in the sense that the definition contains a reference to the thing being defined. A truly circular definition is not very useful:

vorpal:

An adjective used to describe something that is vorpal.

If you saw that definition in the dictionary, you might be annoyed. On the other hand, if you looked up the definition of the factorial function, denoted with the symbol !, you might get something like this:

$0! = 1$

$n! = n\ n - 1\ !$

This definition says that the factorial of 0 is 1, and the factorial of any other value, n, is n multiplied by the factorial of n-1.

So 3! is 3 times 2!, which is 2 times 1!, which is 1 times 0!. Putting it all together, 3! equals 3 times 2 times 1 times 1, which is 6.

If you can write a recursive definition of something, you can write a Python program to evaluate it. The first step is to decide what the parameters should be. In this case it should be clear that factorial takes an integer:

```
def factorial(n):
```

If the argument happens to be 0, all we have to do is return 1:

```
def factorial(n):

if n == 0:
```

```
return 1
```

Otherwise, and this is the interesting part, we have to make a recursive call to find the factorial of n-1 and then multiply it by n:

```
def factorial(n):

if n == 0:

return 1

else:

recurse = factorial(n-1)

result = n * recurse

return result
```

The flow of execution for this program is similar to the flow of countdown, If we call factorial with the value 3:

Since 3 is not 0, we take the second branch and calculate the factorial of n-1...

Since 2 is not 0, we take the second branch and calculate the factorial of n-1...

Since 1 is not 0, we take the second branch and calculate the factorial of n-1...

Since 0 equals 0, we take the first branch and return 1 without making any more recursive calls.

The return value, 1, is multiplied by n, which is 1, and the result is returned.

The return value, 1, is multiplied by n, which is 2, and the result is returned.

The return value (2) is multiplied by n, which is 3, and the result, 6, becomes the return value of the function call that started the whole process.

Figure shows what the stack diagram looks like for this sequence of function calls.

The return values are shown being passed back up the stack. In each frame, the return value is the value of result, which is the product of n and recurse. In the last frame, the local variables recurse and result do not exist, because the branch that creates them does not run.

Leap of Faith

Following the flow of execution is one way to read programs, but it can quickly become overwhelming. An alternative is what I call the “leap of faith”. When you come to a function call, instead of following the flow of execution, you assume that the function works correctly and returns the right result.

In fact, you are already practicing this leap of faith when you use built-in functions. When you call math.cos or math.exp, you don’t examine the bodies of those functions. You just assume that they work because the people who wrote the built-in functions were good programmers.

The same is true when you call one of your own functions. The same is true of recursive programs. When you get to the recursive call, instead of following the flow of execution, you should assume that the recursive call works (returns the correct result) and then ask yourself, “Assuming that I can find the factorial of n-1, can I compute the factorial of n?” It is clear that you can, by multiplying by n.

Of course, it’s a bit strange to assume that the function works correctly when you haven’t finished writing it, but that’s why it’s called a leap of faith!

Checking Types

What happens if we call factorial and give it 1.5 as an argument?

```
>>> factorial(1.5)

RuntimeError: Maximum recursion depth exceeded
```

It looks like an infinite recursion. How can that be? The function has a base case— when n == 0. But if n is not an integer, we can miss the base case and recurse forever.

In the first recursive call, the value of n is 0.5. In the next, it is -0.5. From there, it gets smaller (more negative), but it will never be 0.

We have two choices. We can try to generalize the factorial function to work with floating-point numbers, or we can make factorial check the type of its argument. The first option is called the gamma function. We can use the built-in function isinstance to verify the type of the argument. While we’re at it, we can also make sure the argument is positive:

```
def factorial (n):

if not isinstance(n, int):

print('Factorial is only defined for integers.')

return None

elif n < 0:

print('Factorial is not defined for negative integers.')

return None

elif n == 0:

return 1

else:

return n * factorial(n-1)
```

The first base case handles nonintegers; the second handles negative integers. In both cases, the program prints an error message and returns None to indicate that something went wrong:

```
>>> factorial('fred')
```

```
Factorial is only defined for integers. None

>>> factorial(-2)

Factorial is not defined for negative integers.

None
```

If we get past both checks, we know that n is positive or zero, so we can prove that the recursion terminates.

This program demonstrates a pattern sometimes called a guardian. The first two conditionals act as guardians, protecting the code that follows from values that might cause an error. The guardians make it possible to prove the correctness of the code.

Regulation: AK20	**Subject Code:CSE/CIC** 20APS0526/20APC3605	**Subject Name :** Basics of Python Programming	**AY:** 2023-2024
UNIT-3 (Iteration, Strings, Case Study, Lists)			

UNIT – III

Iteration: Reassignment, Updating variables, The while statement, Break, Square roots, Algorithms.

Strings: A string is a sequence, len, Traversal with a for loop, String slices, Strings are immutable, Searching, Looping and Counting, String methods, The in operator, String comparison.

Case Study: Reading word lists, Search, Looping with indices.

Lists: List is a sequence, Lists are mutable, Traversing a list, List operations, List slices, List methods, Map filter and reduce, Deleting elements, Lists and Strings, Objects and values, Aliasing, List arguments.

Unit Outcomes:

Student should be able to

- Use the data structure list.
- Design programs for manipulating strings.

Iteration

The ability to run a block of statements repeatedly.. Repeating identical or similar tasks without making errors. In a computer program, repetition is also called **iteration.**

Reassignment

It is legal to make more than one assignment to the same variable. A new assignment makes an existing variable refer to a new value (and stop referring to the old value).

```
>>> x = 5
>>>x
5
>>> x = 7
>>>x
7
```

Updating Variables

A common kind of reassignment is an **update**, where the new value of the variable depends on the old.

```
>>> x = x + 1
```

If you try to update a variable that doesn't exist, you get an error, because Python evaluates the right side before it assigns a value to x:

```
>>> x = x + 1
NameError: name 'x' is not defined
```

Before you can update a variable, you have to initialize it, usually with a simple assignment:

```
>>> x = 0
>>> x = x + 1
```

Updating a variable by adding 1 is called an **increment**; subtracting 1 is called as **decrement**.

Introduction (Objects, Values, and Types)

All the data in a Python code is represented by objects or by relations between objects. Every object has an identity, a type and a value.

Identity (Memory Location)

An object's identity **never changes once it has been created**; you may think of it as the **object's address in memory**. The **is operator** compares the identity of two objects; the **id() function** returns an integer representing its identity.

Type (Data Type)

An object's type defines the **possible values and operations** (e.g. "does it have a length?") that type supports. The **type() function** returns the type of an object. An object type is **unchangeable** like the identity.

Value (stand alone and with references (copy))

The value of some objects can change. Objects **whose value can change are** said to be **mutable**; objects **whose value is unchangeable** once they are created are called **immutable (with the concept of identity)**.

The mutability of an object is determined by its **type**. Some of these objects like **lists and dictionaries** are **mutable**, meaning you can change their content without changing their identity. Other objects like **integers, floats, strings** and **tuples** are objects that cannot be changed.

Mutable

Mutable objects can change their value but keep their id().

Note:

1. If any two variables are having same values they share the same memory location
2. If a variable is assigned to another variable both variables share the same value and same address location
3. If you change the original value, then the original value and the address is changed but the copied value will not have this effect and it retains the old value and its memory location
4. For integer and string the above three points are valid and for float everytime the memory location changes.

Immutable

An object with a fixed value. Immutable objects include numbers, strings and tuples. Such an object cannot be altered. A new object has to be created if a different value has to be stored.

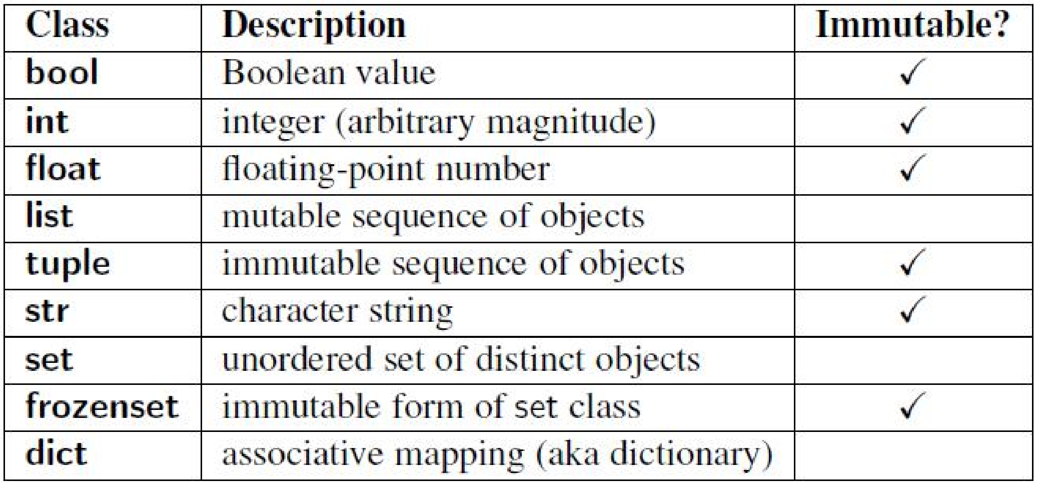
Important note

Some objects contain **references** to other objects, these objects are called **containers**. Some examples of containers are a **tuple**, **list**, and **dictionary**. The **value of an immutable container** that contains a **reference to a mutable object can be changed** if that mutable object is changed. However, the container is still considered immutable because when we talk about the **mutability of a container** only the **identities** of the contained objects are implied.

Having **immutable variables** means that no matter how many times the method is called with the same variable/value, the output will always be the same. Having **mutable variables** means that calling the same method with the same variables may not guarantee the same output, because the variable can be mutated at any time by another method or perhaps, another thread, and that is where you start to go crazy debugging.(Float has to be explained and clarified)

If you want to write most efficient code, you should be the knowing difference between **mutable and immutable** in python. Concatenating string in loops wastes lots of memory, because strings are immutable, concatenating two strings together actually creates a third string which is the combination of the previous two. If you are iterating a lot and building a large string, you will waste a lot of memory creating and throwing away objects. Use list compression joins technique.

Python handles mutable and immutable objects differently**. Immutable** are quicker to access than mutable objects. Also, immutable objects are fundamentally expensive to "change", because doing so involves creating a copy. Changing **mutable** objects is cheap.



Class	Description	Immutable?
bool	Boolean value	✓
int	integer (arbitrary magnitude)	✓
float	floating-point number	✓
list	mutable sequence of objects	
tuple	immutable sequence of objects	✓
str	character string	✓
set	unordered set of distinct objects	
frozenset	immutable form of set class	✓
dict	associative mapping (aka dictionary)	

Storage Model

The first way we can categorize the types is by how many objects can be stored in an object of this type. Python’s types, as well as types from most other languages, can hold either single or multiple values. A type which holds a single literal object we will call atomic or scalar storage, and those which can hold multiple objects we will refer to as container storage. (Container objects are also referred to as composite or compound objects in the documentation, but some of these refer to objects other than types, such as class instances.) Container types bring up the additional issue of whether different types of objects can be stored. All of Python’s container types can hold objects of different types. Table 4.6 categorizes Python’s types by storage model. Although strings may seem like a container type since they “contain” characters (and usually more than one character), they are not considered as such because Python does not have a character type. Thus strings are self-contained literals.

Types Categorized by the Storage Model	
Storage Model Category	**Python Types That Fit Category**
Scalar/atom	Numbers (all numeric types), strings (all are literals)
Container	Lists, tuples, dictionaries

Update Model

Another way of categorizing the standard types is by asking the question, “Once created, can objects be changed, or can their values be updated?” When we introduced Python types early on, we indicated that certain types allow their values to be updated and others do not. Mutable objects are those whose values can be changed, and immutable objects are those whose values cannot be changed.

Types Categorized by the Update Model	
Update Model Category	**Python Types That Fit Category**
Mutable	Lists, dictionaries
Immutable	Numbers, strings, tuples

Now after looking at the table, a thought that must immediately come to mind is, “Wait a minute! What do you mean that numbers and strings are immutable? I’ve done things like the following”:

```
>>> x = 'Python numbers and strings'
>>>id(x)
2939506131952
>>> x = 'are immutable?!? What gives?'
>>>id(x)
2939506760640
>>> i=0
>>>id(i)
140711953831552
>>> i=i+1
>>>i
1
>>>id(i)
140711953831584
```

“They sure as heck don’t look immutable to me!” That is true to some degree, but looks can be deceiving. What is really happening behind the scenes is that the original objects are actually

being replaced in the above examples. Yes, that is right. Read that again.
Rather than referring to the original objects, new objects with the new values were allocated and (re)assigned to the original variable names, and the old objects were garbage-collected. One can confirm this by using the id() BIF to compare object identities before and after such assignments.

If we added calls to id() in our example above, we may be able to see that the objects are being changed, as below:

```
>>> x = 'Python numbers and strings'
>>>print id(x)
16191392
>>> x = 'are immutable?!? What gives?'
>>>print id(x)
16191232
>>> i = 0
>>>print id(i)
7749552
>>> i = i + 1
>>>print id(i)
7749600
```

Your mileage will vary with regard to the object IDs as they will differ between executions. On the flip side, lists can be modified without replacing the original object, as illustrated in the code below:

```
>>>aList = ['ammonia', 83, 85, 'lady']
>>>aList
['ammonia', 83, 85, 'lady']
>>>
>>>aList[2]
85
>>>
>>>id(aList)
135443480
>>>
>>>aList[2] = aList[2] + 1
>>>aList[3] = 'stereo'
>>>aList
['ammonia', 83, 86, 'stereo']
>>>
>>>id(aList)
135443480
```

```
>>>
>>>aList.append('gaudy')
>>>aList.append(aList[2] + 1)
>>>aList
['ammonia', 83, 86, 'stereo', 'gaudy', 87]
>>>
>>>id(aList)
135443480
```

Notice how for each change, the ID for the list remained the same.

Access Model

Although the previous two models of categorizing the types are useful when being introduced to Python, they are not the primary models for differentiating the types. For that purpose, we use the access model. By this, we mean, how do we access the values of our stored data? There are three categories under the access model: direct, sequence, and mapping. The different access models and which types fall into each respective category are given in Table

Direct types indicate single-element, non-container types. All numeric types fit into this category.

Sequence types are those whose elements are sequentially accessible via index values starting at 0. Accessed items can be either single elements or in groups, better known as slices. Types that fall into this category include strings, lists, and tuples. As we mentioned before, Python does not support a character type, so, although strings are literals, they are a sequence type because of the ability to access substrings sequentially.

Mapping types are similar to the indexing properties of sequences, except instead of indexing on a sequential numeric offset, elements (values) are unordered and accessed with a key, thus making mapping types a set of hashed key-value pairs.

Types Categorized by the Access Model

Access Model Category	Types That Fit Category
Direct	Numbers
Sequence	Strings, lists, tuples
Mapping	Dictionaries

Cross-reference chart (see Table) that shows all the standard types, the three different models used for categorization, and where each type fits into these models.

Categorizing the Standard Types

Data Type	Storage Model	Update Model	Access Model
Numbers	Scalar	Immutable	Direct
Strings	Scalar	Immutable	Sequence

Lists	Container	Mutable	Sequence
Tuples	Container	Immutable	Sequence
Dictionaries	Container	Mutable	Mapping

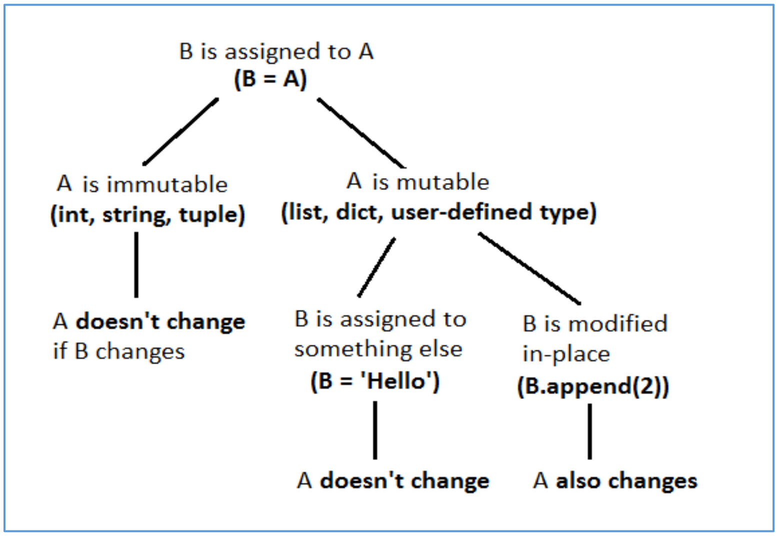
Mutable vs Immutable Objects in Python

Every variable in python holds an instance of an object. There are two types of objects in python i.e. **Mutable** and **Immutable objects**. Whenever an object is instantiated, it is assigned a unique object id. The type of the object is defined at the runtime and it can't be changed afterwards. However, it's state can be changed if it is a mutable object.
To summarise the difference, mutable objects can change their state or contents and immutable objects can't change their state or content.
Immutable Objects : These are of in-built types like **int, float, bool, string, unicode, tuple**. In simple words, an immutable object can't be changed after it is created.
Mutable Objects : These are of type list**,** dict**,** set . Custom classes are generally mutable.

1. Mutable and immutable objects are handled differently in python. Immutable objects are quicker to access and are expensive to change because it involves the creation of a copy.Whereas mutable objects are easy to change.
2. Use of mutable objects is recommended when there is a need to change the size or content of the object.
3. Exception : However, there is an exception in immutability as well. We know that tuple in python is immutable. But the tuple consists of a sequence of names with unchangeable bindings to objects.

The tuple itself isn't mutable but contain items that are mutable.

As a rule of thumb, Generally Primitive-like types are probably immutable and Customized Container-like types are mostly mutable.



The while Statement

```
def countdown(n):
while n > 0:
print(n)
n = n - 1
print('Blastoff!')
```

More formally, here is the flow of execution for a while statement:

1. Determine whether the condition is true or false.
2. If false, exit the while statement and continue execution at the next statement.
3. If the condition is true, run the body and then go back to step 1.

This type of flow is called a **loop** because the third step loops back around to the top. The body of the loop should change the value of one or more variables so that the condition becomes false eventually and the loop terminates. Otherwise the loop will repeat forever, which is called an **infinite loop**.

```
def sequence(n):
while n != 1:
print(n)
if n % 2 == 0:       # n is even
        n = n / 2
else:                # n is odd
        n = n*3 + 1
```

OUTPUT

```
>>>seq(3)
3
10
5.0
16.0
8.0
4.0
  2.0
```

The beauty of a ‘n’ as a powers of two

```
>>>seq(4)
4
2.0
>>>seq(16)
16
8.0
4.0
2.0
```

```
>>>seq(32)
32
16.0
8.0
4.0
2.0
>>>
```

The condition for this loop is n != 1, so the loop will continue until n is 1, which makes the condition false.
Each time through the loop, the program outputs the value of n and then checks whether it is even or odd. If it is even, n is divided by 2. If it is odd, the value of n is replaced with n*3 + 1.
For example, if the argument passed to sequence is 3, the resulting values of n are 3, 10, 5, 16, 8, 4, 2, 1.
Since n sometimes increases and sometimes decreases, there is no obvious proof that n will ever reach 1, or that the program terminates. For some particular values of n, we can prove termination. For example, if the starting value is a power of two, n will be even every time through the loop until it reaches 1. The previous example ends with such a sequence, starting with 16.
The hard question is whether we can prove that this program terminates for all positive values of n. So far, no one has been able to prove it or disprove it!

Break

Sometimes you don’t know it’s time to end a loop until you get halfway through the body. In that case you can use the break statement to jump out of the loop.
For example, suppose you want to take input from the user until they type done. You could write:

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

The loop condition is True, which is always true, so the loop runs until it hits the break statement.
Each time through, it prompts the user with an angle bracket. If the user types done, the break statement exits the loop. Otherwise the program echoes whatever the user types and goes back to the top of the loop. Here’s a sample run:

```
>not done
not done
>done
```

Done!

Square Roots

For example, one way of computing square roots is Newton's method. Suppose that you want to know the square root of a. If you start with almost any estimate, x, you can compute a better estimate with the following formula:
Y=(x+a/x)/2
For example, if a is 4 and x is 3:

```
>>> a = 4
>>> x = 3
>>> y = (x + a/x) / 2
>>>y
2.16666666667
```

The result is closer to the correct answer (sqrt(4) = 2). If we repeat the process with the new estimate, it gets even closer:

```
>>> x = y
>>> y = (x + a/x) / 2
>>>y
2.00641025641
```

After a few more updates, the estimate is almost exact:

```
>>> x = y
>>> y = (x + a/x) / 2
>>>y
2.00001024003
>>> x = y
>>> y = (x + a/x) / 2
>>>y
2.00000000003
```

In general we don't know ahead of time how many steps it takes to get to the right answer, but we know when we get there because the estimate stops changing:

```
>>> x = y
>>> y = (x + a/x) / 2
>>>y
2.0
>>> x = y
>>> y = (x + a/x) / 2
>>>y
2.0
```

When y == x, we can stop. Here is a loop that starts with an initial estimate, x, and improves it until it stops changing:

```
while True:
print(x)
    y = (x + a/x) / 2
if y == x:
break
    x = y
```

For most values of ‘a’ this works fine, but in general it is dangerous to test float equality. Floating-point values are only approximately right: most rational numbers, like 1/3, and irrational numbers, like sqrt (2), can’t be represented exactly with a float.
Rather than checking whether x and y are exactly equal, it is safer to use the built-in function abs to compute the absolute value, or magnitude, of the difference between them:

```
if abs(y-x) < epsilon:
break
```

Where epsilon has a value, like 0.0000001, that determines how close is close enough.

Algorithms

Newton’s method is an example of an **algorithm**: it is a mechanical process for solving a category of problems (in this case, computing square roots).
To understand what an algorithm is, it might help to start with something that is not an algorithm. When you learned to multiply single-digit numbers, you probably memorized the multiplication table. In effect, you memorized 100 specific solutions. That kind of knowledge is not algorithmic.
But if you were “lazy”, you might have learned a few tricks. For example, to find the product of n and 9, you can write n-1 as the first digit and 10-n as the second digit. This trick is a general solution for multiplying any single-digit number by 9. That’s an algorithm!
Similarly, the techniques you learned for addition with carrying, subtraction with borrowing, and long division are all algorithms. One of the characteristics of algorithms is that they do not require any intelligence to carry out. They are mechanical processes where each step follows from the last according to a simple set of rules.
Executing algorithms is boring, but designing them is interesting, intellectually challenging, and a central part of computer science.

Strings

Strings are not like integers, floats, and booleans. A string is a sequence, which means it is an ordered collection of other values.

A String Is a Sequence

A string is a sequence of characters. You can access the characters one at a time with the bracket operator:

```
>>>fruit = 'banana'
>>>letter = fruit[1]
```

The second statement selects character number 1 from fruit and assigns it to letter.
The expression in brackets is called an **index**. The index indicates which character in the sequence you want (hence the name).
But you might not get what you expect:

```
>>>letter
'a'
```

For most people, the first letter of 'banana' is b, not a. But for computer scientists, the index is an offset from the beginning of the string, and the offset of the first letter is zero.

```
>>>letter = fruit[0]
>>>letter
'b'
```

So b is the 0th letter (“zero-eth”) of 'banana', a is the 1th letter (“one-eth”), and n is the 2th letter (“two-eth”).
As an index, you can use an expression that contains variables and operators:

```
>>> i = 1
>>>fruit[i]
'a'
>>>fruit[i+1]
'n'
```

But the value of the index has to be an integer. Otherwise you get:

```
>>>letter = fruit[1.5]
TypeError: string indices must be integers
```

Len

len is a built-in function that returns the number of characters in a string:

```
>>>fruit = 'banana'
>>>len(fruit)
6
```

To get the last letter of a string, you might be tempted to try something like this:

```
>>>length = len(fruit)
>>>last = fruit[length]
IndexError: string index out of range
```

The reason for the IndexError is that there is no letter in 'banana' with the index 6. Since we started counting at zero, the six letters are numbered 0 to 5. To get the last character, you have to subtract 1 from length:

```
>>>last = fruit[length-1]
>>>last
'a'
```

Or you can use negative indices, which count backward from the end of the string. The expression fruit[-1] yields the last letter, fruit[-2] yields the second to last, and so on.

Traversal with a for Loop

A lot of computations involve processing a string one character at a time. Often they start at the beginning, select each character in turn, do something to it, and continue until the end. This pattern of processing is called a **traversal**. One way to write a traversal is with a while loop:

```
fruit="banana"
index = 0
while index <len(fruit):
letter = fruit[index]
print(letter)
index = index + 1
```

This loop traverses the string and displays each letter on a line by itself. The loop condition is index <len(fruit), so when index is equal to the length of the string, the condition is false, and the body of the loop doesn't run. The last character accessed is the one with the index len(fruit)-1, which is the last character in the string.

As an exercise, write a function that takes a string as an argument and displays the letters backward, one per line.

Another way to write a traversal is with a for loop:

```
for letter in fruit:
```

```
print(letter)
```

Each time through the loop, the next character in the string is assigned to the variable letter. The loop continues until no characters are left.

The following example shows how to use concatenation (string addition) and a for loop to generate an **abecedarian series** (that is, in alphabetical order). In Robert McCloskey's book Make Way for Ducklings, the names of the ducklings are Jack, Kack, Lack, Mack, Nack, Ouack, Pack, and Quack. This loop outputs these names in order:

```
prefixes = 'JKLMNOPQ'
suffix = 'ack'
for letter in prefixes:
print(letter + suffix)
```

The output is:

```
Jack
Kack
Lack
Mack
Nack
Oack
Pack
Qack
```

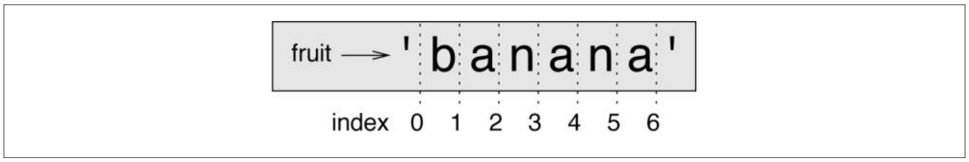
Of course, that's not quite right because "Ouack" and "Quack" are misspelled.

String Slices

A segment of a string is called a **slice**. Selecting a slice is similar to selecting a character:

```
>>> s = 'Monty Python'
>>>s[0:5]
'Monty'
>>>s[6:12]
'Python'
```

The operator [n:m] returns the part of the string from the "n-eth" character to the "m-eth" character, **including the first but excluding the last**. This behavior is counter- intuitive, but it might help to imagine the indices pointing between the characters, as in fig



If you omit the first index (before the colon), the slice starts at the beginning of the string. If you omit the second index, the slice goes to the end of the string:

```
>>>fruit = 'banana'
>>>fruit[:3] # index of [0] to [2]
'ban'
>>>fruit[3:] # index of [3] to [last]
'ana'
```

If the first index is greater than or equal to the second the result is an empty string, represented by two quotation marks:

```
>>>fruit = 'banana'
>>>fruit[3:3]
‘’
```

An empty string contains no characters and has length 0, but other than that, it is the same as any other string.
Continuing this example, what do you think fruit[:] means? Try it and see.

```
>>>fruit = 'banana'
>>>fruit[:]
'banana'
```

Negative Indexing

Use negative indexes to start the slice from the end of the string:
Get the characters from position 6 to position 2 (not included), starting the count from the end of the string:

```
b = "Hindustan"
print(b[-6:-2])
```

OUTPUT

```
Dust
```

```
>>> b="hindustan"
>>> print(b[-6:-2])
dust
>>> b="hindustan"
>>> b[-6:-2]
'dust'
>>> b[2:-6]
'n'
>>> b[2:-5]
```

```
'nd'
>>> b[5:-2]
'st'
```

Strings Are Immutable

It is tempting to use the [] operator on the left side of an assignment, with the intention of changing a character in a string. For example:

```
>>> greeting = 'Hello, world!'

>>>greeting[0] = 'J'

TypeError: 'str' object does not support item assignment
```

The “object” in this case is the string and the “item” is the character you tried to assign. For now, an object is the same thing as a value, but we will refine that definition later.

The reason for the error is that strings are immutable, which means you can’t change an existing string. The best you can do is create a new string that is a variation on the original:

```
>>> greeting = 'Hello, world!'

>>>new_greeting = 'J' + greeting[1:]

>>>new_greeting

'Jello, world!'
```

This example concatenates a new first letter onto a slice of greeting. It has no effect on the original string.

Searching

What does the following function do?

```
def find(word, letter):
   index = 0
   while index <len(word):
      if word[index] == letter:
         return index+1
      index = index + 1
   return -1
```

In a sense, find is the inverse of the [] operator. Instead of taking an index and extracting the

corresponding character, it takes a character and finds the index where that character appears. If the character is not found, the function returns -1.
This is the first example we have seen of a return statement inside a loop. If word[index] == letter, the function breaks out of the loop and returns immediately.
If the character doesn’t appear in the string, the program exits the loop normally and returns -1.
This pattern of computation—traversing a sequence and returning when we find what we are looking for—is called a **search**.
As an exercise, modify find so that it has a third parameter: the index in word where it should start looking.

Looping and Counting

The following program counts the number of times the letter a appears in a string:

```
word = 'banana'
count = 0
for letter in word:
    if letter == 'a':
        count = count + 1
print(count)
```

```
OUTPUT
3
```

This program demonstrates another pattern of computation called a **counter**. The variable count is initialized to 0 and then incremented each time a‘a’ is found. When the loop exits, count contains the result—the total number of a’s.
As an exercise, encapsulate this code in a function named count, and generalize it so that it accepts the string and the letter as arguments.
Then rewrite the function so that instead of traversing the string, it uses the three- parameter version of find from the previous section.

String Methods

Strings provide methods that perform a variety of useful operations. A method is similar to a function—it takes arguments and returns a value—but the syntax is different. For example, the method upper/lower/Capitalize each letter takes a string and returns a new string with all uppercase/lowercase/Capitalize each letter.
Instead of the function syntax upper(word), it uses the method syntax word.upper(), word.lower(),title()-Converts the first character of each word to upper case:

```
>>>word = 'banana'
>>>new_word = word.upper()
>>>new_word
'BANANA'
```

```
>>>new_word = word.lower()
>>>new_word
'banana'
>>>txt = "The rain in Spain stays mainly in the plain"
>>>x = txt.title()
>>>x
The Rain In Spain Stays Mainly In The Plain
```

This form of dot notation specifies the name of the method, upper/lower/title, and the name of the string to apply the method to, word. The empty parentheses indicate that this method takes no arguments.

A method call is called an **invocation**; in this case, we would say that we are invoking upper on word.

As it turns out, there is a string method named find that is remarkably similar to the function we wrote:

```
>>>word = 'banana'
>>>index = word.find('a')
>>>index
1
```

In this example, we invoke find on word and pass the letter we are looking for as a parameter.

Actually, the find method is more general than our function; it can find substrings, not just characters:

The find() method returns the lowest index of the substring if it is found in given string. If its is not found then it returns -1.

Syntax:

str.find(sub,start,end)

sub : It’s the substring which needs to be searched in the given string.

start : Starting position where sub is needs to be checked within the string.

end : Ending position where suffix is needs to be checked within the string.

NOTE : If start and end indexes are not provided then by default it takes 0 and length-1 as starting and ending indexes where ending indexes is not included in our search.

```
>>>word.find('na')
2
```

By default, find starts at the beginning of the string, but it can take a second argument, the index where it should start:

```
>>>word.find('na', 3)
```

```
4
```

This is an example of an **optional argument**. Find can also take a third argument, the index where it should stop:

```
>>>name = 'bob'
>>>name.find('b', 1, 2)
-1
```

This search fails because b does not appear in the index range from 1 to 2, not including 2. Searching up to, but not including, the second index makes find consistent with the slice operator.

The in Operator

The word “in” and “not in”are boolean operators that takes two strings and returns True if the first appears as a substring in the second:

```
>>> x='lion king'
>>> 'l' in x
True
>>> 'lio' in x
True
>>> 'zebra' in x
False
>>> 'zebra' not in x
True
```

For example, the following function prints all the letters from word1 that also appear in word2:

```
def in_both(word1, word2):
    for letter in word1:
        if letter in word2:
            print(letter)
```

With well-chosen variable names, Python sometimes reads like English. You could read this loop, “for (each) letter in (the first) word, if (the) letter (appears) in (the second) word, print (the) letter.”

Here’s what you get if you compare apples and oranges:

```
>>> h("harshitha","hasitha")
h
a
s
```

```
h
i
t
h
a
>>> h("hasitha","harshitha")
h
a
s
i
t
h
a
```

String Comparison

The relational operators work on strings. To see if two strings are equal:

```
if word == 'banana':
print('All right, bananas.')
```

Other relational operations are useful for putting words in alphabetical order:

```
if word < 'banana':
print('Your word, ' + word + ', comes before banana.')
elif word > 'banana':
print('Your word, ' + word + ', comes after banana.')
else:
print('All right, bananas.')
```

Python does not handle uppercase and lowercase letters the same way people do. All the uppercase letters come before all the lowercase letters, so:

```
Your word, Pineapple, comes before banana.
```

A common way to address this problem is to convert strings to a standard format, such as all lowercase, before performing the comparison. Keep that in mind in case you have to defend yourself against a man armed with a Pineapple.

Debugging

When you use indices to traverse the values in a sequence, it is tricky to get the beginning and end of the traversal right. Here is a function that is supposed to compare two words and return

True if one of the words is the reverse of the other, but it contains two errors:

#original version

```
def is_reverse(word1, word2):
    if len(word1) != len(word2):
        return False
    i = 0
    j = len(word2)
    while j > 0:
        if word1[i] != word2[j]:
            return False
        i = i+1
        j = j-1
```

```
>>> r("varun","kurav")
0 4
1 3
2 2
3 1
True
>>> r("madam","MaDaM")
0 4
False
```

#Corrected Version

```
def r(word1, word2):
    #word1=word1.lower() # This statement is used to convert the cases and to check for equality
    #word2=word2.lower() # This statement is used to convert the cases and to check for equality

    if len(word1) != len(word2):
        return False
    i = 0
    j = len(word2)-1# This statement should execute from length -1 as index starts with zero
    while j >= 0: #this statement is not checking the index [0] and it should be while j >= 0:
        print(i,j) #testing the values of the index
        if word1[i] != word2[j]:
            return False
        i = i+1
        j = j-1
    return True
```

```
>>> r("madam","MaDaM")
0 4
```

```
False
>>> r("varun","kurav")
0 4
1 3
2 2
3 1
4 0
False
```

The first if statement checks whether the words are the same length. If not, we can return False immediately. Otherwise, for the rest of the function, we can assume that the words are the same length.

i and j are indices: i traverses word1 forward while j traverses word2 backward. If we find two letters that don’t match, we can return False immediately. If we get through the whole loop and all the letters match, we return True.

If we test this function with the words “pots” and “stop”, we expect the return value
True, but we get an IndexError:

```
>>>is_reverse('pots', 'stop')
...
File "reverse.py", line 15, in is_reverse if word1[i] != word2[j]:
IndexError: string index out of range
```

The index of the last character is 3, so the initial value for j should be len(word2)-1.

Case Study: Word Play

This chapter involves solving word puzzles by searching for words that have certain properties. For example, we’ll find the longest palindromes in English and search for words whose letters appear in alphabetical order.

Reading Word Lists

The built-in function open takes the name of the file as a parameter and returns a **file object** you can use to read the file.

For the exercises in this chapter we need a list of English words. There are lots of word lists available on the Web, but the one most suitable for our purpose is one of the word lists collected and contributed to the public domain by Grady Ward as part of the Moby lexicon project (see http://wikipedia.org/wiki/Moby_Project). It is a list of 113,809 official crosswords; that is, words that are considered valid in crossword puzzles and other word games.

http://greenteapress.com/thinkpython2/code/words.txt

```
>>>fin = open('words.txt')
```

fin is a common name for a file object used for input. The file object provides several methods for reading, including readline, which reads characters from the file until it gets to a newline and returns the result as a string:

```
>>>fin.readline()
"ï»¿aa\n'
```

The **strip()** method returns a copy of the string in which all chars have been stripped from the beginning and the end of the string (default whitespace characters).

Syntax

str.strip("characters to be removed")

Parameters

chars − The characters to be removed from beginning or end of the string.

Return Value

This method returns a copy of the string in which all chars have been stripped from the beginning and the end of the string.

Example

```
str = "*****this is string example....wow!!!*****"
print (str.strip( '*' ))
```

Result

```
this is string example....wow!!!
```

The next word is "aah", which is a perfectly legitimate word, so stop looking at me like that. Or, if it's the whitespace that's bothering you, we can get rid of it with the string method strip:

```
>>>line=fin.readline()
>>>word=line.strip()
>>>word
'aah'
```

You can also use a file object as part of a for loop. This program reads words.txt and prints each word, one per line:

```
fin = open('words.txt')
for line in fin:
word = line.strip()
print(word)
```

```
# Download file "words.txt" here:  http://greenteapress.com/thinkpython2/code/words.txt
# Exercise 1   Write a program that reads words.txt and prints only the words with more than 20
characters
# (not counting whitespace).

def characterCount():
    fin = open("word.txt")
    line=fin.readline()
    count=0
    for line in fin:
        words = line.strip()
        if len(words) > 19:
            count=count+1
            print(words)
    print("Total words=",count)
characterCount()
```

```
OUTPUT
counterdemonstration
counterdemonstrations
counterdemonstrators
hyperaggressivenesses
hypersensitivenesses
microminiaturization
microminiaturizations
representativenesses
Total words= 8
```

```
# Exercise 2
# Write a function called has_no_e that returns True if the given word doesn’t have
# the letter “e” in it.

# Modify your program from the previous section to print only the words that have
# no “e” and compute the percentage of the words in the list that have no “e”.

defhas_no_e_one():
fin = open('words.txt')
for line in fin:
ifline.find("e") == -1:
return("True")
```

OUTPUT

```
True

def has_no_e():
    fin = open("words.txt")
    number_of_words = 0
    count = 0
    for line in fin:
        number_of_words = number_of_words + 1
        words = line.strip()
        if words.find("e") == -1:
            #print(words)
            count = count + 1
    percent_of_words = (count/number_of_words) * 100
    print("Total words without e:", count)
    print("Percent:", percent_of_words)
has_no_e()
```

OUTPUT

```
Total words without e: 37643
Percent: 33.07559156130007
```

# 9.2 Exercises

```
# Download file "words.txt" here:
http://greenteapress.com/thinkpython2/code/words.txt

# Exercise 1   Write a program that reads words.txt and prints only the words with
more than 20 characters

# (not counting whitespace).

defcharacterCount():

    fin =open("words.txt")

for line in fin:

        words =line.strip()

iflen(words) >19:
```

```
print(words)

# Exercise 2

# Write a function called has_no_e that returns True if the given word doesn’t have

# the letter “e” in it.

# Modify your program from the previous section to print only the words that have

# no “e” and compute the percentage of the words in the list that have no “e”.

defhas_no_e_one():

    fin =open('words.txt')

for line in fin:

ifline.find("e") ==-1:

return("True")

defhas_no_e():

    fin =open("words.txt")

number_of_words=0

    count =0

for line in fin:

number_of_words=number_of_words+1

        words =line.strip()

ifwords.find("e") ==-1:

print(words)

            count = count +1

percent_of_words= (count/number_of_words) *100

print("Percent:", percent_of_words)
```

```
# Exercise 3

# Write a function named avoids that takes a word and a string of forbidden letters,

# and that returns True if the word doesn’t use any of the forbidden letters.

defavoids(word, stringHere):

for letters in word:

ifstringHerein word:

returnFalse

returnTrue

# Modify your program to prompt the user to enter a string of forbidden letters

# and then print the number of words that don’t contain any of them. Can you

# find a combination of 5 forbidden letters that excludes the smallest number of
words?

defavoids_forbidden():

user_input=input("Enter a string of forbidden letters!")

   fin =open("words.txt")

   count =0

for line in fin:

      words =line.strip()

ifuser_inputnotin words:

         count = count +1

print(count)

# Exercise 4

# Write a function named uses_only that takes a word and a string of letters, and that
```

```
returns True if the
# word contains only letters in the list. Can you make a sentence using only the
letters acefhlo?
# Other than “Hoe alfalfa?”
defuses_only(word,string_of_letters):
for letter in word:
if letter notinstring_of_letters:
returnFalse
returnTrue
# Exercise 5
# Write a function named uses_all that takes a word and a string of required letters,
# and that returns True if the word uses all the required letters at least once.
# How many words are there that use all the vowels aeiou? How about aeiouy?
defuses_all(word, required_letters):
for letter inrequired_letters:
if letter notin word:
returnFalse
returnTrue
# Exercise 6
# Write a function called is_abecedarian that returns True if the letters in a word
appear in
# alphabetical order (double letters are ok).
# How many abecedarian words are there?
```

```
defis_abcedarian(word):
    i =0
while i <len(word) -1:
if word[i] > word[i +1]:
returnFalse
        i = i +1
returnTrue
# 9.7  Exercises
# Exercise 7
# This question is based on a Puzzler that was broadcast on the radio program Car Talk
# (http://www.cartalk.com/content/puzzlers):
# Give me a word with three consecutive double letters.
# I’ll give you a couple of words that almost qualify, but don’t.
# For example, the word committee, c-o-m-m-i-t-t-e-e. It would be great except
# for the ‘i’ that sneaks in there. Or Mississippi: M-i-s-s-i-s-s-i-p-p-i.
# If you could take out those i’s it would work. But there is a word that has
# three consecutive pairs of letters and to the best of my knowledge this may be
#  the only word. Of course there are probably 500 more but I can only think of one.
#  What is the word?
# Write a program to find it.
defis_double(word):
    i =0
```

```
  count =0
while i <len(word) -1:
if word[i] == word[i+1]:
      count = count +1
    i = i +2
if count ==3:
returnTrue
deffind_double_words():
  fin =open("words.txt")
for line in fin:
    word =line.strip()
ifis_double(word):
print(word)
# Exercise 8   Here’s another Car Talk Puzzler (http://www.cartalk.com/content/puzzlers):
# “I was driving on the highway the other day and I happened to notice my odometer.
# Like most odometers, it shows six digits, in whole miles only. So, if my car had
# 300,000 miles, for example, I’d see 3-0-0-0-0-0.
# “Now, what I saw that day was very interesting. I noticed that the last 4 digits
# were palindromic; that is, they read the same forward as backward.
# For example, 5-4-4-5 is a palindrome, so my odometer could have read 3-1-5-4-4-5.
# “One mile later, the last 5 numbers were palindromic.
# For example, it could have read 3-6-5-4-5-6. One mile after that, the middle

# 4 out of 6 numbers were palindromic. And you ready for this? One mile later,

# all 6 were palindromic!

# “The question is, what was on the odometer when I first looked?”

# Write a Python program that tests all the six-digit numbers and prints any

# numbers that satisfy these requirements

# Version 1

defis_palidrome(num, start):

numString=str(num)

    i = start

    j =len(numString) -1

    count =0

while i <= j:

ifnumString[i] ==numString[j]:

print(numString)

else:

returnFalse

        i = i +1

        j = j -1

# My answer is different than the book's answer:
http://greenteapress.com/thinkpython2/code/cartalk2.py

# In my version, the user can pick where they would like to start testing the number

# to see if it is a palidrome so it works for other numbers besides the ones that
# are given to us in this example.

# Version 2

defis_palidrome_ex(num):

numString=str(num)

ifnumString[::-1] ==numString:

returnTrue

defcar_talk_paldromes(num):

numString=str(num)

ifis_palidrome_ex(num):

print(num)

new_num=numString[2::]

ifis_palidrome_ex(new_num):

print(num)

new_num=numString[1:6]

ifis_palidrome_ex(new_num):

print(num)

new_num=numString[1:5:]

ifis_palidrome_ex(new_num):

print(num)

# My second version tests to see if the specific examples given to us are palidromes

# It only works for numbers that are similar to the example.

# Exercise 9

# Here's another Car Talk Puzzler you can solve with a search
```

```
(http://www.cartalk.com/content/puzzlers):
#  “Recently I had a visit with my mom and we realized that the two digits that
# make up my age when reversed resulted in her age. For example, if she’s 73,
# I’m 37. We wondered how often this has happened over the years but we got
# sidetracked with other topics and we never came up with an answer.
# “When I got home I figured out that the digits of our ages have been reversible
# six times so far. I also figured out that if we’re lucky it would happen again
# in a few years, and if we’re really lucky it would happen one more time after that.
# In other words, it would have happened 8 times over all. So the question is, how
old am I now?”
# Write a Python program that searches for solutions to this Puzzler. Hint:
# you might find the string method zfill useful.
defis_palidrome(motherAge, daughterAge):
motherAge=str(motherAge)
daughterAge=str(daughterAge)
   difference =len(motherAge) -len(daughterAge)
daughterAge=daughterAge.zfill(len(motherAge))
motherAge=motherAge[::-1]
ifmotherAge==daughterAge:
returnTrue
else:
returnFalse
# for ageAtBirth in range (1, 50):
```

```
# count = 0
#for motherAge in range (20, 120):
#    daughterAge = motherAge – ageAtBirth
#    if is_palidrome(motherAge, daughterAge):
#        count = count + 1
#    print(motherAge, daughterAge)
# print(count)
# this is the wrong answer!!
count =0
previousDiffAge=0
formotherAgeinrange (15, 120):
fordaughterAgeinrange(1, 100):
diffAge=motherAge-daughterAge
ifis_palidrome(motherAge, daughterAge) anddiffAge==previousDiffAge:
        count = count +1
if count ==6:
print(motherAge)
print(daugtherAge)
previousDiffAge=diffAge
```

Search

All of the exercises in the previous section have something in common; they can be solved with the search pattern. The simplest example is:

```
def e(word):
    for letter in word:
        if letter == 'e':
            return True
    return False
```

OUTPUT

```
>>> e("eel")
True
>>> e("lee")
True
>>> e("xys")
False
>>> e("feel")
True
```

The for loop traverses the characters in word. If we find the letter “e”, we can immediately return False; otherwise we have to go to the next letter. If we exit the loop normally, that means we didn’t find an “e”, so we return True.
You could write this function more concisely using the in operator, but I started with this version because it demonstrates the logic of the search pattern.
avoids is a more general version of has_no_e but it has the same structure:

```
def a(word, forbidden):
    for letter in word:
        if letter in forbidden:
            print("There is atleast one letter occurance of the string")
            return True
    print("There is no occurance of the string")
    return False
```

OUTPUT

```
>>> a("jackal","zj")
There is atleast one letter occurance of the string
True
>>> a("jackal","zlak")
There is atleast one letter occurance of the string
True
>>> a("jackal","z")
There is no occurance of the string
False
```

We can return False as soon as we find a forbidden letter; if we get to the end of the loop, we return True.
uses_only is similar except that the sense of the condition is reversed:

```
def uses_only(word, available):
    for letter in word:
        if letter not in available:
            print("The first character:",letter,"of the 1st string NOT is having an occurance in the 2nd
string")
            return False
        print("The first character:",letter,"of the 1st string is having an occurance in the 2nd string")
        return True
```

OUTPUT: (Comparison is done from 1st string to the 2ndstring)

```
>>> uses_only("lion","simba")
The first character: l of the 1st string NOT is having an occurance in the 2nd string
False
>>> uses_only("lion","simba the lion")
The first character: l of the 1st string is having an occurance in the 2nd string
True
>>> uses_only("lion","noil")
The first character: l of the 1st string is having an occurance in the 2nd string
True
```

Instead of a list of forbidden letters, we have a list of available letters. If we find a letter in word that is not in available, we can return False.
uses_all is similar except that we reverse the role of the word and the string of letters:

```
def uses_all(word, required):
for letter in required:
if letter not in word:
print("The first character:",letter,"of the 2nd string NOT is having an occurance in the 1st
string")
return False
print("The first character:",letter,"of the 2nd string is having an occurance in the 1st string")
return True
```

OUTPUT: (Comparison is done from 2nd string to the 1st string)

```
>>> uses_all("lion","king")
The first character: k of the 2nd string NOT is having an occurance in the 1st string
False
>>> uses_all("lion king","king")
```

```
The first character: k of the 2nd string is having an occurance in the 1st string
True
>>> uses_all("lion king","ing")
The first character: i of the 2nd string is having an occurance in the 1st string
True
>>> uses_all("lion king","gink")
The first character: g of the 2nd string is having an occurance in the 1st string
True
```

Instead of traversing the letters in word, the loop traverses the required letters. If any of the required letters do not appear in the word, we can return False.
If you were really thinking like a computer scientist, you would have recognized that uses_all was an instance of a previously solved problem, and you would have written:

```
def uses_all(word, required):
   return uses_only(required, word)
```

This is an example of a program development plan called **reduction to a previously solved problem**, which means that you recognize the problem you are working on as an instance of a solved problem and apply an existing solution.

Looping with Indices

I wrote the functions in the previous section with for loops because I only needed the characters in the strings; I didn’t have to do anything with the indices.
For is_abecedarian we have to compare adjacent letters, which is a little tricky with a for loop:

```
def is_abecedarian(word):
   previous = word[0]
   print("previous:",previous)
   for c in word:
      print("c:",c)
      if c < previous:
         print("The Characters in the word is NOT in alphabetical order")
         return False
      previous = c
   print("The Characters in the word is in alphabetical order")
   return True
```

OUTPUT: This program is done by replacing the value everytime with the previous one using for loop

```
>>>is_abecedarian('abcd')
The Characters in the word is in alphabetical order
```

```
True
>>>is_abecedarian('dcba')
The Characters in the word is NOT in alphabetical order
False
>>>is_abecedarian('Jack')
The Characters in the word is in alphabetical order
True
>>>is_abecedarian('jack')
The Characters in the word is NOT in alphabetical order
False
```

An alternative is to use recursion:

```
def is_abecedarian(word):
    if len(word) <= 1:
        print("The Characters in the word is in alphabetical order")
        return True
    if word[0] > word[1]:
        print("The Characters in the word is NOT in alphabetical order")
        return False
    return is_abecedarian(word[1:])
```

OUTPUT: This program is done by replacing the value everytime with the previous one using Looping

```
>>>is_abecedarian('abcd')
The Characters in the word is in alphabetical order
True
>>>is_abecedarian('dcba')
The Characters in the word is NOT in alphabetical order
False
>>>is_abecedarian('Jack')
The Characters in the word is in alphabetical order
True
>>>is_abecedarian('jack')
The Characters in the word is NOT in alphabetical order
False
```

Another option is to use a while loop:

```
def is_abecedarian(word):
    i = 0
while i <len(word)-1:
```

```
if word[i+1] < word[i]:
print("The Characters in the word is NOT in alphabetical order")
return False
        i = i+1
print("The Characters in the word is in alphabetical order")
return True
```

OUTPUT: This program is done by replacing the value everytime with the previous one using while loop

```
>>>is_abecedarian('abcd')
The Characters in the word is in alphabetical order
True
>>>is_abecedarian('dcba')
The Characters in the word is NOT in alphabetical order
False
>>>is_abecedarian('Jack')
The Characters in the word is in alphabetical order
True
>>>is_abecedarian('jack')
The Characters in the word is NOT in alphabetical order
False
```

The loop starts at i=0 and ends when i=len(word)-1. Each time through the loop, it compares the ith character (which you can think of as the current character) to thei+1th character (which you can think of as the next).
If the next character is less than (alphabetically before) the current one, then we have discovered a break in the abecedarian trend, and we return False.
If we get to the end of the loop without finding a fault, then the word passes the test. To convince yourself that the loop ends correctly, consider an example like 'flossy'. The length of the word is 6, so the last time the loop runs is when i is 4, which is the index of the second-to-last character. On the last iteration, it compares the second-to- last character to the last, which is what we want.
Here is a version of is_palindrome that uses two indices: one starts at the beginning and goes up; the other starts at the end and goes down.

```
def is_palindrome(word):
    i = 0
    j = len(word)-1
while i<j:
if word[i] != word[j]:
return False
        i = i+1
        j = j-1
```

```
return True
```

Or we could reduce to a previously solved problem and write:

```
def is_palindrome(word):
return is_reverse(word, word)
```

Lists

A List Is a Sequence

Like a string, a list is a sequence of values. In a string, the values are characters; in a list, they can be any type. The values in a list are called elements or sometimes items.
There are several ways to create a new list; the simplest is to enclose the elements in square brackets ([and]):

```
[10, 20, 30, 40]
['crunchy frog', 'ram bladder', 'lark vomit']
```

The first example is a list of four integers. The second is a list of three strings. The elements of a list don't have to be the same type. The following list contains a string, a float, an integer, and (lo!) another list:

```
['spam', 2.0, 5, [10, 20]]
```

A list within another list is nested.
A list that contains no elements is called an empty list; you can create one with empty brackets, [].
As you might expect, you can assign list values to variables:

```
>>>cheeses = ['Cheddar', 'Edam', 'Gouda']
>>>numbers = [42, 123]
>>>empty = []
>>>print(cheeses, numbers, empty)
['Cheddar', 'Edam', 'Gouda'] [42, 123] []
```

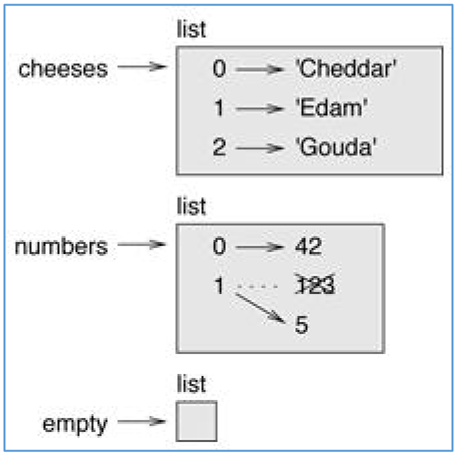
Lists Are Mutable

The syntax for accessing the elements of a list is the same as for accessing the characters of a string—the bracket operator. The expression inside the brackets specifies the index. Remember that the indices start at 0:

```
>>>cheeses[0]
'Cheddar'
```

Unlike strings, lists are mutable. When the bracket operator appears on the left side of an assignment, it identifies the element of the list that will be assigned:

```
>>>numbers = [42, 123]
>>>numbers[1] = 5
>>>numbers
[42, 5]
```



Lists are represented by boxes with the word “list” outside and the elements of the list inside. Cheeses refer to a list with three elements indexed 0, 1 and 2. Numbers contains two elements; the diagram shows that the value of the second element has been reassigned from 123 to 5. Empty refers to a list with no elements.

List indices work the same way as string indices:

- Any integer expression can be used as an index.
- If you try to read or write an element that does not exist, you get an IndexError.
- If an index has a negative value, it counts backward from the end of the list.

The in operator also works on lists:

```
>>>cheeses = ['Cheddar', 'Edam', 'Gouda']
>>> 'Edam' in cheeses
True
>>> 'Brie' in cheeses
False
```

Traversing a List

The most common way to traverse the elements of a list is with a for loop. The syntax is the same as for strings:

```
>>> n=[1,2,3,4,5,6,7,8,9,10]
>>> n
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
>>> type(n)
<class 'list'>
>>> for k in n:
        print(k)

1
2
3
4
5
6
7
8
9
10
>>>
```

This works well if you only need to read the elements of the list. But if you want to write or update the elements, you need the indices. A common way to do that is to combine the built-in functions range and len:

```
>>> for i in range(len(n)):
        n[i]=n[i]*2
        print(n[i])
2
4
6
8
10
12
14
16
18
20
```

This loop traverses the list and updates each element. len returns the number of elements in the list. range returns a list of indices from 0 to n-1, where n is the length of the list. Each time through the loop, i gets the index of the next element. The assignment statement in the body uses

i to read the old value of the element and to assign the new value.
A for loop over an empty list never runs the body:

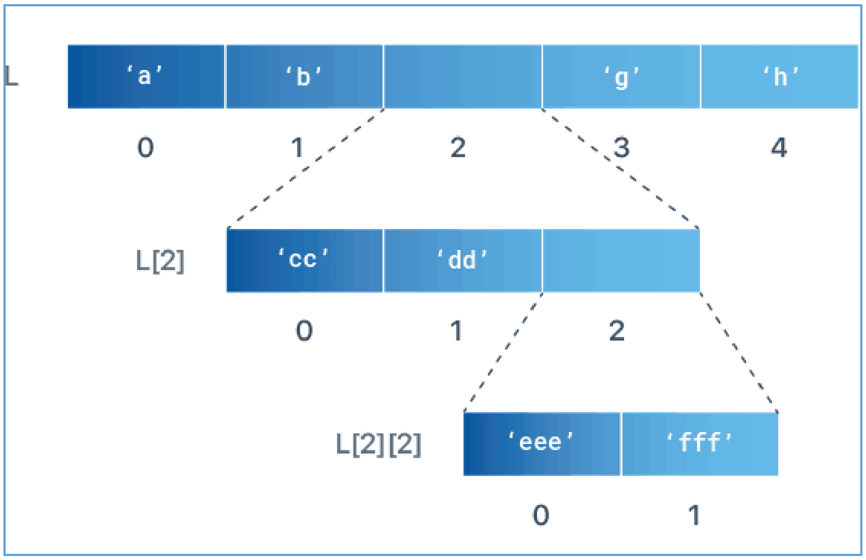
```
for x in []:
print('This never happens.')
```

Although a list can contain another list, the nested list still counts as a single element. The length of this list is four:

```
>>> a=[1,2,3,[4,5,6],7]
>>> a
[1, 2, 3, [4, 5, 6], 7]
>>> len(a)
5
```

Access Nested List Items by Index

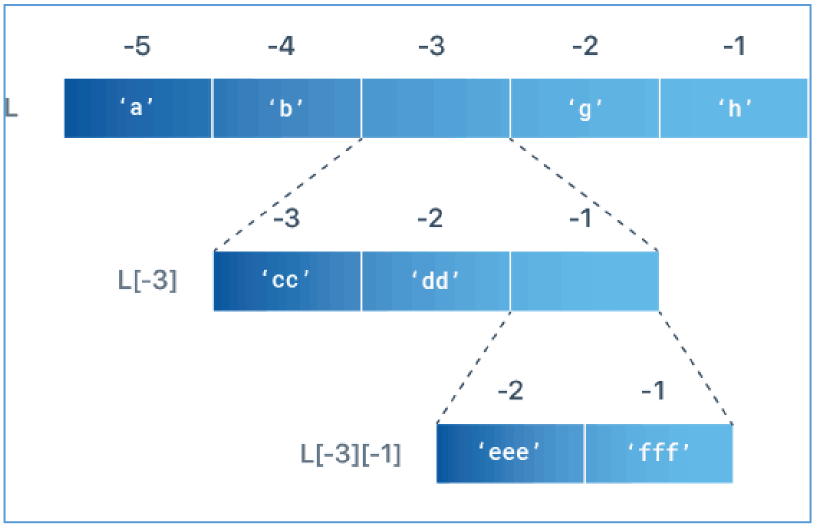
You can access individual items in a nested list using multiple indexes.
The indexes for the items in a nested list are illustrated as below:



```
L = ['a', 'b', ['cc', 'dd', ['eee', 'fff']], 'g', 'h']
print(L[2])
# Prints ['cc', 'dd', ['eee', 'fff']]
print(L[2][2])
# Prints ['eee', 'fff']
print(L[2][2][0])
# Prints eee
```

Negative List Indexing In a Nested List

You can access a nested list by negative indexing as well.
Negative indexes count backward from the end of the list. So, `L[-1]` refers to the last item, `L[-2]` is the second-last, and so on.
The negative indexes for the items in a nested list are illustrated as below:



```
L = ['a', 'b', ['cc', 'dd', ['eee', 'fff']], 'g', 'h']
print(L[-3])
# Prints ['cc', 'dd', ['eee', 'fff']]
print(L[-3][-1])
# Prints ['eee', 'fff']
print(L[-3][-1][-2])
# Prints eee
```

List Operations

The + operator concatenates lists:

```
>>> a = [1, 2, 3]
>>> b = [4, 5, 6]
>>> c = a + b
>>>c
[1, 2, 3, 4, 5, 6]
```

The * operator repeats a list a given number of times:

```
>>> [0] * 4
```

```
[0, 0, 0, 0]
>>> [1, 2, 3] * 3
[1, 2, 3, 1, 2, 3, 1, 2, 3]
```

The first example repeats [0] four times. The second example repeats the list [1, 2, 3] three times.

List Slices

The slice operator also works on lists
Note:the traversing of slices is from left to right

1. If both the indices are positive the second index-1 is taken. The first index should be always lesser than the second index.

```
>>> lion
['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j', 'k']
>>> lion[5]
'f'
>>> lion[9]
'j'
>>> lion[5:9]
['f', 'g', 'h', 'i']
>>> lion[9:5]
[]
```

2. If both the indices are negative the second index+1 is taken. The first index should be always lesser than the second index.

```
>>> lion
['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j', 'k']
>>> lion[-9]
'c'
>>> lion[-5]
'g'
>>> lion[-9:-5]
['c', 'd', 'e', 'f']
>>> lion[-5:-9]
[]
```

3. If First index is positive and second index is negative, it will include the first index and second index -1 is taken. The first index position should be always lesser than the second index position.

```
 >>> lion
['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j', 'k']
```

```
>>> lion[2]
'c'
>>> lion[-1]
'k'
>>> lion[2:-1]
['c', 'd', 'e', 'f', 'g', 'h', 'i', 'j']
>>> lion[2:-5]
['c', 'd', 'e', 'f']
>>> lion[5:-2]
['f', 'g', 'h', 'i']
```

4. If First index is negative and second index is positive, it will include the first index and the second index-1 is taken. The first index should be always lesser than the second index.

```
>>> lion
['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j', 'k']
>>> lion[-5]
'g'
>>> lion[9]
'j'
>>> lion[-5:9]
['g', 'h', 'i']
>>> lion[-5:2]
[]
```

0	1	2	3	4	5	6	7	Operation	Description
e	q	u	a	t	i	o	n	s[:]	Full List
e	q	u	a					s[:4]	From First
				t	i	o	n	s[4:]	Till Last
	q	u	a	t				s[1:5]	Between
0	1	2	3	4	5	6	7		
e	q	u	a	t	i	o	n		
-8	-7	-6	-5	-4	-3	-2	-1		Description
e	q	u	a	t	i	o	n	s[:]	Full List

e	q	u	a					s[:-4]	From First	
				t	i	o	n	**s[-4:]**	Till last - We cannot get "tion" using s[-4:0]	
	q	u	a	t				s[-7:-3]	Between	
	q	u	a	t				s[1:-3]	MIX	[+ve:-ve]
	q	u	a	t				s[-7:5]	MIX	[-ve:+ve]

If **you omit the first index, the slice starts at the beginning.**
If you **omit the second, the slice goes to the end.**
So if you **omit both**, the slice is a copy of the **whole list**:

```
>>>t[:]
['a', 'b', 'c', 'd', 'e', 'f']
```

Since lists are mutable, it is often useful to make a copy before performing operations that modify lists.
A slice operator on the left side of an assignment can update multiple elements **(Note: only the left boundary is considered and you can update n number of values)**

List Methods

Python provides methods that operate on lists. For example, append adds a new element to the end of a list:

```
>>> t = ['a', 'b', 'c']
>>>t.append('d')
>>>t
['a', 'b', 'c', 'd']
```

extend takes a list as an argument and appends all of the elements:

```
>>> t1 = ['a', 'b', 'c']
>>> t2 = ['d', 'e']
>>>t1.extend(t2)
>>> t1
['a', 'b', 'c', 'd', 'e']
```

This example leaves t2 unmodified.
sort arranges the elements of the list from low to high:

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

```
>>>t.sort()
>>>t
['a', 'b', 'c', 'd', 'e']

>>> t = ['d', 'c', 'e', 'b', 'a']
>>> t.sort(reverse=True)
>>> t
['e', 'd', 'c', 'b', 'a']
```

Sorted ()

The sorted() function returns a sorted list from the items in an iterable.

The sorted() function sorts the elements of a given iterable in a specific order (either ascending or descending) and returns the sorted iterable as a list.

The syntax of the sorted() function is:

sorted(iterable, key=None, reverse=False)

Parameters for the sorted() function
sorted() can take a maximum of three parameters:

1. iterable - A sequence (string, tuple, list) or collection (set, dictionary, frozen set) or any other iterator.
2. reverse (Optional) - If True, the sorted list is reversed (or sorted in descending order). Defaults to False if not provided.
3. key (Optional) - A function that serves as a key for the sort comparison. Defaults to None.

Example 1: Sort string, list, and tuple

```
>>> a
[65, 48, 2, 78, 22, 1, 665, 4]
>>> sorted(a)
[1, 2, 4, 22, 48, 65, 78, 665]
>>> t=sorted(a)
>>> t
[1, 2, 4, 22, 48, 65, 78, 665]
>>> a
[65, 48, 2, 78, 22, 1, 665, 4]
']

>>> p= ['e', 'a', 'u', 'o', 'i']
>>> p
['e', 'a', 'u', 'o', 'i']
```

```
>>> sorted(p)
['a', 'e', 'i', 'o', 'u']
>>> sorted(p,reverse=True)
['u', 'o', 'i', 'e', 'a']
>>>
>>> s="python"
>>> sorted(s)
['h', 'n', 'o', 'p', 't', 'y']
>>> sorted(s,reverse=True)
['y', 't', 'p', 'o', 'n', 'h']
```

Most list methods are void; they modify the list and return None. If you accidentally write t = t.sort(), you will be disappointed with the result.

Map, Filter and Reduce

To add up all the numbers in a list, you can use a loop like this:

```
a=[1,2,3,4,5]
def add():
        total=0
        for x in a:
                total+=x
        return total
```

total is initialized to 0. Each time through the loop, x gets one element from the list. The += operator provides a short way to update a variable. This **augmented assignment statement**,

```
total += x
```

is equivalent to

```
total = total + x
```

As the loop runs, total accumulates the sum of the elements; a variable used this way is sometimes called an **accumulator**.

Adding up the elements of a list is such a common operation that Python provides it as a built-in function, sum:

```
>>> t = [1, 2, 3]
```

```
>>>sum(t)
6
```

An operation like this that combines a sequence of elements into a single value is sometimes called **reduce**.

Sometimes you want to traverse one list while building another. For example, the following function takes a list of strings and returns a new list that contains capitalized strings:

```
t=['a','b','c','d','e']
def cap(t):
    res=[]
    for s in t:
        res.append(s.capitalize())
    return res
```

res is initialized with an empty list; each time through the loop, we append the next element. So res is another kind of accumulator.

An operation like capitalize_all is sometimes called a **map** because it “maps” a function (in this case the method capitalize) onto each of the elements in a sequence.

Another common operation is to select some of the elements from a list and return a sublist. For example, the following function takes a list of strings and returns a list that contains only the uppercase strings:

```
t=['A','b','C','d','e']
def only_upper(t):
    res = []
    for s in t:
        if s.isupper():
            res.append(s)
    return res
```

isupper is a string method that returns True if the string contains only uppercase letters.

An operation like only_upper is called a **filter** because it selects some of the elements and filters out the others.

Most common list operations can be expressed as a combination of map, filter and reduce.

Deleting Elements

There are several ways to delete elements from a list. If you know the index of the element you want, you can use **pop**:

```
>>> t = ['a', 'b', 'c']
>>> x = t.pop(1)
>>>t ['a', 'c']
```

```
>>>x
'b'
```

pop modifies the list and returns the element that was removed. If you don't provide an index, it deletes and returns the last element.
If you don't need the removed value, you can use the del operator:

```
>>> t = ['a', 'b', 'c']
>>>del t[1]
>>>t
['a', 'c']
```

If you know the element you want to remove (but not the index), you can use remove:

```
>>> t = ['a', 'b', 'c']
>>>t.remove('b')
>>>t
['a', 'c']
```

The return value from remove is None.
To remove more than one element, you can use del with a slice index:

```
>>> t = ['a', 'b', 'c', 'd', 'e', 'f']
>>>del t[1:5]
>>>t
['a', 'f']
```

As usual, **the slice selects all the elements up to but not including the second index.**

Lists and Strings

A string is a sequence of characters and a list is a sequence of values, but a list of characters is not the same as a string. To convert from a string to a list of characters, you can use list:

```
>>> s = 'spam'
>>> t = list(s)
>>>t
['s', 'p', 'a', 'm']
```

Because **list is the name of a built-in function**, you should avoid using it as a variable name. The **list** function **breaks a string into individual letters**. If you want to **break a string into words**, you can use the **split** method:

```
>>> s = 'pining for the fjords'
>>> t = s.split()
>>>t
['pining', 'for', 'the', 'fjords']
```

An optional argument called a delimiter specifies which characters to use as word boundaries. The following example uses a hyphen as a delimiter:

```
>>> s = 'spam-spam-spam'
>>>delimiter = '-'
>>> t = s.split(delimiter)
>>>t
['spam', 'spam', 'spam']
```

```
>>> j="lion****king**9*8-45/89"
>>> h="*"
>>> k=j.split(h)
>>> k
['lion', '', '', '', 'king', '', '9', '8-45/89']
```

join is the inverse of split. It takes a list of strings and concatenates the elements. join is a string method, so you have to invoke it on the delimiter and pass the list as a parameter:

```
>>> t = ['pining', 'for', 'the', 'fjords']
>>>delimiter = ' '
>>> s = delimiter.join(t)
>>>s
'pining for the fjords'
```

```
>>> s="please,read,the,following"
>>> d=","
>>> t=s.split(d)
>>> t
['please', 'read', 'the', 'following']
>>> k=d.join(t)
>>> k
'please,read,the,following'
>>>
'
>>> x="lion king is my favourite movie"
>>> y=x.split()
>>> y
['lion', 'king', 'is', 'my', 'favourite', 'movie']
```

```
>>> d="*"
>>> ab=d.join(y)
>>> ab
'lion*king*is*my*favourite*movie'
>>> e=""
>>>
>>> ba=e.join(y)
>>> ba
'lionkingismyfavouritemovie'
>>> e=" "
>>> ba=e.join(y)
>>> ba
'lion king is my favourite movie'
>>>
```

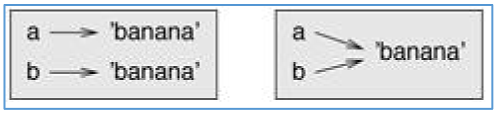
In this case the delimiter is a space character, so join puts a space between words. To concatenate strings without spaces, you can use the empty string, '', as a delimiter.

Objects and Values

If we run these assignment statements:

```
a = 'banana'
b = 'banana'
```

We know that a and b both refer to a string, but we don’t know whether they refer to the same string. There are two possible states, shown in the fig

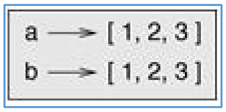


In one case, a and b refer to two different objects that have the same value. In the second case, they refer to the same object.
To check whether two variables refer to the same object, you can use the is operator:

```
>>> a = 'banana'
>>> b = 'banana'
>>>a is b
True
```

In this example, Python only created one string object, and both a and b refer to it. But when you create two lists, you get two objects:

```
>>> a = [1, 2, 3]
>>> b = [1, 2, 3]
>>>a is b
False
```

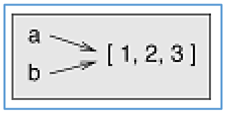


In this case we would say that the two lists are **equivalent**, because they have the same elements**, but not identical**, because they are not the same object. If two objects are identical, they are also equivalent, but if they are equivalent, they are not necessarily identical.
Until now, we have been using “object” and “value” interchangeably, but it is more precise to say that an object has a value. If you evaluate [1, 2, 3], you get a list object whose value is a sequence of integers. If another list has the same elements, we say it has the same value, but it is not the same object.

Aliasing

If a refers to an object and you assign b = a, then both variables refer to the same object:

```
>>> a = [1, 2, 3]
>>> b = a
>>>b is a
True
```



The association of a variable with an object is called a **reference**. In this example, there are two references to the same object.
An object with more than one reference has more than one name, so we say that the object is **aliased**.
If the aliased object is mutable, changes made with one alias affect the other:

```
>>>b[0] = 42
>>>a
[42, 2, 3]
```

Although this behavior can be useful, it is error-prone. In general, it is safer to avoid aliasing when you are working with mutable objects.

For immutable objects like strings, aliasing is not as much of a problem. In this example:

```
a = 'banana'
b = 'banana'
```

It almost never makes a difference whether a and b refer to the same string or not.

List Arguments

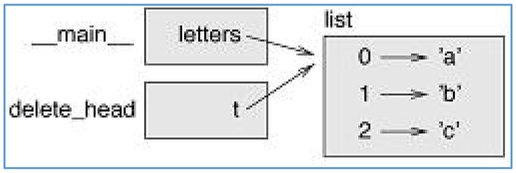
When you pass a list to a function, the function gets a reference to the list. If the function modifies the list, the caller sees the change. For example, delete_head removes the first element from a list:

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

Here’s how it is used:

```
>>>letters = ['a', 'b', 'c']
>>>delete_head(letters)
>>>letters
['b', 'c']
```

The parameter t and the variable letters are aliases for the same object.



Since the list is shared by two frames, I drew it between them.

It is important to distinguish between operations that modify lists and operations that create new lists. For example, the append method modifies a list, but the + operator creates a new list:

```
>>> t1 = [1, 2]
>>> t2 = t1.append(3)
>>> t1
[1, 2, 3]
>>> t2
None
```

append modifies the list and returns None:

```
>>> t3 = t1 + [4]
>>> t1
[1, 2, 3]
>>> t3
[1, 2, 3, 4]
>>> t2
```

The + operator creates a new list and leaves the original list unchanged.
This difference is important when you write functions that are supposed to modify lists. For example, this function does not delete the head of a list:

```
def bad_delete_head(t):
t = t[1:]          # WRONG!
```

The slice operator creates a new list and the assignment makes t refer to it, but that doesn’t affect the caller.

```
>>> t4 = [1, 2, 3]
>>>bad_delete_head(t4)
>>> t4
[1, 2, 3]
```

At the beginning of bad_delete_head, t and t4 refer to the same list. At the end, t refers to a new list, but t4 still refers to the original, unmodified list.
An alternative is to write a function that creates and returns a new list. For example,tail returns all but the first element of a list:

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

This function leaves the original list unmodified. Here’s how it is used:

```
>>>letters = ['a', 'b', 'c']
>>>rest = tail(letters)
>>>rest
['b', 'c']
```

Debugging
Careless use of lists (and other mutable objects) can lead to long hours of debugging. Here are some common pitfalls and ways to avoid them:

- Most list methods modify the argument and return None. This is the opposite of the string methods, which return a new string and leave the original alone.

If you are used to writing string code like this:

```
word = word.strip()
```

It is tempting to write list code like this:

```
t = t.sort()        # WRONG!
```

Because sort returns None, the next operation you perform with t is likely to fail.
Before using list methods and operators, you should read the documentation carefully and then test them in interactive mode.

- Pick an idiom and stick with it.

Part of the problem with lists is that there are too many ways to do things. For example, to remove an element from a list, you can use pop, remove, del, or even a slice assignment.
To add an element, you can use the append method or the + operator. Assuming that t is a list and x is a list element, these are correct:

```
t.append(x)
t = t + [x]
t += [x]
```

And these are wrong:

```
t.append([x])   # WRONG!
t = t.append(x)# WRONG!
t + [x]  # WRONG!
t = t + x          # WRONG!
```

Try out each of these examples in interactive mode to make sure you understand what they do. Notice that only the last one causes a runtime error; the other three are legal, but they do the wrong thing.

- Make copies to avoid aliasing.

If you want to use a method like sort that modifies the argument, but you need to keep the original list as well, you can make a copy:

```
>>> t = [3, 1, 2]
>>> t2 = t[:]
>>>t2.sort()
>>>t
[3, 1, 2]
>>> t2
[1, 2, 3]
```

In this example you could also use the built-in function sorted, which returns a new, sorted list and leaves the original alone:

```
>>> t2 = sorted(t)
>>>t
[3, 1, 2]
>>> t2
[1, 2, 3]
```

Regulation: AK20	**Subject Code:CSE/CIC** 20APS0526/20APC3605	**Subject Name :** Basics of Python Programming	**AY:** 2023-2024
UNIT-4 (Dictionaries, Tuples, Files, Classes and Objects)			

UNIT – IV

A Dictionary Is a Mapping

A dictionary is like a list, but more general. In a list, the indices have to be integers; in a dictionary they can be (almost) any type.

A dictionary contains a collection of indices, which are called **keys**, and a collection of values. Each key is associated with a single value. The association of a key and a value is called **a key-value pair** or sometimes an **item**.

In mathematical language, a dictionary represents a **mapping** from keys to values, so you can also say that each key "maps to" a value. As an example, we'll build a dictionary that maps from English to Spanish words, so the keys and the values are all strings.

The function dict creates a new dictionary with no items. Because dict is the name of a built-in function, you should avoid using it as a variable name.

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

The squiggly brackets, {}, represent an empty dictionary. To add items to the dictionary, you can use square brackets:

```
>>>d[1]="lion"
```

This line creates an item that maps from the key '1' to the value 'lion''. If we print the dictionary again, we see a key-value pair with a colon between the key and value:

```
>>>d["two"]=2
>>>d[1.5]="float"
>>>d["list"]=[1,"simba",1.414]
>>>d
{1: 'lion', 'two': 2, 1.5: 'float', 'list': [1, 'simba', 1.414]}
>>>type(d)
<class 'dict'>
>>>type(d[1])
<class 'str'>
>>>type(d['list'])
<class 'list'>
```

This output format is also an input format. For example, you can create a new dictionary with three items:

```
>>> e={"first":"mufasa",2:32768,1.414:123.456}
>>>e
{'first': 'mufasa', 2: 32768, 1.414: 123.456}
```

Dictionary Updation

Addition

```
>>> a={"name":"jack","age":26}
>>>a
{'name': 'jack', 'age': 26}
>>>id(a)
59051056
>>>a["address"]="Tirupati"
>>>a
{'name': 'jack', 'age': 26, 'address': 'Tirupati'}
>>>id(a)
59051056
>>>a["age"]=52
>>>a
{'name': 'jack', 'age': 52, 'address': 'Tirupati'}
>>>id(a)
59051056
```

Deletion

```
# Removing elements from a dictionary

# create a dictionary
>>> squares = {1: 1, 2: 4, 3: 9, 4: 16, 5: 25}
>>> squares
{1: 1, 2: 4, 3: 9, 4: 16, 5: 25}
>>>id(squares)
61911320
# remove a particular item, returns its value
# Output: 16
>>>print(squares.pop(4))
16
>>> squares
{1: 1, 2: 4, 3: 9, 5: 25}
```

```
# remove an arbitrary item, return (key,value)
# Output: (5, 25)
>>>print(squares.popitem())
(5, 25)
>>> squares
{1: 1, 2: 4, 3: 9}
# remove all items
squares.clear()
>>> squares
{}
# delete the dictionary itself
>>>del squares
>>> squares
Traceback (most recent call last):
  File "<pyshell#100>", line 1, in <module>
squares
NameError: name 'squares' is not defined
```

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

```
>>>e[1.414]
123.456
```

The key '1.414' always maps to the value '123.456' so the order of the items doesn’t matter. If the key isn’t in the dictionary, you get an exception:

```
>>>e[7]
Traceback (most recent call last):
  File "<pyshell#27>", line 1, in <module>
e[7]
KeyError: 7
```

The len function works on dictionaries; it returns the number of key-value pairs:

```
>>>len(e)
3
```

The in operator works on dictionaries, too; it tells you whether something appears as a key in the dictionary (appearing as a value is not good enough).(NOTE: we cannot search with values using in function directly, we have to use only key)

```
>>> "mufasa" in e
False
>>> "first" in e
True
>>> 2 in e
True
```

To see whether something appears as a value in a dictionary, you can use the method values, which returns a collection of values, and then use the in operator:

```
>>> a={24:50,12:5,20:45}
>>> 5 in a.values()
True
>>> 12 in a.keys()
True

>>>val=e.values()
>>>val
dict_values(['mufasa', 32768, 123.456])
>>> 'mufasa' in val
True
```

The in operator uses different algorithms for lists and dictionaries. For lists, it searches the elements of the list in order. As the list gets longer, the search time gets longer in direct proportion.

For dictionaries, Python uses an algorithm called a **hashtable** that has a remarkable property: the in operator takes about the same amount of time no matter how many items are in the dictionary.

Dictionary as a Collection of Counters

Suppose you are given a string and you want to count how many times each letter appears. There are several ways you could do it:

1. You could create 26 variables, one for each letter of the alphabet. Then you could traverse the string and, for each character, increment the corresponding counter, probably using a chained conditional.
2. You could create a list with 26 elements. Then you could convert each character to a number (using the built-in function ord), use the number as an index into the list, and increment the appropriate counter.
3. You could create a dictionary with characters as keys and counters as the corresponding values. The first time you see a character, you would add an item to the dictionary. After that you would increment the value of an existing item.

Each of these options performs the same computation, but each of them implements that computation in a different way.

An **implementation** is a way of performing a computation; some implementations are better than others. For example, an advantage of the dictionary implementation is that we don’t have to know ahead of time which letters appear in the string and we only have to make room for the letters that do appear. Here is what the code might look like:

```
def histogram(s):
        d = dict()
        for c in s:
                if c not in d:
                        d[c] = 1
                else:
                        d[c] += 1
        return d
```

The name of the function is histogram, which is a statistical term for a collection of counters (or frequencies).

The first line of the function creates an empty dictionary. The for loop traverses the string. Each time through the loop, if the character c is not in the dictionary, we create a new item with key c and the initial value 1 (since we have seen this letter once). If c is already in the dictionary we increment d[c].

Here’s how it works:

```
h = histogram('brontosaurus')
>>>h
{'b': 1, 'r': 2, 'o': 2, 'n': 1, 't': 1, 's': 2, 'a': 1, 'u': 2}
```

The histogram indicates that the letters ’a’ and 'b' appear once; 'o' appears twice, and so on.

Dictionaries have a method called get that takes a key and a default value. If the key appears in the dictionary, get returns the corresponding value; otherwise it returns the default value. For example:

```
>>> h = histogram('brontosaurus')
>>>h.get('r',0)
2
>>>h.get('z',0)
0
```

As an exercise, use get to write histogram more concisely. You should be able to eliminate the if statement.

Looping and Dictionaries

If you use a dictionary in a for statement, it traverses the keys of the dictionary. For example, print_hist prints each key and the corresponding value:

```
def histogram(s):
    d = dict()
    for c in s:
        if c not in d:
            d[c] = 1
        else:
            d[c] += 1
    return d

def print_hist(h):
    for c in h:
        print(c,h[c])
```

Here's what the output looks like:

```
>>> h = histogram('brontosaurus')
>>>print_hist(h)
b 1
r 2
o 2
n 1
t 1
s 2
a 1
u 2
```

Again, the keys are in no particular order. To traverse the keys in sorted order, you can use the built-in function sorted:

```
>>>for key in sorted(h):
    print(key,h[key])


a 1
b 1
```

```
n 1
o 2
r 2
s 2
t 1
u 2
```

Sorted ()

The sorted() function returns a sorted list from the items in an iterable.

The sorted() function sorts the elements of a given iterable in a specific order (either ascending or descending) and returns the sorted iterable as a list.

The syntax of the sorted() function is:

```
sorted(iterable, key=None, reverse=False)
```

Parameters for the sorted() function
sorted() can take a maximum of three parameters:

1. iterable - A sequence (string, tuple, list) or collection (set, dictionary, frozen set) or any other iterator.
2. reverse (Optional) - If True, the sorted list is reversed (or sorted in descending order). Defaults to False if not provided.
3. key (Optional) - A function that serves as a key for the sort comparison. Defaults to None.

```
>>> a={24:50,12:5,20:45}
>>>type(a)
<class 'dict'>
>>>a
{24: 50, 12: 5, 20: 45}
>>>sorted(a.keys())
[12, 20, 24]
>>>sorted(a.values())
[5, 45, 50]
>>>sorted(a.items())
[(12, 5), (20, 45), (24, 50)]
>>>sorted(a.values() ,  reverse= True)
[50, 45, 5]
>>>sorted(a.keys() ,  reverse= True)
[24, 20, 12]
```

```
>>>sorted(a.items() ,  reverse= True)
[(24, 50), (20, 45), (12, 5)]


>>> b={"king":70,"Flin":100,"hat":50}
>>>type(b)
<class 'dict'>
>>>b
{'king': 70, 'Flin': 100, 'hat': 50}
>>>sorted(b.keys())
['Flin', 'hat', 'king']
>>>sorted(b.values())
[50, 70, 100]
>>>sorted(b.items())
[('Flin', 100), ('hat', 50), ('king', 70)]
>>>sorted(b.keys() ,  reverse= True)
['king', 'hat', 'Flin']
>>>sorted(b.values() ,  reverse= True)
[100, 70, 50]
>>>sorted(b.items() ,  reverse= True)
[('king', 70), ('hat', 50), ('Flin', 100)]

>>> c={3.1:8,2.75:20,2.9:15}
>>>type(c)
<class 'dict'>
>>>c
{3.1: 8, 2.75: 20, 2.9: 15}
>>>sorted(c.values())
[8, 15, 20]
>>>sorted(c.keys())
[2.75, 2.9, 3.1]
>>>sorted(c.items())
[(2.75, 20), (2.9, 15), (3.1, 8)]

d={'hat':50,3.14:4.5,25:"lion"}
>>>type(d)
<class 'dict'>
>>>sorted(d.values())
Traceback (most recent call last):
  File "<pyshell#55>", line 1, in <module>
sorted(d.values())
TypeError: '<' not supported between instances of 'str' and 'float'
>>>sorted(c.keys())
```

```
Traceback (most recent call last):
  File "<pyshell#56>", line 1, in <module>
sorted(c.keys())
TypeError: '<' not supported between instances of 'float'
```

Reverse Lookup

Given a dictionary d and a key k, it is easy to find the corresponding value v = d[k]. This operation is called a **lookup**.

But what if you have v and you want to find k? You have two problems: first, there might be more than one key that maps to the value v. Depending on the application, you might be able to pick one, or you might have to make a list that contains all of them. Second, there is no simple syntax to do a reverse lookup; you have to search. Here is a function that takes a value and returns the first key that maps to that value:

```
def histogram(s):
        d = dict()
        for c in s:
                if c not in d:
                        d[c] = 1
                else:
                        d[c] += 1
        return d

defprint_hist(h):
        for c in h:
                print(c,h[c])

defreverse_lookup(d, v):
        for k in d:
                if d[k] == v:
                        return k
        raiseLookupError()
```

This function is yet another example of the search pattern, but it uses a feature we haven't seen before: raise. The **raise statement** causes an exception; in this case it causes a LookupError, which is a built-in exception used to indicate that a lookup operation failed.

If we get to the end of the loop, that means v doesn't appear in the dictionary as a value, so we raise an exception. Here is an example of a successful reverse lookup:

```
>>> h = histogram('brontosaurus')
>>> k=reverse_lookup(h,2)
>>>k
'r'
```

And an unsuccessful one:

```
>>> k=reverse_lookup(h,3)
Traceback (most recent call last):
  File "<pyshell#62>", line 1, in <module>
    k=reverse_lookup(h,3)
  File "C:/Users/Admin/AppData/Local/Programs/Python/Python38/test12.py", line 18, in reverse_lookup
raiseLookupError()
LookupError
```

The effect when you raise an exception is the same as when Python raises one: it prints a traceback and an error message.

The raise statement can take a detailed error message as an optional argument. For example:

```
def histogram(s):
        d = dict()
        for c in s:
                if c not in d:
                        d[c] = 1
                else:
                        d[c] += 1
        return d

defprint_hist(h):
        for c in h:
                print(c,h[c])

defreverse_lookup(d, v):
        for k in d:
                if d[k] == v:
                return k
        raiseLookupError('value does not appear in the dictionary')

>>> h = histogram('brontosaurus')
>>> k=reverse_lookup(h,2)
>>>k
```

```
'r'
>>> k=reverse_lookup(h,3)
Traceback (most recent call last):
  File "<pyshell#67>", line 1, in <module>
    k=reverse_lookup(h,3)
  File "C:/Users/Admin/AppData/Local/Programs/Python/Python38/dictionary.py", line 162, in reverse_lookup
raiseLookupError('value does not appear in the dictionary')
LookupError: value does not appear in the dictionary
```

A reverse lookup is much slower than a forward lookup; if you have to do it often, or if the dictionary gets big, the performance of your program will suffer.

Dictionaries and Lists

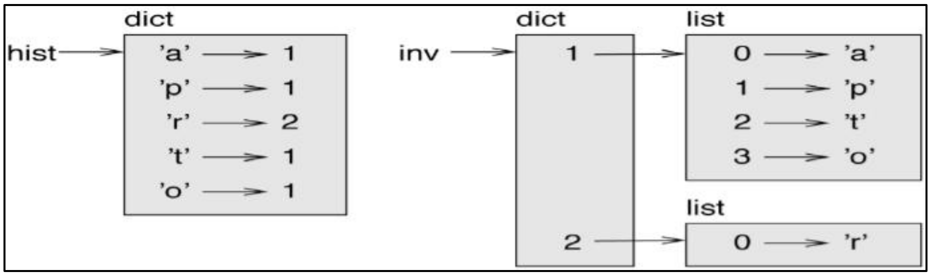
Lists can appear as values in a dictionary. For example, if you are given a dictionary that maps from letters to frequencies, you might want to invert it; that is, create a dictionary that maps from frequencies to letters. Since there might be several letters with the same frequency, each value in the inverted dictionary should be a list of letters. Here is a function that inverts a dictionary:

```
def histogram(s):
        d = dict()
        for c in s:
                if c not in d:
                        d[c] = 1
                else:
                        d[c] += 1
        return d

definvert_dict(d):
        inverse = dict()
        for key in d:
                val = d[key]
                ifval not in inverse:
                        inverse[val] = [key]
                else:
                        inverse[val].append(key)
        return inverse
```

Each time through the loop, key gets a key from h and val gets the corresponding value. If val is not in inverse, that means we haven’t seen it before, so we create a new item and initialize it with a **singleton** (a list that contains a single element). Otherwise we have seen this value before, so we append the corresponding key to the list. Here is an example:

```
>>> h = histogram('brontosaurus')
>>>h
{'b': 1, 'r': 2, 'o': 2, 'n': 1, 't': 1, 's': 2, 'a': 1, 'u': 2}
>>>inverse=invert_dict(h)
>>>inverse
{1: ['b', 'n', 't', 'a'], 2: ['r', 'o', 's', 'u']}
```

Figure is a state diagram showing hist and inverse. A dictionary is represented as a box with the type dict above it and the key-value pairs inside. If the values are integers, floats or strings, I draw them inside the box, but I usually draw lists outside the box, just to keep the diagram simple.



Lists can be values in a dictionary, as this example shows, but they cannot be keys. Here’s what happens if you try:

```
>>> t=[1,2,3]
>>> d=dict()
>>>d[t]='oops'
Traceback (most recent call last):
  File "<pyshell#74>", line 1, in <module>
d[t]='oops'
TypeError: unhashable type: 'list'
```

#In dictionary we can't assign a dictionary or set as a key.
#In dictionary we can assign a list as a key

```
>>> s={[10,20]:"lion",20:"simba",50:"mufasa"}
Traceback (most recent call last):
  File "<pyshell#126>", line 1, in <module>
    s={[10,20]:"lion",20:"simba",50:"mufasa"}
TypeError: unhashable type: 'list'
>>> s={(10,20):"lion",20:"simba",50:"mufasa"}
>>>s
```

```
{(10, 20): 'lion', 20: 'simba', 50: 'mufasa'}
>>>s[(100,200)]="jackal"
>>>s
{(10, 20): 'lion', 20: 'simba', 50: 'mufasa', (100, 200): 'jackal'}
>>>s[(100,200)]
'jackal'
>>>s[(10,20)]
'lion'
>>> t={10:20,20:30,40:50}
>>>t
{10: 20, 20: 30, 40: 50}
>>>t[(100,200)]=1000
>>>t
{10: 20, 20: 30, 40: 50, (100, 200): 1000}
>>>t[{200:300}]=2000
Traceback (most recent call last):
  File "<pyshell#5>", line 1, in <module>
t[{200:300}]=2000
TypeError: unhashable type: 'dict'
>>>t[{200,300}]=2000
Traceback (most recent call last):
  File "<pyshell#6>", line 1, in <module>
t[{200,300}]=2000
TypeError: unhashable type: 'set'
```

I mentioned earlier that a dictionary is implemented using a hashtable and that means that the keys have to be **hashable**.

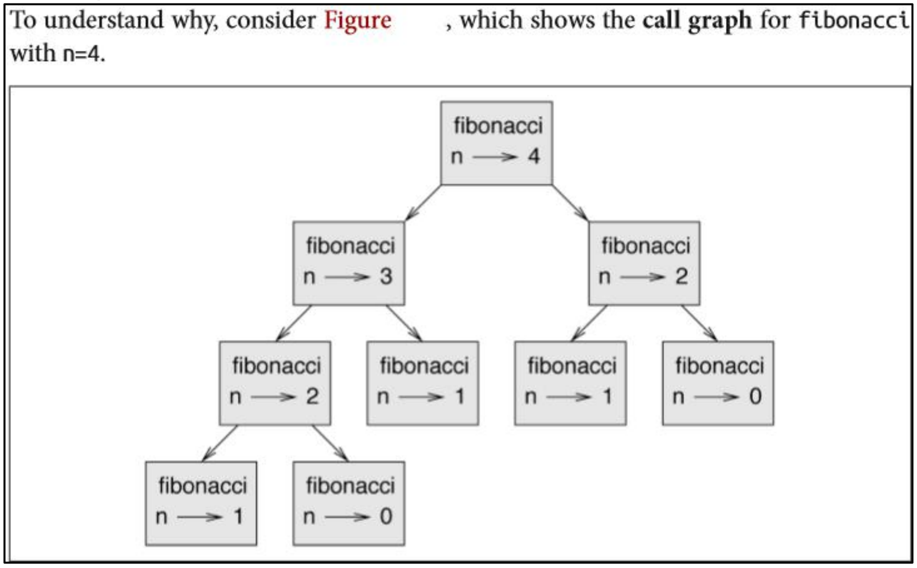
A **hash** is a function that takes a value (of any kind) and returns an integer. Dictionaries use these integers, called hash values, to store and look up key-value pairs.

This system works fine if the keys are immutable. But if the keys are mutable, like lists, bad things happen. For example, when you create a key-value pair, Python hashes the key and stores it in the corresponding location. If you modify the key and then hash it again, it would go to a different location. In that case you might have two entries for the same key, or you might not be able to find a key. Either way, the dictionary wouldn't work correctly.

That's why keys have to be hashable, and why mutable types like lists aren't. The simplest way to get around this limitation is to use tuples. Since dictionaries are mutable, they can't be used as keys, but they can be used as values.

Memos

If you played with the fibonacci function, you might have noticed that the bigger the argument you provide, the longer the function takes to run. Furthermore, the runtime increases quickly.



To understand why, consider Figure , which shows the **call graph** for `fibonacci` with n=4.

A call graph shows a set of function frames, with lines connecting each frame to the frames of the functions it calls. At the top of the graph, fibonacci with n=4 calls fibonacci with n=3 and n=2. In turn, fibonacci with n=3 calls fibonacci with n=2 and n=1. And so on.

Count how many times fibonacci(0) and fibonacci(1) are called. This is an inefficient solution to the problem, and it gets worse as the argument gets bigger.

One solution is to keep track of values that have already been computed by storing them in a dictionary. **A previously computed value that is stored for later use is called a memo**. Here is a "memoized" version of fibonacci:

```
known = {0:0, 1:1}
deffibonacci(n):
        if n in known:
                return known[n]
        res = fibonacci(n-1) + fibonacci(n-2)
        known[n] = res
        return res

>>>fibonacci(4)
3
>>>fibonacci(3)
2
>>>fibonacci(2)
1
```

```
>>>fibonacci(10)
55
The original Dictionary {0: 0, 1: 1}
```

Modified Program

```
known = {0:0, 1:1}
print("The original Dictionary",known)
def f(n):
print("entry:",n,known)
if n in known:
return known[n]
res = f(n-1) + f(n-2)
known[n] = res
print("exit:",n,known)
return res
```

OUTPUT

```
The original Dictionary {0: 0, 1: 1}
>>>fibonacci(5)
entry: 5 {0: 0, 1: 1}
entry: 4 {0: 0, 1: 1}
entry: 3 {0: 0, 1: 1}
entry: 2 {0: 0, 1: 1}
entry: 1 {0: 0, 1: 1}
entry: 0 {0: 0, 1: 1}
exit: 2 {0: 0, 1: 1, 2: 1}
entry: 1 {0: 0, 1: 1, 2: 1}
exit: 3 {0: 0, 1: 1, 2: 1, 3: 2}
entry: 2 {0: 0, 1: 1, 2: 1, 3: 2}
exit: 4 {0: 0, 1: 1, 2: 1, 3: 2, 4: 3}
entry: 3 {0: 0, 1: 1, 2: 1, 3: 2, 4: 3}
exit: 5 {0: 0, 1: 1, 2: 1, 3: 2, 4: 3, 5: 5}
5
The original Dictionary {0: 0, 1: 1}
>>>fibonacci(10)
entry: 10 {0: 0, 1: 1}
entry: 9 {0: 0, 1: 1}
entry: 8 {0: 0, 1: 1}
entry: 7 {0: 0, 1: 1}
entry: 6 {0: 0, 1: 1}
entry: 5 {0: 0, 1: 1}
```

```
entry: 4 {0: 0, 1: 1}
entry: 3 {0: 0, 1: 1}
entry: 2 {0: 0, 1: 1}
entry: 1 {0: 0, 1: 1}
entry: 0 {0: 0, 1: 1}
exit: 2 {0: 0, 1: 1, 2: 1}
entry: 1 {0: 0, 1: 1, 2: 1}
exit: 3 {0: 0, 1: 1, 2: 1, 3: 2}
entry: 2 {0: 0, 1: 1, 2: 1, 3: 2}
exit: 4 {0: 0, 1: 1, 2: 1, 3: 2, 4: 3}
entry: 3 {0: 0, 1: 1, 2: 1, 3: 2, 4: 3}
exit: 5 {0: 0, 1: 1, 2: 1, 3: 2, 4: 3, 5: 5}
entry: 4 {0: 0, 1: 1, 2: 1, 3: 2, 4: 3, 5: 5}
exit: 6 {0: 0, 1: 1, 2: 1, 3: 2, 4: 3, 5: 5, 6: 8}
entry: 5 {0: 0, 1: 1, 2: 1, 3: 2, 4: 3, 5: 5, 6: 8}
exit: 7 {0: 0, 1: 1, 2: 1, 3: 2, 4: 3, 5: 5, 6: 8, 7: 13}
entry: 6 {0: 0, 1: 1, 2: 1, 3: 2, 4: 3, 5: 5, 6: 8, 7: 13}
exit: 8 {0: 0, 1: 1, 2: 1, 3: 2, 4: 3, 5: 5, 6: 8, 7: 13, 8: 21}
entry: 7 {0: 0, 1: 1, 2: 1, 3: 2, 4: 3, 5: 5, 6: 8, 7: 13, 8: 21}
exit: 9 {0: 0, 1: 1, 2: 1, 3: 2, 4: 3, 5: 5, 6: 8, 7: 13, 8: 21, 9: 34}
entry: 8 {0: 0, 1: 1, 2: 1, 3: 2, 4: 3, 5: 5, 6: 8, 7: 13, 8: 21, 9: 34}
exit: 10 {0: 0, 1: 1, 2: 1, 3: 2, 4: 3, 5: 5, 6: 8, 7: 13, 8: 21, 9: 34, 10: 55}
55
```

known is a dictionary that keeps track of the Fibonacci numbers we already know. It starts with two items: 0 maps to 0 and 1 maps to 1.

Whenever fibonacci is called, it checks known. If the result is already there, it can return immediately. Otherwise it has to compute the new value, add it to the dictionary, and return it.

If you run this version of fibonacci and compare it with the original, you will find that it is much faster.

Global Variables

In the previous example, known is created outside the function, so it belongs to the special frame called __main__. Variables in __main__ are sometimes called **global** because they can be accessed from any function. Unlike local variables, which disappear when their function ends, global variables persist from one function call to the next.

It is common to use global variables for **flags**; that is, boolean variables that indicate (“flag”) whether a condition is true. For example, some programs use a flag named verbose to control the level of detail in the output:

```
verbose = True
def example1():
        if verbose:
        print('Running example1')

>>>example1()
Running example1
```

If you try to reassign a global variable, you might be surprised. The following example is supposed to keep track of whether the function has been called:

```
been_called = False

def example2():
been_called = True        # WRONG
```

But if you run it you will see that the value of been_called doesn’t change. The problem is that example2 creates a new local variable named been_called. The local variable goes away when the function ends, and has no effect on the global variable. To reassign a global variable inside a function you have to **declare** the global variable before you use it:

```
been_called = False

def example2():
        globalbeen_called
        been_called = True
```

The **global statement** tells the interpreter something like, “In this function, when I say been_called, I mean the global variable; don’t create a local one.”

Here’s an example that tries to update a global variable:

```
count = 0
def example3():
count = count + 1         # WRONG
```

If you run it you get:

```
>>>example3()
Traceback (most recent call last):
  File "<pyshell#83>", line 1, in <module>
```

```
example3()
  File "C:/Users/Admin/AppData/Local/Programs/Python/Python38/dictionary.py", line 253, in
example3
count = count + 1          # WRONG
UnboundLocalError: local variable 'count' referenced before assignment
```

Python assumes that count is local, and under that assumption you are reading it before writing it. The solution, again, is to declare count global:

```
def example3():
          global count
          count += 1

>>>example3()
Traceback (most recent call last):
  File "<pyshell#84>", line 1, in <module>
example3()
  File "C:/Users/Admin/AppData/Local/Programs/Python/Python38/dictionary.py", line 267, in
example3
count += 1
NameError: name 'count' is not defined
```

```
count=1
def example3():
global count
count += 1
print(count)
```

```
OUTPUT
>>>example3()
2
>>>example3()
```

If a global variable refers to a mutable value, you can modify the value without declaring the variable:

```
known = {0:0, 1:1}
def example4():
          known[2] = 1
          print(known)
```

```
OUPUT
>>>example4()
```

```
{0: 0, 1: 1, 2: 1}
```

So you can add, remove and replace elements of a global list or dictionary, but if you want to reassign the variable, you have to declare it:

```
def example5():
global known
known = dict()
```

Global variables can be useful, but if you have a lot of them, and you modify them frequently, they can make programs hard to debug.

Tuples

Tuples are immutable

A tuple is a sequence of values. The values can be any type, and they are indexed by integers, so in that respect tuples are a lot like lists. The important difference is that tuples are immutable. Syntactically, a tuple is a comma-separated list of values:

```
>>> t='a','b','c','d','e'
>>>t
('a', 'b', 'c', 'd', 'e')
>>>type(t)
<class 'tuple'>
```

Although it is not necessary, it is common to enclose tuples in parentheses:

```
>>> S=('1','B','4','D','5','F')
>>> S
('1', 'B', '4', 'D', '5', 'F')
>>>type(S)
<class 'tuple'>
```

To create a tuple with a single element, you have to include a final comma:

```
>>> r='a',
>>>type(r)
<class 'tuple'>
```

A value in parentheses is not a tuple:

```
>>> z=('lion')
```

```
>>>type(z)
<class 'str'>
```

Another way to create a tuple is the built-in function tuple. With no argument, it creates an empty tuple:

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

If the argument is a sequence (string, list or tuple), the result is a tuple with the elements of the sequence:

```
>>> y=tuple('lionking')
>>>type(y)
<class 'tuple'>
>>>y
('l', 'i', 'o', 'n', 'k', 'i', 'n', 'g')
```

Because tuple is the name of a built-in function, you should avoid using it as a variable name. Most list operators also work on tuples. The bracket operator indexes an element:

```
>>>y
('l', 'i', 'o', 'n', 'k', 'i', 'n', 'g')
>>>y[0]
'l'
>>>y[4]
'k'
```

And the slice operator selects a range of elements

```
>>>y[1:3]
('i', 'o')
```

But if you try to modify one of the elements of the tuple, you get an error:

```
>>>y[0]='k'
Traceback (most recent call last):
  File "<pyshell#24>", line 1, in <module>
y[0]='k'
TypeError: 'tuple' object does not support item assignment
```

Because tuples are immutable, you can't modify the elements. But you can replace one tuple with another:

```
>>> y=('k',)+y[1:]
>>>y
('k', 'i', 'o', 'n', 'k', 'i', 'n', 'g')
```

This statement makes a new tuple and then makes 'y' refer to it.

The relational operators work with tuples and other sequences; Python starts by comparing the first element from each sequence. If they are equal, it goes on to the next elements, and so on, until it finds elements that differ. Subsequent elements are not considered (even if they are really big).

```
>>> (0,1,2)<(0,3,4)
True
>>> (0,1,20000)<(0,3,4)
True
>>> ('a','c')<('a','z')
True
>>> ('a','z')<('a','a')
False
>>> ('a','b','z')<('a','b','c')
False
>>> (0.0,1.1,2.4)<(0.0,3.2,1.0)
True
>>> ([1],[2],[2000])<([0],[3],[4])
False
```

Tuple assignment

It is often useful to swap the values of two variables. With conventional assignments, you have to use a temporary variable. For example, to swap a and b:

```
>>> a=(1,2,3)
>>>type(a)
<class 'tuple'>
>>> b=(4,5,6)
>>>type(b)
<class 'tuple'>
>>>temp=a
>>> a=b
>>> b=temp
>>>a
(4, 5, 6)
```

```
>>>b
(1, 2, 3)
>>>temp
(1, 2, 3)
```

This solution is cumbersome; tuple assignment is more elegant:

```
>>>a
(4, 5, 6)
>>>b
(1, 2, 3)
>>>a,b=b,a
>>>a
(1, 2, 3)
>>>b
(4, 5, 6)
```

The left side is a tuple of variables; the right side is a tuple of expressions. Each value is assigned to its respective variable. All the expressions on the right side are evaluated before any of the assignments. The number of variables on the left and the number of values on the right have to be the same:

```
>>>a,b=1,2,3
Traceback (most recent call last):
  File "<pyshell#51>", line 1, in <module>
a,b=1,2,3
ValueError: too many values to unpack (expected 2)
```

More generally, the right side can be any kind of sequence (string, list or tuple). For example, to split an email address into a user name and a domain, you could write:

```
>>>email='badri2005@gmail.com'
>>>email
'badri2005@gmail.com'
>>>uname,domain=email.split('@')
>>>uname
'badri2005'
>>>domain
'gmail.com'
```

Tuples as Return Values

Strictly speaking, a function can only return one value, but if the value is a tuple, the effect is the same as returning multiple values. For example, if you want to divide two integers and compute the quotient and remainder, it is inefficient to compute x/y and then x%y. It is better to compute them both at the same time. The built-in function **divmod** takes two arguments and returns a tuple of two values, the quotient and remainder. You can store the result as a tuple

```
>>> t=divmod(25,3)
>>>t
(8, 1)
>>>type(t)
<class 'tuple'>
```

Or use tuple assignment to store the elements separately:

```
>>>quot,rem=divmod(25,3)
>>>quot
8
>>>rem
1
```

Here is an example of a function that returns a tuple:

```
>>> t=divmod(25,3)
>>>defmin_max():
        return min(t),max(t)

>>>t
(8, 1)
>>>min_max()
(1, 8)

>>> t=(1,2,45,89,56,-5,80)
>>>min_max()
(-5, 89)
```

max and min are built-in functions that find the largest and smallest elements of a sequence. min_max computes both and returns a tuple of two values.

Variable-Length Argument Tuples

Functions can take a variable number of arguments. A parameter name that begins with "*" **gathers** arguments into a tuple. For example, printall takes any number of arguments and prints them:

```
>>>def printall(*args):
        print(args)
```

The gather parameter can have any name you like, but args is conventional. Here’s how the function works:

```
>>>printall("lion",12,3.414)
('lion', 12, 3.414)
>>>type(printall)
<class 'function'>
```

The complement of gather is **scatter**. If you have a sequence of values and you want to pass it to a function as multiple arguments, you can use the * operator. For example, divmod takes exactly two arguments; it doesn’t work with a tuple:

```
>>> t=(7,3)
>>>divmod(t)
Traceback (most recent call last):
  File "<pyshell#76>", line 1, in <module>
divmod(t)
TypeError: divmod expected 2 arguments, got 1
>>>
```

But if you scatter the tuple, it works:

```
>>> t=(7,3)
>>>divmod(*t)
(2, 1)
```

Many of the built-in functions use variable-length argument tuples. For example, max and min can take any number of arguments

```
>>>max(-85,2,-3,-100)
2
>>>min(-85,2,-3,-100)
-100
```

But sum does not.

```
>>>sum(1,2,3)
Traceback (most recent call last):
  File "<pyshell#81>", line 1, in <module>
sum(1,2,3)
TypeError: sum() takes at most 2 arguments (3 given)
```

As an exercise, write a function called small that takes any number of arguments and returns their sum

sum(iterable, start)

iterable :iterable can be anything list , tuples or dictionaries , but most importantly it should be numbers.
start : this start is added to the sum of numbers in the iterable.
If start is not given in the syntax , it is assumed to be 0.

`sum(a)`

a is the list , it adds up all the numbers in the list a and takes start to be 0, so returning only the sum of the numbers in the list.

`sum(a, start)`

this returns the sum of the list + start

```
numbers = [1,2,3,4,5,1,4,5]

# start parameter is not provided
add = sum(numbers)
print(add)

# start = 10
add = sum(numbers, 10)
print(add)

output
25
35


>>>def sum_all(*args):
          return sum(args)

>>>sum_all(1,2,3)
6
>>>sum_all(1,2,3,4,5,6,7,8,9,10)
55
```

Lists and Tuples

zip is a built-in function that takes two or more sequences and returns a list of tuples where each tuple contains one element from each sequence. The name of the function refers to a zipper, which joins and interleaves two rows of teeth.
This example zips a string and a list:

```
>>> s='abc'
>>> t=[0,1,2]
>>>zip(s,t)
<zip object at 0x011D8568>
```

The result is a **zip object** that knows how to iterate through the pairs. The most common use of zip is in a for loop:

```
>>>for pair in zip(s,t):
        print (pair)


('a', 0)
('b', 1)
('c', 2)
```

A zip object is a kind of **iterator**, which is any object that iterates through a sequence. Iterators are similar to lists in some ways, but unlike lists, you can't use an index to select an element from an iterator.
If you want to use list operators and methods, you can use a zip object to make a list:

```
>>>list(zip(s,t))
[('a', 0), ('b', 1), ('c', 2)]
```

The result is a list of tuples; in this example, each tuple contains a character from the string and the corresponding element from the list.
If the sequences are not the same length, the result has the length of the shorter one.

```
>>>list(zip('lionking','mufasa'))
[('l', 'm'), ('i', 'u'), ('o', 'f'), ('n', 'a'), ('k', 's'), ('i', 'a')]
>>> list(zip('mufasa','lionking'))
[('m', 'l'), ('u', 'i'), ('f', 'o'), ('a', 'n'), ('s', 'k'), ('a', 'i')]
```

You can use tuple assignment in a for loop to traverse a list of tuples:

```
>>> s='lionking'
>>> t=[0,1,2,3,4,5]
>>>s
```

```
'lionking'
>>>t
[0, 1, 2, 3, 4, 5]
>>> type(s)
<class 'str'>
>>>type(t)
<class 'list'>
>>>for pair in zip(s,t):
          print (pair)


('a', 0)
('b', 1)
('c', 2)
>>> m=zip(s,t)
>>>m
<zip object at 0x011D8568>
```

Each time through the loop, Python selects the next tuple in the list and assigns the elements to letter and number. The output of this loop is:

```
>>>for letter, number in m:
          print(number,letter)


0 l
1 i
2 o
3 n
4 k
5 i
```

If you combine zip, for and tuple assignment, you get a useful idiom for traversing two (or more) sequences at the same time. For example, has_match takes two sequences, t1 and t2, and returns True if there is an index i such that t1[i] == t2[i]:

```
>>> x=(1,2,3,4,5,6)
>>>type(x)
<class 'tuple'>
>>> y=(5,6,7,8,9,10)
>>>type(y)
<class 'tuple'>
z=(1,6,3,9,4,10,6)
```

```
>>>type(z)
<class 'tuple'>

>>>def has_match(a,b):
        For i,j in zip(a,b):
                if i==j:
                        print (i)
                else:
                        print("No match")

>>>has_match(x,z)
1
No match
3
No match
No match
No match
>>>has_match(z,y)
No match
6
No match
No match
No match
10
```

If you need to traverse the elements of a sequence and their indices, you can use the built-in function enumerate:

```
>>>for index,element in enumerate('lionking'):
        print(index,element)
```

The result from enumerate is an enumerate object, which iterates a sequence of pairs; each pair contains an index (starting from 0) and an element from the given sequence. In this example, the output is

```
0 l
1 i
2 o
3 n
4 k
5 i
6 n
7 g
```

Dictionaries and Tuples

Dictionaries have a method called items that returns a sequence of tuples, where each tuple is a key-value pair.

```
>>>d={'a':200,'b':500,'c':100}
>>>type(d)
<class 'dict'>
>>> e=d.items()
>>>e
dict_items([('a', 200), ('b', 500), ('c', 100)])
>>> type(e)
<class 'dict_items'>
```

The result is a dict_items object, which is an iterator that iterates the key-value pairs. You can use it in a for loop like this:

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


a 200
b 500
c 100
```

As you should expect from a dictionary, the items are in no particular order. Going in the other direction, you can use a list of tuples to initialize a new dictionary:

```
>>> t=[('a', 200), ('b', 500), ('c', 100)]
>>>t
[('a', 200), ('b', 500), ('c', 100)]
>>>type(t)
<class 'list'>
>>> d=dict(t)
>>>d
{'a': 200, 'b': 500, 'c': 100}
>>>type(d)
<class 'dict'>
```

Combining dict with zip yields a concise way to create a dictionary:

```
>>> d=dict(zip('abc',range(3)))
>>>d
{'a': 0, 'b': 1, 'c': 2}
```

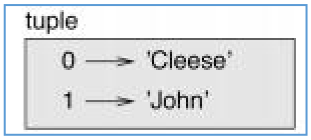
The dictionary method update also takes a list of tuples and adds them, as key-value pairs, to an existing dictionary. It is common to use tuples as keys in dictionaries (primarily because you can't use lists). For example, a telephone directory might map from last-name, first-name pairs to telephone numbers. Assuming that we have defined last, first and number, we could write:

```
directory[last, first] = number
```

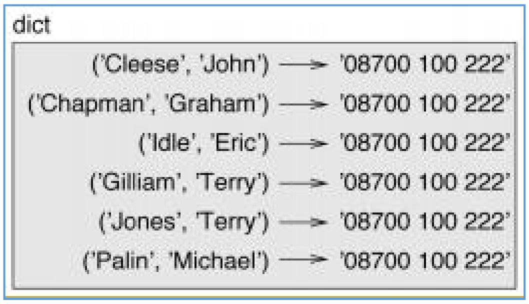
The expression in brackets is a tuple. We could use tuple assignment to traverse this dictionary:

```
for last, first in directory:
        print(first, last, directory[last,first])
```

This loop traverses the keys in directory, which are tuples. It assigns the elements of each tuple to last and first, then prints the name and corresponding telephone number. There are two ways to represent tuples in a state diagram. The more detailed version shows the indices and elements just as they appear in a list. For example, the tuple ('Cleese', 'John') would appear as in Figure



But in a larger diagram you might want to leave out the details. For example, a diagram of the telephone directory might appear as in Figure


dict
('Cleese', 'John') → '08700 100 222'
('Chapman', 'Graham') → '08700 100 222'
('Idle', 'Eric') → '08700 100 222'
('Gilliam', 'Terry') → '08700 100 222'
('Jones', 'Terry') → '08700 100 222'
('Palin', 'Michael') → '08700 100 222'

Here the tuples are shown using Python syntax as a graphical shorthand.

Sequences of Sequences

I have focused on lists of tuples, but almost all of the examples in this chapter also work with lists of lists, tuples of tuples, and tuples of lists. To avoid enumerating the possible combinations, it is sometimes easier to talk about sequences of sequences.

In many contexts, the different kinds of sequences (strings, lists and tuples) can be used interchangeably. So how should you choose one over the others?

To start with the obvious, strings are more limited than other sequences because the elements have to be characters. They are also immutable. If you need the ability to change the characters in a string (as opposed to creating a new string), you might want to use a list of characters instead.

Lists are more common than tuples, mostly because they are mutable. But there are a few cases where you might prefer tuples:

1. In some contexts, like a return statement, it is syntactically simpler to create a tuple than a list.
2. If you want to use a sequence as a dictionary key, you have to use an immutable type like a tuple or string.
3. If you are passing a sequence as an argument to a function, using tuples reduces the potential for unexpected behaviour due to aliasing.

Because tuples are immutable, they don’t provide methods like sort and reverse, which modify existing lists. But Python provides the built-in function sorted, which takes any sequence and returns a new list with the same elements in sorted order, and reversed, which takes a sequence and returns an iterator that traverses the list in reverse order.

Files

This chapter introduces the idea of “persistent” programs that keep data in permanent storage, and shows how to use different kinds of permanent storage, like files and databases.

Persistence

Most of the programs we have seen so far are transient in the sense that they run for a short time and produce some output, but when they end, their data disappears. If you run the program again, it starts with a clean slate.

Other programs are persistent: they run for a long time (or all the time); they keep at least some of their data in permanent storage (a hard drive, for example); and if they shut down and restart, they pick up where they left off. Examples of persistent programs are operating systems, which run pretty much whenever a computer is on, and web servers, which run all the time, waiting for requests to come in on the network.

One of the simplest ways for programs to maintain their data is by reading and writing text files. We have already seen programs that read text files; in this chapter we will see programs that write them.

An alternative is to store the state of the program in a database. In this chapter I will present a simple database and a module, pickle, that makes it easy to store program data.

Reading and Writing

A text file is a sequence of characters stored on a permanent medium like a hard drive, flash memory, or CD-ROM.

To write a file, you have to open it with mode 'w' as a second parameter:

```
>>>fout=open('words.txt','w')
```

If the file already exists, opening it in write mode clears out the old data and starts fresh, so be careful! If the file doesn't exist, a new one is created. open returns a file object that provides methods for working with the file. The write method puts data into the file:

```
>>> line1="This is Badrinath,\n"
>>>fout.write(line1)
19
```

The return value is the number of characters that were written. The file object keeps track of where it is, so if you call write again, it adds the new data to the end of the file:

```
>>> line2="This is Simba son of Mufasa"
>>>fout.write(line2)
27
```

When you are done writing, you should close the file:

```
>>>fout.close()
```

If you don't close the file, it gets closed for you when the program ends.

Format Operator

The argument of write has to be a string, so if we want to put other values in a file, we have to convert them to strings. The easiest way to do that is with str:

```
>>> x=52
>>>fout.write(str(x))
Traceback (most recent call last):
  File "<pyshell#16>", line 1, in <module>
fout.write(str(x))
ValueError: I/O operation on closed file.
```

An alternative is to use the **format operator, %**. When applied to integers, % is the modulus operator. But when the first operand is a string, % is the format operator.

The first operand is the **format string**, which contains one or more **format sequences**, which specify how the second operand is formatted. The result is a string. For example, the format sequence '%d' means that the second operand should be for- matted as a decimal integer:

```
>>>camels=42
>>> '%d' %camels
'42'
```

The result is the string '42', which is not to be confused with the integer value 42.

A format sequence can appear anywhere in the string, so you can embed a value in a sentence:

```
>>> 'i have spotted %d camels.' % camels
'i have spotted 42 camels.'
```

If there is more than one format sequence in the string, the second argument has to be a tuple. Each format sequence is matched with an element of the tuple, in order.
The following example uses '%d' to format an integer, '%g' to format a floating-point number, and '%s' to format a string:

```
>>> 'In %d years I have spotted %g %s .' % (3,0.1,'camels')
'In 3 years I have spotted 0.1 camels .'
```

The number of elements in the tuple has to match the number of format sequences in the string. Also, the types of the elements have to match the format sequences:

```
>>> '%d %d %d' %(1,2)
Traceback (most recent call last):
  File "<pyshell#20>", line 1, in <module>
    '%d %d %d' %(1,2)
TypeError: not enough arguments for format string
>>> '%d' % 'd
```

In the first example, there aren’t enough elements; in the second, the element is the wrong type.

Filenames and Paths

Files are organized into **directories** (also called “folders”). Every running program has a “current directory”, which is the default directory for most operations. For example, when you open a file for reading, Python looks for it in the current directory.
The os module provides functions for working with files and directories (“os” stands for “operating system”). os.getcwd returns the name of the current directory:

```
>>> import os
>>>cwd=os.getcwd()
>>>cwd
'C:\\Users\\admin\\AppData\\Local\\Programs\\Python\\Python38-32'
```

cwd stands for “current working directory”. The result in this example is /home/dins dale, which is the home directory of a user named dinsdale.
A string like '/home/dinsdale' that identifies a file or directory is called a path. A simple filename, like memo.txt, is also considered a path, but it is a relative path because it relates to the current directory. If the current directory is /home/dinsdale, the filename memo.txt would refer to

/home/dinsdale/memo.txt. A path that begins with / does not depend on the current directory; it is called an **absolute path.** To find the absolute path to a file, you can use os.path.abspath:

```
>>>os.path.abspath('words.txt')
'C:\\Users\\admin\\AppData\\Local\\Programs\\Python\\Python38-32\\words.txt'
```

os.path provides other functions for working with filenames and paths. For example, os.path.exists checks whether a file or directory exists:

```
>>>os.path.exists('words.txt')
True
```

If it exists, os.path.isdir checks whether it's a directory:

```
>>>os.path.exists('words.txt')
True
>>>os.path.exists('words123.txt')
False
>>>os.path.isdir('C:\\Users\\admin\\AppData\\Local\\Programs\\Python\\Python38-32\\')
True
```

Similarly, os.path.isfile checks whether it's a file.
os.listdir returns a list of the files (and other directories) in the given directory:

```
>>>os.listdir(cwd)
['06-10-2020.txt', '1.py', '2.py', 'api-ms-win-core-console-l1-1-0.dll', 'api-ms-win-core-datetime-
l1-1-0.dll', 'api-ms-win-core-debug-l1-1-0.dll', 'api-ms-win-core-errorhandling-l1-1-0.dll', 'api-
ms-win-core-file-l1-1-0.dll', 'api-ms-win-core-file-l1-2-0.dll', 'api-ms-win-core-file-l2-1-0.dll',
'api-ms-win-core-handle-l1-1-0.dll', 'api-ms-win-core-heap-l1-1-0.dll', 'api-ms-win-core-
interlocked-l1-1-0.dll', 'api-ms-win-core-libraryloader-l1-1-0.dll', 'api-ms-win-core-localization-
l1-2-0.dll', 'api-ms-win-core-memory-l1-1-0.dll', 'api-ms-win-core-namedpipe-l1-1-0.dll', 'api-
ms-win-core-processenvironment-l1-1-0.dll', 'api-ms-win-core-processthreads-l1-1-0.dll', 'api-
ms-win-core-processthreads-l1-1-1.dll', 'api-ms-win-core-profile-l1-1-0.dll', 'api-ms-win-core-
rtlsupport-l1-1-0.dll', 'api-ms-win-core-string-l1-1-0.dll', 'api-ms-win-core-synch-l1-1-0.dll', 'api-
ms-win-core-synch-l1-2-0.dll', 'api-ms-win-core-sysinfo-l1-1-0.dll', 'api-ms-win-core-timezone-
l1-1-0.dll', 'api-ms-win-core-util-l1-1-0.dll', 'api-ms-win-crt-conio-l1-1-0.dll', 'api-ms-win-crt-
convert-l1-1-0.dll', 'api-ms-win-crt-environment-l1-1-0.dll', 'api-ms-win-crt-filesystem-l1-1-
0.dll', 'api-ms-win-crt-heap-l1-1-0.dll', 'api-ms-win-crt-locale-l1-1-0.dll', 'api-ms-win-crt-math-
l1-1-0.dll', 'api-ms-win-crt-multibyte-l1-1-0.dll', 'api-ms-win-crt-private-l1-1-0.dll', 'api-ms-win-
crt-process-l1-1-0.dll', 'api-ms-win-crt-runtime-l1-1-0.dll', 'api-ms-win-crt-stdio-l1-1-0.dll', 'api-
ms-win-crt-string-l1-1-0.dll', 'api-ms-win-crt-time-l1-1-0.dll', 'api-ms-win-crt-utility-l1-1-0.dll',
'concrt140.dll', 'DLLs', 'Doc', 'IDAN.py', 'include', 'Lib', 'libs', 'LICENSE.txt', 'msvcp140.dll',
'msvcp140_1.dll', 'msvcp140_2.dll', 'msvcp140_atomic_wait.dll', 'msvcp140_codecvt_ids.dll',
'NEWS.txt', 'python.exe', 'python3.dll', 'python38.dll', 'pythonw.exe', 'replace.py', 'Scripts',
'search.py', 'Strings.docx', 'strings.py', 't.py', 'tcl', 'te.py', 'test.txt', 'test1.py', 'test2.py', 'test3.py',
'teste.txt', 'Tools', 'ucrtbase.dll', 'vcamp140.dll', 'vccorlib140.dll', 'vcomp140.dll',
'vcruntime140.dll', 'w.txt', 'word.txt', 'words.txt', 'words.xlsx', 'words1.txt', 'words6.txt', 'wordstxt',
'Z.py', 'ZFG.py']
```

To demonstrate these functions, the following example "walks" through a directory, prints the names of all the files, and calls itself recursively on all the directories:

```
>>>def walk(dirname):
        for name in os.listdir(dirname):
                path=os.path.join(dirname,name)
                ifos.path.isfile(path):
                        print(path)
                else:
                        walk(path)


>>>walk(cwd)

C:\Users\admin\AppData\Local\Programs\Python\Python38-32\06-10-2020.txt
C:\Users\admin\AppData\Local\Programs\Python\Python38-32\1.py
C:\Users\admin\AppData\Local\Programs\Python\Python38-32\2.py
.
.
.
```

os.path.join takes a directory and a filename and joins them into a complete path. The os module provides a function called walk that is similar to this one but more versatile.

Catching Exceptions

A lot of things can go wrong when you try to read and write files. If you try to open a file that doesn't exist, you get an IOError:

```
>>>fin=open('bad_file')
Traceback (most recent call last):
  File "<pyshell#50>", line 1, in <module>
fin=open('bad_file')
FileNotFoundError: [Errno 2] No such file or directory: 'bad_file'
```

If you don't have permission to access a file:

```
>>>fout = open('/etc/passwd', 'w')
```

```
PermissionError: [Errno 13] Permission denied: '/etc/passwd'
```

And if you try to open a directory for reading, you get

```
>>>fin = open('/home')
IsADirectoryError: [Errno 21] Is a directory: '/home'
```

To avoid these errors, you could use functions like os.path.exists and os.path.isfile, but it would take a lot of time and code to check all the possibilities (if “Errno 21” is any indication, there are at least 21 things that can go wrong).
It is better to go ahead and try—and deal with problems if they happen—which is exactly what the try statement does. The syntax is similar to an if...else statement:

```
try:
fin = open('bad_file')
except:
print('Something went wrong.')
```

Python starts by executing the try clause. If all goes well, it skips the except clause and proceeds. If an exception occurs, it jumps out of the try clause and runs the except clause.
Handling an exception with a try statement is called **catching** an exception. In this example, the except clause prints an error message that is not very helpful. In gen-eral, catching an exception gives you a chance to fix the problem, or try again, or at least end the program gracefully

Databases

A database is a file that is organized for storing data. Many databases are organized like a dictionary in the sense that they map from keys to values. The biggest difference between a database and a dictionary is that the database is on disk (or other permanent storage), so it persists after the program ends.
The module dbm provides an interface for creating and updating database files. As an example, I’ll create a database that contains captions for image files.
Opening a database is similar to opening other files:

```
>>> import dbm
>>>db=dbm.open('captions','c')
>>>type(db)
<class 'dbm.dumb._Database'>
```

The mode 'c' means that the database should be created if it doesn’t already exist. The result is a database object that can be used (for most operations) like a dictionary. When you create a new item, dbm updates the database file:

```
>>>db['cleese.png'] = 'Photo of John Cleese.'
```

When you access one of the items, dbm reads the file:

```
>>>db['cleese.png']
b'Photo of John Cleese.'
```

The result is a **bytes object**, which is why it begins with b. A bytes object is similar to a string in many ways. When you get farther into Python, the difference becomes important, but for now we can ignore it. If you make another assignment to an existing key, dbm replaces the old value:

```
>>>db['cleese.png'] = 'Photo of John Cleese doing a silly walk.'
>>>db['cleese.png']
b'Photo of John Cleese doing a silly walk.'
```

Some dictionary methods, like keys and items, don't work with database objects. But iteration with a for loop works:

```
>>>for key in db:
        print(key,db[key])


b'cleese.png'b'Photo of John Cleese doing a silly walk.'
```

As with other files, you should close the database when you are done:

```
>>>db.close()
```

Pickling

A limitation of dbm is that the keys and values have to be strings or bytes. If you try to use any other type, you get an error.

The pickle module can help. It translates almost any type of object into a string suitable for storage in a database, and then translates strings back into objects.

pickle.dumps takes an object as a parameter and returns a string representation (dumps is short for "dump string"):

```
>>> import pickle
>>> t=[1,2,3]
>>>pickle.dumps(t)
b'\x80\x04\x95\x0b\x00\x00\x00\x00\x00\x00\x00]\x94(K\x01K\x02K\x03e.'
```

The format isn't obvious to human readers; it is meant to be easy for pickle to interpret. pickle.loads ("load string") reconstitutes the object:

```
>>> t1=[5,6,7]
>>> s=pickle.dumps(t1)
>>> t2=pickle.loads(s)
>>> t2
[5, 6, 7]
```

Although the new object has the same value as the old, it is not (in general) the same object:

```
>>> t1=[5,6,7]
>>> s=pickle.dumps(t1)
>>> t2=pickle.loads(s)
>>> t2
[5, 6, 7]
>>>id(t1)
55156424
>>>id(t2)
55156488
>>> t1
[5, 6, 7]
>>> t2
[5, 6, 7]
>>> t1==t2
True
>>>t1 is t2
False
```

In other words, pickling and then unpickling has the same effect as copying the object. You can use pickle to store non-strings in a database. In fact, this combination is so common that it has been encapsulated in a module called **shelve**.

Pipes

Most operating systems provide a command-line interface, also known as a shell. Shells usually provide commands to navigate the file system and launch applications. For example, in Unix you can change directories with cd, display the contents of a directory with ls, and launch a web browser by typing (for example) firefox.

Any program that you can launch from the shell can also be launched from Python using a **pipe object**, which represents a running program.

For example, the Unix command ls -l normally displays the contents of the current directory in long format. You can launch ls with os.popen :

```
>>> import os
```

```
>>>cmd='ls-l'
>>>fp=os.popen(cmd)
>>> fp
<os._wrap_close object at 0x02ED7DF0>
```

The argument is a string that contains a shell command. The return value is an object that behaves like an open file. You can read the output from the ls process one line at a time with readline or get the whole thing at once with read:

```
>>> import os
>>>cmd='ls-l'
>>>fp=os.popen(cmd)
>>>res=fp.read()
```

When you are done, you close the pipe like a file:

```
>>> import os
>>>cmd='ls-l'
>>>fp=os.popen(cmd)
>>>res=fp.read()
>>>stat=fp.close()
>>>print(stat)
1
```

The return value is the final status of the ls process; None means that it ended normally (with no errors). For example, most Unix systems provide a command called md5sum that reads the contents of a file and computes a “checksum”. This command provides an efficient way to check whether two files have the same contents. The probability that different contents yield the same checksum is very small (that is, unlikely to happen before the universe collapses).

You can use a pipe to run md5sum from Python and get the result:

```
>>> import os
>>>filename='1.py'
>>>cmd='md5sum'+filename
>>>fp=os.popen(cmd)
>>>res=fp.read()
>>>stat=fp.close()
>>>print(res)

>>>print(stat)
1
```

Writing Modules

Any file that contains Python code can be imported as a module. For example, suppose you have a file named wc.py with the following code:

```
def linecount(filename):
        count=0
        for line in open(filename):
                count+=1
        return count

print(linecount('1.py'))

>>>
= RESTART: C:/Users/admin/AppData/Local/Programs/Python/Python38-32/wc.py
7
```

If you run this program, it reads itself and prints the number of lines in the file, which is 7. You can also import it like this:

```
>>> import wc
7
```

Now you have a module object wc:

```
>>>wc
<module 'wc' from 'C:/Users/admin/AppData/Local/Programs/Python/Python38-32\\wc.py'>
```

The module object provides linecount:

```
>>>wc.linecount('wc.py')
7
```

So that’s how you write modules in Python.
The only problem with this example is that when you import the module it runs the test code at the bottom. Normally when you import a module, it defines new functions but it doesn’t run them. Programs that will be imported as modules often use the following idiom:

```
if __name__ == '__main__':
print(linecount('wc.py'))
```

__name__ is a built-in variable that is set when the program starts. If the program is running as a script, __name__ has the value '__main__': in that case, the test code runs. Otherwise, if the module is being imported, the test code is skipped.

As an exercise, type this example into a file named wc.py and run it as a script. Then run the Python interpreter and import wc. What is the value of __name__ when the module is being imported? Warning: If you import a module that has already been imported, Python does nothing. It does not re-read the file, even if it has changed.

If you want to reload a module, you can use the built-in function reload, but it can be tricky, so the safest thing to do is restart the interpreter and then import the module again.

Debugging

When you are reading and writing files, you might run into problems with white space. These errors can be hard to debug because spaces, tabs and newlines are normally invisible:

```
>>> k='1 2\t 3\n 4'
>>> k
'1 2\t 3\n 4'
>>>print(k)
1 2	3
 4
```

The built-in function repr can help. It takes any object as an argument and returns a string representation of the object. For strings, it represents whitespace characters with backslash sequences:

```
>>>print(repr(k))
'1 2\t 3\n 4'
```

This can be helpful for debugging.

One other problem you might run into is that different systems use different characters to indicate the end of a line. Some systems use a newline, represented \n. Others use a return character, represented \r. Some use both. If you move files between different systems, these inconsistencies can cause problems. For most systems, there are applications to convert from one format to another.

Classes and Objects

Programmer-Defined Types

We have used many of Python's built-in types; now we are going to define a new type. As an example, we will create a type called **Point** that represents a point in two dimensional space. In mathematical notation, points are often written in parentheses with a comma separating the coordinates. For example, (0,0) represents the origin, and (x,y) represents the point x units to the right and y units up from the origin.

There are several ways we might represent points in Python:

- We could store the coordinates separately in two variables, x and y.
- We could store the coordinates as elements in a list or tuple.
- We could create a new type to represent points as objects.

Creating a new type is more complicated than the other options, but it has advantages that will be apparent soon.

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

```
>>>class Point:
        """Represents a point in 2-D space"""
```

The header indicates that the new class is called Point. The body is a docstring that explains what the class is for. You can define variables and methods inside a class definition, but we will get back to that later.

Defining a class named Point creates a **class object**

```
>>> Point
<class '__main__.Point'>
```

Because Point is defined at the top level, its “full name” is __main__.Point. The class object is like a factory for creating objects. To create a Point, you call Point as if it were a function:

```
>>>blank =Point()
>>>blank
<__main__.Point object at 0x03185D48>
>>> cyan=Point()
>>> cyan
<__main__.Point object at 0x02D770B8>
```

The return value is a reference to a Point object, which we assign to blank.

Creating a new object is called **instantiation**, and the object is an **instance** of the class.

When you print an instance, Python tells you what class it belongs to and where it is stored in memory (the prefix 0x means that the following number is in hexadecimal).

```
>>> a=Jam()
>>> a
<__main__.Jam object at 0x02D77478>
>>> b=Jam()
>>> b
<__main__.Jam object at 0x02D77700>
```

Every object is an instance of some class, so "object" and "instance" are interchangeable. But in this chapter I use "instance" to indicate that I am talking about a programmer-defined type.

Attributes

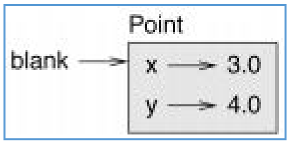
You can assign values to an instance using dot notation:

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

This syntax is similar to the syntax for selecting a variable from a module, such as math.pi or string.whitespace. In this case, though, we are assigning values to named elements of an object. These elements are called **attributes**.

As a noun, "AT-trib-ute" is pronounced with emphasis on the first syllable, as opposed to "a-TRIB-ute", which is a verb.

The following diagram shows the result of these assignments. A state diagram that shows an object and its attributes is called an **object diagram**



The variable blank refers to a Point object, which contains two attributes. Each attribute refers to a floating-point number. You can read the value of an attribute using the same syntax:

```
>>>blank.x=3.4
>>>blank.y=4.5
>>>blank.x
3.4
>>>blank.y
4.5
>>> x=blank.x
>>>x
3.4
```

The expression blank.x means, "Go to the object blank refers to and get the value of x." In the example, we assign that value to a variable named x. There is no conflict between the variable x and the attribute x.

You can use dot notation as part of any expression. For example:

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

```
>>>blank.x
3.0
>>>blank.y
4.0
>>> x=blank.x
>>>x
3.0
>>> '(%g,%g)' % (blank.x,blank.y)
'(3,4)'
>>>distance=math.sqrt(blank.x**2+blank.y**2)
>>>distance
5.0
```

You can pass an instance as an argument in the usual way. For example:

```
def print_point(p):
        print('(%g,%g)'%(p.x,p.y))
```

print_point takes a point as an argument and displays it in mathematical notation. To invoke it, you can pass blank as an argument:

```
>>>print_point(blank)
(3,4)
```

Inside the function, p is an alias for blank, so if the function modifies p, blank changes. As an exercise, write a function called distance_between_points that takes two Points as arguments and returns the distance between them.

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.
"""
```

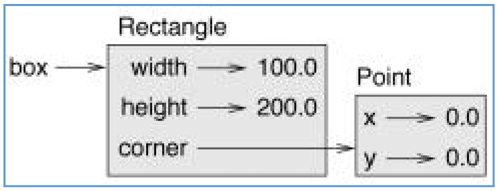
The docstring lists the attributes: width and height are numbers; corner is a Point object that specifies the lower-left corner.

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
```

The expression box.corner.x means, “Go to the object box refers to and select the attribute named corner; then go to that object and select the attribute named x.”

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.
As an exercise, write a function named move_rectangle that takes a Rectangle and two numbers named dx and dy. It should change the location of the rectangle by adding dx to the x coordinate of corner and adding dy to the y coordinate of corner.

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:

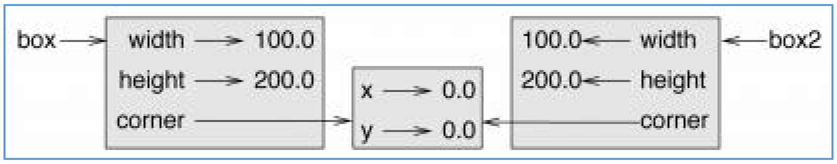
```
def print_point(p):
        print('(%g,%g)'%(p.x,p.y))

>>>print_point(p1)
(3,4)
>>>print_point(p2)
(3,4)
>>>p1 is p2
False
>>> p1==p2
False
>>>id(p1)
59446104
>>>id(p2)
59446152
>>> p1
<__main__.Point object at 0x02D77448>
>>> p2
<__main__.Point object at 0x02D77760>
```

The is operator indicates that p1 and p2 are not the same object, which is what we expected. But you might have expected == to yield True because these points contain the same data. In that case, you will be disappointed to learn that for instances, the default behavior of the == operator is the same as the is operator; it checks object identity, not object equivalence. That's because for programmer-defined types, Python doesn't know what should be considered equivalent. At least, not yet.

If you use copy.copy to duplicate a Rectangle, you will find that it copies the Rectangle object but not the embedded Point:

```
>>> box2=copy.copy(box)
>>>box2 is box
False
>>>box2.corner is box.corner
True
>>>id(box2)
59636608
>>>id(box)
59446056
>>>box
<__main__.Rectangle object at 0x038B1328>
>>> box2
<__main__.Rectangle object at 0x038DFB80>
```

This operation is called a **shallow copy** because it copies the object and any references it contains, but not the embedded objects.



For most applications, this is not what you want. In this example, invoking grow_rectangle on one of the Rectangles would not affect the other, but invoking move_rectangle on either would affect both! This behaviour is confusing and error-prone.
Fortunately, 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. You will not be surprised to learn that this operation is called a **deep copy.**

```
>>> box3=copy.deepcopy(box)
>>> box3
<__main__.Rectangle object at 0x038DFE80>
>>>id(box3)
59637376
>>>box
<__main__.Rectangle object at 0x038B1328>
>>>id(box)
59446056
>>>box3 is box
False
>>>box3.corner is box.corner
False
```

box3 and box are completely separate objects.
As an exercise, write a version of move_rectangle that creates and returns a new Rectangle instead of modifying the old one.

Debugging

When you start working with objects, you are likely to encounter some new exceptions. If you try to access an attribute that doesn't exist, you get an AttributeError:

```
>>> p=Point()
>>>p.x=3
>>>p.y=4
>>>p.z
Traceback (most recent call last):
  File "<pyshell#82>", line 1, in <module>
p.z
AttributeError: 'Point' object has no attribute 'z'
```

If you are not sure what type an object is, you can ask:

```
>>>type(p)
<class '__main__.Point'>
```

You can also use isinstance to check whether an object is an instance of a class:

```
>>>isinstance(p,Point)
True
```

If you are not sure whether an object has a particular attribute, you can use the builtin function hasattr:

```
>>>hasattr(p,'x')
True
>>>hasattr(p,'z')
False
```

The first argument can be any object; the second argument is a string that contains the name of the attribute.

You can also use a try statement to see if the object has the attributes you need:

```
>>> try:
                        x=p.x
exceptAttributeError:
                        x=0
```

This approach can make it easier to write functions that work with different types; more on that topic is coming up in “Polymorphism”

Regulation: AK20	**Subject Code:CSE/CIC** 20APS0526/20APC3605	**Subject Name :** Basics of Python Programming	**AY:** 2023-2024
UNIT-5 (Classes and Functions, Methods, Inheritance, The Goodies)			

UNIT – V

Classes and Functions: Time, Pure functions, Modifiers, Prototyping versus Planning

Classes and Methods: Object oriented features, Printing objects, 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.

The Goodies: Conditional expressions, List comprehensions, Generator expressions, any and all, Sets, Counters, defaultdict, Named tuples, Gathering keyword Args

Classes and Functions

Time

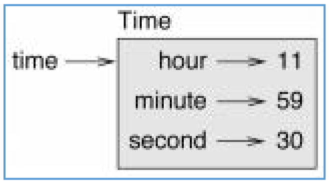
As another example of a programmer-defined type, we’ll define a class called Time that records the time of day. The class definition looks like this:

```
>>> 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
```

As an exercise, write a function called print_time that takes a Time object and prints it in the form hour:minute:second. Hint: the format sequence '%.2d' prints an integer using at least two digits, including a leading zero if necessary.
Write a boolean function called is_after that takes two Time objects, t1 and t2, and returns True if t1 follows t2 chronologically and False otherwise. Challenge: don’t use an if statement.



Pure Functions

In the next few sections, we'll write two functions that add time values. They demonstrate two kinds of functions: pure functions and modifiers. They also demonstrate a development plan I'll call **prototype and patch**, which is a way of tackling a complex problem by starting with a simple prototype and incrementally dealing with the complications.
Here is a simple prototype of add_time:

```
>>> def add_time(t1, t2):
 sum = Time()
 sum.hour = t1.hour + t2.hour
 sum.minute = t1.minute + t2.minute
 sum.second = t1.second + t2.second
 return sum
```

The function creates a new Time object, initializes its attributes, and returns a reference to the new object. This is called a **pure function** because it does not modify any of the objects passed to it as arguments and it has no effect, like displaying a value or getting user input, other than returning a value. To test this function, I'll create two Time objects: **start** contains the start time of a movie, like Monty Python and the Holy Grail, and **duration** contains the runtime of the movie, which is 1 hour 35 minutes. add_time figures out when the movie will be done:

```
>>> start=Time()
>>> start.hour=9
>>> start.minute=45
>>> start.second=0
>>> duration=Time()
>>> duration.hour=1
>>> duration.minute=35
>>> duration.second=0
>>> done=add_time(start,duration)
>>>def print_time(time):
 print('%.2d:%.2d:%.2d' % (time.hour, time.minute, time.second))
>>> print_time(done)
10:80:00
```

The result, 10:80:00, might not be what you were hoping for. The problem is that this function does not deal with cases where the number of seconds or minutes adds up to more than sixty. When that happens, we have to "carry" the extra seconds into the minute column or the extra minutes into the hour column.

Here's an improved version:

```
def add_time(t1, t2):
    s = Time()
    s.hour = t1.hour + t2.hour
    s.minute = t1.minute + t2.minute
    s.second = t1.second + t2.second
    if s.second >= 60:
        s.second -= 60
        s.minute += 1
        if s.minute >= 60:
            s.minute -= 60
            s.hour += 1
            return s
```

Although this function is correct, it is starting to get big. We will see a shorter alternative later.

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
```

The first line performs the basic operation; the remainder deals with the special cases we saw before.
Is this function correct? What happens if seconds is much greater than 60?
In that case, it is not enough to carry once; we have to keep doing it until time.second is less than 60. One solution is to replace the if statements with while statements. That would make the function correct, but not very efficient. As an exercise, write a correct version of increment that doesn’t contain any loops.
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.

In general, I recommend that you write pure functions whenever it is reasonable and resort to modifiers only if there is a compelling advantage. This approach might be called a **functional programming style**. As an exercise, write a "pure" version of increment that creates and returns a new Time object rather than modifying the parameter.

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).

An alternative is **designed development**, in which high-level insight into the problem can make the programming much easier. In this case, the insight is that a Time object is really a three-digit number in base 60. The second attribute is the "ones column", the minute attribute is the "sixties column", and the hour attribute is the "thirty-six hundreds column". When we wrote add_time and increment, we were effectively doing addition in base 60, which is why we had to carry from one column to the next.

This observation suggests another approach to the whole problem—we can convert Time objects to integers and take advantage of the fact that the computer knows how to do integer arithmetic.

Here is a function that converts Times to integers:

```
>>> def time_to_int(time):
        minutes=time.hour*60+time.minute
        seconds=minutes*60+time.second
        return seconds
```

And here is a function that converts an integer to a Time (recall that divmod divides the first argument by the second and returns the quotient and remainder as a tuple):

```
>>> def int_to_time(seconds):
        time=Time()
        minutes,time.second=divmod(seconds,60)
        time.hour,time.minute=divmod(minutes,60)
        return time
```

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)
```

This version is shorter than the original, and easier to verify. As an exercise, rewrite increment using time_to_int and int_to_time.

In some ways, converting from base 60 to base 10 and back is harder than just dealing with times. Base conversion is more abstract; our intuition for dealing with time values is better.

But if we have the insight to treat times as base 60 numbers and make the investment of writing the conversion functions (time_to_int and int_to_time), we get a program that is shorter, easier to read and debug, and more reliable.

It is also easier to add features later. For example, imagine subtracting two Times to find the duration between them. The naive approach would be to implement subtraction with borrowing. Using the conversion functions would be easier and more likely to be correct.

Ironically, sometimes making a problem harder (or more general) makes it easier (because there are fewer special cases and fewer opportunities for error).

Debugging

A Time object is well-formed if the values of minute and second are between 0 and 60 (including 0 but not 60) and if hour is positive. hour and minute should be integral values, but we might allow second to have a fraction part.

Requirements like these are called **invariants** because they should always be true. To put it a different way, if they are not true, something has gone wrong.

Writing code to check invariants can help detect errors and find their causes. For example, you might have a function like valid_time that takes a Time object and returns False if it violates an invariant:

```
def valid_time(time):
    if time.hour < 0 or time.minute < 0 or time.second < 0:
        return False
    if time.minute >= 60 or time.second >= 60:
        return False
    return True
```

At the beginning of each function you could check the arguments to make sure they are valid:

```
def add_time(t1, t2):
    if not valid_time(t1) or not valid_time(t2):
        raise ValueError('invalid Time object in add_time')
    seconds = time_to_int(t1) + time_to_int(t2)
    return int_to_time(seconds)
```

Or you could use an assert statement, which checks a given invariant and raises an exception if it fails:

```
def add_time(t1, t2):
    assert valid_time(t1) and valid_time(t2)
    seconds = time_to_int(t1) + time_to_int(t2)
    return int_to_time(seconds)
```

assert statements are useful because they distinguish code that deals with normal conditions from code that checks for errors.

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.

For example, the Time class corresponds to the way people record the time of day, and the functions we defined correspond to the kinds of things people do with times. Similarly, the Point and Rectangle classes correspond to the mathematical concepts of a point and a rectangle.

So far, we have not taken advantage of the features Python provides to support object-oriented programming. These features are not strictly necessary; most of them provide alternative syntax for things we have already done. But in many cases, the alternative is more concise and more accurately conveys the structure of the program.

For example, in Time1.py there is no obvious connection between the class definition and the function definitions that follow. With some examination, it is apparent that every function takes at least one Time object as an argument.

This observation is the motivation for **methods**; a method is a function that is associated with a particular class. We have seen methods for strings, lists, dictionaries and tuples. In this chapter, we will define methods for programmer-defined types.

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.

In the next few sections, we will take the functions from the previous two chapters and transform them into methods. This transformation is purely mechanical; you can do it by following a sequence of steps. If you are comfortable converting from one form to another, you will be able to choose the best form for whatever you are doing

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 = 3
start.minute = 15
start.second = 00
print_time(start)
```

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)
03:15:00
```

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()
03:15:00
```

In this use of dot notation, print_time is the name of the method (again), and start is the object the method is invoked on, which is called the **subject**. Just as the subject of a sentence is what the sentence is about, the subject of a method invocation is what the method is about.

Inside the method, the subject is assigned to the first parameter, so in this case start is assigned to time. By convention, the first parameter of a method is called self, so it would be more common to write print_time like this:

```
class Time:
        def print_time(self):
                print('%.2d:%.2d:%.2d' % (self.hour, self.minute, self.second))
```

The reason for this convention is an implicit metaphor:

- The syntax for a function call, print_time(start), suggests that the function is the active agent. It says something like, “Hey print_time! Here’s an object for you to print.”
- In object-oriented programming, the objects are the active agents. A method invocation like start.print_time() says “Hey start! Please print yourself.”

This change in perspective might be more polite, but it is not obvious that it is useful. In the examples we have seen so far, it may not be. But sometimes shifting responsibility from the functions onto the objects makes it possible to write more versatile functions (or methods), and makes it easier to maintain and reuse code.

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. This mechanism can be confusing, especially if you make an error. For example, if you invoke increment with two arguments, you get:

```
>>> end = start.increment(1337, 460)
```

```
TypeError: increment() takes 2 positional arguments but 3 were given
```

The error message is initially confusing, because there are only two arguments in parentheses. But the subject is also considered an argument, so all together that's three. By the way, a **positional argument** is an argument that doesn't have a parameter name; that is, it is not a keyword argument. In this 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 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
```

When I write a new class, I almost always start by writing __init__, which makes it easier to instantiate objects, and __str__, which is useful for debugging.

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__. So there is a lot happening behind the scenes! Changing the behavior of an operator so that it works with programmer-defined types is called **operator overloading**. For every operator in Python there is a corresponding special method, like __add__.

Type-Based Dispatch

In the previous section we added two Time objects, but you also might want to add an integer to a Time object. The following is a version of __add__ that checks the type of other and invokes either add_time or increment:

```
# inside class Time:
def __add__(self, other):
        if isinstance(other, Time):
                return self.add_time(other)
        else:
                return self.increment(other)

def add_time(self, other):
        seconds = self.time_to_int() + other.time_to_int()
        return int_to_time(seconds)

def increment(self, seconds):
        seconds += self.time_to_int()
        return int_to_time(seconds)
```

The built-in function isinstance takes a value and a class object, and returns True if the value is an instance of the class. If other is a Time object, __add__ invokes add_time. Otherwise it assumes that the parameter is a number and invokes increment. This operation is called a **type-based dispatch** because it dispatches the computation to different methods based on the type of the arguments. Here are examples that use the + operator with different types:

```
>>> start = Time(9, 45)
>>> duration = Time(1, 35)
>>> print(start + duration)
11:20:00
>>> print(start + 1337)
10:07:17
```

Unfortunately, this implementation of addition is not commutative. If the integer is the first operand, you get

```
>>> print(1337 + start)
TypeError: unsupported operand type(s) for +: 'int' and 'instance'
```

The problem is, instead of asking the Time object to add an integer, Python is asking an integer to add a Time object, and it doesn't know how. But there is a clever solution for this problem: the special method __radd__, which stands for "right-side add". This method is invoked when a Time object appears on the right side of the + operator. Here's the definition:

```
# inside class Time:
def __radd__(self, other):
        return self.__add__(other)
```

And here's how it's used:

```
>>> print(1337 + start)
10:07:17
```

As an exercise, write an add method for Points that works with either a Point object or a tuple:

- If the second operand is a Point, the method should return a new Point whose x coordinate is the sum of the x coordinates of the operands, and likewise for the y coordinates.
- If the second operand is a tuple, the method should add the first element of the tuple to the x coordinate and the second element to the y coordinate, and return a new Point with the result.

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, in "Dictionary as a Collection of Counters" we used 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. For example, the built-in function sum, which adds the elements of a sequence, works as long as the elements of the sequence support addition. Since Time objects provide an add method, they work with sum:

```
>>> t1 = Time(7, 43)
>>> t2 = Time(7, 41)
>>>t3 = Time(7, 37)
>>> total = sum([t1, t2, t3])
>>> print(total)
23:01:00
```

In general, if all of the operations inside a function work with a given type, the function works with that type. The best kind of polymorphism is the unintentional kind, where you discover that a function you already wrote can be applied to a type you never planned for.

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 errorprone 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.

Debugging

It is legal to add attributes to objects at any point in the execution of a program, but if you have objects with the same type that don't have the same attributes, it is easy to make mistakes. It is considered a good idea to initialize all of an object's attributes in the init method. If you are not sure whether an object has a particular attribute, you can use the builtin function hasattr.
Another way to access attributes is the built-in function vars, which takes an object and returns a dictionary that maps from attribute names (as strings) to their values:

```
>>> p = Point(3, 4)
>>> vars(p)
{'y': 4, 'x': 3}
```

For purposes of debugging, you might find it useful to keep this function handy:

```
def print_attributes(obj):
        for attr in vars(obj):
                print(attr, getattr(obj, attr))
```

print_attributes traverses the dictionary and prints each attribute name and its corresponding value. The built-in function getattr takes an object and an attribute name (as a string) and returns the attribute's value.

Inheritance

The language feature most often associated with object-oriented programming is **inheritance**. Inheritance is the ability to define a new class that is a modified version of an existing class. In this chapter I demonstrate inheritance using classes that repre- sent playing cards, decks of cards, and poker hands.

Card Objects

There are 52 cards in a deck, each of which belongs to 1 of 4 suits and 1 of 13 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 --> 3
Hearts --> 2
Diamonds --> 1
Clubs --> 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 --> 11
Queen --> 12
King --> 13

I am using the --> 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])
```

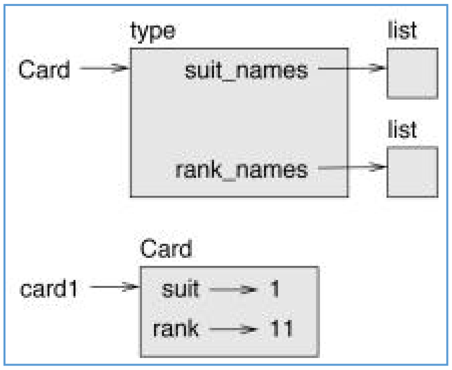
Variables like suit_names and rank_names, which are defined inside a class but outside of any method, are called class attributes because they are associated with the class object Card. This term distinguishes them from variables like suit and rank, which are called **instance attributes** because they 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. Every card has its own suit and rank, but there is only one copy of suit_names and rank_names.

Putting it all together, the expression Card.rank_names[self.rank] means “use the attribute rank from the object self as an index into the list rank_names from the class Card, and select the appropriate string.” The first element of rank_names is None because there is no card with rank zero. By including None as a place-keeper, we get a mapping with the nice property that the index 2 maps to the string '2', and so on. To avoid this tweak, we could have used a dictionary instead of a list.

With the methods we have so far, we can create and print cards:

```
>>> card1 = Card(2, 11)
>>> print(card1)
Jack of Hearts
```

Figure is a diagram of the Card class object and one Card instance. Card is a class object; its type is type. card1 is an instance of Card, so its type is Card. To save space, I didn’t draw the contents of suit_names and rank_names.



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 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 out- rank 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
```

As an exercise, write an __lt__ method for Time objects. You can use tuple comparison, but you also might consider comparing integers.

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 52 cards:

```
class Deck:
        def __init__(self):
                self.cards = []
```

```
for suit in rang(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 itera tion 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 new lines.

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)
```

Don't forget to import random. As an exercise, write a Deck method named sort that uses the list method sort to sort the cards in a Deck. sort uses the __lt__ method we defined to determine the order

Inheritance

Inheritance is the ability to define a new class that is a modified version of an existing class. As an example, let's say we want a class to represent a "hand", that is, the cards held by one player. A hand is similar to a deck: both are made up of a collection of cards, and both require operations like adding and removing cards.

A hand is also different from a deck; there are operations we want for hands that don't make sense for a deck. For example, in poker we might compare two hands to see which one wins. In bridge, we might compute a score for a hand in order to make a bid.

This relationship between classes—similar, but different—lends itself to inheritance. 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. In some games, cards are moved from one hand to another, or from a hand back to the deck. You can use move_cards for any of these operations: self can be either a Deck or a Hand, and hand, despite the name, can also be a Deck.

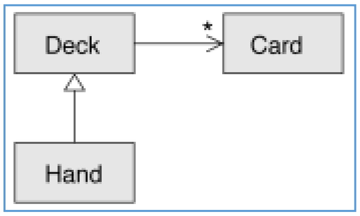
Inheritance is a useful feature. Some programs that would be repetitive without inheritance can be written more elegantly with it. Inheritance can facilitate code reuse, since you can customize the behavior of parent classes without having to mod- ify them. In some cases, the inheritance structure reflects 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 indi- vidual 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 shows the relationships between Card, Deck and Hand.



The arrow with a hollow triangle head represents an IS-A relationship; in this case it indicates that Hand inherits from Deck. The standard arrowhead represents a HAS-A relationship; in this case a Deck has ref- erences to Card objects. The star (*) near the arrowhead 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

The previous chapters demonstrate a development plan we might call “objectoriented design”. 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 discov- ered function interfaces by encapsulation and generalization, we can discover class interfaces by **data encapsulation**.

Markov analysis, from “Markov Analysis”, you’ll see that 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 structures (which makes for some interesting generated text). 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.

Debugging

Inheritance can make debugging difficult because when you invoke a method on an object, it might be hard to figure out which method will be invoked. Suppose you are writing a function that works with Hand objects. You would like it to work with all kinds of Hands, like PokerHands, BridgeHands, etc. If you invoke a method like shuffle, you might get the one defined in Deck, but if any of the sub- classes override this method, you'll get that version instead. This behavior is usually a good thing, but it can be confusing.

Any time you are unsure about the flow of execution through your program, the simplest solution is to add print statements at the beginning of the relevant methods. If Deck.shuffle prints a message that says something like Running Deck.shuffle, then as the program runs it traces the flow of execution.

As an alternative, you could use this function, which takes an object and a method name (as a string) and returns the class that provides the definition of the method:

```
def find_defining_class(obj, meth_name):
        for ty in type(obj).mro():
                if meth_name in ty.__dict__:
                        return ty
```

Here's an example:

```
>>> hand = Hand()
>>> find_defining_class(hand, 'shuffle')
<class ‘Card.Deck’>
```

So the shuffle method for this Hand is the one in Deck. find_defining_class uses the mro method to get the list of class objects (types) that will be searched for methods. “MRO” stands for “method resolution order”, which is the sequence of classes Python searches to “resolve” a method name.

Here's a design suggestion: when you override a method, the interface of the new method should be the same as the old. It should take the same parameters, return the same type, and obey the same preconditions and postconditions. If you follow this rule, you will find that any function designed to work with an instance of a parent class, like a Deck, will also work with instances of child classes like a Hand and Poker- Hand. If you violate this rule, which is called the “Liskov substitution principle”, your code will collapse like (sorry) a house of cards.

The Goodies

One of my goals for this book has been to teach you as little Python as possible. When there were two ways to do something, I picked one and avoided mentioning the other. Or sometimes I put the second one into an exercise. Now I want to go back for some of the good bits that got left behind. Python provides a number of features that are not really necessary—you can write good code without them—but with them you can sometimes write code that's more concise, readable or efficient, and sometimes all three.

Conditional Expressions

Conditional statements are often used to choose one of two values; for example:

```
if x > 0:
    y = math.log(x)
else:
    y = float('nan')
```

This statement checks whether x is positive. If so, it computes math.log. If not, math.log would raise a ValueError. To avoid stopping the program, we generate a "NaN", which is a special floating-point value that represents "Not a Number". We can write this statement more concisely using a **conditional expression**:

```
y = math.log(x) if x > 0 else float('nan')
```

You can almost read this line like English: "y gets log-x if x is greater than 0; other- wise it gets NaN". Recursive functions can sometimes be rewritten using conditional expressions. For example, here is a recursive version of factorial:

```
def factorial(n):
    if n == 0:
        return 1
    else:
        return n * factorial(n-1)
```

We can rewrite it like this:

```
def factorial(n):
    return 1 if n == 0 else n * factorial(n-1)
```

Another use of conditional expressions is handling optional arguments. For example, here is the init method from GoodKangaroo:

```
def __init__(self, name, contents=None):
    self.name = name
    if contents == None:
        contents = []
    self.pouch_contents = contents
```

We can rewrite this one like this:

```
def __init__(self, name, contents=None):
    self.name = name
    self.pouch_contents = [] if contents == None else contents
```

In general, you can replace a conditional statement with a conditional expression if both branches contain simple expressions that are either returned or assigned to the same variable.

List Comprehensions

In “Map, Filter and Reduce” we saw the map and filter patterns. For example, this function takes a list of strings, maps the string method capitalize to the elements, and returns a new list of strings:

```
def capitalize_all(t):
    res = []
    for s in t:
        res.append(s.capitalize())
    return res
```

We can write this more concisely using a **list comprehension:**

```
def capitalize_all(t):
    return [s.capitalize() for s in t]
```

The bracket operators indicate that we are constructing a new list. The expression inside the brackets specifies the elements of the list, and the for clause indicates what sequence we are traversing. The syntax of a list comprehension is a little awkward because the loop variable, s in this example, appears in the expression before we get to the definition.
List comprehensions can also be used for filtering. For example, this function selects only the elements of t that are uppercase, and returns a new list:

```
def only_upper(t):
    res = []
    for s in t:
        if s.isupper():
            res.append(s)
    return res
```

We can rewrite it using a list comprehension:

```
def only_upper(t):
        return [s for s in t if s.isupper()]
```

List comprehensions are concise and easy to read, at least for simple expressions. And they are usually faster than the equivalent for loops, sometimes much faster. So if you are mad at me for not mentioning them earlier, I understand. But, in my defense, list comprehensions are harder to debug because you can't put a print statement inside the loop. I suggest that you use them only if the computation is simple enough that you are likely to get it right the first time. And for beginners that means never.

Generator Expressions

Generator expressions are similar to list comprehensions, but with parentheses instead of square brackets:

```
>>> g = (x**2 for x in range(5))
>>> g
<generator object <genexpr> at 0x7f4c45a786c0>
```

The result is a generator object that knows how to iterate through a sequence of values. But unlike a list comprehension, it does not compute the values all at once; it waits to be asked. The built-in function next gets the next value from the generator:

```
>>> next(g)
0
>>> next(g)
1
```

When you get to the end of the sequence, next raises a StopIteration exception. You can also use a for loop to iterate through the values:

```
>>> for val in g:
 ...        print(val)
4
9
16
```

The generator object keeps track of where it is in the sequence, so the for loop picks up where next left off. Once the generator is exhausted, it continues to raise StopException:

```
>>> next(g)
```

```
StopIteration
```

Generator expressions are often used with functions like sum, max, and min:

```
>>> sum(x**2 for x in range(5))
30
```

any and all

Python provides a built-in function, any, that takes a sequence of boolean values and returns True if any of the values are True. It works on lists:

```
>>> any([False, False, True])
True
```

But it is often used with generator expressions:

```
>>> any(letter == 't' for letter in 'monty')
True
```

That example isn't very useful because it does the same thing as the in operator. But we could use any to rewrite some of the search functions. For example, we could write avoids like this:

```
def avoids(word, forbidden):
        return not any(letter in forbidden for letter in word)
```

The function almost reads like English: "word avoids forbidden if there are not any forbidden letters in word." Using any with a generator expression is efficient because it stops immediately if it finds a True value, so it doesn't have to evaluate the whole sequence. Python provides another built-in function, all, that returns True if every element of the sequence is True.

Sets

I use dictionaries to find the words that appear in a document but not in a word list. The function I wrote takes d1, which contains the words from the document as keys, and d2, which contains the list of words. It returns a dictionary that contains the keys from d1 that are not in d2:

```
def subtract(d1, d2):
        res = dict()
        for key in d1:
                if key not in d2:
                        res[key] = None
        return res
```

In all of these dictionaries, the values are None because we never use them. As a result, we waste some storage space. Python provides another built-in type, called a set, that behaves like a collection of dictionary keys with no values. Adding elements to a set is fast; so is checking membership. And sets provide methods and operators to compute common set operations.
For example, set subtraction is available as a method called difference or as an operator, -. So we can rewrite subtract like this:

```
def subtract(d1, d2):
    return set(d1) - set(d2)
```

The result is a set instead of a dictionary, but for operations like iteration, the behavior is the same. Some of the exercises in this book can be done concisely and efficiently with sets. For example, here is a solution to has_duplicates, from Exercise, that uses a dictionary:

```
def has_duplicates(t):
    d = {}
    for x in t:
        if x in d:
            return True
        d[x] = True
    return False
```

When an element appears for the first time, it is added to the dictionary. If the same element appears again, the function returns True. Using sets, we can write the same function like this:

```
def has_duplicates(t):
    return len(set(t)) < len(t)
```

An element can only appear in a set once, so if an element in t appears more than once, the set will be smaller than t. If there are no duplicates, the set will be the same size as t. For example, here’s a version of uses_only with a loop:

```
def uses_only(word, available):
    for letter in word:
        if letter not in available:
            return False
    return True
```

uses_only checks whether all letters in word are in available. We can rewrite it like this:

```
def uses_only(word, available):
    return set(word) <= set(available)
```

The <= operator checks whether one set is a subset or another, including the possibility that they are equal, which is true if all the letters in word appear in available.

Counters

A Counter is like a set, except that if an element appears more than once, the Counter keeps track of how many times it appears. If you are familiar with the mathematical idea of a **multiset**, a Counter is a natural way to represent a multiset.
Counter is defined in a standard module called collections, so you have to import it. You can initialize a Counter with a string, list, or anything else that supports iteration:

```
>>> from collections import Counter
 >>> count = Counter('parrot')
>>> count
Counter({'r': 2, 't': 1, 'o': 1, 'p': 1, 'a': 1})
```

Counters behave like dictionaries in many ways; they map from each key to the number of times it appears. As in dictionaries, the keys have to be hashable. Unlike dictionaries, Counters don't raise an exception if you access an element that doesn't appear. Instead, they return 0:

```
>>> count['d']
0
```

We can use Counters to rewrite is_anagram:

```
def is_anagram(word1, word2):
        return Counter(word1) == Counter(word2)
```

If two words are anagrams, they contain the same letters with the same counts, so their Counters are equivalent. Counters provide methods and operators to perform set-like operations, including addition, subtraction, union and intersection. And they provide an often-useful method, most_common, which returns a list of value-frequency pairs, sorted from most common to least:

```
>>> count = Counter('parrot')
>>> for val, freq in count.most_common(3):
...     print(val, freq)
r 2
p 1
a 1
```

defaultdict

The collections module also provides defaultdict, which is like a dictionary except that if you access a key that doesn't exist, it can generate a new value on the fly. When you create a defaultdict, you provide a function that's used to create new values. A function used to create objects is sometimes called a **factory**. The built-in functions that create lists, sets, and other types can be used as factories:

```
>>> from collections import defaultdict
>>> d = defaultdict(list)
```

Notice that the argument is list, which is a class object, not list(), which is a new list. The function you provide doesn't get called unless you access a key that doesn't exist:

```
>>> t = d['new key']
>>> t
[]
```

The new list, which we're calling t, is also added to the dictionary. So if we modify t, the change appears in d:

```
>>> t.append('new value')
>>> d
defaultdict(, {'new key': ['new value']})
```

If you are making a dictionary of lists, you can often write simpler code using defaultdict. I make a dictionary that maps from a sorted string of letters to the list of words that can be spelled with those letters. For example, 'opst' maps to the list ['opts', 'post', 'pots', 'spot', 'stop', 'tops']. Here's the original code:

```
def all_anagrams(filename):
        d = {}
        for line in open(filename):
                word = line.strip().lower()
                t = signature(word)
                if t not in d:
                        d[t] = [word]
                else:
                        d[t].append(word)
        return d
```

This can be simplified using setdefault:

```
def all_anagrams(filename):
        d = {}
```

```
    for line in open(filename):
        word = line.strip().lower()
        t = signature(word)
        d.setdefault(t, []).append(word)
    return d
```

This solution has the drawback that it makes a new list every time, regardless of whether it is needed. For lists, that’s no big deal, but if the factory function is complicated, it might be. We can avoid this problem and simplify the code using a defaultdict:

```
def all_anagrams(filename):
    d = defaultdict(list)
    for line in open(filename):
        word = line.strip().lower()
        t = signature(word)
        d[t].append(word)
    return d
```

Named Tuples

Many simple objects are basically collections of related values. For example, the Point object defined contains two numbers, x and y. When you define a class like this, you usually start with an init method and a str method:

```
class Point:
    def __init__(self, x=0, y=0):
        self.x = x self.y = y

    def __str__(self):
        return '(%g, %g)' % (self.x, self.y)
```

This is a lot of code to convey a small amount of information. Python provides a more concise way to say the same thing:

```
from collections import namedtuple
Point = namedtuple('Point', ['x', 'y'])
```

The first argument is the name of the class you want to create. The second is a list of the attributes Point objects should have, as strings. The return value from namedtuple is a class object:

```
>>> Point
<class ‘__main_.Point’>
```

Point automatically provides methods like __init__ and __str__ so you don’t have to write them. To create a Point object, you use the Point class as a function:

```
>>> p = Point(1, 2)
>>> p
Point(x=1, y=2)
```

The init method assigns the arguments to attributes using the names you provided. The str method prints a representation of the Point object and its attributes. You can access the elements of the named tuple by name:

```
>>> p.x, p.y
(1, 2)
```

But you can also treat a named tuple as a tuple:

```
>>> p[0], p[1]
(1, 2)
>>> x, y = p
>>> x, y
(1, 2)
```

Named tuples provide a quick way to define simple classes. The drawback is that sim- ple classes don’t always stay simple. You might decide later that you want to add methods to a named tuple. In that case, you could define a new class that inherits from the named tuple:

```
class Pointier(Point):
        # add more methods here
```

Or you could switch to a conventional class definition.

Gathering Keyword Args

In “Variable-Length Argument Tuples”, we saw how to write a function that gathers its arguments into a tuple:

```
def printall(*args):
        print(args)
```

You can call this function with any number of positional arguments (that is, arguments that don’t have keywords):

```
>>> printall(1, 2.0, '3')
(1, 2.0, '3')
```

But the * operator doesn’t gather keyword arguments:

```
>>> printall(1, 2.0, third='3')
TypeError: printall() got an unexpected keyword argument 'third'
```

To gather keyword arguments, you can use the ** operator:

```
def printall(*args, **kwargs):
        print(args, kwargs)
```

You can call the keyword gathering parameter anything you want, but kwargs is a common choice. The result is a dictionary that maps keywords to values:

```
>>> printall(1, 2.0, third='3')
(1, 2.0) {'third': '3'}
```

If you have a dictionary of keywords and values, you can use the scatter operator, **, to call a function:

```
>>> d = dict(x=1, y=2)
>>> Point(**d)
Point(x=1, y=2)
```

Without the scatter operator, the function would treat d as a single positional argument, so it would assign d to x and complain because there’s nothing to assign to y:

```
>>> d = dict(x=1, y=2)
>>> Point(d)
Traceback (most recent call last):
File "<stdin>", line 1, in module
TypeError: __new__() missing 1 required positional argument: 'y'
```

When you are working with functions that have a large number of parameters, it is often useful to create and pass around dictionaries that specify frequently used options.

Regulation: AK20	**Subject Code:CSE/CIC** 20APS0526/20APC3605	**Subject Name :** Basics of Python Programming	**AY:** 2023-2024
UINITWISE QUESTIONS			

BASICS OF PYTHON PROGRAMMING

PART-A

2 MARKS QUESTIONS:

UNIT-1

1. Define python?
2. What is the difference between interactive mode and script mode of executing a python code? (Or)What are the types of modes in the python?
3. What is a function? Mention the type of function used in Python.
4. How to check the number of keywords in Python.
5. “There is no difference between single quotes and double quotes while creating the string”. Justify the statement.
6. What happens if a semicolon (;) is placed at the end of a Python statement?
7. What is a variable? Write the rules for writing a variable?
8. Write syntax for defining function.
9. What is Indentation?
10. Define the scope and lifetime of a variable in Python.
11. Why is * called string repetition operator?
12. What are membership operators? Give examples for usage.
13. What are different types of values? Give examples.
14. After Executing the below python code, we will get an error. Can you explain why?

```
a = 1

b = 2

print(a == b)

print(b == c)
```

15. Write a function in Python programming to print Hello message.
16. Difference between global and local in Python?
17. Write the rules for identifiers.
18. Write program that leads to indentation error?
19. Write the purpose of stack diagram?

Regulation: AK20	**Subject Code:CSE/CIC** 20APS0526/20APC3605	**Subject Name :** Basics of Python Programming	**AY:** 2023-2024
UINITWISE QUESTIONS			

UNIT-2

1. Explain Fruitful function with an example.
2. How are nested for loops used in python.
3. What is docstring?
4. What is Boolean Expression? Give an example.
5. What is range() function and how it is used in lists?
6. What is encapsulation and its uses? Give example.
7. Define recursion. Write recursive function for finding factoring of given number.
8. Describe about any two conditional statements in python?
9. Write a Python program to read input data from the keyboard.
10. Describe about any two conditional statements in python?
11. List various types of logical operators in Python?
12. What is the importance of fruitful functions?
13. How to define multi-line string literals? Give an example.
14. What is refactoring? Explain it with an example?
15. Describe Keyboard input. with an example
16. Define the conditional execution with example?

UNIT-3

1.List the standard data types of python?
2.List the mutable data types and immutable data types.
3.What is the use of upper() and lower() functions in string? Give an example.
4.How to develop a pass in python.
5.Write a short notes on continue statement with example.
6.What is the purpose & use of break statement? Give an example.
7.What is meant by updating a variable? Give example.
8.Define string slicing?
9.How the values stored in a list are accessed ?
10. What is the difference between append() and insert() method on list data type?
11. How to remove spaces from a string.
12. How strings can be compared in Python?
13. List the escape characters can be used in strings.
14. Define list .Give one example.

UNIT-4

1. What are the advantages of Tuple over List?
2. How does del operation work on dictionaries? Give an example.
3. What are the two built-in methods that you can use on tuples?
4. Define dictionary and define the importance & uses of dictionary in python.
5. Compare dictionary with list.
6. Distinguish between Dictionaries and Tuples?
7. Distinguish between files and modules?
8. Write the python code to convert a list to a tuple?
9. What are the features of tuple in Python?
10. What is difference persistent program and transient program?
11. Difference between List and Dictionary?
12. Demonstrate format operator with a suitable example.
13. Write differences between list and tuple?
14. Write program to display the items in the Dictionary?

UNIT-5

1. Explain about List Comprehensions with a simple example.
2. What are mutable and immutable objects in python? Distinguish Mutable and Immutable Objects in Python?
3. Why objects are mutable?
4. Define inheritance? Give example.
5. What is _str_ method. what are the significance and uses of __str__method?
6. Explain the term constructor with an example.
7. Using list-comprehension, how would you generate a list with the following elements:

   [0, 1, 2, 3, 4, 5, 6, 7]

8. Is it possible to convert a class object into a floating type value?
9. Write the advantages of operator overloading
10. What is mean by type based dispatch?
11. Write the purpose of List Comprehensions?

PART-B

5 MARKS QUESTIONS:

UNIT-1

Regulation: AK20	**Subject Code:CSE/CIC** 20APS0526/20APC3605	**Subject Name :** Basics of Python Programming	**AY:** 2023-2024
UINITWISE QUESTIONS			

1. Explain how to write and execute a program in python and illustrate the steps for writing a python program to check whether the number is palindrome or not by using functions.
2. Define comment and list out different types of comments with syntax?
3. Show an example how precedence of operators effects an expression evaluation.

   What are the different flow control statements available in python? Explain with suitable examples.
4. Explain membership, bitwise and logical operators in python with an example.
5. With the necessary examples explain various keywords available in Python3.
6. What is an expression? What is order of operators in evaluation of expression? Illustrate it with suitable example.
7. Explain with example, function definition, use of function, parameters and arguments to the function.
8. Using the concept of functions, write a Python program to calculate the area of a circle, square and rectangle.
9. Distinguish between local and global variables with suitable examples.
10. Explain the different types of parameters in Python with suitable examples.
11. Explain unary & Relational operators with an example.
12. Estimate the features of Python programming language in detail.
13. Define variable in python and list the rules of python variables?
14. Python variables do not have specific types. Justify this statement with the help of an example.
15. Python has developed as an open source project. Justify this statement.
16. What are identifiers? Discuss the rules to name an identifier.
17. Distinguish between actual and formal parameter?
18. Demonstrate how a function calls another function. Justify your answer with an example?
19. Write a program add.py that takes 2 numbers as command line aruguments and print its sum.
20. Write about assignment statements with example program?
21. Explain the procedure to run a simple python script with an example program.
22. Write about function and benefits with example program?

UNIT-2

1. Write a python program using nested for loop to print the following pattern?

```
1

2 2

3 3 3

4 4 4 4
```

5   5 5 5 5

2. Describe about input statements in Python examples.
3. Describe in detail about conditional execution, Alternative Execution, Chained Conditionals and Nested Conditionals with an example
4. Explain about Boolean functions with suitable examples?
5. Explain about turtle module with example
6. Write a python program to read one subject marks and print pass or fail. Use single return Values function with argument.
7. Describe with suitable examples, fruitful functions and incremental development.
8. What is recursion? How recursion is differ from infinite recursion. Give suitable example.
9. Write a Python program to find Perimeter of rectangle
10. Compose a turtle object to draw shapes of the following using Object Oriented concepts: Star, Circle and polygons of side (4 to 10 sides).
11. Explain the concept of composition in functions with a suitable example?
12. What are the different loop control statements available in Python? Explain with suitable examples.
13. What is recursive function? Write a python program to calculate factorial of a number using recursive function?
14. What are the advantages and disadvantages of recursion function?
15. What is a fruitful function? Explain with the help of an example?
16. Analyze the importance of turtle module with simple program.
17. Define the recursive function and write the program for Fibonacci series using recursion.

UNIT-3

1. Briefly describe about break, continue and pass statements with suitable examples
2. Write a Python function that takes two lists and returns True if they have at least one common member?
3. Describe the following with suitable examples:
    i. Creating the List
    ii. Accessing values in the Lists
    iii. Updating the List
    iv. Deleting the list Elements
    v. Inserting elements into the List
    vi. Calculate the sum of all the elements of a List
    vii. Count the number of occurrences of an element in a list
    viii. Sorting of list

ix. Lists are Mutable
x. Appending the elements to the list
xi. Slicing of lists

4. Discuss the following operations on strings
   i) Length of string
   ii) Indexing in strings
   iii) counting substrings in a string
5. Write a Python program to display all positions of a substring in a given main string.
6. What is string? Discuss with examples, string traversal with for loop and while loop.
7. Write a program that counts the number of times the letter ‘a’ appears in a string.
8. What is a list? Explain with examples, creation of list, traversing a list, and list slicing.
9. What is string? Explain string concatenation and string repletion operator with example.
10. Assume input string contains only alphabet symbols and digits. Write a Python program to sort characters of the string, first alphabet symbols followed by digits.
11. Write a Python program for the following requirement:
    Input : a4b3c2
    Output : aaaabbbcc
12. Explain about the following functions/methods used on list with examples:
    i) count() ii)index() iv) append() iv) extend() v) remove() vi) copy() vii) reverse
13. Write a short note on the below functions in Python programming with programs
    i) filter() ii) map() iii) reduce()
14. Design a Python program to count the number of words in a text file.
15. What is String Traversal? Explain traversal with a for loop.
16. Explain string slicing in Python with an example?
17. Define syntax of a while loop in Python programming language with an example?
18. Create a new string made of the first, middle, and last characters of each input string
19. List out the methods of string and write simple program.

<u>UNIT-4</u>

1. Write the syntax to create, open and close a file? Illustrate with an example python program.
2. Write the syntax of the following with examples regarding Exception Handling in Python Programming:

   i try

ii try-except

iii try-except-else

iv try-except-else-finally

3. Discuss the following methods on dictionary
   i)index()
   ii)sorted()
   iii)max()
4. Design a program that reads a file, breaks each line into words, strips whitespace and punctuation from the words, and converts them to lowercase.
5. What is a tuple? Describe with example statements for creating a tuple, tuple assignment, and tuple as return values.
6. What is a file? Explain with examples, file format operator, file name and paths, catching exceptions in files, reading and writing a file.
7. Write a Python program that takes a sentence as input from the user and computes the frequency of each letter. Use a variable of dictionary type to maintain the count.
8. Write Python program to check for the presence of a key in the dictionary and find the sum of all its values.
9. Explain the need for Pickle module. Write Python program to save dictionary in Python Pickle.
10. List out the following:
    a) Dictionary as a Collection of Counters
    b) List and Tuple Slices.
11. Explain with example of closing a file in Python?
12. Write a program to input any two tuples and interchange the tuple values?
13. Write a Python program to read a word and print the number of letters, vowels and percentage of vowels in the word using a dictionary.
14. Explain the steps to create a Dictionary in Python and list out Dictionary Built-in functions.
15. Explain the steps to read config files and write Log files in Python.
16. What are the different modes of opening a file in Python? Explain the Python ‘open()’ built-in function.
17. Explain about write() and writelines() functions in Python.
18. Explain with an example of reading and writing files in python?
19. Illustrate the ways of creating the tuple and the tuple assignment with suitable programs?
20. Create class and objects and explain them?
21. Write a program to search a string or word in the given file (take any file)?
22. Write short notes on dictionaries and tuples?
23. In detail discuss about file operation modes with example program?

Regulation: AK20	**Subject Code:CSE/CIC** 20APS0526/20APC3605	**Subject Name :** Basics of Python Programming	**AY:** 2023-2024
UINITWISE QUESTIONS			

UNIT-5

1. Write a simple python program to create a class and an object in python?
2. Explain the significance of inheritance in OOP with an example in Python? Write a python program to show inheritance in python programming?
3. What is __init__method? Explain the significance of __init__ method in Python with suitable example.
4. Explain about the key features of object oriented programming in detail. (Or) Explain in detail about Object oriented Features in Python
5. Differentiate between compile-time and run-time polymorphism.
6. Write a python script to determine the time differences between two given times in HH:MM:SS format.(0<=HH<=23, 0<=MM<=59,0<=SS<=59)?
7. Create the following with examples.

   i. Creating a class
   ii. Constructor
   iii. The self-variable

8. Illustrate with suitable examples, operator overloading and polymorphism.
9. Consider a Rectangle Class and Create Two Rectangle Objects. Write Python program to Check Whether the Area of the First Rectangle is Greater than Second by Overloading > Operator.
10. Write Python Program to Demonstrate Multiple Inheritance with Method Overriding.
11. Write a Python program to perform union, intersection and difference operation using set.
12. Using the concept of classes and inheritance, write code for preparing a card, adding cards to deck, comparing cards, sorting of a card deck and printing the deck of cards.
13. Illustrate List Comprehension with suitable examples.
14. What is a class? What is the relation between an object and a class? Write a program which shows how to define a class, how to access member functions and how to create and access objects in Python.
15. Summarize Add, Remove, Shuffle and Sort of a card deck.
16. Write a note on Counters and Sets.
17. Compare and contrast Pure functions and Modifiers with suitable example?
18. What is class diagram? Draw the graphical representation of class Diagram?
19. Explain the List Accessing Methods and List Comprehension. Write Python code snippets to illustrate Method overriding and Method overloading concepts.
20. Describe the steps to create Classes and Objects in Python. List out the Built-in Class attributes.
21. How can polymorphism be implemented in Python? Illustrate with a code snippet.
22. Explain in detail about Operator Overloading in Python?
23. Explain about method overloading with example program?

24. Apply the inheritance concept with suitable example?
25. Explain Object oriented features?