

UNIT - 1

Introduction to Python

Installation

Python is a widely used general-purpose, high level programming language. It was initially designed by Guido van Rossum in 1991 and developed by Python Software Foundation. It was mainly developed for emphasis on code readability, and its syntax allows programmers to express concepts in fewer lines of code.

Python is a programming language that lets you work quickly and integrate systems more efficiently.

There are two major Python versions- Python 2 and Python 3.

- On 16 October 2000, Python 2.0 was released with many new features.
- On 3rd December 2008, Python 3.0 was released with more testing and includes new features.

Beginning with Python programming:

1) Finding an Interpreter:

Before we start Python programming, we need to have an interpreter to interpret and run our programs. There are certain online interpreters like <https://ide.geeksforgeeks.org/> , <http://ideone.com/> or <http://codepad.org/> that can be used to start Python without installing an interpreter.

Installation(Windows):

There are many interpreters available freely to run Python scripts like IDLE (Integrated Development Environment) which is installed when you install the python software from <http://python.org/downloads/>

Installation steps:

Steps to be followed and remembered:

Step 1: Select Version of Python to Install.

Step 2: Download Python Executable Installer.

Step 3: Run Executable Installer.

Step 4: Verify Python Was Installed On Windows

Step 5: Verify Pip Was Installed.

Step 6: Add Python Path to Environment Variables (Optional)



2) Writing first program:

Script Begins

Statement1

Statement2

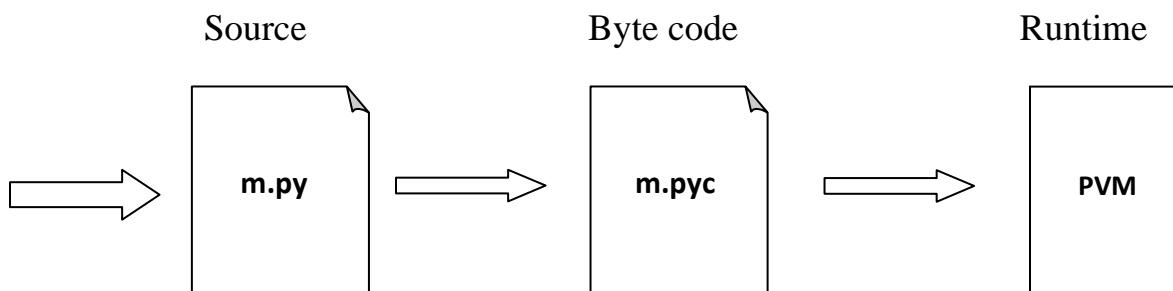
Statement3

Script Ends

How a Program Works

Python Code Execution:

Python's traditional runtime execution model: Source code you type is translated to byte code, which is then run by the Python Virtual Machine (PVM). Your code is automatically compiled, but then it is interpreted.



Source code extension is .py
Byte code extension is .pyc (Compiled **python code**)

There are two modes for using the Python interpreter:

- Interactive Mode
- Script Mode

Running Python in interactive mode:

Without passing python script file to the interpreter, directly execute code to Python prompt. Once you're inside the python interpreter, then you can start.

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

Relevant output is displayed on subsequent lines without the >>> symbol

```
>>> x=[0,1,2]
```

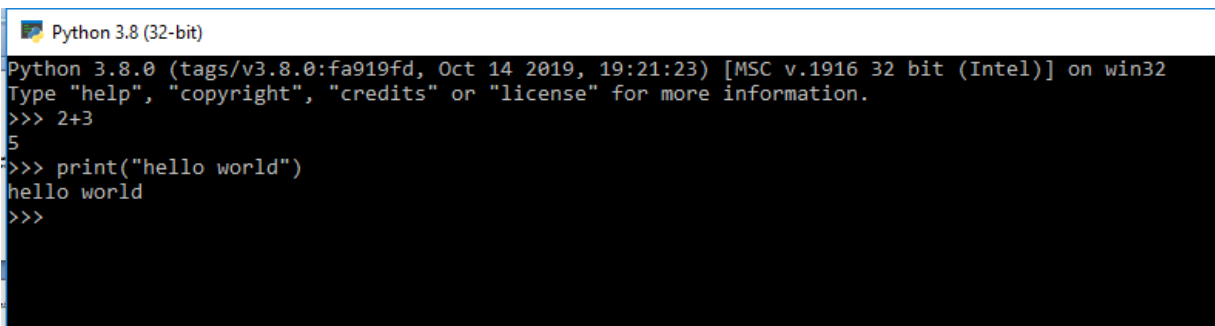
Quantities stored in memory are not displayed by default.

```
>>> x
```

#If a quantity is stored in memory, typing its name will display it. [0, 1, 2]

```
>>> 2+3
```

```
5
```



```
Python 3.8 (32-bit)
Python 3.8.0 (tags/v3.8.0:fa919fd, Oct 14 2019, 19:21:23) [MSC v.1916 32 bit (Intel)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> 2+3
5
>>> print("hello world")
hello world
>>>
```

The chevron at the beginning of the 1st line, i.e., the symbol >>> is a prompt the python interpreter uses to indicate that it is ready. If the programmer types 2+6, the interpreter replies 8.

Running Python in script mode:

Alternatively, programmers can store Python script source code in a file with the .py extension, and use the interpreter to execute the contents of the file. To execute the script by the interpreter, you have to tell the interpreter the name of the file. For example, if you have a script name MyFile.py and you're working on Unix, to run the script you have to type:

python MyFile.py

Working with the interactive mode is better when Python programmers deal with small pieces of code as you can type and execute them immediately, but when the code is more than 2-4 lines, using the script for coding can help to modify and use the code in future.

Example:

```
C:\Users\MRCET\AppData\Local\Programs\Python\Python38-32\pyyy>python e1.py
resource open
the no cant be divisibile zero division by zero
resource close
finished
```

Using Python

The following are the primary factors to use python in day-to-day life:

1. Python is object-oriented

Structure supports such concepts as polymorphism, operation overloading and multiple inheritances.

2. Indentation

Indentation is one of the greatest features in python.

3. It's free (open source)

Downloading python and installing python is free and easy

4. It's Powerful

- Dynamic typing
- Built-in types and tools
- Library utilities
- Third party utilities (e.g. Numeric, NumPy, sciPy)
- Automatic memory management

5. It's Portable

- Python runs virtually every major platform used today
- As long as you have a compatible python interpreter installed, python programs will run in exactly the same manner, irrespective of platform.

6. It's easy to use and learn

- No intermediate compile

- Python Programs are compiled automatically to an intermediate form called byte code, which the interpreter then reads.
- This gives python the development speed of an interpreter without the performance loss inherent in purely interpreted languages.
- Structure and syntax are pretty intuitive and easy to grasp.

7. Interpreted Language

Python is processed at runtime by python Interpreter

8. Interactive Programming Language

Users can interact with the python interpreter directly for writing the programs

9. Straight forward syntax

The formation of python syntax is simple and straight forward which also makes it popular.

We can use Python everywhere. The most common important application areas are as follows:

1. For developing Desktop Applications

The Applications which are running on a single systems (i.e., Stand alone applications)

Eg: Simple Calculator application

2. For developing Web Applications

Eg: Gmail Application, Online E-commerce applications, Facebook application, Blog applications etc.,

3. For Network Applications

Eg: Chatting applications, Client-Server applictaions etc.,

4. For Games development

5. For Data Analysis Applications

6. For Machine Learning applications

7. For developing Artificial Intelligence, Deep Learning, Neural Network Applications

8. For IOT

9. For Data Sciene

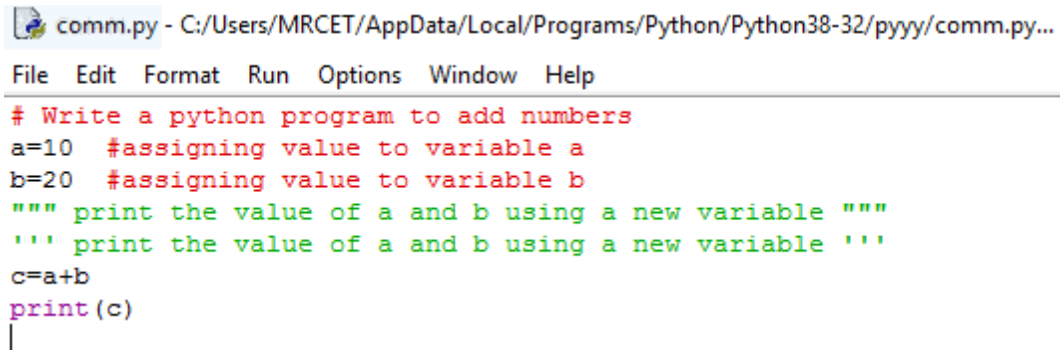
That's why Python is called as General Purpose Programming Language.

Comments:

Single-line comments begin with a hash (#) symbol and are useful in mentioning that the whole line should be considered as a comment until the end of line.

A Multi line comment is useful when we need to comment on many lines. In python, triple double quote (“ “ “) and single quote(‘ ‘ ‘)are used for multi-line commenting.

Example:



```
comm.py - C:/Users/MRCET/AppData/Local/Programs/Python/Python38-32/pyyy/comm.py...
File Edit Format Run Options Window Help
# Write a python program to add numbers
a=10 #assigning value to variable a
b=20 #assigning value to variable b
""" print the value of a and b using a new variable """
''' print the value of a and b using a new variable '''
c=a+b
print(c)
```

Output:

```
C:/Users/NRCM/AppData/Local/Programs/Python/Python38-32/pyyy/comm.py
30
```

Variables:

Variables are nothing but reserved memory locations to store values. This means that when you create a variable you reserve some space in memory.

Based on the data type of a variable, the interpreter allocates memory and decides what can be stored in the reserved memory. Therefore, by assigning different data types to variables, you can store integers, decimals or characters in these variables.

Rules for Python variables:

- A variable name must start with a letter or the underscore character
- A variable name cannot start with a number
- A variable name can only contain alpha-numeric characters and underscores (A-z, 0-9, and _)
- Variable names are case-sensitive (age, Age and AGE are three different variables)

Assigning Values to Variables:

Python variables do not need explicit declaration to reserve memory space. The declaration happens automatically when you assign a value to a variable. The equal sign (=) is used to assign values to variables.

The operand to the left of the = operator is the name of the variable and the operand to the right of the = operator is the value stored in the variable.

For example –

```
a= 100      # An integer assignment
```

```
b = 1000.0 # A floating point
```

```
c = "John" # A string
```

```
print (a)
```

```
print (b)
```

```
print (c)
```

This produces the following result –

```
100
```

```
1000.0
```

```
John
```

Multiple Assignments:

Python allows you to assign a single value to several variables simultaneously. For

Example:

```
a = b = c = 1
```

Here, an integer object is created with the value 1, and all three variables are assigned to the same memory location. You can also assign multiple objects to multiple variables.

For example –

```
a,b,c = 1,2,"nrcm"
```

Here, two integer objects with values 1 and 2 are assigned to variables a and b respectively, and one string object with the value "john" is assigned to the variable c.

Output Variables:

The Python print statement is often used to output variables.

Variables do not need to be declared with any particular type and can even change type after they have been set.

```
x = 5 # x is of type int
```

```
x = "mrcet " # x is now of type str print(x)
```

Output: mrcet

To combine both text and a variable, Python uses the “+” character:

Example

```
x = "awesome" print("Python is " + x)
```

Output

Python is awesome

You can also use the + character to add a variable to another variable:

Example

```
x = "Python is " y = "awesome" z = x + y print(z)
```

Output:

Python is awesome

You can also use the + character to add a variable to another variable:

Example

```
x = "Python is "
```

```
y = "awesome"
```

```
z = x + y
```

```
print(z)
```

Output:

Python is awesome

Types of Variables:

Python supports 2 types of variables.

1. Global Variables
2. Local Variables

Local Scope:

A variable which is defined inside a function is local to that function. It is accessible from the point at which it is defined until the end of the function, and exists for as long as the function is executing.

```
# create a local variable
```

```
def f1():
```

```
    y = "local"
```

```
    print(y)
```

```
f1()
```

Output:

Local If we try to access the local variable outside the scope for example

Global Scope:

A variable which is defined in the main body of a file is called a global variable. It will be visible throughout the file, and also inside any file which imports that file.

The variable defined inside a function can also be made global by using the global statement.

1. Global Variables

The variables which are declared outside of function are called global variables.

These variables can be accessed in all functions of that module.

Consider the following example:

```
a = 10 # Global Variables
def f1():
    a = 20 # Local variable to the function 'f1'
    print(a) # 20
def f2():
    print(a) # 10
f1()
f2()
20
10
```

Suppose our requirement is, we don't want local variable. Can you please refer the local variable as the global variable only? One special keyword is used, called as global.

Global keyword:

We can use global keyword for the following 2 purposes:

1. To declare global variables explicitly inside function.
2. To make global variable available to the function so that we can perform required modifications.

Expressions:

An expression is a combination of values, variables, and operators. An expression is evaluated using assignment operator.

Examples: $Y=x + 17$

```
>>> x=10
```

```
>>> z=x+20
```

```
>>> z
```

```
30
```

```
>>> x=10
```

```
>>> y=20
>>> c=x+y
>>> c
30
```

A value all by itself is a simple expression, and so is a variable.

```
>>> y=20
>>> y
20
```

Python also defines expressions only contain identifiers, literals, and operators. So, Identifiers: Any name that is used to define a class, function, variable module, or object is an identifier.

Literals: These are language-independent terms in Python and should exist independently in any programming language. In Python, there are the string literals, byte literals, integer literals, floating point literals, and imaginary literals.

Operators: In Python you can implement the following operations using the corresponding tokens.

Operator	Token
add	+
subtract	-
multiply	*
Integer Division	/
remainder	%
Binary left shift	<<
Binary right shift	>>

and	&
or	
Less than	<
Greater than	>
Less than or equal to	<=
Greater than or equal to	>=
Check equality	==
Check not equal	!=

Some of the python expressions are:

Generator expression:

Syntax: (compute(var) for var in iterable)

```
>>> x = (i for i in 'abc') #tuple comprehension
```

```
>>> x
```

```
<generator object <genexpr> at 0x033EEC30>
```

```
>>> print(x)
```

```
<generator object <genexpr> at 0x033EEC30>
```

You might expect this to print as ('a', 'b', 'c') but it prints as <generator object <genexpr> at 0x02AAD710> The result of a tuple comprehension is not a tuple: it is actually a generator. The only thing that you need to know now about a generator now is that you can iterate over it, but ONLY ONCE.

Conditional expression:

Syntax: true_value if Condition else false_value

```
>>> x = "1" if True else "2"
```

```
>>> x
```

'1'

Statements:

A statement is an instruction that the Python interpreter can execute. We have normally two basic statements, the assignment statement and the print statement. Some other kinds of statements that are if statements, while statements, and for statements generally called as control flows.

Examples:

An assignment statement creates new variables and gives them values:

```
>>> x=10
>>> college="nrcm"
```

An print statement is something which is an input from the user, to be printed / displayed on to the screen (or) monitor.

```
>>> print("nrcm colege")
nrcm college
```

Precedence of Operators:

Operator precedence affects how an expression is evaluated.

For example, $x = 7 + 3 * 2$; here, x is assigned 13, not 20 because operator * has higher precedence than +, so it first multiplies $3*2$ and then adds into 7.

Example 1:

```
>>> 3+4*2
11
```

Multiplication gets evaluated before the addition operation

```
>>> (10+10)*2
40
```

Parentheses () overriding the precedence of the arithmetic operators

Example 2:

```
a = 20
b = 10
c = 15
d = 5
e = 0
e = (a + b) * c / d #( 30 * 15 ) / 5
print("Value of (a + b) * c / d is ", e)
e = ((a + b) * c) / d # (30 * 15 ) / 5
print("Value of ((a + b) * c) / d is ", e)
```

```
e = (a + b) * (c / d); # (30) * (15/5)
print("Value of (a + b) * (c / d) is ", e)
e = a + (b * c) / d; # 20 + (150/5)
print("Value of a + (b * c) / d is ", e)
```

Output:

```
C:/Users/MRCET/AppData/Local/Programs/Python/Python38-32/pyyy/opprec.py
Value of (a + b) * c / d is 90.0
Value of ((a + b) * c) / d is 90.0
Value of (a + b) * (c / d) is 90.0
Value of a + (b * c) / d is 50.0
```

Type conversions

In Python we need not to declare the type of the variables.

In Python, type concept is applicable (int,float .. types are there in Python), but we are not required to declare type explicitly.

In Python, whenever we are assigning some value to a variable, based on the provided value, automatically type will be considered. Such type of programming languages is known as Dynamically Typed Programming Languages.

```
a = 10
print(type(a))
<class 'int'>
a = 10.5
print(type(a))
<class 'float'>
a = True
print(type(a))
<class 'bool'>
a = "Karthi"
print(type(a))
<class 'str'>
```

In programming, type conversion is the process of converting data of one type to another. For example: converting int data to str.

There are two types of type conversion in Python.

Implicit Conversion - automatic type conversion

Explicit Conversion - manual type conversion

Python Implicit Type Conversion

In certain situations, Python automatically converts one data type to another. This is known as implicit type conversion.

Example 1: Converting integer to float

Let's see an example where Python promotes the conversion of the lower data type (integer) to the higher data type (float) to avoid data loss.

```
integer_number = 123
```

```
float_number = 1.23
```

```
new_number = integer_number + float_number
```

```
# display new value and resulting data type
```

```
print("Value:",new_number)
```

```
print("Data Type:",type(new_number))
```

output:

```
Value: 124.23
```

```
Data Type: <class 'float'>
```

In the above example, we have created two variables: integer_number and float_number of int and float type respectively.

Then we added these two variables and stored the result in new_number.

Explicit Type Conversion

In Explicit Type Conversion, users convert the data type of an object to required data type.

We use the built-in functions like int(), float(), str(), etc to perform explicit type conversion.

This type of conversion is also called typecasting because the user casts (changes) the data type of the objects.

Example 2: Addition of string and integer Using Explicit Conversion

```
num_string = '12'  
num_integer = 23
```

```
print("Data type of num_string before Type Casting:",type(num_string))
```

```
# explicit type conversion  
num_string = int(num_string)
```

```
print("Data type of num_string after Type Casting:",type(num_string))
```

```
num_sum = num_integer + num_string
```

```
print("Sum:",num_sum)  
print("Data type of num_sum:",type(num_sum))
```

Run Code

Output:

Data type of num_string before Type Casting: <class 'str'>

Data type of num_string after Type Casting: <class 'int'>

Sum: 35

Data type of num_sum: <class 'int'>

In the above example, we have created two variables: num_string and num_integer with str and int type values respectively. Notice the code,

```
num_string = int(num_string)
```

Here, we have used int() to perform explicit type conversion of num_string to integer type.

After converting num_string to an integer value, Python is able to add these two variables.

Finally, we got the num_sum value i.e 35 and data type to be int.

Type Conversion is the conversion of an object from one data type to another data type.

Implicit Type Conversion is automatically performed by the Python interpreter.

Python avoids the loss of data in Implicit Type Conversion.

Explicit Type Conversion is also called Type Casting, the data types of objects are converted using predefined functions by the user.

In Type Casting, loss of data may occur as we enforce the object to a specific data type.

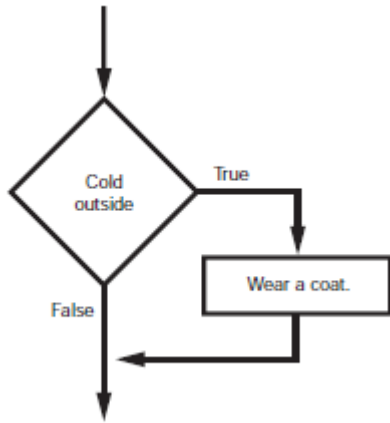
1. `int(a, base)`: This function converts any data type to integer. 'Base' specifies the base in which string is if the data type is a string.
2. `float()`: This function is used to convert any data type to a floating-point number.
3. `ord()` : This function is used to convert a character to integer.
4. `hex()` : This function is to convert integer to hexadecimal string.
5. `oct()` : This function is to convert integer to octal string.
6. `tuple()` : This function is used to convert to a tuple.
7. `set()` : This function returns the type after converting to set.
8. `list()` : This function is used to convert any data type to a list type.
9. `dict()` : This function is used to convert a tuple of order (key,value) into a dictionary.
10. `str()` : Used to convert integer into a string.
11. `complex(real,imag)`: This function converts real numbers to complex(real,imag) number.
12. `chr(number)`: This function converts number to its corresponding ASCII character.

Decision Structure and Boolean Logic

Python

If Statement

The if statement is used to create a decision structure, which allows a program to have more than one path of execution. The if statement causes one or more statements to execute only when a Boolean expression is true.



```
if condition:  
    statement  
    statement  
    etc.
```

Simple Decision Structure

General Format of Decision Structure

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

Relational Operators

- 1) Check the Operators

```
x=1  
x>=1 ← This is an expression  
True
```

```
Use of if  
sales=5000  
if sales==5000:  
    print(sales)
```

- 2) Perform the problem. Calculate the average Test Score.

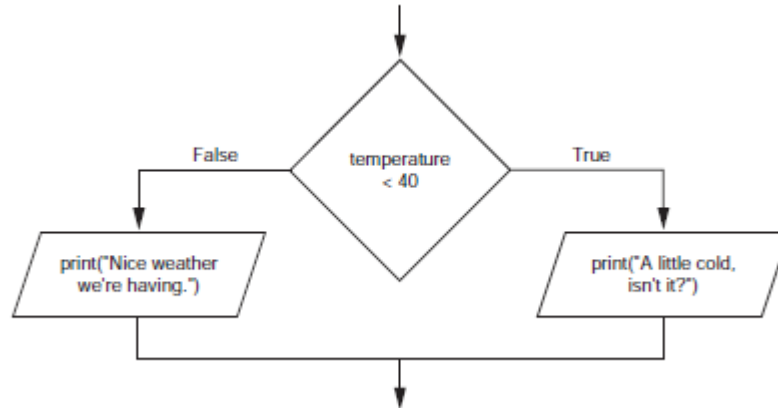
```
high score=95  
test1 = int(input('Enter the score for test 1:'))  
test2 = int(input('Enter the score for test 2:'))  
test3 = int(input('Enter the score for test 3:'))  
# Calculate the average test score.  
average = (test1 + test2 + test3) / 3
```

```
# Print the average.  
print('The average score is', average)
```

```
# If the average is a high score,
# congratulate the user.
if average >= high_score:
    print('Congratulations!')
    print("That is a great average!")
```

3) If-else statement

An if-else statement will execute one block of statements if its condition is true, or another block if its condition is false.



```
if condition:
    statement
    statement
    etc.
else:
    statement
    statement
    etc.
```

General format of if-else Structure

Eg.

```
sales=5000
```

```
if sales==5000:
```

```
    print(sales)
```

```
else:
```

```
    print("this is alternative")
```

4) Comparing Strings

Python allows you to compare strings. This allows you to create decision structures that test the value of a string.

```
>>> name1='mary'
```

```
>>> name2='mark'
```

```
>>> if name1==name2:
```

```
    print('the names are the same')
```

```
else:
```

```
    print('the names are not same')
```

Nested if-elif-else statement **To test more than one condition, a decision structure can be nested inside another decision structure.**

```
number = int(input('Please input a number in the range of 1 through 7: '))
```

```
if number == 1:
    print('Monday')
elif number == 2:
    print('Tuesday')
elif number == 3:
    print('Wednesday')
elif number == 4:
    print('Thursday')
elif number == 5:
    print('Friday')
elif number == 6:
    print('Saturday')
elif number == 7:
    print('Sunday')
else:
    print('Error')
```

Examples:

```
mass = float(input('Please enter an object\'s mass: '))
```

```
weight = mass * 9.8
```

```
if weight > 500:
    print('It is too heavy.')
elif weight < 100:
    print('It is too light.')
else:
    print(weight, 'N', sep="")
```

5) Logical operators

Operator	Meaning
and	The and operator connects two Boolean expressions into one compound expression. Both subexpressions must be true for the compound expression to be true.
or	The or operator connects two Boolean expressions into one compound expression. One or both subexpressions must be true for the compound expression to be true. It is only necessary for one of the subexpressions to be true, and it does not matter which.
not	The not operator is a unary operator, meaning it works with only one operand. The operand must be a Boolean expression. The not operator reverses the truth of its operand. If it is applied to an expression that is true, the operator returns false. If it is applied to an expression that is false, the operator returns true.

```
>>> if not(temp>100):
        print("this is true")
```

6) **Short Circuit Evaluations** If 1st sub expression is false other sub expression will not be checked

But in OR if 1st expression is true other will not be checked.

Boolean Variable

A Boolean Variable can reference one of two values: True or False

A flag is a variable that signals when

some condition exists in the program. When the flag variable is set to False, it indicates the condition does not exist. When the flag variable is set to True, it means the condition does exist.

```
Sales=4000.0
```

```
if Sales>=5000.0:
```

```
    sales_quota=True
```

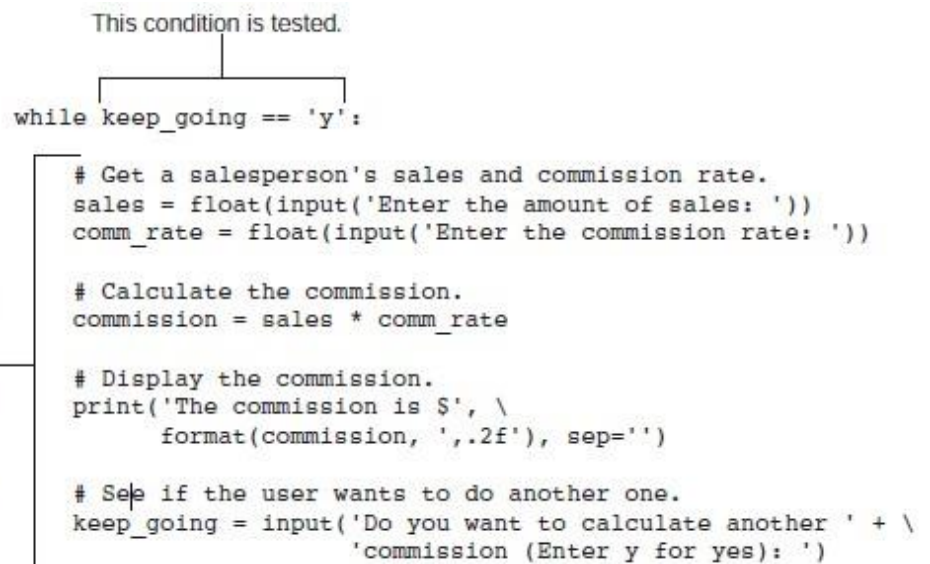
```
else:
```

```
    sales_quota=False
```

7) **Nested Loops**

A condition-controlled loop uses a true/false condition to control the number of times that it repeats. A count-controlled loop repeats a specific number of times

```
while condition:
    statement
    statement
    etc.
```



```
number = float(input('Enter a positive number or enter a negative number ' + \
```

```
                    'if you want to end: '))
```

```
total = 0
```

```

while number > 0:
    total += number
    number = float(input('Enter a positive number: '))
    print('The sum is', total)
Example to perform
# This program assists a technician in the process
# of checking a substance's temperature.

# Create a variable to represent the maximum
# temperature.
max_temp = 102.5

# Get the substance's temperature.
temperature = float(input("Enter the substance's Celsius temperature: "))

# As long as necessary, instruct the user to
# adjust the thermostat.
while temperature > max_temp:
    print('The temperature is too high.')
    print('Turn the thermostat down and wait')
    print('5 minutes. Then take the temperature')
    print('again and enter it.')
    temperature = float(input('Enter the new Celsius temperature:'))

# Remind the user to check the temperature again
# in 15 minutes.
print('The temperature is acceptable.')
print('Check it again in 15 minutes.')

```

8) For Loop : Count controlled Loop

```

for variable in [value1, value2, etc.]:
    statement
    statement
    etc.

```

Example

```

print('I will display the numbers 1 through 5.')
for num in [1, 2, 3, 4, 5]:
    print(num)

```

9) Using Range functions with for loop

```

>>> for x in range(5):
    print(x)
Eg. for number in range(1,11):
    square=number**2
    print(number, '\t', square)

```

Operator	Example Usage	Equivalent To
+=	x += 5	x = x + 5
--	y -= 2	y = y - 2
*=	z *= 10	z = z * 10
/=	a /= b	a = a / b
%=	c %= 3	c = c % 3

```
>>> for num in range(1,10,2):
    print(num)
```

Highest to Lowest

```
range(10,0,-1)
```

```
10, 9, 8, 7, 6, 5, 4, 3, 2, 1
```

10) Nested Loops

A loop inside another loop is nested loop

```
>>> for hours in range(24):
    for minutes in range(60):
        for seconds in range(60):
            print(hours,':', minutes,':', seconds)
```

11) Perform the Problem

Print the following

```
*****
*****
*****
*****
*****
```

12) Functions

```
>>> def message():
    print('hello')
    print('Friend')
message()
```

Data Types in Python

Every value in Python has a data type. Since everything is an object in Python programming, data types are actually classes, and variables are instance (object) of these classes.

There are various data types in Python. Some of the important types are listed below

DATA TYPES IN PYTHON

- ❖ Python Numbers
- ❖ Python List
- ❖ Python Tuple
- ❖ Python Strings
- ❖ Python Set
- ❖ Python Dictionary

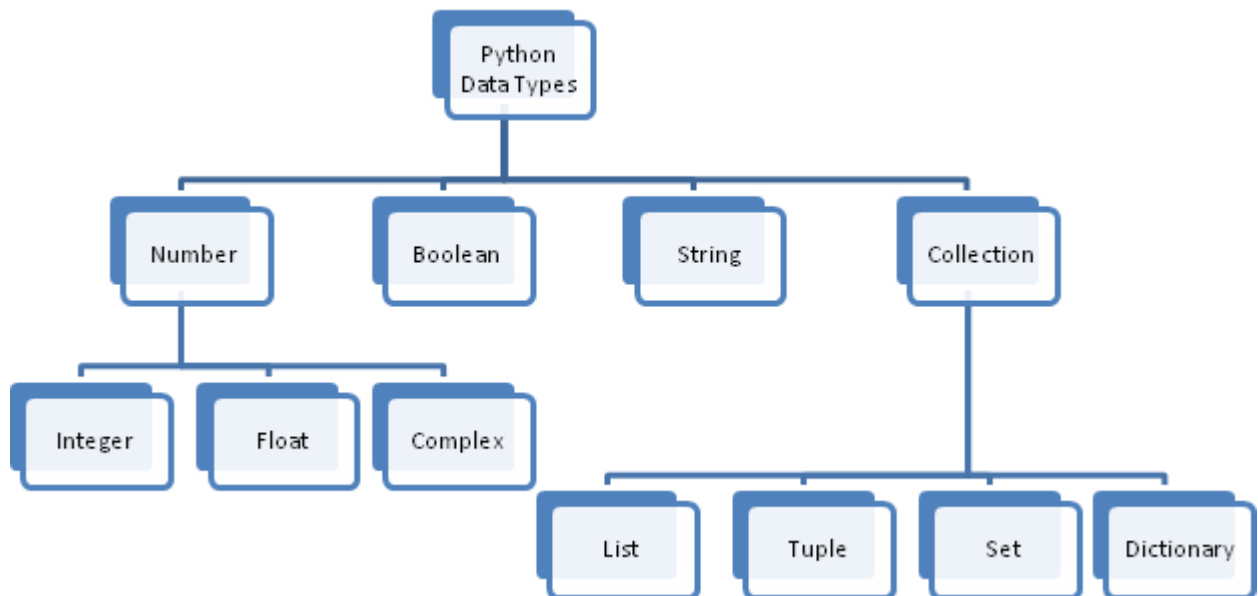


Chart : Python Data Types

1. Python Data Type – Numeric

Python numeric data type is used to hold numeric values like;

Data Type	Use
Int	holds signed integers of non-limited length.
Long	holds long integers(exists in Python 2.x,
Float	holds floating precision numbers and it's
complex	holds complex numbers.

2. Python Data Type – String

String is a **sequence of characters**. Python supports Unicode characters. Generally strings are represented by either single or double quotes.

```
>>> s1 = "This is a string"
>>> s2= '''a multiline String'''
```

Single Line String	"hello world"																																																																																																																																																									
Multi Line String	""""Gwalior Madhya Pradesh""""																																																																																																																																																									
Raw String	r"raw \n string" [Used when we want to have a string that containsbackslash and don't want it to be treated as an escape character.]																																																																																																																																																									
Character	"C" [Single letter]																																																																																																																																																									
Unicode string	<p>u"\u0938\u0902\u0917\u0940\u0924\u093E" will print सं ीता</p> <table border="1"> <thead> <tr> <th></th> <th>0</th> <th>1</th> <th>2</th> <th>3</th> <th>4</th> <th>5</th> <th>6</th> <th>7</th> <th>8</th> <th>9</th> <th>A</th> <th>B</th> <th>C</th> <th>D</th> <th>E</th> <th>F</th> </tr> </thead> <tbody> <tr> <td>U+090x</td> <td>ॐ</td> <td>ॐ</td> <td>ॐ</td> <td>ॐ</td> <td>ॐ</td> <td>ॐ</td> <td>ॐ</td> <td>ॐ</td> <td>ॐ</td> <td>ॐ</td> <td>ॐ</td> <td>ॐ</td> <td>ॐ</td> <td>ॐ</td> <td>ॐ</td> <td>ॐ</td> </tr> <tr> <td>U+091x</td> <td>ॐ</td> <td>ॐ</td> <td>ॐ</td> <td>ॐ</td> <td>ॐ</td> <td>ॐ</td> <td>ॐ</td> <td>ॐ</td> <td>ॐ</td> <td>ॐ</td> <td>ॐ</td> <td>ॐ</td> <td>ॐ</td> <td>ॐ</td> <td>ॐ</td> <td>ॐ</td> </tr> <tr> <td>U+092x</td> <td>ॐ</td> <td>ॐ</td> <td>ॐ</td> <td>ॐ</td> <td>ॐ</td> <td>ॐ</td> <td>ॐ</td> <td>ॐ</td> <td>ॐ</td> <td>ॐ</td> <td>ॐ</td> <td>ॐ</td> <td>ॐ</td> <td>ॐ</td> <td>ॐ</td> <td>ॐ</td> </tr> <tr> <td>U+093x</td> <td>ॐ</td> <td>ॐ</td> <td>ॐ</td> <td>ॐ</td> <td>ॐ</td> <td>ॐ</td> <td>ॐ</td> <td>ॐ</td> <td>ॐ</td> <td>ॐ</td> <td>ॐ</td> <td>ॐ</td> <td>ॐ</td> <td>ॐ</td> <td>ॐ</td> <td>ॐ</td> </tr> <tr> <td>U+094x</td> <td>ॐ</td> <td>ॐ</td> <td>ॐ</td> <td>ॐ</td> <td>ॐ</td> <td>ॐ</td> <td>ॐ</td> <td>ॐ</td> <td>ॐ</td> <td>ॐ</td> <td>ॐ</td> <td>ॐ</td> <td>ॐ</td> <td>ॐ</td> <td>ॐ</td> <td>ॐ</td> </tr> <tr> <td>U+095x</td> <td>ॐ</td> <td>ॐ</td> <td>ॐ</td> <td>ॐ</td> <td>ॐ</td> <td>ॐ</td> <td>ॐ</td> <td>ॐ</td> <td>ॐ</td> <td>ॐ</td> <td>ॐ</td> <td>ॐ</td> <td>ॐ</td> <td>ॐ</td> <td>ॐ</td> <td>ॐ</td> </tr> <tr> <td>U+096x</td> <td>ॐ</td> <td>ॐ</td> <td>ॐ</td> <td>ॐ</td> <td>ॐ</td> <td>ॐ</td> <td>ॐ</td> <td>ॐ</td> <td>ॐ</td> <td>ॐ</td> <td>ॐ</td> <td>ॐ</td> <td>ॐ</td> <td>ॐ</td> <td>ॐ</td> <td>ॐ</td> </tr> <tr> <td>U+097x</td> <td>ॐ</td> <td>ॐ</td> <td>ॐ</td> <td>ॐ</td> <td>ॐ</td> <td>ॐ</td> <td>ॐ</td> <td>ॐ</td> <td>ॐ</td> <td>ॐ</td> <td>ॐ</td> <td>ॐ</td> <td>ॐ</td> <td>ॐ</td> <td>ॐ</td> <td>ॐ</td> </tr> </tbody> </table>		0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	U+090x	ॐ	ॐ	ॐ	ॐ	ॐ	ॐ	ॐ	ॐ	ॐ	ॐ	ॐ	ॐ	ॐ	ॐ	ॐ	ॐ	U+091x	ॐ	ॐ	ॐ	ॐ	ॐ	ॐ	ॐ	ॐ	ॐ	ॐ	ॐ	ॐ	ॐ	ॐ	ॐ	ॐ	U+092x	ॐ	ॐ	ॐ	ॐ	ॐ	ॐ	ॐ	ॐ	ॐ	ॐ	ॐ	ॐ	ॐ	ॐ	ॐ	ॐ	U+093x	ॐ	ॐ	ॐ	ॐ	ॐ	ॐ	ॐ	ॐ	ॐ	ॐ	ॐ	ॐ	ॐ	ॐ	ॐ	ॐ	U+094x	ॐ	ॐ	ॐ	ॐ	ॐ	ॐ	ॐ	ॐ	ॐ	ॐ	ॐ	ॐ	ॐ	ॐ	ॐ	ॐ	U+095x	ॐ	ॐ	ॐ	ॐ	ॐ	ॐ	ॐ	ॐ	ॐ	ॐ	ॐ	ॐ	ॐ	ॐ	ॐ	ॐ	U+096x	ॐ	ॐ	ॐ	ॐ	ॐ	ॐ	ॐ	ॐ	ॐ	ॐ	ॐ	ॐ	ॐ	ॐ	ॐ	ॐ	U+097x	ॐ	ॐ	ॐ	ॐ	ॐ	ॐ	ॐ	ॐ	ॐ	ॐ	ॐ	ॐ	ॐ	ॐ	ॐ	ॐ
	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F																																																																																																																																										
U+090x	ॐ	ॐ	ॐ	ॐ	ॐ	ॐ	ॐ	ॐ	ॐ	ॐ	ॐ	ॐ	ॐ	ॐ	ॐ	ॐ																																																																																																																																										
U+091x	ॐ	ॐ	ॐ	ॐ	ॐ	ॐ	ॐ	ॐ	ॐ	ॐ	ॐ	ॐ	ॐ	ॐ	ॐ	ॐ																																																																																																																																										
U+092x	ॐ	ॐ	ॐ	ॐ	ॐ	ॐ	ॐ	ॐ	ॐ	ॐ	ॐ	ॐ	ॐ	ॐ	ॐ	ॐ																																																																																																																																										
U+093x	ॐ	ॐ	ॐ	ॐ	ॐ	ॐ	ॐ	ॐ	ॐ	ॐ	ॐ	ॐ	ॐ	ॐ	ॐ	ॐ																																																																																																																																										
U+094x	ॐ	ॐ	ॐ	ॐ	ॐ	ॐ	ॐ	ॐ	ॐ	ॐ	ॐ	ॐ	ॐ	ॐ	ॐ	ॐ																																																																																																																																										
U+095x	ॐ	ॐ	ॐ	ॐ	ॐ	ॐ	ॐ	ॐ	ॐ	ॐ	ॐ	ॐ	ॐ	ॐ	ॐ	ॐ																																																																																																																																										
U+096x	ॐ	ॐ	ॐ	ॐ	ॐ	ॐ	ॐ	ॐ	ॐ	ॐ	ॐ	ॐ	ॐ	ॐ	ॐ	ॐ																																																																																																																																										
U+097x	ॐ	ॐ	ॐ	ॐ	ॐ	ॐ	ॐ	ॐ	ॐ	ॐ	ॐ	ॐ	ॐ	ॐ	ॐ	ॐ																																																																																																																																										

3. Python Data Type – List

List is a versatile data type exclusive in Python. In a sense it is same as array in C/C++. But interesting thing about list in Python is it can simultaneously hold different type of data.

Formally list is a ordered sequence of some data written using squarebrackets ([]) and commas(,)

```
my_list = []  
# empty list
```

```
my_list = [1, 2, 3]  
# list of integers
```

```
my_list = [1, "Hello", 3.4]  
# list with mixed data types
```

Also, a list can even have another list as an item. This is called nested list.

```
# nested list  
my_list = ["mouse", [8, 4, 6], ['a']]
```

4. Python Tuple

Tuple is an ordered sequences of items same as list. The only difference is that tuples are immutable. Tuples once created cannot be modified.

Tuples are used to write-protect data and are usually faster than list as it cannot change dynamically. It is defined within parentheses () where items are separated by commas

```
>>> t = (50, 'Learning is fun', 1+3j, 45.67) # Can store mixed data types
```

Advantages of Tuple over List

- Since tuple are immutable, iterating through tuple is faster than with list. So there is a slight performance boost.
- We generally use tuple for heterogeneous (different) datatypes and list for homogeneous (similar) datatypes.
- Tuples that contain immutable elements can be used as key for a dictionary. With list, this is not possible.
- If you have data that doesn't change, implementing it as tuple will guarantee that it remains write-protected.

5. Python Set

Set is an unordered collection of unique items. Set is defined by values separated by comma inside braces { }. Items in a set are not ordered.

```
a = {5,2,3,1,4}
```

printing set variable

```
print("a = ", a)
```

data type of variable a

```
print(type(a))      # <class 'set'>
```

6. Python Dictionary

Dictionary is an unordered collection of key-value pairs.

It is generally used when we have a huge amount of data. Dictionaries are optimized for retrieving data. We must know the key to retrieve the value.

In Python, dictionaries are defined within Curly braces { } with each item being a pair in the form **key: value**.

Key and value can be of any type.

```
Python 3.6.5 Shell
File Edit Shell Debug Options Window Help
Python 3.6.5 (v3.6.5:f59c0932b4, Mar 28 2018, 16:07:46) [MSC v.1900 32 bit (Intel)] on win32
Type "copyright", "credits" or "license()" for more information.
>>> d1={'Name': 'Manasvi', 'Class': 9}
>>> d1
{'Name': 'Manasvi', 'Class': 9}
>>> d1['Name']
'Manasvi'
>>> d1['Class']
9
>>> |
```



Dictionary Values can be printed using key eg d1['Name']

Unit - 2

Python Control Statements

In any programming language a program may execute sequentially, selectively or iteratively. Every programming language provides constructs to support Sequence, Selection and Iteration. In Python all these construct can broadly categorized in 2 categories.

- A. Conditional Control Construct
(Selection, Iteration)
- B. Un- Conditional Control Construct
(pass, break, continue, exit(), quit())

Python have following types of control statements

1. Selection (branching) Statement
2. Iteration (looping) Statement
3. Jumping (break / continue)Statement

Conditional Control
Statements

Python Selection Statements

Un Conditional Control
Statements

Python have following types of selection statements

1. if statement
2. if else statement
3. Ladder if else statement (if-elif-else)
4. Nested if statement

Python If statements

This construct of python program consist of one if condition with one block of statements. When condition becomes true then executes the block given below it.

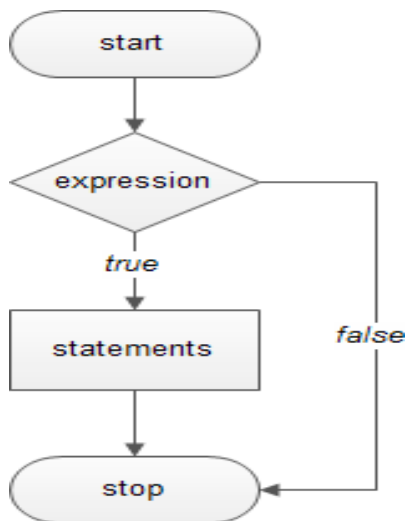
Syntax:




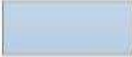

if (condition):

.....
.....
.....

Flow Chart: it is a graphical representation of steps an algorithm to solve a problem.

Flowchart



Symbol	Name	Function
	Start/end	An oval represents a start or end point
	Arrows	A line is a connector that shows relationships between the representative shapes
	Input/Output	A parallelogram represents input or output
	Process	A rectangle represents a process
	Decision	A diamond indicates a decision

Example:

```
Age=int(input("Enter Age: "))
```

```
If ( age>=18):
```

```
    Print("You are eligible for vote")
```

```
If(age<0):
```

```
    Print("You entered Negative Number")
```

Python if - else statements

This construct of python program consist of **one if condition with two blocks**. When condition becomes true then executes the block given below it. If condition evaluates result as false, it will executes the block given below else.

Syntax:

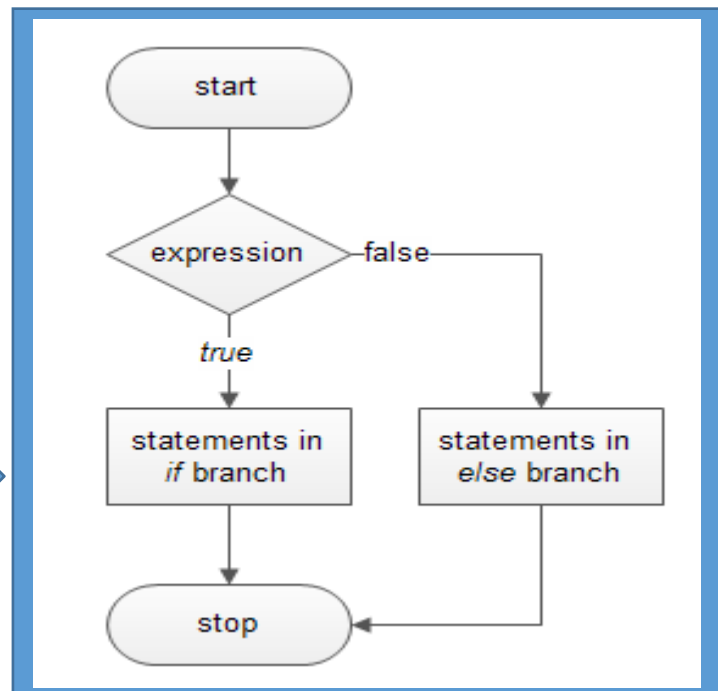
```
if ( condition):
```

```
    .....
```

```
else:
```

```
    .....
```

Flowchart 



Example-1:

```
Age=int(input("Enter Age: "))
```

```
if ( age>=18):
```

```
    print("You are eligible for vote")
```

```
else:
```

```
    print("You are not eligible for vote")
```

Example-2:

```
N=int(input("Enter Number: "))
```

```
if(n%2==0):
```

```
    print(N," is Even Number")
```

```
Else:
```

```
    print(N," is Odd Number")
```

Python Ladder if else statements (if-elif-else)

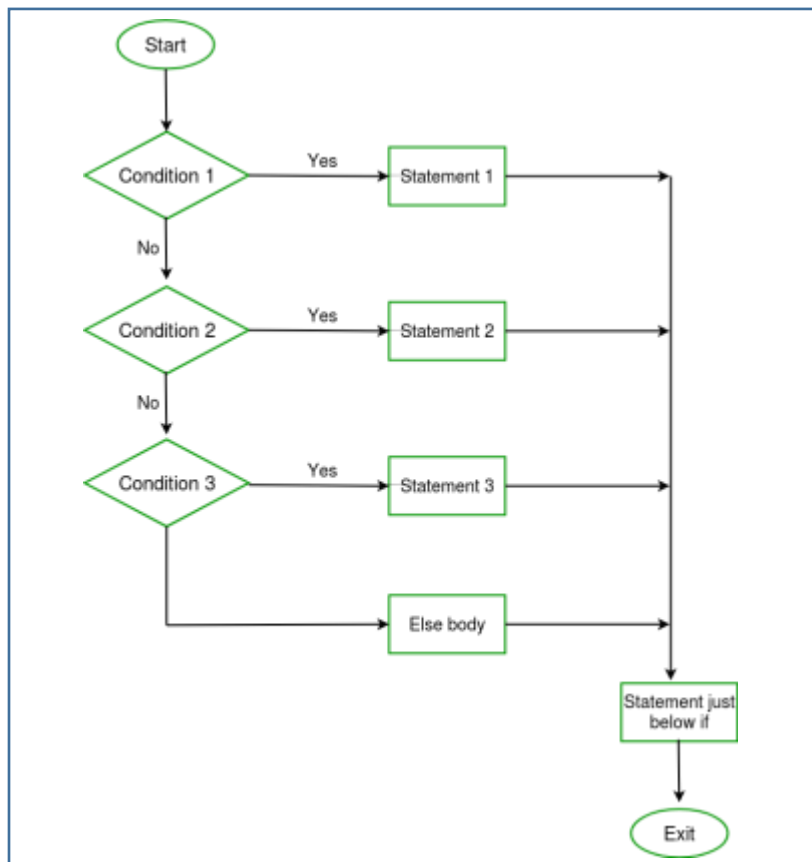
This construct of python program consist of **more than one if condition**. When first condition evaluates result as true then executes the block given below it. If condition evaluates result as false, it transfer the control at else part to test another condition. So, it is **multi-decision making** construct.

Syntax:

```
if ( condition-1):  
    .....  
    .....  
elif (condition-2):  
    .....  
    .....  
elif (condition-3):  
    .....  
    .....  
else:  
    .....  
    .....
```

Example:

```
num=int(input("Enter Number: "))  
If ( num>=0):  
    Print("You entered positive number")  
elif ( num<0):  
    Print("You entered Negative number")  
else:  
    Print("You entered Zero ")
```



Python Nested if statements

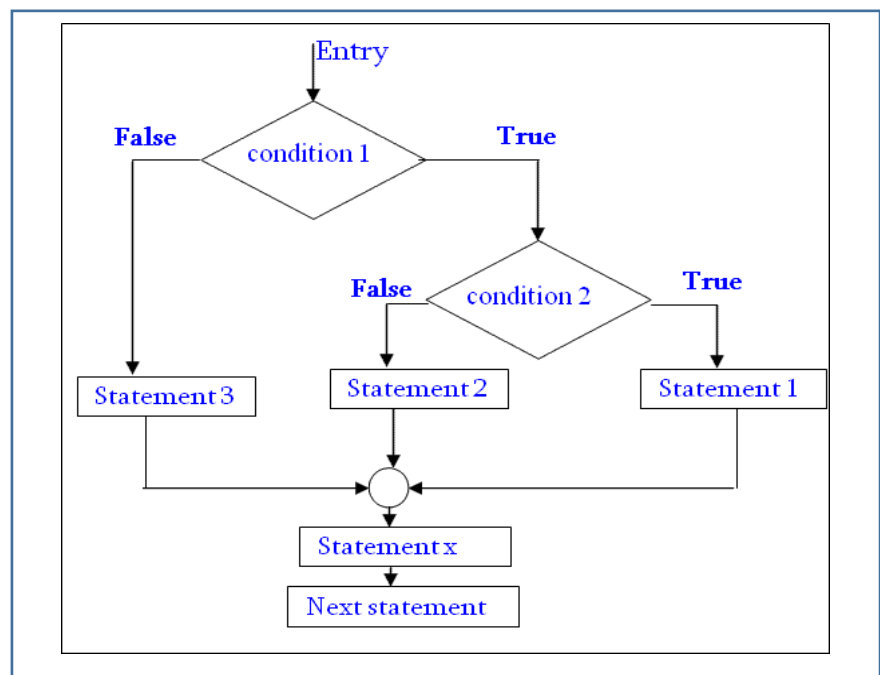
It is the construct where one if condition take part inside of other if condition. This construct consist of **more than one if condition**. Block executes when condition becomes false and next condition evaluates when first condition became true.

So, it is also **multi-decision making** construct.

Syntax:

```
if ( condition-1):  
    if (condition-2):  
        .....  
        .....  
    else:  
        .....  
        .....  
else:  
    .....  
    .....
```

FlowChart



Example:

```
num=int(input("Enter Number: "))  
If ( num<=0):  
    if ( num<0):  
        Print("You entered Negative number")  
    else:  
        Print("You entered Zero ")  
else:  
    Print("You entered Positive number")
```


Program: find largest number out of given three numbers

```
x=int(input("Enter First Number: "))
y=int(input("Enter Second Number: "))
z=int(input("Enter Third Number: "))
if(x>y and x>z):
    largest=x
elif(y>x and y>z):
    largest=y
elif(z>x and z>y):
    largest=z
print("Largest Value in %d, %d and %d is: %d"%(x,y,z,largest))
```

Program: calculate simple interest

Formula: principle x (rate/100) x time

```
p=float(input("Enter principle amount: "))
r=float(input("Enter rate of interest: "))
t=int(input("Enter time in months: "))
si=p*r*t/100
print("Simple Interest=",si)
```

Program: calculate EMI

Input the following to arrive at your Equal Monthly Installment -EMI:

1. Loan Amount: Input the desired loan amount that you wish to avail.
2. Loan Tenure (In Years): Input the desired loan term for which you wish to avail the loan.
3. Interest Rate (% P.A.): Input interest rate.
4. $EMI = \frac{P \cdot R \cdot (1+R)^N}{[(1+R)^N - 1]}$

```
P=int(input("Enter loan amount: "))
```

```

YR=float(input("Enter rate of interest P.A. : "))
T=int(input("Enter tenure(Installments) in years: "))
MR=YR/(12*100) # Monthly Rate
EMI=(P*MR*(1+MR)**T)/(((1+MR)**T)-1)
print("Principle Amount: ",P)
print("Rate of Interest(Yearly): ",YR)
print("No. of Installments: ",T)
print("EMI Amount: ",EMI)

```

Program: Sorting of three number. (Ascending and Descending)

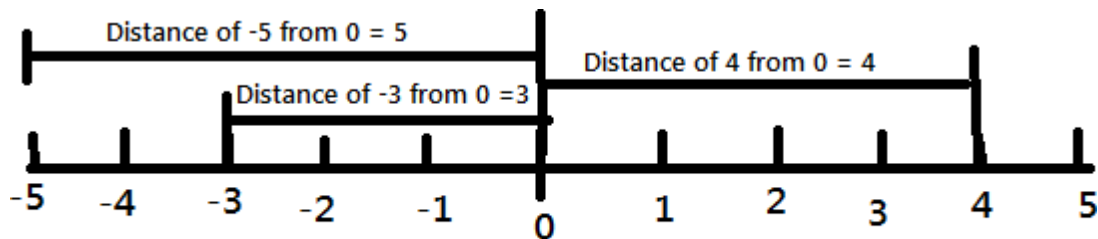
```

x=int(input("Enter First Number: "))
y=int(input("Enter Second Number: "))
z=int(input("Enter Third Number: "))
min=max=mid=None
if(x>=y and x>=z):
    if(y>=z):
        min,mid,max=z,y,x
    else:
        min,mid,max=y,z,x
elif(y>=x and y>=z):
    if(x>=z):
        min,mid,max=z,x,y
    else:
        min,mid,max=x,z,y
elif(z>=x and z>=y):
    if(x>=y):
        min,mid,max=y,x,z
    else:
        min,mid,max=x,y,z
print("Numbers in Ascending Order: ",min,mid,max)
print("Numbers in Descending Order: ",max,mid,min)

```

Program: Absolute Value

Absolute value of a given number is always measured as positive number. This number is the distance of given number from the 0(Zero). The input value may be integer, float or complex number in Python. The absolute value of given number may be integer or float.



Concept of Absolute Value

(i). Absolute Value of -5 is 5 (ii) Absolute Value of -3 is 3 (iii) Absolute Value of 4 is 4

```
n=float(input("Enter a number to find absolute value: "))
print("Absolute Value using abs(): ",abs(n))
if(n-int(n)>=0 or n-int(n)<=0): # This code is used to identify that number is float or int type.
    pass
else:
    n=int(n)
if(n<0):
    print("Absolute Value= ",n*-1)
else:
    print("Absolute Value= ",n)
```

Program: Calculate the Total selling price after levying the GST (Goods and Service Tax) as CGST and SGST on sale.

CGST (Central Govt. GST), SGST (State Govt. GST)

Sale amount	CGST Rate	SGST Rate
0-50000	5%	5%
Above 50000	18%	18%

```
amt=float(input("Enter total Sale Amount: "))
if(amt<=50000):
    rate=5
else:
    rate=18
cgst=sgst=amt*rate/100
tot_amt=amt+cgst+sgst
print("Amount of Sale: ",amt)
print("GST rate of Sale: ",rate)
print("CGST of Sale: ",cgst)
print("SGST of Sale: ",sgst)
print("Total Payable Amount of Sale: ",tot_amt)
```

Python Iteration Statements

The iteration (Looping) constructs mean to execute the block of statements again and again depending upon the result of condition. This repetition of statements continues till condition meets True result. As soon as condition meets false result, the iteration stops.

Python supports following types of iteration statements

1. while
2. for

Four Essential parts of Looping:

- i. Initialization of control variable
- ii. Condition testing with control variable
- iii. Body of loop Construct
- iv. Increment / decrement in control variable

Python while loop

The while loop is conditional construct that executes a block of statements again and again till given condition remains true. Whenever condition meets result false then loop will terminate.

Syntax:

Initialization of control variable

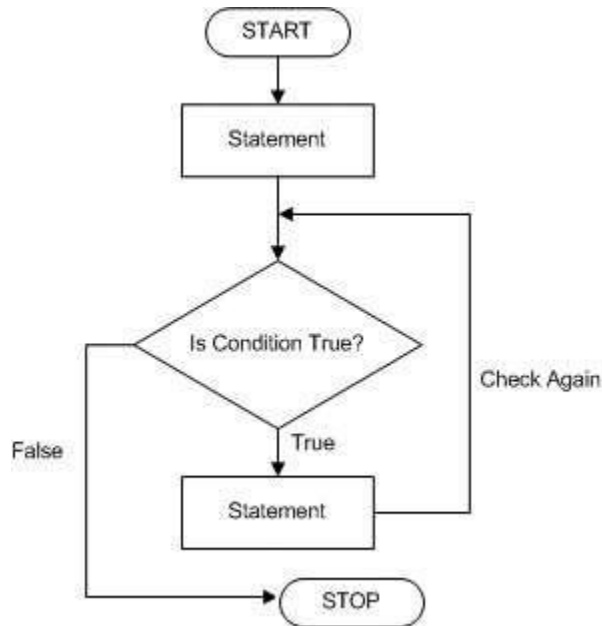
while (condition):

.....

Updation in control variable

.....

Flowchart



Example: print 1 to 10 numbers

```
num=1 # initialization
```

```
while(num<=10): # condition testing
```

```
    print(num, end=" ")
```

Body of loop

```
        num += 1 # Increment
```

Example: Sum of 1 to 10 numbers.

```
num=1
```

```
sum=0
```

```
while(num<=10):
```

```
    sum += num
```

```
    num += 1
```

```
print("The Sum of 1- 10 numbers: ",sum)
```

Example: Enter per day sale amount and find average sale for a week.

Python range() Function

The range() function returns a sequence of numbers, starting from 0 by default, and increments by 1 (by default), and ends at a specified number. The common format of range() is as given below:

range (start value, stop value, step value)

Where all 3 parameters are of integer type
Start value is Lower Limit
Stop value is Upper Limit
Step value is Increment / Decrement

Start and Step Parameters are optional default value will be as Start=0 and Step=1

Note: The Lower Limit is included but Upper Limit is not included in result.

Example

range(5) => sequence of 0,1,2,3,4
range(2,5) => sequence of 2,3,4
range(1,10,2) => sequence of 1,3,5,7,9
range(5,0,-1) => sequence of 5,4,3,2,1
range(0,-5) => sequence of [] blank list (default Step is +1)
range(0,-5,-1) => sequence of 0, -1, -2, -3, -4
range(-5,0,1) => sequence of -5, -4, -3, -2, -1
range(-5,1,1) => sequence of -5, -4, -3, -2, -1, 0

L=list(range(1,20,2))

Print(L) Output: [1, 3, 5, 7, 9, 11, 13, 15, 17, 19]

Python for loop

A for loop is used for iterating over a sequence (that is either a list, a tuple, a string etc.) With for loop we can execute a set of statements, and for loop can also execute once for each element in a list, tuple, set etc.

Example: print 1-10 numbers

```
for num in range(1,11,1):  
    print(num, end=" ")
```

Output: 1 2 3 4 5 6 7 8 9 10

Example: print 10-1 numbers

```
for num in range(10,0,-1):  
    print(num, end=" ")
```

Output: 10 9 8 7 6 5 4 3 2 1

Print each element in a fruit list:

```
fruits = ["mango", "apple", "grapes", "cherry"]
```

```
for x in fruits:
```

```
    print(x)
```

output:

```
mango  
apple  
grapes  
cherry
```

```
for x in "TIGER":
```

```
    print(x)
```

output:

```
T  
I  
G  
E  
R
```

Membership Operators:

The “in” and “not in” are membership operators. These operators check either given value is available in sequence or not. The “in” operator returns Boolean True result if value exist in sequence otherwise returns Boolean False.

The “not in” operator also returns Boolean True / False result but it works opposite to “in” operator.

else in for Loop

The **else** keyword in for loop specifies a block of code to be executed when the loop is finished:

```
for x in range(4):
    print(x, end=" ")
else:
    print("\nFinally finished!")
```

output: 0 1 2 3
Finally finished!

Nested Loops

A nested loop is a loop inside another loop.

```
city = ["Jaipur", "Delhi", "Mumbai"]
fruits = ["apple", "mango", "cherry"]
for x in city:
    for y in fruits:
        print(x, ":", y)
```

output:

```
Jaipur : apple
Jaipur : mango
Jaipur : cherry
Delhi : apple
Delhi : mango
Delhi : cherry
Mumbai : apple
Mumbai : mango
Mumbai : cherry
```

Un- Conditional Control Construct

(pass, break, continue, exit(), quit())

pass Statement (Empty Statement)

The pass statement do nothing, but it used to complete the syntax of programming concept. Pass is useful in the situation where user does not requires any action but syntax requires a statement. The Python compiler encounters pass statement then it do nothing but transfer the control in flow of execution.

```
a=int(input('Enter first Number: '))
b=int(input('Enter Second Number: '))
if(b==0):
    pass
else:
    print('a/b=',a/b)

for x in [0, 1, 2]:
    pass
```

Jumping Statements

break Statement

The jump- break statement enables to skip over a part of code that used in loop even if the loop condition remains true. It terminates to that loop in which it lies. The execution continues from the statement which find out of loop terminated by break.

```
n=1
while(n<=5):
    print("n=",n)
    k=1
    while(k<=5):
        if(k==3):
            break
        print("k=",k, end=" ")
        k+=1
    n+=1
    print()
```

Output:

```
n= 1
k= 1 k= 2
n= 2
k= 1 k= 2
n= 3
k= 1 k= 2
n= 4
k= 1 k= 2
n= 5
k= 1 k= 2
```

Exit the loop when x is "banana":

```
fruits = ["apple", "banana", "cherry"]
for x in fruits:
    if x == "banana":
        break
    print(x)
```

output: apple

Continue Statement

Continue statement is also a jump statement. With the help of continue statement, some of statements in loop, skipped over and starts the next iteration. It forcefully stop the current iteration and transfer the flow of control at the loop controlling condition.

```
i = 0
while i <=10:
    i+=1
    if (i%2==1):
        continue
    print(i, end=" ")
```

output: 2 4 6 8 10

```
fruits = ["apple", "banana", "cherry"]
for x in fruits:
    if x == "banana":
        continue
    print(x)
```

output:

apple
cherry

FILES, EXCEPTIONS, MODULES, PACKAGES

Files and exception: text files, reading and writing files, command line arguments, errors and exceptions, handling exceptions, modules (datetime, time, OS , calendar, math module), Explore packages.

Files, Exceptions, Modules, Packages:

Files and exception:

A **file** is some information or data which stays in the computer storage devices. Python gives you easy ways to manipulate these files. Generally files divide in two categories, text file and binary file. Text files are simple text where as the binary files contain binary data which is only readable by computer.

- **Text files:** In this type of file, Each line of text is terminated with a special character called EOL (End of Line), which is the new line character ('\n') in python by default.
- **Binary files:** In this type of file, there is no terminator for a line and the data is stored after converting it into machine understandable binary language.

An **exception** is an event, which occurs during the execution of a program that disrupts the normal flow of the program's instructions. In general, when a Python script encounters a situation that it cannot cope with, it raises an exception. An exception is a Python object that represents an error.

Text files:

We can create the text files by using the syntax:

Variable name=open ("file.txt", file mode) For

ex: `f= open ("hello.txt","w+")`

- We declared the variable f to open a file named hello.txt. **Open** takes 2 arguments, the file that we want to open and a string that represents the kinds of permission or operation we want to do on the file
- Here we used "w" letter in our argument, which indicates write and the plus sign that means it will create a file if it does not exist in library

- The available option beside "w" are "r" for read and "a" for append and plus sign means if it is not there then create it

File Modes in Python:

Mode	Description
'r'	This is the default mode. It Opens file for reading.
'w'	This Mode Opens file for writing. If file does not exist, it creates a new file.If file exists it truncates the file.
'x'	Creates a new file. If file already exists, the operation fails.
'a'	Open file in append mode. If file does not exist, it creates a new file.
't'	This is the default mode. It opens in text mode.
'b'	This opens in binary mode.
'+'	This will open a file for reading and writing (updating)

Reading and Writing files:

The following image shows how to create and open a text file in notepad from commandprompt

```
Microsoft Windows [Version 10.0.14393]
(c) 2016 Microsoft Corporation. All rights reserved.

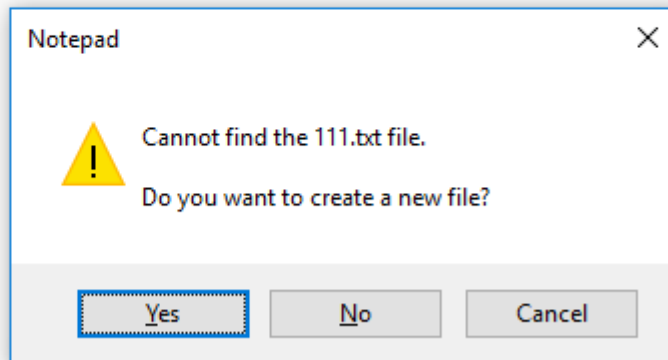
C:\Users\MRCET\AppData\Local\Programs\Python\Python38-32\filess>start notepad hello.txt

C:\Users\MRCET\AppData\Local\Programs\Python\Python38-32\filess>type hello.txt
Hello mrcet
good morning
how r u
C:\Users\MRCET\AppData\Local\Programs\Python\Python38-32\filess>
```

(or)

```
C:\Users\MRCET\AppData\Local\Programs\Python\Python38-32\filess>notepad 111.txt
```

Hit on enter then it shows the following whether to open or not?



Click on “yes” to open else “no” to cancel

Write a python program to open and read a file

```
a=open(“one.txt”,”r”) print(a.read())
```

a.close()

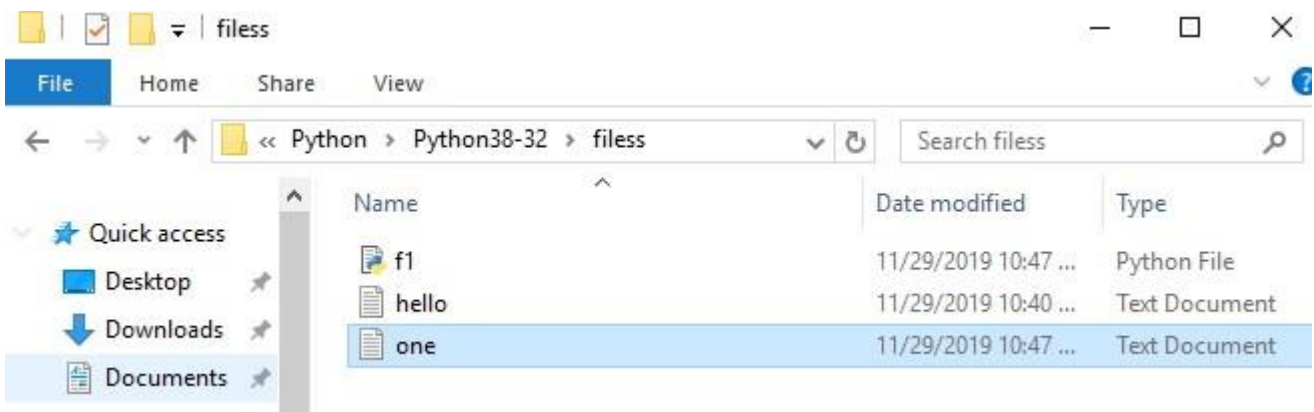
Output:

C:/Users/AppData/Local/Programs/Python/Python38-32/files/f1.py welcome to python programming

(or)

```
C:\Users\MRCET\AppData\Local\Programs\Python\Python38-32\files>python f1.py
welcome to python programming
```

Note: All the program files and text files need to saved together in a particular file then only the program performs the operations in the given file mode



f.close() ---- This will close the instance of the file somefile.txt stored # Write a

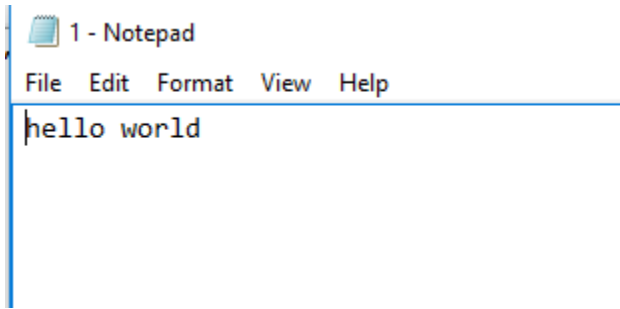
python program to open and write “hello world” into a file?

```
f=open("1.txt","a")
```

```
f.write("hello world")
```

```
f.close()
```

Output:



(or)

```
C:\Users\MRCET\AppData\Local\Programs\Python\Python38-32\files>type 1.txt  
hello world
```

Note: In the above program the 1.txt file is created automatically and adds hello world into txt file

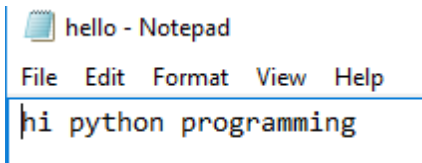
If we keep on executing the same program for more than one time then it appends the data many times

```
C:\Users\MRCET\AppData\Local\Programs\Python\Python38-32\files>type 1.txt  
hello world  
hello world
```

Write a python program to write the content “hi python programming” for the existing file.

```
f=open("1.txt",'w')  
f.write("hi python programming")  
f.close()
```

Output:



In the above program the hello.txt file consists of data like

```
C:\Users\MRCET\AppData\Local\Programs\Python\Python38-32\files>type hello.txt  
Hello mrcet  
good morning  
how r u
```

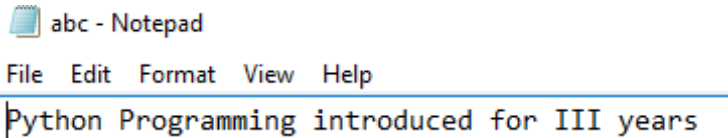
But when we try to write some data on to the same file it overwrites and saves with the current data (check output)

```
C:\Users\MRCET\AppData\Local\Programs\Python\Python38-32\files>type hello.txt
hi python programming
```

Write a python program to open and write the content to file and read it.

```
fo=open("abc.txt","w+")
fo.write("Python Programming")
print(fo.read())
fo.close()
```

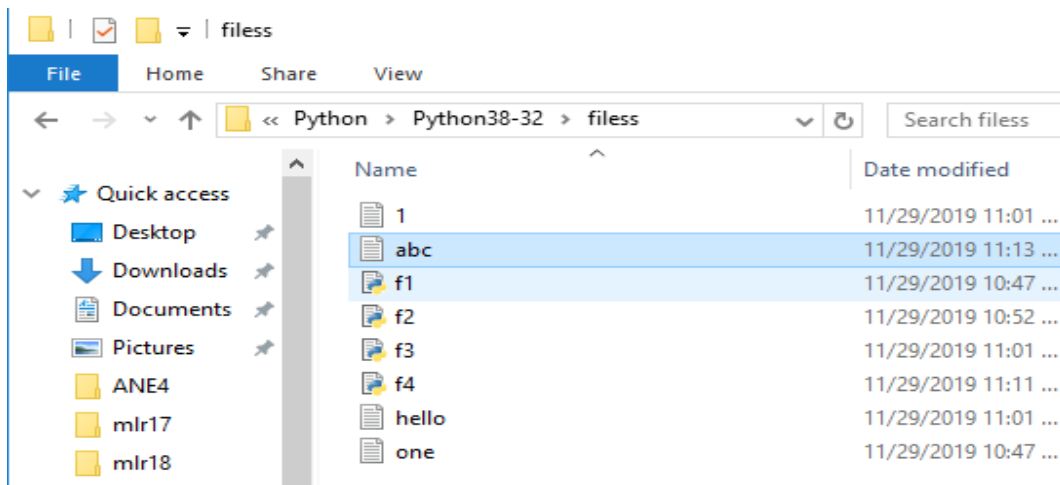
Output:



(or)

```
C:\Users\MRCET\AppData\Local\Programs\Python\Python38-32\files>type abc.txt
Python Programming introduced for III years
```

Note: It creates the abc.txt file automatically and writes the data into it



Command line arguments:

The command line arguments must be given whenever we want to give the input before the start of the script, while on the other hand, `raw_input()` is used to get the input while the python program / script is running.

The command line arguments in python can be processed by using either 'sys' module, 'argparse' module and 'getopt' module.

'sys' module :

Python sys module stores the command line arguments into a list, we can access it using `sys.argv`. This is very useful and simple way to read command line arguments as String.

sys.argv is the list of commandline arguments passed to the Python program. `argv` represents all the items that come along via the command line input, it's basically an array holding the command line arguments of our program

```
>>> sys.modules.keys() -- this prints so many dict elements in the form of list.
```

```
# Python code to demonstrate the use of 'sys' module for command line argumentsimport sys
```

```
# command line arguments are stored in the form# of list
```

```
in sys.argv
```

```
argumentList = sys.argv
```

```
print(argumentList)
```

```
# Print the name of file
```

```
print(sys.argv[0])
```

```
# Print the first argument after the name of file
```

```
#print(sys.argv[1])
```

Output:

```
C:/Users/AppData/Local/Programs/Python/Python38-32/cmdnlinarg.py
```

```
['C:/Users/AppData/Local/Programs/Python/Python38-32/cmdnlinarg.py']
```

```
C:/Users/AppData/Local/Programs/Python/Python38-32/cmdnlinarg.py
```

Note: Since my list consist of only one element at '0' index so it prints only that listelement, if we try to access at index position '1' then it shows the error like,

IndexError: list index out of range

```
import sys

print(type(sys.argv))
print('The command line arguments are:')for i in
sys.argv:
    print(i)
```

Output:

```
C:/User/AppData/Local/Programs/Python/Python38-32/symod.py ==
<class 'list'>
The command line arguments are: C:/Users/AppData/Local/Programs/Python/Python38-
32/symod.py
```

write a python program to get python version.

```
import sys
print("System version is:")
print(sys.version) print("Version
Information is:")
print(sys.version_info)
```

Output:

```
C:/Users/AppData/Local/Programs/Python/Python38-32/pyyy/s1.py =System version is:
3.8.0 (tags/v3.8.0:fa919fd, Oct 14 2019, 19:21:23) [MSC v.1916 32 bit (Intel)]
Version Information is:
sys.version_info(major=3, minor=8, micro=0, releaselevel='final', serial=0)
```

'argparse' module :

Python getopt module is very similar in working as the C getopt() function for parsing command-line parameters. Python getopt module is useful in parsing command line arguments where we want user to enter some options too.

```
>>> parser = argparse.ArgumentParser(description='Process some integers.')
```

```
import argparse
```

```
parser = argparse.ArgumentParser()  
print(parser.parse_args())
```

‘getopt’ module :

Python argparse module is the preferred way to parse command line arguments. It provides a lot of options such as positional arguments, default value for arguments, help message, specifying data type of argument etc

It parses the command line options and parameter list. The signature of this function is mentioned below:

```
getopt.getopt(args, shortopts, longopts=[ ])
```

- args are the arguments to be passed.
- shortopts is the options this script accepts.
- Optional parameter, longopts is the list of String parameters this function accepts which should be supported. Note that the -- should not be prepended with option names.

-h ----- print help and usage message

-m ----- accept custom option value

-d ----- run the script in debug mode

```
import getopt
```

```
import sys
```

```
argv = sys.argv[0:]
```

```
try:
```

```
    opts, args = getopt.getopt(argv, 'hm:d', ['help', 'my_file='])#print(opts)
```

```
    print(args)
```

```
except getopt.GetoptError:
```

```
    # Print a message or do something useful
```

```
    print('Something went wrong!') sys.exit(2)
```

Output:

C:/Users/AppData/Local/Programs/Python/Python38-32/gtopt.py ==

['C:/Users/AppData/Local/Programs/Python/Python38-32/gtopt.py']

Errors and Exceptions:

Python Errors and Built-in Exceptions: Python (interpreter) raises exceptions when it encounters **errors**. When writing a program, we, more often than not, will encounter errors. Error caused by not following the proper structure (syntax) of the language is called syntax error or parsing error

ZeroDivisionError:

ZeroDivisionError in Python indicates that the second argument used in a division (or modulo) operation was zero.

OverflowError:

OverflowError in Python indicates that an arithmetic operation has exceeded the limits of the current Python runtime. This is typically due to excessively large float values, as integer values that are too big will opt to raise memory errors instead.

ImportError:

It is raised when you try to import a module which does not exist. This may happen if you made a typing mistake in the module name or the module doesn't exist in its standard path. In the example below, a module named "non_existing_module" is being imported but it doesn't exist, hence an import error exception is raised.

IndexError:

An IndexError exception is raised when you refer a sequence which is out of range. In the example below, the list abc contains only 3 entries, but the 4th index is being accessed, which will result in an IndexError exception.

TypeError:

When two unrelated type of objects are combined, TypeError exception is raised. In example below, an int and a string is added, which will result in TypeError exception.

IndentationError:

Unexpected indent. As mentioned in the "expected an indentedblock" section, Python not only insists on indentation, it insists on consistent indentation. You are free to choose the number of spaces of indentation to use, but you then need to stick with it.

Syntax errors:

These are the most basic type of error. They arise when the Python parser is unable to understand a line of code. Syntax errors are almost always fatal, i.e. there is almost never a way to successfully execute a piece of code containing syntax errors.

Run-time error:

A run-time error happens when Python understands what you are saying, but runs into trouble when following your instructions.

Key Error :

Python raises a KeyError whenever a dict() object is requested (using the format a = adict[key]) and the key is not in the dictionary.

Value Error:

In Python, a value is the information that is stored within a certain object. To encounter a ValueError in Python means that is a problem with the content of the object you tried to assign the value to.

Python has many built-in exceptions which forces your program to output an error when something in it goes wrong. In Python, users can define such exceptions by creating a new class. This exception class has to be derived, either directly or indirectly, from Exception class.

Different types of exceptions:

- ArrayIndexOutOfBoundsException.
- ClassNotFoundException.
- FileNotFoundException.
- IOException.
- InterruptedException.
- NoSuchFieldException.
- NoSuchMethodException

HANDLING EXCEPTION

Exception

An Exception is an event which occurs during the execution of a program that disrupts the normal flow of the program's instructions. If a python script encounters a situation it cannot cope up, it raises an exception

HOW TO HANDLE EXCEPTION?

There are four blocks which help to handle the exception. They are

- try block
- except statement
- else block
- finally block

i) try block

- In the try block a programmer will write the suspicious code that may raise an exception. One can defend their program from a run time error by placing the codes inside the try block.

Syntax:

```
try:  
    #The operations here
```

ii) except statement

- Except statement should be followed by the try block.
- A single try block can have multiple except statement.
- The except statement which handles the exception.
- Multiple except statements require multiple exception names to handle the exception separately.

```
except Exception 1:  
    #Handle Exception 1  
except Exception 2:  
    #Handle Exception 2
```

iii) else block

If there is no exception in the program the else block will get executed.

Syntax:

```
else:  
    #If no Exception, it will execute
```


iv) finally block:

A finally block will always execute whether an exception happens or not the block will always execute.

```
finally:  
    #Always Execute
```

Syntax:

```
try:  
    #The operations here  
except Exception 1:  
    #Handle Exception 1  
except Exception 2:  
    #Handle Exception 2  
else:  
    #If no Exception, it will execute  
finally:  
    #Always Execute
```

(i) Write a python program to write a file with exception handling.

```
try:  
    f=open("test.txt", 'w+')  
    f.write("My First File")  
    f.seek(0)  
    print(f.read())  
except IOError:  
    print("File not Found")  
else:  
    print("File not Found")  
    f.close()  
finally:  
    print("Close a file")
```

(ii) Write a python program to read a file which raises an exception

Assume test.txt is not created in the computer and the following program is executed which raises an exception.

```

try:
    f=open("test.txt", 'w+')
    f.write("My First File")
    f.seek(0)
    print(f.read())
except IOError:
    print("File not Found")
else:
    print("File not Found")
    f.close()
finally:
    print("Close a file")

```

a) Except Clause with no exception

An except statement without the exception name, the python interpreter will consider as default 'Exception' and it will catch all exceptions.

Syntax:

```

try:
    #The operations here
except:
    #Handles Exception
else:
    #If no Exception, it will execute
finally:
    #Always Execute

```

b) Except clause with multiple Exception

An except statement can have multiple exceptions, We call it by exception name

Syntax:

```

try:
    #The operations here
except (Exception 1, Exception 2):
    #Handles Exception
else:
    #If no Exception, it will execute
finally:
    #Always Execute

```

(i) Write a python program to read a file with multiple exceptions

```
try:
    f=open("test.txt", 'w+')
    f=write("My First File")
    print(f.read())
except (IOError, ValueError,
ZeroDivisionError):
    print("File not Found")
else:
    print("File not Found")
    f.close()
finally:
    print("Close a file")
```

c) Argument of an Exception

An exception can have an argument which is a value that gives additional information about the problem. The content of an argument will vary by the exception.

```
try:
    #The operations here
except Exception Type, Argument:
    #Handles Exception with Argument
else:
    #If no Exception, it will execute
finally:
    #Always Execute
```

d) Raising an Exception

You can raise exceptions in several ways by using raise statement

Syntax:

```
raise Exception, Argument:
```

Example:

```
def fun(level):
    if level<10:
        raise "Invalid Level", level
fun(5) #raise an Exception
fun(11)
```

TYPES OF EXCEPTION

There are two types of Exception:

- Built-in Exception
- User Defined Exception

i) Built-in Exception

There are some built-in exceptions which should not be changed.

The Syntax for all Built-in Exception

```
except Exception_Name
```

<u>S.No</u>	<u>Exception Name</u>	<u>Description</u>
1	Exception	It is the Base class for all Exceptions
2	<u>ArithmeticError</u>	It is the Base class for all Errors that occur on numeric calculations.
3	ZeroDivisionError	Raised when a number is divided by zero
4	IOError	Raised when an Input or Output operation fails such as open() function. When a file is not exist in the folder
5	TypeError	Raised when operation or function is invalid for a specified data type.
6	ValueError	Raised when built-in function for a data type has valid arguments and it has invalid values.
7	RuntimeError	Raised when a generated error does not fall into any category.
8	KeyboardInterrupt	Raised when the user interrupts program execution by pressing <u>Ctrl+C</u>
9	<u>FloatingPointError</u>	Raised when floating calculation Fails.

<u>S.No</u>	<u>Exception Name</u>	<u>Description</u>
10	<u>AssertionError</u>	Raised in case of failure of assert statement.
11	<u>OverflowError</u>	Raised when a calculation exceeds maximum limit for a numeric types.
12	<u>StandardError</u>	Base class for all built-in exception except ' <u>StopIteration</u> and <u>SystemExit</u> '
13	StopIteration	Raised when next() method of an iteration does not exist
14	SystemExit	Raised by the <u>sys.exit()</u> function
15	SyntaxError	Raised when there is an error in Python Syntax
16	IndentationError	Raised when Indentation is not specified properly
17	<u>AssertionError</u>	Raised in case of failure of assert statement.

USER DEFINED EXCEPTION

In Python a user can create their own exception by deriving classes from standard Exceptions. There are two steps to create a user defined exception.

Step-1

A User Defined Exception should be derived from standard Built-in Exceptions.

Step-2

After referring base class the user defined exception can be used in the program

```
class NetworkError(RuntimeError):
def __init__ (self,arg):
    self.args=arg
try:
    raise NetworkError("Bad host name")
except NetworkError.e:
    print(e.args)
```

ASSERTION

An assertion is a sanity check which can turn on (or) turn off when the program is in testing mode.

- The assertion can be used by assert keyword. It will check whether the input is valid and after the operation it will check for valid output.

Syntax:

```
assert(Expression)
```

Example

```
def add(x):  
    assert(x<0)  
    return(x)  
add(10)
```

FUNCTIONS:

Function is a sub program which consists of set of instructions used to perform a specific task. A large program is divided into basic building blocks called function.

Need For Function:

- When the program is too complex and large they are divided into parts. Each part is separately coded and combined into single program. Each subprogram is called as function.
- Debugging, Testing and maintenance becomes easy when the program is divided into subprograms.
- Functions are used to avoid rewriting same code again and again in a program.
- Function provides code re-usability
- The length of the program is reduced.

Types of function:

Functions can be classified into two categories:

i) user defined function

ii) Built in function

i) Built in functions

- Built in functions are the functions that are already created and stored in python.
- These built in functions are always available for usage and accessed by a programmer. It cannot be modified.

Built in function Description

```
>>>max(3,4) # returns largest element 4
>>>min(3,4) # returns smallest element 3
>>>len("hello") #returns length of an object 5
>>>range(2,8,1) #returns range of given values [2, 3, 4, 5, 6, 7]
>>>round(7.8) #returns rounded integer of the given number 8.0
>>>chr(5) #returns a character (a string) from an integer '\x05'
>>>float(5) #returns float number from string or integer 5.0
>>>int(5.0) # returns integer from string or float 5
>>>pow(3,5) #returns power of given number 243
>>>type( 5.6) #returns data type of object to which it belongs <type 'float'>
>>>t=tuple([4,6.0,7]) # to create tuple of items from list (4, 6.0, 7)
>>>print("good morning") # displays the given object - Good morning
>>>input("enter name: ") # reads and returns the given string
enter name : George
```

ii)User Defined Functions:

- User defined functions are the functions that programmers create for their requirement and use.
- These functions can then be combined to form module which can be used in other programs by importing them.
- Advantages of user defined functions:
- Programmers working on large project can divide the workload by making different functions.

- If repeated code occurs in a program, function can be used to include those codes and execute when needed by calling that function.

Function definition: (Sub program)

- def keyword is used to define a function.
- Give the function name after def keyword followed by parentheses in which arguments are given.
- End with colon (:)
- Inside the function add the program statements to be executed
- End with or without return statement

Syntax:

```
def fun_name(Parameter1,Parameter2...Parameter n):  
statement1  
statement2...  
statement n  
return[expression]
```

Example:

```
def my_add(a,b):  
c=a+b  
return c
```

Function Calling: (Main Function)

- Once we have defined a function, we can call it from another function, program or even the Python prompt.
- To call a function we simply type the function name with appropriate arguments.

Example:

```
x=5  
y=4  
my_add(x,y)
```

Flow of Execution:

- The order in which statements are executed is called the flow of execution
- Execution always begins at the first statement of the program.
- Statements are executed one at a time, in order, from top to bottom.
- Function definitions do not alter the flow of execution of the program, but remember that statements inside the function are not executed until the function is called.
- Function calls are like a bypass in the flow of execution. Instead of going to the next statement, the flow jumps to the first line of the called function, executes all the statements there, and then comes back to pick up where it left off.

Note: When you read a program, don't read from top to bottom. Instead, follow the flow of execution. This means that you will read the def statements as you are scanning from top to bottom, but you should skip the statements of the function definition until you reach a point where that function is called.

Function Prototypes:

- i. Function without arguments and without return type
- ii. Function with arguments and without return type
- iii. Function without arguments and with return type
- iv. Function with arguments and with return type

i) Function without arguments and without return type

In this type no argument is passed through the function call and no output is return to main function.

The sub function will read the input values perform the operation and print the result in the same block.

ii) Function with arguments and without return type

Arguments are passed through the function call but output is not return to the main function

iii) Function without arguments and with return type

In this type no argument is passed through the function call but output is return to the main function.

iv) Function with arguments and with return type

In this type arguments are passed through the function call and output is return to the main function.

Without Return Type

Without argument

```
def add():  
    a=int(input("enter a"))  
    b=int(input("enter b"))  
    c=a+b  
    print(c)
```

add()

OUTPUT:

```
enter a 5  
enter b 10  
15
```

With return type

Without argument

```
def add():  
    a=int(input("enter a"))  
    b=int(input("enter b"))  
    c=a+b  
    return c
```

c=add()

print(c)

OUTPUT:

```
enter a 5
```

With argument

```
def add(a,b):  
    c=a+b  
    print(c)  
a=int(input("enter a"))  
b=int(input("enter b"))
```

add(a,b)

OUTPUT:

```
enter a 5  
enter b 10  
15
```

With argument

```
def add(a,b):  
    c=a+b  
    return c  
a=int(input("enter a"))  
b=int(input("enter b"))
```

c=add(a,b)

print(c)

OUTPUT:

```
enter a 5
```

enter b 10
15

enter b 10
15

Parameters And Arguments:

Parameters:

- Parameters are the value(s) provided in the parenthesis when we write function header.
- These are the values required by function to work.
- If there is more than one value required, all of them will be listed in parameter list separated by comma.
- Example: `def my_add(a,b):`

Arguments :

- Arguments are the value(s) provided in function call/invoke statement.
- List of arguments should be supplied in same way as parameters are listed.
- Bounding of parameters to arguments is done 1:1, and so there should be same number and type of arguments as mentioned in parameter list.
- Example: `my_add(x,y)`

6.7 RETURN STATEMENT:

- The return statement is used to exit a function and go back to the place from where it was called.
- If the return statement has no arguments, then it will not return any values. But exits from function.

Syntax:

```
return[expression]
```

Example:

```
def my_add(a,b):  
    c=a+b  
    return c
```

```
x=5
```

```
y=4
```

```
print(my_add(x,y))
```

Output:

```
9
```

ARGUMENTS TYPES:

1. Required Arguments
2. Keyword Arguments
3. Default Arguments
4. Variable length Arguments

- Required Arguments: The number of arguments in the function call should match exactly with the function definition.

```
def my_details( name, age ):  
    print("Name: ", name)
```

```
        print("Age ", age)
    return
my_details("george",56)
```

Output:

Name: george

Age 56

□ Keyword Arguments:

Python interpreter is able to use the keywords provided to match the values with parameters even though if they are arranged in out of order.

```
def my_details( name, age ):
    print("Name: ", name)
    print("Age ", age)
    return
```

```
my_details(age=56,name="george")
```

Output:

Name: george

Age 56

□ Default Arguments:

Assumes a default value if a value is not provided in the function call for that argument.

```
def my_details( name, age=40 ):
    print("Name: ", name)
    print("Age ", age)
    return
```

```
my_details(name="george")
```

Output:

Name: george

Age 40

□ Variable length Arguments

If we want to specify more arguments than specified while defining the function, variable length arguments are used. It is denoted by * symbol before parameter.

```
def my_details(*name ):
    print(*name)
    my_details("rajan","rahul","micheal",ärjun")
```

Output:

rajan rahul micheal ärjun

What is Python Module

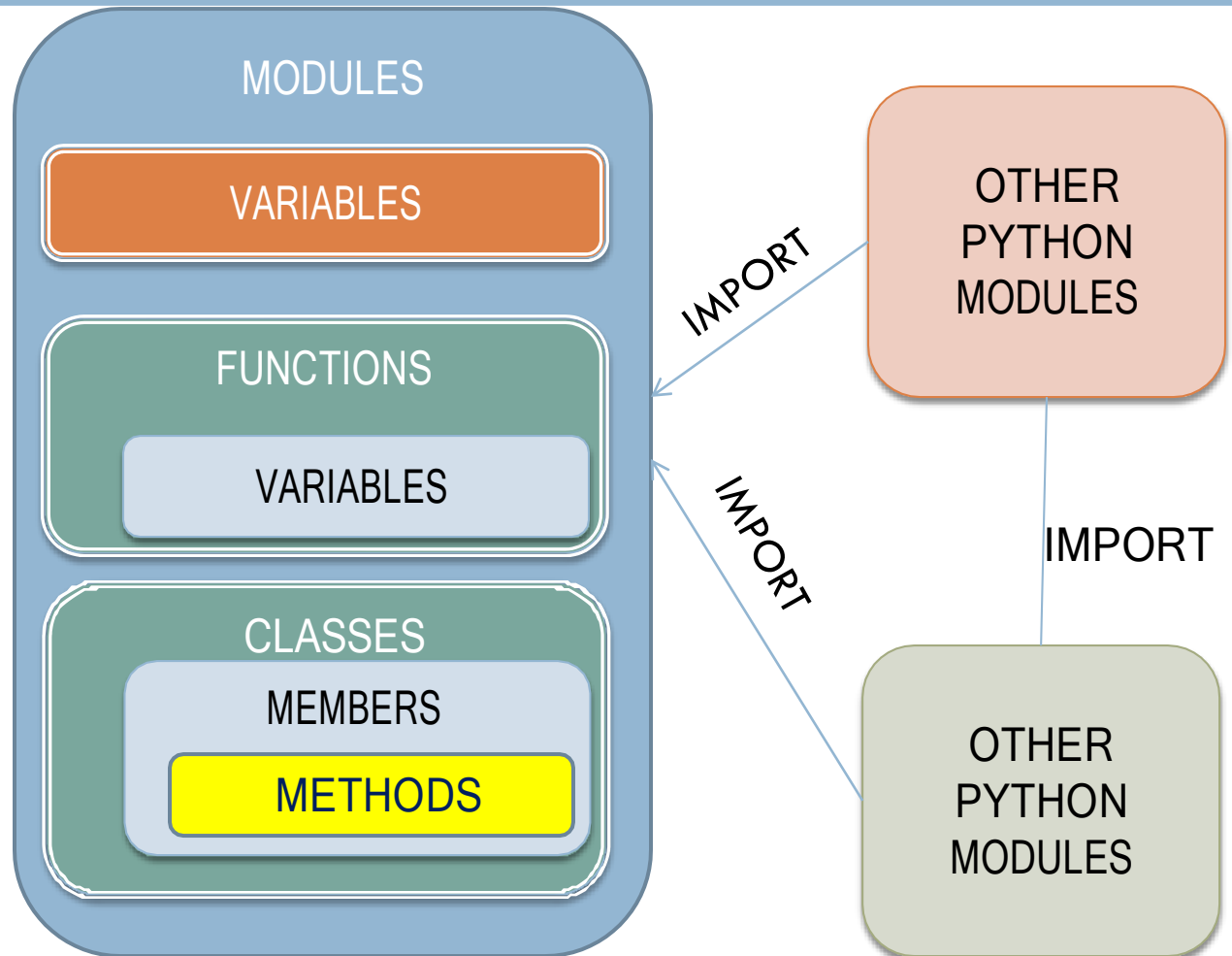
- A Module is a file containing Python definitions (docstrings) , functions, variables, classes and statements.
- Act of partitioning a program into individual components(modules) is called modularity. A module is a separate unit in itself.
 - ▮ It reduces its complexity to some degree
 - ▮ It creates numbers of well-defined, documented boundaries within program.
 - ▮ Its contents can be reused in other program, without having to rewrite or recreate them.

Structure of Python module

- A python module is simply a normal python file(.py) and contains functions, constants and other elements.
- Python module may contains following objects:

docstring	Triple quoted comments. Useful for documentation Purpose
Variables and constants	For storing values
Classes	To create blueprint of any object
Objects	Object is an instance of class. It represent class in real world
Statements	Instruction
Functions	Group of statements

Composition/Structure of python module



Importing Python modules

- To import entire module
 - `import <module name>`
 - **Example:** `import math`
- To import specific function/object from module:
 - `from <module_name> import <function_name>`
 - **Example:** `from math import sqrt`
- `import *` : can be used to import all names from module into current calling module

Accessing function/constant of imported module

- To use function/constant/variable of imported module we have to specify module name and function name separated by dot(.). This format is known as dot notation.
 - `<module_name>.<function_name>`
 - **Example:** `print(math.sqrt(25))`
- However if only particular function is imported using **from** then module name before function name is not required. We will see examples with next slides.

Types of Modules

- There are various in-built module in python, we will discuss few of them
 - ▮ Math module
 - ▮ Random module
 - ▮ Statistical module

Math module

- This module provides various function to perform arithmetic operations.
- Example of functions in math modules are:

sqrt	ceil	floor	pow
fabs	sin	cos	tan

- Example of variables in math modules are:
 - ▮ pi
 - ▮ e

Math module functions

- **sqrt(x)** : this function returns the square root of number(x).

```
>>> import math
>>> print(math.sqrt(49))
7.0
```

module name is required before function name here

- **pow(x,y)** : this function returns the $(x)^y$

```
>>> from math import pow
>>> print(pow(2, 6))
64.0
```

module name is not required before function name here

- **ceil(x)** : this function return the x rounded to next integer.

```
>>> import math
>>> print(math.ceil(45.25))
46
```

Math module functions

- **floor(x)** : this function returns the x rounded to previous integer.

```
>>> import math
>>> print(math.floor(5.9))
5
```

- **fabs(x)** : this function returns absolute value of float x. absolute value means number without any sign

```
>>> import math
>>> a=-8.5
>>> print(math.fabs(a))
8.5
```

Math module functions

- **cos(x)** : it return cosine of x (measured in radian)

```
>>> import math
>>> print(math.cos(90))
-0.4480736161291701
```

- **tan(x)** : it return tangent of x (measured in radian)

```
>>> import math
>>> print(math.tan(45))
1.6197751905438615
```

- **pi** : return the constant value of pi (22/7)

```
>>> print(math.pi)
3.141592653589793
```

- **e** : return the constant value of constant e

```
>>> print(math.e)
2.718281828459045
```

Using Random Module

- Python has a module namely random that provides random - number generators. Random number means any number generated within the given range.
- To generate random number in Python we have to import random module
- 2 most common method to generate random number in python are :
 - random() function
 - randint(a,b) function

random() function

- It is floating point random number generator between 0.0 to 1.0. here lower limit is inclusive where as upper limit is less than 1.0.
- $0 \leq N < 1$
- Examples:

```
>>> import random
>>> a = random.random()
>>> print(a)
0.08888880146536
>>> |
```

Output is less than 1

random() function

- To generate random number between given range of values using random(), the following format should be used:
 - ▮ $\text{Lower_range} + \text{random()} * (\text{upper_range} - \text{lower_range})$
 - ▮ For example to generate number between 10 to 50:
 - $10 + \text{random()} * (40)$

randint() function

- Another way to generate random number is randint() function, but it generate integer numbers.
- Both the given range values are inclusive i.e. if we generate random number as :
 - ▮ randint(20,70)
 - In above example random number between 20 to 70 will be taken. (including 20 and 70 also)

```
>>> import random
>>> a = random.randint(10,20)
>>> print(a)
18
>>> .
```

randrange() function

- This function is also used to generate random number within given range.
- Syntax
 - ▮ randrange(start,stop,step)

```
import random
n1 = random.randrange(5,15)
n2 = random.randrange(5,15)
n3 = random.randrange(5,15)
n4 = random.randrange(5,15)
print(n1,n2,n3,n4)
```

```
11 8 5 12
```

It will generate random number between 5 to 14

random output between 5 to 14, may vary

randrange() function

```
import random
for i in range(20):
    n1 = random.randrange(1, 30, 2)
    print(n1, end='\t')
```

It will generate random number between 1 to 29 with stepping of 2 i.e. it will generate number with gap of 2 i.e. 1,3,5,7 and so on

```
25    11    15    9     3     7     19    13    17    7
27    11    27    5     21    7     17    9     25    7
```

Statistical Module

- This module provides functions for calculating mathematical statistics of numeric (Real-valued) data.
- We will deal with 3 basic function under this module
 - ▮ Mean
 - ▮ Median
 - ▮ mode

Mean

- The mean is the average of all numbers and is sometimes called the arithmetic mean.

```
>>> import statistics
>>> mynum = [10,20,30,40,50,60,70,80,90,100]
>>> mymean = statistics.mean(mynum)
>>> print(mymean)
55
```

55, is the average of all numbers in the list

Median

- The median is the middle number in a group of numbers.

```
>>> import statistics
>>> mynum = [10,20,30,40,50,60,70,80,90]
>>> mymedian = statistics.median(mynum)
>>> print(mymedian)
50
```

With odd number of elements it will simply return the middle position value

```
>>> import statistics
>>> mynum = [10,20,30,40,50,60,70,80,90,100]
>>> mymedian = statistics.median(mynum)
>>> print(mymedian)
55.0
```

With even number of elements, it will return the average of value at mid + mid-1 i.e. $(50+60)/2 = 55.0$

Mode

- The mode is the number that occurs most often within a set of numbers i.e. most common data in list.

```
>>> import statistics
>>> mynum = [10,20,10,40,20,10,70,80,90]
>>> mymode = statistics.mode(mynum)
>>> print(mymode)
10
```

Here, 10 occurs most in the list.

UNIT-3

INTRODUCTION

A sequence is an orderly collection of items and each item is indexed by an integer. Following sequence data types in Python were also briefly introduced.

- Strings
- Lists
- Tuples
- Another data type 'Dictionary'

STRINGS

String is a sequence which is made up of one or more UNICODE characters. Here the character can be a letter, digit, whitespace or any other symbol. A string can be created by enclosing one or more characters in single, double or triple quote.

Example

```
>>> str1 = 'Hello World!'
>>> str2 = "Hello World!"
>>> str3 = """Hello World!"""
>>> str4 = '''Hello World!'''
```

str1, str2, str3, str4 are all string variables having the same value 'Hello World!'. Values stored in str3 and str4 can be extended to multiple lines using triple codes as can be seen in the following example:

```
>>> str3 = """Hello World!
welcome to the world of Python"""
>>> str4 = '''Hello World!
welcome to the world of Python'''
```

Accessing Characters in a String

Each individual character in a string can be accessed using a technique called indexing. The index specifies the character to be accessed in the string and is written in square brackets ([]). The index of the first character (from left) in the string is 0 and the last character is n-1 where n is the length of the string. If we give index value out of this range then we get an IndexError. The index must be an integer (positive, zero or negative).

#initializes a string str1

```
>>> str1 = 'Hello World!'
#gives the first character of str1
>>> str1[0] 'H'
#gives seventh character of str1
>>> str1[6] 'W'
#gives last character of str1
>>> str1[11] '!'
#gives error as index is out of range
>>> str1[15]
```

IndexError: string index out of range

The index can also be an expression including variables and operators but the expression must evaluate to an integer.

#an expression resulting in an integer index #so gives 6th character of str1

```
>>> str1[2+4] 'W'
```

#gives error as index must be an integer

```
>>> str1[1.5]
```

TypeError: string indices must be integers

Python allows an index value to be negative also. Negative indices are used when we want to access the characters of the string from right to left. Starting from right hand side, the first character has the index as -1 and the last character has the index $-n$ where n is the length of the string. Table below shows the indexing of characters in the string 'Hello World!' in both the cases, i.e., positive and negative indices.

```
>>> str1[-1] #gives first character from right
```

```
'!'
```

```
>>> str1[-12] #gives last character from right 'H'
```

Table Indexing of characters in string 'Hello World!'

Positive Indices	0	1	2	3	4	5	6	7	8	9	10	11
String	H	e	l	l	o		W	o	r	l	d	!
Negative Indices	-12	-11	-10	-9	-8	-7	-6	-5	-4	-3	-2	-1

An inbuilt function len() in Python returns the length of the string that is passed as parameter. For example, the length of string str1 = 'Hello World!' is 12.

#gives the length of the string str1

```
>>> len(str1) 12
```

#length of the string is assigned to n

```
>>> n = len(str1)
```

```
>>> print(n) 12
```

#gives the last character of the string

```
>>> str1[n-1] '!' 
```

#gives the first character of the string

```
>>> str1[-n] 'H'
```

String is Immutable

A string is an immutable data type. It means that the contents of the string cannot be changed after it has been created. An attempt to do this would lead to an error.

```
>>> str1 = "Hello World!"
```

#if we try to replace character 'e' with 'a'

```
>>> str1[1] = 'a'
```

TypeError: 'str' object does not support item assignment

STRING OPERATIONS

As we know that string is a sequence of characters. Python allows certain operations on string data type, such as concatenation, repetition, membership and slicing. These operations are

explained in the following subsections with suitable examples.

Concatenation

To concatenate means to join. Python allows us to join two strings using concatenation operator plus which is denoted by symbol +.

```
>>> str1 = 'Hello'    #First string
>>> str2 = 'World!'  #Second string
>>> str1 + str2      #Concatenated strings 'HelloWorld!'
#str1 and str2 remain same
>>> str1             #after this operation. 'Hello'
>>> str2 'World!'
```

Repetition

Python allows us to repeat the given string using repetition operator which is denoted by symbol *.

```
#assign string 'Hello' to str1
>>> str1 = 'Hello'
#repeat the value of str1 2 times
>>> str1 * 2 'HelloHello'
#repeat the value of str1 5 times
>>> str1 * 5 'HelloHelloHelloHelloHello'
```

Note: str1 still remains the same after the use of repetition operator.

Membership

Python has two membership operators 'in' and 'not in'. The 'in' operator takes two strings and returns True if the first string appears as a substring in the second string, otherwise it returns False.

```
>>> str1 = 'Hello World!'
>>> 'W' in str1 True
>>> 'Wor' in str1 True
>>> 'My' in str1 False
```

The 'not in' operator also takes two strings and returns True if the first string does not appear as a substring in the second string, otherwise returns False.

```
>>> str1 = 'Hello World!'
>>> 'My' not in str1 True
>>> 'Hello' not in str1 False
```

Slicing

In Python, to access some part of a string or substring, we use a method called slicing. This can be done by specifying an index range. Given a string str1, the slice operation str1 [n: m] returns the part of the string str1 starting from index n (inclusive) and ending at m (exclusive). In other words, we can say that str1 [n: m] returns all the characters starting from str1 [n] till str1 [m-1]. The numbers of characters in the substring will always be equal to difference of two indices m and n, i.e., (m-n).

```
>>> str1 = 'Hello World!'
#gives substring starting from index 1 to 4
```

```

>>> str1[1:5]
'ello'
#gives substring starting from 7 to 9
>>> str1[7:10]
'orl'
#index that is too big is truncated down to #the end of the string
>>> str1[3:20]
'lo World!'
#first index > second index results in an
#empty " string
>>> str1[7:2]
If the first index is not mentioned, the slice starts
from index.
#gives substring from index 0 to 4
>>> str1[:5]
'Hello'
If the second index is not mentioned, the slicing is done till the length of the string.
#gives substring from index 6 to end
>>> str1[6:]
'World!'

```

The slice operation can also take a third index that specifies the ‘step size’. For example, str1[n:m:k], means every kth character has to be extracted from the string str1 starting from n and ending at m-1. By default, the step size is one.

```

>>> str1[0:10:2]
'HloWr'
>>> str1[0:10:3]
'HIWI'
Negative indexes can also be used for slicing. #characters at index -6,-5,-4,-3 and -2 are #sliced
>>> str1[-6:-1]
'World'

```

If we ignore both the indexes and give step size as -1
 #str1 string is obtained in the reverse order

```

>>> str1[::-1]
'dlroW olleH'

```

TRAVERSING A STRING

We can access each character of a string or traverse a string using for loop and while loop.

String Traversal Using for Loop:

```

>>> str1 = 'Hello World!'
>>> for ch in str1:
print(ch,end = ")
Hello World! #output of for loop

```

In the above code, the loop starts from the first character of the string str1 and automatically ends when the last character is accessed.

String Traversal Using while Loop:

```
>>> str1 = 'Hello World!'
```

```
>>> index = 0
```

#len(): a function to get length of string

```
>>> while index < len(str1):
```

```
print(str1[index],end = ") index += 1
```

Hello World! #output of while loop

Here while loop runs till the condition index

< len(str) is True, where index varies from 0 to len(str1) -1.

STRING METHODS AND BUILT-IN FUNCTIONS Python has several built-in functions that allow us to work with strings. Table below describes some of the commonly used built-in functions for string manipulation.

Table Built-in functions for string manipulations

Method	Description	Example
len()	Returns length of the given string	>>> str1 = 'Hello World!' len(str1) >>> 12
title()	Returns the string with first letter of every word in the string in uppercase and rest in lowercase	>>> str1 = 'hello WORLD!' >>> str1.title() 'Hello World!'

lower()	Returns the string with all uppercase letters converted to lowercase	>>> str1 = 'hello WORLD!' >>> str1.lower() 'hello world!'
upper()	Returns the string with all lowercase letters converted to uppercase	>>> str1 = 'hello WORLD!' >>> str1.upper() 'HELLO WORLD!'
count(str, start, end)	Returns number of times substring str occurs in the given string. If we do not give start index and end index then searching starts from index 0 and ends at length of the string	>>> str1 = 'Hello World! Hello Hello' >>> str1.count('Hello',12,25) 2 >>> str1.count('Hello') 3
find(str,start, end)	Returns the first occurrence of index of substring str occurring in the given string. If we do not give start and end then searching starts from index 0 and ends at length of the string. If the	>>> str1 = 'Hello World! Hello Hello' >>> str1.find('Hello',10,20) 13 >>> str1.find('Hello',15,25) 19

	substring is not present in the given string, then the function returns -1	<pre>>>> str1.find('Hello') 0 >>> str1.find('Hee') -1</pre>
index(str, start, end)	Same as find() but raises an exception if the substring is not present in the given string	<pre>>>> str1 = 'Hello World! Hello Hello' >>> str1.index('Hello') 0 >>> str1.index('Hee') ValueError: substring not found</pre>
endswith()	Returns True if the given string ends with the supplied substring otherwise returns False	<pre>>>> str1 = 'Hello World!' >>> str1.endswith('World!') True >>> str1.endswith('l') True >>> str1.endswith('lde') False</pre>
startswith()	Returns True if the given string starts with the supplied substring otherwise returns False	<pre>>>> str1 = 'Hello World!' >>> str1.startswith('He') True >>> str1.startswith('Hee') False</pre>
isalnum()	Returns True if characters of the given string are either alphabets or numeric. If whitespace or special symbols are part of the given string or the string is empty it returns False	<pre>>>> str1 = 'HelloWorld' >>> str1.isalnum() True >>> str1 = 'HelloWorld2' >>> str1.isalnum() True >>> str1 = 'HelloWorld!!' >>> str1.isalnum() False</pre>
islower()	Returns True if the string is non-empty and has all lowercase alphabets, or has at least one character as lowercase alphabet and rest are non-alphabet characters	<pre>>>> str1 = 'hello world!' >>> str1.islower() True >>> str1 = 'hello 1234' >>> str1.islower() True >>> str1 = 'hello ??'</pre>

		<pre>>>> str1.islower() True >>> str1 = '1234' >>> str1.islower() False >>> str1 = 'Hello World!' >>> str1.islower() False</pre>
isupper()	Returns True if the string is non-empty and has all uppercase alphabets, or has at least one character as uppercase character and rest are non-alphabet characters	<pre>>>> str1 = 'HELLO WORLD!' >>> str1.isupper() True >>> str1 = 'HELLO 1234' >>> str1.isupper() True >>> str1 = 'HELLO ??' >>> str1.isupper() True >>> str1 = '1234' >>> str1.isupper() False >>> str1 = 'Hello World!' >>> str1.isupper() False</pre>
isspace()	Returns True if the string is non-empty and all characters are white spaces (blank, tab, newline, carriage return)	<pre>>>> str1 = ' \n \t \r' >>> str1.isspace() True >>> str1 = 'Hello \n' >>> str1.isspace() False</pre>
istitle()	Returns True if the string is non-empty and title case, i.e., the first letter of every word in the string in uppercase and rest in lowercase	<pre>>>> str1 = 'Hello World!' >>> str1.istitle() True >>> str1 = 'hello World!' >>> str1.istitle() False</pre>
lstrip()	Returns the string after removing the spaces only on the left of the string	<pre>>>> str1 = ' Hello World! ' >>> str1.lstrip() 'Hello World! '</pre>

rstrip()	Returns the string after removing the spaces only on the right of the string	>>> str1 = ' Hello World!' >>> str1.rstrip() ' Hello World!'
strip()	Returns the string after removing the spaces both on the left and the right of the string	>>> str1 = ' Hello World!' >>> str1.strip() 'Hello World!'
replace(oldstr, newstr)	Replaces all occurrences of old string with the new string	>>> str1 = 'Hello World!' >>> str1.replace('o','*') 'Hell* W*rld!' >>> str1 = 'Hello World!' >>> str1.replace('World','Country') 'Hello Country!' >>> str1 = 'Hello World! Hello' >>> str1.replace('Hello','Bye') 'Bye World! Bye'
join()	Returns a string in which the characters in the string have been joined by a separator	>>> str1 = ('HelloWorld!') >>> str2 = '-' #separator >>> str2.join(str1) 'H-e-l-l-o-W-o-r-l-d-!'
partition()	Partitions the given string at the first occurrence of the substring (separator) and returns the string partitioned into three parts. Substring before the separator Separator Substring after the separator If the separator is not found in the string, it returns the whole string itself and two empty strings	>>> str1 = 'India is a Great Country' >>> str1.partition('is') ('India ', 'is', ' a Great Country') >>> str1.partition('are') ('India is a Great Country', '', '')
split()	Returns a list of words delimited by the specified substring. If no delimiter is given then words are separated by space.	>>> str1 = 'India is a Great Country' >>> str1.split() ['India','is','a','Great','Country'] >>> str1 = 'India is a Great Country' >>> str1.split('a') ['Indi', ' is ', ' Gre', ' t Country']

HANDLING STRINGS

In this section, we will learn about user defined functions in Python to perform different operations on strings.

Program Write a program with a user defined function to count the number of times a character (passed as argument) occurs in the given string.

#Program 1

#Function to count the number of times a character occurs in a #string

```
def charCount(ch,st):
count = 0
for character in st:
if character == ch:
count += 1
return count #end of function
st = input("Enter a string: ")
ch = input("Enter the character to be searched: ")
count = charCount(ch,st)
print("Number of times character",ch,"occurs in the string is:",count)
```

Output:

Enter a string: Today is a Holiday

Enter the character to be searched: a

Number of times character a occurs in the string is: 3

Program Write a program with a user defined function with string as a parameter which replaces all vowels in the string with '*'.
all vowels in the string with '*'.

#Program 2

#Function to replace all vowels in the string with '*' def replaceVowel(st):

#create an empty string newstr = "

for character in st:

#check if next character is a vowel if character in 'aeiouAEIOU':

#Replace vowel with * newstr += '*'

else:

newstr += character return newstr

#end of function

st = input("Enter a String: ")

st1 = replaceVowel(st)

print("The original String is:",st)

print("The modified String is:",st1)

Output:

Enter a String: Hello World

The original String is: Hello World

The modified String is: H*ll* W*rld

Program Write a program to input a string from the user and print it in the reverse order without creating a new string.

#Program 3

```
#Program to display string in reverse order
st = input("Enter a string: ")
for i in range(-1,-len(st)-1,-1):
print(st[i],end="")
```

Output:

```
Enter a string: Hello World
dlroW olleH
```

Program Write a program which reverses a string passed as parameter and stores the reversed string in a new string. Use a user defined function for reversing the string.

#Program 4

```
#Function to reverse a string
def reverseString(st):
newstr = "      #create a new string length = len(st)
for i in range(-1,-length-1,-1):
newstr += st[i]
return newstr #end of function
st = input("Enter a String: ")
st1 = reverseString(st)
print("The original String is:",st)
print("The reversed String is:",st1)
```

Output:

```
Enter a String: Hello World
The original String is: Hello World
The reversed String is: dlroW olleH
```

Program Write a program using a user defined function to check if a string is a palindrome or not. (A string is called palindrome if it reads same backwards as forward. For example, Kanak is a palindrome.)

#Program 5

```
#Function to check if a string is palindrome or not
def checkPalin(st):
i = 0
j = len(st) - 1
while(i <= j):
if(st[i] != st[j]):
return False
i += 1
j -= 1
return True #end of function
st = input("Enter a String: ")
result = checkPalin(st)
if result == True:
print("The given string",st,"is a palindrome")
```

else:

```
print("The given string",st,"is not a palindrome")
```

Output 1:

Enter a String: kanak

The given string kanak is a palindrome

Output 2:

Enter a String: computer

The given string computer is not a palindrome

Lists:

list operations, list slices, list methods, list loop, mutability, aliasing, cloning lists, list parameters, list comprehension; Tuples: tuple assignment, tuple as return value, tuple comprehension; Dictionaries: operations and methods, comprehension; Lists, Tuples, Dictionaries:

List:

- It is a general purpose most widely used in data structures
- List is a collection which is ordered and changeable and allows duplicate members. (Grow and shrink as needed, sequence type, sortable).
- To use a list, you must declare it first. Do this using square brackets and separate values with commas.
- We can construct / create list in many ways.

Example:

```
>>> list1=[1,2,3,'A','B',7,8,[10,11]]
```

```
>>> print(list1)
```

```
[1, 2, 3, 'A', 'B', 7, 8, [10, 11]]
```

```
-----  
>>> x=list()
```

```
>>> x
```

```
[]
```

```
-----  
>>> tuple1=(1,2,3,4)
```

```
>>> x=list(tuple1)
```

```
>>> x
```

```
[1, 2, 3, 4]
```

List operations:

These operations include indexing, slicing, adding, multiplying, and checking for membership.

Basic List Operations:

Lists respond to the + and * operators much like strings; they mean concatenation and repetition here too, except that the result is a new list, not a string.

PythonExpression	Results	Description
len([1, 2, 3])	3	Length
[1, 2, 3] + [4, 5, 6]	[1, 2, 3, 4, 5, 6]	Concatenation
['Hi!'] * 4	['Hi!', 'Hi!', 'Hi!', 'Hi!']	Repetition
3 in [1, 2, 3]	True	Membership
for x in [1, 2, 3]: print x,	1 2 3	Iteration

Indexing, Slicing, and Matrixes

Because lists are sequences, indexing and slicing work the same way for lists as they do for strings.

Assuming following input –

```
L = ['nrcm', 'college', 'NRCM!']
```

Python Expression	Results	Description
L[2]	NRCM!	Offsets start at zero
L[-2]	college	Negative: count from the right
L[1:]	['college', 'NRCM!']	Slicing fetches sections
L[3]	IndexError	Index out of Range

List slices:

```
>>> list1=range(1,6)
>>> list1
range(1, 6)
>>> print(list1)
range(1, 6)
>>> list1=[1,2,3,4,5,6,7,8,9,10]
>>> list1[1:]
[2, 3, 4, 5, 6, 7, 8, 9, 10]
>>> list1[:1]
[1]
>>> list1[2:5]
[3, 4, 5]
>>> list1[:6]
[1, 2, 3, 4, 5, 6]
>>> list1[1:2:4]
[2]
>>> list1[1:8:2]
[2, 4, 6, 8]
```

List methods:

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

- Del()
- Append()
- Extend()
- Insert()
- Pop()
- Remove()
- Reverse()
- Sort()

Delete: Delete a list or an item from a list

```
>>> x=[5,3,8,6]
>>> del(x[1]) #deletes the index position 1 in a list
>>> x
[5, 8, 6]
-----
>>> del(x)
```

```
>>> x # complete list gets deleted
```

Append: Append an item to a list

```
>>> x=[1,5,8,4]
>>> x.append(10)
```

```
>>> x
```

```
[1, 5, 8, 4, 10]
```

Extend: Append a sequence to a list.

```
>>> x=[1,2,3,4]
```

```
>>> y=[3,6,9,1]
```

```
>>> x.extend(y)
```

```
>>> x
```

```
[1, 2, 3, 4, 3, 6, 9, 1]
```

Insert: To add an item at the specified index, use the insert () method:

```
>>> x=[1,2,4,6,7]
```

```
>>> x.insert(2,10) #insert(index no, item to be inserted)
```

```
>>> x
```

```
[1, 2, 10, 4, 6, 7]
```

```
-----  
>>> x.insert(4,['a',11])
```

```
>>> x
```

```
[1, 2, 10, 4, ['a', 11], 6, 7]
```

Pop: The pop() method removes the specified index, (or the last item if index is not specified) or simply pops the last item of list and returns the item.

```
>>> x=[1, 2, 10, 4, 6, 7]
```

```
>>> x.pop()
```

```
7
```

```
>>> x
```

```
[1, 2, 10, 4, 6]
```

```
-----  
>>> x=[1, 2, 10, 4, 6]
```

```
>>> x.pop(2)
```

```
10
```

```
>>> x
```

```
[1, 2, 4, 6]
```

Remove: The remove() method removes the specified item from a given list.

```
>>> x=[1,33,2,10,4,6]
```

```
>>> x.remove(33)
```

```
>>> x
```

```
[1, 2, 10, 4, 6]
```

```
>>> x.remove(4)
```

```
>>> x
```

```
[1, 2, 10, 6]
```

Reverse: Reverse the order of a given list.

```
>>> x=[1,2,3,4,5,6,7]
```

```
>>> x.reverse()
```

```
>>> x
```

```
[7, 6, 5, 4, 3, 2, 1]
```

Sort: Sorts the elements in ascending order

```
>>> x=[7, 6, 5, 4, 3, 2, 1]
```

```
>>> x.sort()
```

```
>>> x
[1, 2, 3, 4, 5, 6, 7]
-----
>>> x=[10,1,5,3,8,7]
>>> x.sort()
>>> x
[1, 3, 5, 7, 8, 10]
```

List loop:

Loops are control structures used to repeat a given section of code a certain number of times or until a particular condition is met.

Method #1: For loop

```
#list of items
list = ['M','R','C','E','T']
i = 1
#Iterating over the list
for item in list:
    print ('college ',i,' is ',item)
    i = i+1
```

Output:

```
C:/Users/NRCM/AppData/Local/Programs/Python/Python38-32/pyyy/lis.py
college 1 is N
college 2 is R
college 3 is C
college 4 is M
```

Method #2: For loop and range()

In case we want to use the traditional for loop which iterates from number x to number y.

```
# Python3 code to iterate over a list
list = [1, 3, 5, 7, 9]
```

```
# getting length of list
length = len(list)
```

```
# Iterating the index
# same as 'for i in range(len(list))'
for i in range(length):
    print(list[i])
```

Output:

```
C:/Users/NRCM/AppData/Local/Programs/Python/Python38-32/pyyy/listloop.py
1
3
5
7
9
```

Method #3: using while loop

```
# Python3 code to iterate over a list
list = [1, 3, 5, 7, 9]
```

```
# Getting length of list
```

```
length = len(list)
```

```
i = 0
```

```
# Iterating using while loop
```

```
while i < length:
```

```
    print(list[i])
```

```
    i += 1
```

Mutability:

A mutable object can be changed after it is created, and an immutable object can't.

Append: Append an item to a list

```
>>> x=[1,5,8,4]
```

```
>>> x.append(10)
```

```
>>> x
```

```
[1, 5, 8, 4, 10]
```

Extend: Append a sequence to a list.

```
>>> x=[1,2,3,4]
```

```
>>> y=[3,6,9,1]
```

```
>>> x.extend(y)
```

```
>>> x
```

Delete: Delete a list or an item from a list

```
>>> x=[5,3,8,6]
```

```
>>> del(x[1]) #deletes the index position 1 in a list
```

```
>>> x
```

```
[5, 8, 6]
```

Insert: To add an item at the specified index, use the insert () method:

```
>>> x=[1,2,4,6,7]
```

```
>>> x.insert(2,10) #insert(index no, item to be inserted)
```

```
>>> x
```

```
[1, 2, 10, 4, 6, 7]
```

```
-----  
>>> x.insert(4,['a',11])
```

```
>>> x
```

```
[1, 2, 10, 4, ['a', 11], 6, 7]
```

Pop: The pop() method removes the specified index, (or the last item if index is not specified) or simply pops the last item of list and returns the item.

```
>>> x=[1, 2, 10, 4, 6, 7]
```

```
>>> x.pop()
```

```
7
```

```
>>> x
```

```
[1, 2, 10, 4, 6]
```

```
-----  
>>> x=[1, 2, 10, 4, 6]
```

```
>>> x.pop(2)
```

```
10
```

```
>>> x
```

```
[1, 2, 4, 6]
```

Remove: The remove() method removes the specified item from a given list.

```
>>> x=[1,33,2,10,4,6]
```

```
>>> x.remove(33)
```

```
>>> x
```

```
[1, 2, 10, 4, 6]
```

```
>>> x.remove(4)
```

```
>>> x
```

```
[1, 2, 10, 6]
```

Reverse: Reverse the order of a given list.

```
>>> x=[1,2,3,4,5,6,7]
```

```
>>> x.reverse()
```

```
>>> x
```

```
[7, 6, 5, 4, 3, 2, 1]
```

Sort: Sorts the elements in ascending order

```
>>> x=[7, 6, 5, 4, 3, 2, 1]
```

```
>>> x.sort()
```

```
>>> x
```

```
[1, 2, 3, 4, 5, 6, 7]
```

```
-----  
>>> x=[10,1,5,3,8,7]
```

```
>>> x.sort()
```

```
>>> x
```

```
[1, 3, 5, 7, 8, 10]
```

Aliasing:

1. An alias is a second name for a piece of data, often easier (and more useful) than making a copy.
2. If the data is immutable, aliases don't matter because the data can't change.
3. But if data can change, aliases can result in lot of hard – to – find bugs.
4. Aliasing happens whenever one variable's value is assigned to another variable.

For example:

```
a = [81, 82, 83]
```

```
b = [81, 82, 83]
```

```
print(a == b)
```

```
print(a is b)
```

```
b = a
```

```
print(a == b)
```

```
print(a is b)
```

```
b[0] = 5
```

```
print(a)
```

Output:

```
C:/Users/NRCM/AppData/Local/Programs/Python/Python38-32/pyyy/alia.py
```

```
True
```

```
False
```

```
True
```


True

[5, 82, 83]

Because the same list has two different names, a and b, we say that it is aliased. Changes made with one alias affect the other. In the example above, you can see that a and b refer to the same list after executing the assignment statement b = a.

Cloning Lists:

If we want to modify a list and also keep a copy of the original, we need to be able to make a copy of the list itself, not just the reference. This process is sometimes called cloning, to avoid the ambiguity of the word copy.

The easiest way to clone a list is to use the slice operator. Taking any slice of a creates a new list. In this case the slice happens to consist of the whole list.

Example:

```
a = [81, 82, 83]
```

```
b = a[:] # make a clone using slice
```

```
print(a == b)
```

```
print(a is b)
```

```
b[0] = 5
```

```
print(a)
```

```
print(b)
```

Output:

```
C:/Users/NRCM/AppData/Local/Programs/Python/Python38-32/pyyy/clo.py
```

```
True
```

```
False
```

```
[81, 82, 83]
```

```
[5, 82, 83]
```

Now we are free to make changes to b without worrying about a

List parameters:

Passing a list as an argument actually passes a reference to the list, not a copy of the list.

Since lists are mutable, changes made to the elements referenced by the parameter change the same list that the argument is referencing.

for example, the function below takes a list as an argument and multiplies each element in the list by 2:

```
def doubleStuff(List):
```

```
    """ Overwrite each element in aList with double its value. """
```

```
    for position in range(len(List)):
```

```
        List[position] = 2 * List[position]
```

```
things = [2, 5, 9]
```

```
print(things)
```

```
doubleStuff(things)
```

```
print(things)
```

Output:

```
C:/Users/NRCM/AppData/Local/Programs/Python/Python38-32/lipar.py ==
```

```
[2, 5, 9]
```

```
[4, 10, 18]
```

List comprehension:

List:

List comprehensions provide a concise way to create lists. Common applications are to make new lists where each element is the result of some operations applied to each member of another sequence or iterable, or to create a subsequence of those elements that satisfy a certain condition.

For example, assume we want to create a list of squares, like:

```
>>> list1=[]
>>> for x in range(10):
list1.append(x**2)
>>> list1
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
(or)
```

This is also equivalent to

```
>>> list1=list(map(lambda x:x**2, range(10)))
>>> list1
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
(or)
```

Which is more concise and readable.

```
>>> list1=[x**2 for x in range(10)]
>>> list1
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

Similarly some examples:

```
>>> x=[m for m in range(8)]
>>> print(x)
[0, 1, 2, 3, 4, 5, 6, 7]
>>> x=[z**2 for z in range(10) if z>4]
>>> print(x)
[25, 36, 49, 64, 81]
>>> x=[x ** 2 for x in range (1, 11) if x % 2 == 1]
>>> print(x)
[1, 9, 25, 49, 81]
>>> a=5
>>> table = [[a, b, a * b] for b in range(1, 11)]
>>> for i in table:
print(i)
[5, 1, 5]
[5, 2, 10]
[5, 3, 15]
[5, 4, 20]
[5, 5, 25]
[5, 6, 30]
[5, 7, 35]
[5, 8, 40]
[5, 9, 45]
[5, 10, 50]
```

Tuples:

A tuple is a collection which is ordered and unchangeable. In Python tuples are written with round brackets.

- Supports all operations for sequences.
- Immutable, but member objects may be mutable.
- If the contents of a list shouldn't change, use a tuple to prevent items from accidentally being added, changed, or deleted.
- Tuples are more efficient than list due to python's implementation.

We can construct tuple in many ways:

```
X=() #no item tuple
```

```
X=(1,2,3)
```

```
X=tuple(list1)
```

```
X=1,2,3,4
```

Example:

```
>>> x=(1,2,3)
```

```
>>> print(x)
```

```
(1, 2, 3)
```

```
>>> x
```

```
(1, 2, 3)
```

```
-----  
>>> x=()
```

```
>>> x
```

```
()
```

```
-----  
>>> x=[4,5,66,9]
```

```
>>> y=tuple(x)
```

```
>>> y
```

```
(4, 5, 66, 9)
```

```
-----  
>>> x=1,2,3,4
```

```
>>> x
```

```
(1, 2, 3, 4)
```

Some of the operations of tuple are:

- Access tuple items
- Change tuple items
- Loop through a tuple
- Count()
- Index()
- Length()

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

```
>>> x=('a','b','c','g')
```

```
>>> print(x[2])
```

```
c
```

Change tuple items: Once a tuple is created, you cannot change its values. Tuples are unchangeable.

```
>>> x=(2,5,7,'4',8)
```

```
>>> x[1]=10
```

Traceback (most recent call last):

```
File "<pyshell#41>", line 1, in <module>
```

```
x[1]=10
```

TypeError: 'tuple' object does not support item assignment

```
>>> x
```

```
(2, 5, 7, '4', 8) # the value is still the same
```

Loop through a tuple: We can loop the values of tuple using for loop

```
>>> x=4,5,6,7,2,'aa'
```

```
>>> for i in x:
```

```
print(i)
```

```
4
```

```
5
```

```
6
```

```
7
```

```
2
```

```
aa
```

Count (): Returns the number of times a specified value occurs in a tuple

```
>>> x=(1,2,3,4,5,6,2,10,2,11,12,2)
```

```
>>> x.count(2)
```

```
4
```

Index (): Searches the tuple for a specified value and returns the position of where it was found

```
>>> x=(1,2,3,4,5,6,2,10,2,11,12,2)
```

```
>>> x.index(2)
```

```
1
```

(Or)

```
>>> x=(1,2,3,4,5,6,2,10,2,11,12,2)
```

```
>>> y=x.index(2)
```

```
>>> print(y)
```

```
1
```

Length (): To know the number of items or values present in a tuple, we use len().

```
>>> x=(1,2,3,4,5,6,2,10,2,11,12,2)
```

```
>>> y=len(x)
```

```
>>> print(y)
```

```
12
```

Tuple Assignment

Python has tuple assignment feature which enables you to assign more than one variable at a time. In here, we have assigned tuple 1 with the college information like college name, year, etc. and another tuple 2 with the values in it like number (1, 2, 3... 7).

For Example,

Here is the code,

```
>>> tup1 = ('NRCM', 'eng college','2004','cse', 'it','csit');
```

```
>>> tup2 = (1,2,3,4,5,6,7);
>>> print(tup1[0])
```

NRCM

```
>>> print(tup2[1:4])
(2, 3, 4)
```

Tuple 1 includes list of information of NRCM

Tuple 2 includes list of numbers in it

We call the value for [0] in tuple and for tuple 2 we call the value between 1 and 4

Run the above code- It gives name NRCM for first tuple while for second tuple it gives number (2, 3, 4)

Tuple as return values:

A Tuple is a comma separated sequence of items. It is created with or without (). Tuples are immutable.

A Python program to return multiple values from a method using tuple

This function returns a tuple

```
def fun():
    str = "NRCM college"
    x = 20
    return str, x; # Return tuple, we could also
    # write (str, x)
```

Driver code to test above method

```
str, x = fun() # Assign returned tuple
print(str)
print(x)
```

Output:

```
C:/Users/NRCM/AppData/Local/Programs/Python/Python38-32/tupretval.py
NRCM college
20
```

- Functions can return tuples as return values.

```
def circleInfo(r):
    """ Return (circumference, area) of a circle of radius r """
    c = 2 * 3.14159 * r
    a = 3.14159 * r * r
    return (c, a)
```

```
print(circleInfo(10))
```

Output:

```
C:/Users/NRCM/AppData/Local/Programs/Python/Python38-32/functupretval.py
(62.8318, 314.159)
```

```
def f(x):
    y0 = x + 1
    y1 = x * 3
    y2 = y0 ** y3
    return (y0, y1, y2)
```

Tuple comprehension:

Tuple Comprehensions are special: The result of a tuple comprehension is special. You might expect it to produce a tuple, but what it does is produce a special "generator" object that we can iterate over.

For example:

```
>>> x = (i for i in 'abc') #tuple comprehension
>>> x
<generator object <genexpr> at 0x033EEC30>
>>> print(x)
<generator object <genexpr> at 0x033EEC30>
```

You might expect this to print as ('a', 'b', 'c') but it prints as <generator object <genexpr> at 0x02AAD710> The result of a tuple comprehension is not a tuple: it is actually a generator. The only thing that you need to know now about a generator now is that you can iterate over it, but ONLY ONCE.

So, given the code

```
>>> x = (i for i in 'abc')
>>> for i in x:
print(i)
a
b
c
```

Create a list of 2-tuples like (number, square):

```
>>> z=[(x, x**2) for x in range(6)]
>>> z
[(0, 0), (1, 1), (2, 4), (3, 9), (4, 16), (5, 25)]
```

Set:

Similarly to list comprehensions, set comprehensions are also supported:

```
>>> a = {x for x in 'abracadabra' if x not in 'abc'}
>>> a
{'r', 'd'}
>>> x={3*x for x in range(10) if x>5}
>>> x
{24, 18, 27, 21}
```

Dictionaries:

A dictionary is a collection which is unordered, changeable and indexed. In Python dictionaries are written with curly brackets, and they have keys and values.

- Key-value pairs
- Unordered

We can construct or create dictionary like:

```
X={1:'A',2:'B',3:'c'}
X=dict([('a',3) ('b',4)])
X=dict('A'=1,'B'=2)
```

Example:

```
>>> dict1 = {"brand":"NRCM","model":"college","year":2004}
>>> dict1
{'brand': 'NRCM', 'model': 'college', 'year': 2004}
```

Operations and methods:

Methods that are available with dictionary are tabulated below. Some of them have already been used in the above examples.

Method	Description
clear()	Remove all items from the dictionary.
copy()	Return a shallow copy of the dictionary.
fromkeys(seq[, v])	Return a new dictionary with keys from seq and value equal to v (defaults to None).
get(key[,d])	Return the value of key. If key doesnot exit, return d (defaults to None).
items()	Return a new view of the dictionary's items (key, value).
keys()	Return a new view of the dictionary's keys.
pop(key[,d])	Remove the item with key and return its value or d if key is not found. If d is not provided and key is not found, raises KeyError.
popitem()	Remove and return an arbitrary item (key, value). Raises KeyError if the dictionary is empty.
setdefault(key[,d])	If key is in the dictionary, return its value. If not, insert key with a value of d and return d (defaults to None).
update([other])	Update the dictionary with the key/value pairs from other, overwriting existing keys.
values()	Return a new view of the dictionary's values

Below are some dictionary operations:

To access specific value of a dictionary, we must pass its key,

```
>>> dict1 = {"brand":"NRCM","model":"college","year":2004}
>>> x=dict1["brand"]
>>> x
'NRCM'
```

To access keys and values and items of dictionary:

```
>>> dict1 = {"brand":"NRCM","model":"college","year":2004}
>>> dict1.keys()
dict_keys(['brand', 'model', 'year'])
>>> dict1.values()
dict_values(['NRCM', 'college', 2004])
>>> dict1.items()
dict_items([('brand', 'NRCM'), ('model', 'college'), ('year', 2004)])
```

```
>>> for items in dict1.values():
print(items)
NRCM
college
```

2004

```
>>> for items in dict1.keys():
```

```
print(items)
```

```
brand
```

```
model
```

```
year
```

```
>>> for i in dict1.items():
```

```
print(i)
```

```
('brand', 'NRCM')
```

```
('model', 'college')
```

```
('year', 2004)
```

Some more operations like:

- Add/change
- Remove
- Length
- Delete

Add/change values: You can change the value of a specific item by referring to its key name

```
>>> dict1 = {"brand": "nrcm", "model": "college", "year": 2004}
```

```
>>> dict1["year"] = 2005
```

```
>>> dict1
```

```
{'brand': 'nrcm', 'model': 'college', 'year': 2005}
```

Remove(): It removes or pop the specific item of dictionary.

```
>>> dict1 = {"brand": "nrcm", "model": "college", "year": 2004}
```

```
>>> print(dict1.pop("model"))
```

```
college
```

```
>>> dict1
```

```
{'brand': 'nrcm', 'year': 2005}
```

Delete: Deletes a particular item.

```
>>> x = {1:1, 2:4, 3:9, 4:16, 5:25}
```

```
>>> del x[5]
```

```
>>> x
```

Length: we use len() method to get the length of dictionary.

```
>>> {1: 1, 2: 4, 3: 9, 4: 16}
```

```
{1: 1, 2: 4, 3: 9, 4: 16}
```

```
>>> y = len(x)
```

```
>>> y
```

```
4
```

Iterating over (key, value) pairs:

```
>>> x = {1:1, 2:4, 3:9, 4:16, 5:25}
```

```
>>> for key in x:
```

```
print(key, x[key])
```

```
1 1
```

```
2 4
```

```
3 9
```

```
4 16
```

```
5 25
```



```
>>> for k,v in x.items():
print(k,v)
1 1
2 4
3 9
4 16
5 25
```

List of Dictionaries:

```
>>> customers = [{"uid":1,"name":"John"},
{"uid":2,"name":"Smith"},
{"uid":3,"name":"Andersson"},
]
>>> >>> print(customers)
[{'uid': 1, 'name': 'John'}, {'uid': 2, 'name': 'Smith'}, {'uid': 3, 'name': 'Andersson'}]
## Print the uid and name of each customer
>>> for x in customers:
print(x["uid"], x["name"])
1 John
2 Smith
3 Andersson
## Modify an entry, This will change the name of customer 2 from Smith to Charlie
>>> customers[2]["name"]="charlie"
>>> print(customers)
[{'uid': 1, 'name': 'John'}, {'uid': 2, 'name': 'Smith'}, {'uid': 3, 'name': 'charlie'}]
## Add a new field to each entry
>>> for x in customers:
x["password"]="123456" # any initial value
>>> print(customers)
[{'uid': 1, 'name': 'John', 'password': '123456'}, {'uid': 2, 'name': 'Smith', 'password':
'123456'}, {'uid': 3, 'name': 'charlie', 'password': '123456'}]
## Delete a field
>>> del customers[1]
>>> print(customers)
[{'uid': 1, 'name': 'John', 'password': '123456'}, {'uid': 3, 'name': 'charlie', 'password':
'123456'}]
>>> del customers[1]
>>> print(customers)
[{'uid': 1, 'name': 'John', 'password': '123456'}]
## Delete all fields
>>> for x in customers:
del x["uid"]
>>> x
{'name': 'John', 'password': '123456'}
```

Comprehension:

Dictionary comprehensions can be used to create dictionaries from arbitrary key and value expressions:

```
>>> z={x: x**2 for x in (2,4,6)}
>>> z
{2: 4, 4: 16, 6: 36}
>>> dict11 = {x: x*x for x in range(6)}
>>> dict11
{0: 0, 1: 1, 2: 4, 3: 9, 4: 16, 5: 25}
```

UNIT-4

Classes and objects

User-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 (a little) more complicated than the other options, but it has advantages that will be apparent soon.

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

```
class Point(object):  
    """represents a point in 2-D space"""
```

This header indicates that the new class is a `Point`, which is a kind of object, which is a built-in type.

The body is a docstring that explains what the class is for. You can define variables and functions inside a class definition, but we will get back to that later.

Defining a class named `Point` creates a class object.

```
>>> print 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()  
>>> print blank  
<__main__.Point instance at 0xb7e9d3ac>
```

The return value is a reference to a `Point` object, which we assign to `blank`. Creating a new object is called instantiation, and the object is an instance of the class.

When you print an instance, Python tells you what class it belongs to and where it is stored in memory (the prefix 0x means that the following number is in hexadecimal).

Attributes

You can assign values to an instance using dot notation:

```
>>> blank.x = 3.0
```

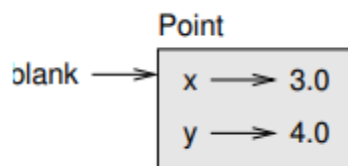
```
>>> blank.y = 4.0
```

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-TRIBute,” 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:

```
>>> print blank.y
```

```
4.0
```

```
>>> x = blank.x
```

```
>>> print x
```

```
3.0
```

The expression `blank.x` means, “Go to the object `blank` refers to and get the value of `x`.” In this case, 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:

```
>>> print '(%g, %g)' % (blank.x, blank.y) (3.0, 4.0)
```

```
>>> distance = math.sqrt(blank.x**2 + blank.y**2)
```

```
>>> print distance 5.0
```

You can pass an instance as an argument in the usual way. For example:

```
def print_point(p):
```

```
    print '(%g, %g)' % (p.x, p.y)
```

`print_point` takes a point as an argument and displays it in mathematical notation. To invoke it, you can pass `blank` as an argument:

```
>>> print_point(blank)
```

```
(3.0, 4.0)
```

Inside the function, `p` is an alias for `blank`, so if the function modifies `p`, `blank` changes.

Below Exercise Write a function called distance that takes two Points as arguments and returns the distance between them.

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(object):  
    """represent a rectangle.  
    attributes: width, height, corner. """
```

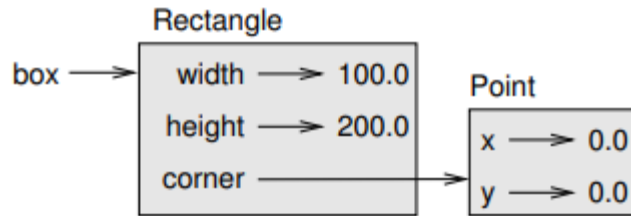
The docstring lists the attributes: width and height are numbers; corner is a Point object that specifies the lower-left corner.

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`.”

The figure shows the state of this object:



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(box):
    p = Point()
    p.x = box.corner.x + box.width/2.0
    p.y = box.corner.y + box.height/2.0
    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)
(50.0, 100.0)
  
```

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:

```
>>> print box.width
```

```
100.0
```

```
>>> print box.height
```

```
200.0
```

```
>>> grow_rectangle(box, 50, 100)
```

```
>>> print box.width
```

```
150.0
```

```
>>> print box.height
```

```
300.0
```

Inside the function, `rect` is an alias for `box`, so if the function modifies `rect`, `box` changes.

Exercise 15.2 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`.

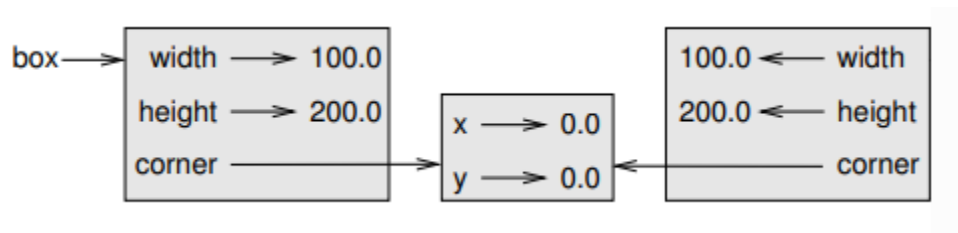
```
>>> print_point(p1)
(3.0, 4.0)
>>> print_point(p2)
(3.0, 4.0)
>>> p1 is p2
False
>>> p1 == p2
False
```

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. This behavior can be changed—we'll see how later.

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
```

Here is what the object diagram looks like:



This operation is called a shallow copy because it copies the object and any references it contains, but not the embedded objects.

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 behavior is confusing and error-prone.

Fortunately, the copy module contains 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 is box
```

```
False
```

```
>>> box3.corner is box.corner
```

```
False
```

box3 and box are completely separate objects.

Exercise 15.3 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()
```

```
>>> print p.z
```

```
AttributeError: Point instance has no attribute 'z'
```

If you are not sure what type an object is, you can ask:

```
>>> type(p)
```

```
<type '__main__.Point'>
```

If you are not sure whether an object has a particular attribute, you can use the built-in 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.

Classes and functions

Time

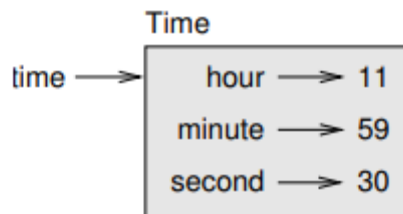
As another example of a user-defined type, we'll define a class called `Time` that records the time of day. The class definition looks like this:

```
class Time(object):  
    """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
```

The state diagram for the `Time` object looks like this:



Exercise 16.1 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.

Exercise 16.2 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 run time of the movie, which is one hour 35 minutes.

`add_time` figures out when the movie will be done.

```
>>> start = Time()
>>> start.hour = 9
>>> start.minute = 45
>>> start.second = 0
>>> duration = Time()
>>> duration.hour = 1
>>> duration.minute = 35
>>> duration.second = 0
>>> done = add_time(start, duration)
>>> print_time(done)
10:80:00
```

The result, 10:80:00 might not be what you were hoping for. The problem is that this function does not deal with cases where the number of seconds or minutes adds

up to more than sixty. When that happens, we have to “carry” the extra seconds into the minute column or the extra minutes into the hour column.

Here’s an improved version:

```
def add_time(t1, t2):
    sum = Time()
    sum.hour = t1.hour + t2.hour
    sum.minute = t1.minute + t2.minute
    sum.second = t1.second + t2.second
    if sum.second >= 60:
        sum.second -= 60
        sum.minute += 1
    if sum.minute >= 60:
        sum.minute -= 60
        sum.hour += 1
    return sum
```

Although this function is correct, it is starting to get big. We will see a shorter alternative later.

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 the parameter seconds is much greater than sixty?

In that case, it is not enough to carry once; we have to keep doing it until time.second is less than sixty. One solution is to replace the if statements with while statements. That would make the function correct, but not very efficient.

Prototyping versus planning

The development plan I am demonstrating is called “prototype and patch.” For each function, I wrote a prototype that performed the basic calculation and then tested it, patching errors along the way.

This approach can be effective, especially if you don’t yet have a deep understanding of the problem. But incremental corrections can generate code that is unnecessarily complicated—since it deals with many special cases—and unreliable—since it is hard to know if you have found all the errors.

An alternative is planned 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 (see wikipedia.org/wiki/Sexagesimal)! 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 the function that converts integers to Times (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.

Exercise 16.5 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 naïve 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 minutes and seconds are between 0 and 60 (including 0 but not 60) and if hours is positive. hours and minutes should be integral values, but we might allow seconds to have a fraction part.

These kind of requirements are called invariants because they should always be true. To put it a different way, if they are not true, then something has gone wrong.

Writing code to check your invariants can help you 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.hours < 0 or time.minutes < 0 or time.seconds < 0:  
        return False  
    if time.minutes >= 60 or time.seconds >= 60:  
        return False  
    return True
```

Then 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.

It is not easy to define object-oriented programming, but we have already seen some of its characteristics:

- Programs are made up of object definitions and function definitions, and most of the computation is expressed in terms of operations on objects.
- Each object definition corresponds to some object or concept in the real world, and the functions that operate on that object correspond to the ways real-world objects interact.

For example, the Time class defined in Chapter 16 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 the Time program, 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 user-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 simply 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

In Chapter 16, we defined a class named Time and in Exercise 16.1, you wrote a function named print_time:

```
class Time(object):  
  
    """represents the time of day.  
  
    attributes: hour, minute, second"""
```



```
def print_time(time):  
    print '%.2d:%.2d:%.2d' % (time.hour, time.minute, time.second)
```

To call this function, you have to pass a Time object as an argument:

```
>>> start = Time()  
>>> start.hour = 9  
>>> start.minute = 45  
>>> start.second = 00  
>>> print_time(start)  
  
09:45:00
```

To make `print_time` a method, all we have to do is move the function definition inside the class definition. Notice the change in indentation.

```
class Time(object):  
    def print_time(time):  
        print '%.2d:%.2d:%.2d' % (time.hour, time.minute, time.second)
```

Now there are two ways to call `print_time`. The first (and less common) way is to use function syntax:

```
>>> Time.print_time(start)  
  
09:45:00
```

In this use of dot notation, `Time` is the name of the class, and `print_time` is the name of the method.

`start` is passed as a parameter.

The second (and more concise) way is to use method syntax:

```
>>> start.print_time()  
  
09:45:00
```

In this use of dot notation, `print_time` is the name of the method (again), and `start` is the object the method is invoked on, which is called the subject. Just as the subject of a sentence is what the sentence is about, the subject of a method invocation is what the method is about.

Inside the method, the subject is assigned to the first parameter, so in this case `start` is assigned to `time`.

By convention, the first parameter of a method is called `self`, so it would be more common to write `print_time` like this:

```
class Time(object):
    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, and makes it easier to maintain and reuse code.

Exercise 17.1 Rewrite `time_to_int` (from Section 16.4) as a method. It is probably not appropriate to rewrite `int_to_time` as a method; it’s not clear what object you would invoke it on!

Another example

Here’s a version of `increment` (from Section 16.3) rewritten as a method:

```
# inside class Time:
    def increment(self, seconds):
        seconds += self.time_to_int()
        return int_to_time(seconds)
```

This version assumes that `time_to_int` is written as a method, as in Exercise 17.1. 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 exactly 2 arguments (3 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.

A more complicated example

`is_after` (from Exercise 16.2) 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.

Exercise 17.2 Write an `init` method for the `Point` class that takes `x` and `y` as optional parameters and assigns them to the corresponding attributes.

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.

Exercise 17.3 Write a `str` method for the `Point` class. Create a `Point` object and print it.

Operator overloading

By defining other special methods, you can specify the behavior of operators on user-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 quite a lot happening behind the scenes!

Changing the behavior of an operator so that it works with user-defined types is called operator overloading. For every operator in Python there is a corresponding special method, like `__add__`.

For more details, see docs.python.org/ref/specialnames.html.

Exercise 17.4 Write an `add` method for the `Point` class.

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 to do that. 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
```

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 will actually work for any kind of sequence. For example, in Section 11.1 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 can 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, then 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.

Debugging

It is legal to add attributes to objects at any point in the execution of a program, but if you are a stickler for type theory, it is a dubious practice to have objects of the same type with different attribute sets. It is usually 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 built-in function `hasattr` (see Section 15.7).

Another way to access the attributes of an object is through the special attribute `__dict__`, which is a dictionary that maps attribute names (as strings) and values:

```
>>> p = Point(3, 4)
>>> print p.__dict__
{'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 obj.__dict__:
```

```
        print attr, getattr(obj, attr)
```

`print_attributes` traverses the items in the object's 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

In this chapter we will develop classes to represent playing cards, decks of cards, and poker hands. If you don't play poker, you can read about it at wikipedia.org/wiki/Poker, but you don't have to; I'll tell you what you need to know for the exercises.

If you are not familiar with Anglo-American playing cards, you can read about them at wikipedia.org/wiki/Playing_cards.

Card objects

There are fifty-two cards in a deck, each of which belongs to one of four suits and one of thirteen ranks. The suits are Spades, Hearts, Diamonds, and Clubs (in descending order in bridge). The ranks are Ace, 2, 3, 4, 5, 6, 7, 8, 9, 10, Jack, Queen, and King. Depending on the game that you are playing, an Ace may be higher than King or lower than 2.

If we want to define a new object to represent a playing card, it is obvious what the attributes should be: rank and suit. It is not as obvious what type the attributes should be. One possibility is to use strings containing words like 'Spade' for suits and 'Queen' for ranks. One problem with this implementation is that it would not be easy to compare cards to see which had a higher rank or suit.

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

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

Spades 7 → 3

Hearts 7 → 2

Diamonds 7 → 1

Clubs 7→ 0

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

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

Jack 7→ 11

Queen 7→ 12

King 7→ 13

I am using the 7→ symbol to make it clear that these mappings are not part of the Python program.

They are part of the program design, but they don't appear explicitly in the code.

The class definition for Card looks like this:

```
class Card(object):
    """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 distinguished 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.

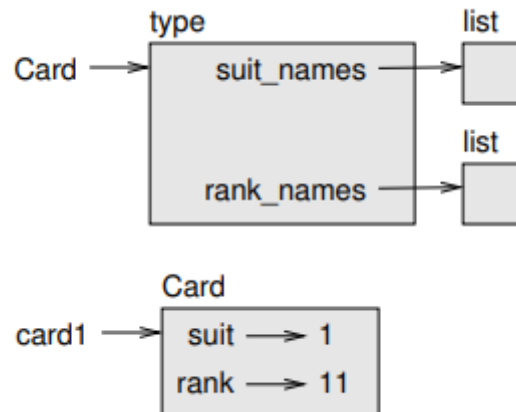
With the methods we have so far, we can create and print cards:

```
>>> card1 = Card(2, 11)
```

```
>>> print card1
```

Jack of Hearts

Here is a diagram that shows the `Card` class object and one `Card` instance:



Card is a class object, so it has type type. card1 has type Card. (To save space, I didn't draw the contents of suit_names and rank_names).

Comparing cards

For built-in types, there are conditional operators (<, >, ==, etc.) that compare values and determine when one is greater than, less than, or equal to another. For user-defined types, we can override the behavior of the built-in operators by providing a method named `__cmp__`.

`__cmp__` takes two parameters, `self` and `other`, and returns a positive number if the first object is greater, a negative number if the second object is greater, and 0 if they are equal to each other.

The correct ordering for cards is not obvious. For example, which is better, the 3 of Clubs or the 2 of Diamonds? One has a higher rank, but the other has a higher suit. In order to compare cards, you have to decide whether rank or suit is more important.

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

With that decided, we can write `__cmp__`:

```
# inside class Card:
def __cmp__(self, other):
    # check the suits
    if self.suit > other.suit: return 1
    if self.suit < other.suit: return -1
    # suits are the same... check ranks
    if self.rank > other.rank: return 1
    if self.rank < other.rank: return -1
    # ranks are the same... it's a tie
    return 0
```

You can write this more concisely using tuple comparison:

```
# inside class Card:
def __cmp__(self, other):
```

```
t1 = self.suit, self.rank
t2 = other.suit, other.rank

return cmp(t1, t2)
```

The built-in function `cmp` has the same interface as the method `__cmp__`: it takes two values and returns a positive number if the first is larger, a negative number if the second is larger, and 0 if they are equal.

Decks

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

The following is a class definition for `Deck`. The `init` method creates the attribute `cards` and generates the standard set of fifty-two cards:

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

The easiest way to populate the deck is with a nested loop. The outer loop enumerates the suits from 0 to 3. The inner loop enumerates the ranks from 1 to 13. Each iteration creates a new `Card` with the current suit and rank, and appends it to `self.cards`.

Printing the deck

Here is a `__str__` method for `Deck`:

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

This method demonstrates an efficient way to accumulate a large string: building a list of strings and then using `join`. The built-in function `str` invokes the `__str__` method on each card and returns the string representation.

Since we invoke `join` on a newline character, the cards are separated by newlines. Here's what the result looks like:

```
>>> deck = Deck()
```

```
>>> print deck
```

```
Ace of Clubs
```

```
2 of Clubs
```

```
3 of Clubs
```

```
...
```

```
10 of Spades
```

```
Jack of Spades
```

```
Queen of Spades
```

```
King of Spades
```

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

Add, remove, shuffle and sort

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

```
#inside class Deck:
```

```
def pop_card(self):
```

```
    return self.cards.pop()
```

Since `pop` removes the last card in the list, we are dealing from the bottom of the deck. In real life bottom dealing is frowned upon¹, but in this context it's ok.

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 function without doing much real work is sometimes called a veneer. The metaphor comes from woodworking, where it is common to glue a thin layer of good quality wood to the surface of a cheaper piece of wood.

In this case we are defining a “thin” method that expresses a list operation in terms that are appropriate for decks.

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.

Exercise 18.2 Write a Deck method named sort that uses the list method sort to sort the cards in a Deck. sort uses the `__cmp__` method we defined to determine sort order.

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.

It is called “inheritance” because the new class inherits the methods of the existing class. Extending this metaphor, the existing class is called the parent and the new class is called the child.

As an example, let’s say we want a class to represent a “hand,” that is, the set of cards held by one player. A hand is similar to a deck: both are made up of a set 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.

The definition of a child class is like other class definitions, but the name of the parent class appears 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.

Hand also 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
```

So when you create a Hand, Python invokes this init method:

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

But 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`.

Exercise 18.3 Write a `Deck` method called `deal_hands` that takes two parameters, the number of hands and the number of cards per hand, and that creates new `Hand` objects, deals the appropriate number of cards per hand, and returns a list of `Hand` objects.

Inheritance is a useful feature. Some programs that would be repetitive without inheritance can be written more elegantly with it. Inheritance can facilitate code reuse, since you can customize the behavior of parent classes without having to modify them. In some cases, the inheritance structure reflects the natural structure of the problem, which makes the program 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 scattered among several modules. Also, many of the things that can be done using inheritance can be done as well or better without it.

Class diagrams

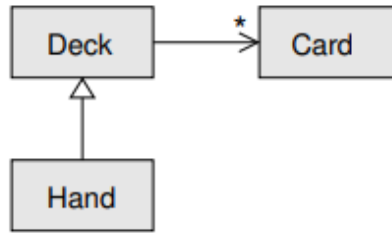
So far we have seen stack diagrams, which show the state of a program, and object diagrams, which show the attributes of an object and their values. These diagrams represent a snapshot in the execution of a program, so they change as the program runs.

They are also highly detailed; for some purposes, too detailed. A class diagram is a more abstract representation of the structure of a program. Instead of showing individual objects, it shows classes and the relationships between them.

There are several kinds of relationship between classes:

- Objects in one class might contain references to objects in another class. For example, each `Rectangle` contains a reference to a `Point`, and each `Deck` contains references to many `Cards`. This kind of relationship is called HAS-A, as in, “a `Rectangle` has a `Point`.”
- One class might inherit from another. This relationship is called IS-A, as in, “a `Hand` is a kind of a `Deck`.”
- One class might depend on another in the sense that changes in one class would require changes in the other.

A class diagram is a graphical representation of these relationships². For example, this diagram 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 arrow head represents a HAS-A relationship; in this case a Deck has references to Card objects.

The star (*) near the arrow head is a multiplicity; it indicates how many Cards a Deck has. A multiplicity can be a simple number, like 52, a range, like 5..7 or a star, which indicates that a Deck can have any number of Cards.

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.

Exercise 18.4 Read TurtleWorld.py, World.py and Gui.py and draw a class diagram that shows the relationships among the classes defined there.

Debugging

Inheritance can make debugging a challenge because when you invoke a method on an object, you might not know 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 subclasses override this method, you'll get that version instead.

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()
>>> print 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.”

Here's a program design suggestion: whenever you override a method, the interface of the new method should be the same as the old. It should take the same arguments, return the same type, and obey the same preconditions and postconditions. If you obey this rule, you will find that any function designed to work with an instance of a superclass, like a Deck, will also work with instances of subclasses like a Hand or PokerHand.

If you violate this rule, your code will collapse like (sorry) a house of cards.

UNIT-5

Graphical User Interfaces

Most people do not judge a book by its cover. They are interested in its contents, not its appearance. However, users judge a software product by its user interface because they have no other way to access its functionality. With the exception of Chapter 7, in which we explored graphics and image processing, this book has focused on programs that present a terminal-based user interface. This type of user interface is perfectly adequate for some applications, and it is the simplest and easiest for beginning programmers to code.

However, 99% of the world's computer users never see such a user interface. Instead, most interactive computer software employs a **graphical user interface** or **GUI** (or its close relative, the touchscreen interface). A GUI displays text as well as small images (called icons) that represent objects such as folders, files of different types, command buttons, and drop-down menus. In addition to entering text at the keyboard, the user of a GUI can select some of these icons with a pointing device, such as a mouse, and move them around on the display. Commands can be activated by pressing the enter key or control keys, by pressing a command button, by selecting a drop-down menu item, or by double-clicking on some icons with the mouse. Put more simply, a GUI displays all information, including text, graphically to its users and allows them to manipulate this information directly with a pointing device.

In this chapter, you will learn how to develop GUIs. Much GUI-based programming requires you to use existing classes, objects, and their methods, as you did in previous chapters. Along the way, you will also learn how to develop new classes of objects, such as application windows, by extending or repurposing existing classes. Rather than defining a new class of objects from scratch, you will create a customized version of an existing class by the mechanisms of **subclassing** and **inheritance**. GUI programming provides an engaging area for learning these techniques, which play a prominent role in modern software development.

The Behavior of Terminal-Based Programs and GUI-Based Programs

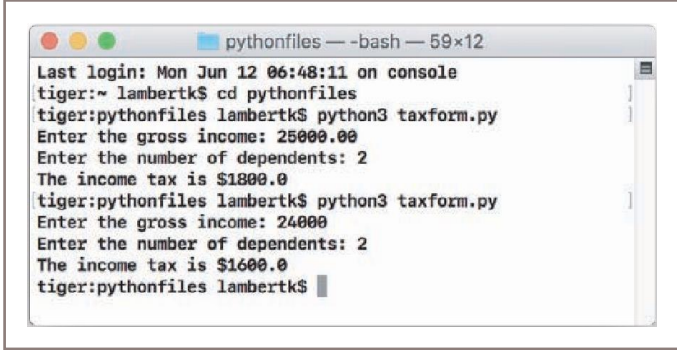
The transition to GUIs involves making a significant adjustment to your thinking. A GUI program is event driven, meaning that it is inactive until the user clicks a button or selects a menu option. In contrast, a terminal-based program maintains constant control over the interactions with the user. Put differently, a terminal-based program prompts users to enter successive inputs, whereas a GUI program puts users in charge, allowing them to enter inputs in any order and waiting for them to press a command button or select a menu option.

To make this difference clear, we begin by examining the look and behavior of two different versions of the same program from a user's point of view. This program, first introduced in Chapter 2, computes and displays a person's income tax, given two inputs—the gross income and the number of dependents. The first version of the program includes a terminal-based user interface, whereas the second version uses a graphical user interface.

Although both programs perform the same function, their behavior, look, and feel from a user's perspective are quite different.

The Terminal-Based Version

The terminal-based version of the program prompts the user for his gross income and number of dependents. After the user enters his inputs, the program responds by computing and displaying his income tax. The program then terminates execution. A sample session with the program is shown in Figure 8-1.



```
pythonfiles -- -bash -- 59x12
Last login: Mon Jun 12 06:48:11 on console
tiger:~ lambertk$ cd pythonfiles
tiger:pythonfiles lambertk$ python3 taxform.py
Enter the gross income: 25000.00
Enter the number of dependents: 2
The income tax is $1800.0
tiger:pythonfiles lambertk$ python3 taxform.py
Enter the gross income: 24000
Enter the number of dependents: 2
The income tax is $1600.0
tiger:pythonfiles lambertk$
```

Figure 8-1 A session with the terminal-based tax calculator program

The terminal-based user interface has several obvious effects on its users:

- The user is constrained to reply to a definite sequence of prompts for inputs. Once an input is entered, there is no way to back up and change it.
- To obtain results for a different set of input data, the user must run the program again. At that point, all of the inputs must be re-entered.

Each of these effects poses a problem for users that can be solved by converting the interface to a GUI.

The GUI-Based Version

The GUI-based version of the program displays a **window** that contains various components, also called **widgets**. Some of these components look like text, while others provide visual cues as to their use. Figure 8-2 shows snapshots of a sample session with this version of the program. The snapshot on the left shows the interface at program start-up, while the snapshot on the right shows the interface after the user has entered inputs and clicked the **Compute** button. The program was run on a Macintosh; on a Windows- or Linux-based PC, the windows look slightly different.

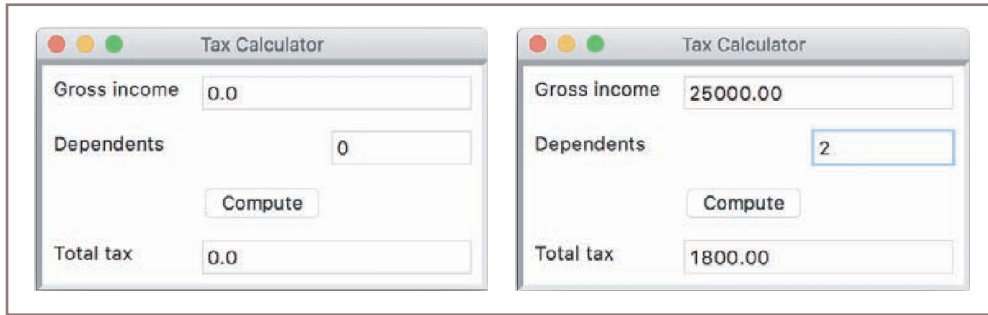


Figure 8-2 A GUI-based tax calculator program

The window in Figure 8-2 contains the following components:

- A **title bar** at the top of the window. This bar contains the title of the program, “Tax Calculator.” It also contains three colored disks. Each disk is a **command button**. The user can use the mouse to click the left disk to quit the program, the middle disk to minimize the window, or the right disk to zoom the window. The user can also move the window around the screen by holding the left mouse button on the title bar and dragging the mouse.
- A set of **labels** along the left side of the window. These are text elements that describe the inputs and outputs. For example, “Gross income” is one label.
- A set of **entry fields** along the right side of the window. These are boxes within which the program can output text or receive it as input from the user. The first two entry fields will be used for inputs, while the last field will be used for the output. At program start-up, the fields contain default values, as shown in the window on the left side of Figure 8-2.
- A single command button labeled **Compute**. When the user uses the mouse to press this button, the program responds by using the data in the two input fields to compute the income tax. This result is then displayed in the output field. Sample input data and the corresponding output are shown in the window on the right side of Figure 8-2.
- The user can also alter the size of the window by holding the mouse on its lower-right corner and dragging in any direction.

Although this review of features might seem tedious to anyone who regularly uses GUI-based programs, a careful inventory is necessary for the programmer who builds them. Also, a close study of these features reveals the following effects on users:

- The user is not constrained to enter inputs in a particular order. Before she presses the **Compute** button, she can edit any of the data in the two input fields.
- Running different data sets does not require re-entering all of the data. The user can edit just one value and press the **Compute** button to observe different results.

When we compare the effects of the two interfaces on users, the GUI seems to be a definite improvement on the terminal-based user interface. The improvement is even more noticeable as the number of command options increases and the information to be presented grows in quantity and complexity.

Event-Driven Programming

Rather than guide the user through a series of prompts, a GUI-based program opens a window and waits for the user to manipulate window components with the mouse. These user-generated events, such as mouse clicks, trigger operations in the program to respond by pulling in inputs, processing them, and displaying results. This type of software system is event-driven, and the type of programming used to create it is called **event-driven programming**.

Like any complex program, an event-driven program is developed in several steps. In the analysis step, the types of window components and their arrangement in the window are determined. Because GUI-based programs are almost always object based, this becomes a matter of choosing among GUI component classes available in the programming language or inventing new ones if needed. Graphic designers and cognitive psychologists might be called in to assist in this phase, if the analysts do not already possess this type of expertise. To a certain extent, the number, types, and arrangement of the window components depend on the nature of the information to be displayed and also depend on the set of commands that will be available to the user for manipulating that information.

Let us return to the example of the tax calculator program to see how it might be structured as an event-driven program. The GUI in this program consists of the window and its components, including the labeled entry fields and the **Compute** button. The action triggered when this button is clicked is a method call. This method fetches the input values from the input fields and performs the computation. The result is then sent to the output field to be displayed.

Once the interactions among these resources have been determined, their coding can begin. This phase consists of several steps:

1. Define a new class to represent the main application window.
2. Instantiate the classes of window components needed for this application, such as labels, fields, and command buttons.
3. Position these components in the window.
4. Register a method with each window component in which an event relevant to the application might occur.
5. Define these methods to handle the events.
6. Define a **main** function that instantiates the window class and runs the appropriate method to launch the GUI.

In coding tfile program, you could initially skip steps 4 and 5, wfliefl concern responding to user events. Tflis would allow you to preview and refine tfile window and its layout, even tfloughfl tfile command buttons and otfler GUI elements lack functionality.

In tfile sections tflat follow, we explore tfilese elements of GUI-based, event-driven programing witfl examples in Pytflon.

Coding Simple GUI-Based Programs

In tflis section, we sflow some examples of simple GUI-based programs in Pytflon. Pytflon's standard `tkinter` module includes classes for windows and numerous types of window components, but its use can be cflallenging for beginners. Tflerefore, tflis book uses a custom, open-source module called `breezypythongui`, wflie occasionally relying upon some of tfile simpler resources of `tkinter`. You will find tfile code, documentation, and installation instructions for tfile `breezypythongui` module at <http://home.wlu.edu/~lambertk/breezypythongui/>. We start witfl some sflort demo programs tflat illustrate some basic GUI components, and, in later sections, we develop some examples witfl more significant functionality.

A Simple "Hello World" Program

Our first demo program defines a class for a main window tflat displays a greeting. Figure 8-3 sflows a screenshot of tfile window.



Figure 8-3 Displaying a label with text in a window

As in all of our GUI-based programs, a new window class **extends** tfile `EasyFrame` class. By “extends,” we mean “repurposes” or “provides extra functionality for.” Tfile `EasyFrame` class provides tfile basic functionality for any window, sucfl as tfile command buttons in tfile title bar. Our new class, named `LabelDemo`, provides additional functionality to tfile `EasyFrame` class. Here is tfile code for tfile program:

```
"""
File: labeldemo.py
"""

from breezypythongui import EasyFrame

class LabelDemo(EasyFrame):
    """Displays a greeting in a window."""

    def __init__(self):
        """Sets up the window and the label."""
```



```
        EasyFrame._init_(self)
        self.addLabel(text = "Hello world!", row = 0, column = 0)

def main():
    """Instantiates and pops up the window."""
    LabelDemo().mainloop()

if __name__ == "__main__":
    main()
```

We will speak more generally about class definitions shortly. For now, note that this program performs the following steps:

1. Import the `EasyFrame` class from the `breezypythongui` module. This class is a subclass of `tkinter`'s `Frame` class, which represents a top-level window. In many GUI programs, this is the only import you will need.
2. Define the `LabelDemo` class as a subclass of `EasyFrame`. The `LabelDemo` class describes the window's layout and functionality for this application.
3. Define an `__init__` method in the `LabelDemo` class. This method is automatically run when the window is created. The `__init__` method runs a method with the same name on the `EasyFrame` class and then sets up any window components to display in the window. In this case, the `addLabel` method is run on the window itself. The `addLabel` method creates a window component, a **label object** with the text "Hello world!," and adds it to the window at the grid position (0, 0).
4. The last five lines of code define a `main` function and check to see if the Python code file is being run as a program. If this is true, the `main` function is called to create an instance of the `LabelDemo` class. The `mainloop` method is then run on this object. At this point, the window pops up for viewing. Note that `mainloop`, as the name implies, enters a loop. The Python Virtual Machine runs this loop behind the scenes. Its purpose is to wait for user events, as mentioned earlier. The loop terminates when the user clicks the window's close box.

Because steps 1 and 4 typically have the same format in each program, they will be omitted from the text of many of the program examples that follow.

A Template for All GUI Programs

Writing the code to pop up a window that says "Hello world!" might seem like a lot of work. However, the good news is that the structure of a GUI program is always the same, no matter how complex the application becomes. Here is the template for this structure:

```
from breezypythongui import EasyFrame

Other imports

class ApplicationName(EasyFrame):

    The __init__ method definition

    Definitions of event handling methods

def main():
    ApplicationName().mainloop()

if __name__ == "__main__":
    main()
```

A GUI application window is always represented as a class `tfl` that extends `EasyFrame`. The `__init__` method initializes the window by setting its attributes and populating it with appropriate GUI components. In our example, Python runs the `tfl` method automatically when the constructor function `LabelDemo` is called. The event handling methods provide the responses of the application to user events (not relevant in the example program). The last lines of code, beginning with the definition of the `main` function, create an instance of the application window class and run the `mainloop` method on this instance. The window then pops up and waits for user events. Pressing the window's close button will quit the program normally. If you have launched the program from an IDLE window, you can run it again after quitting by entering `main()` at the shell prompt.

The Syntax of Class and Method Definitions

Note that the syntax of class and method definitions is a bit like the syntax of function definitions. Each definition has a one-line header that begins with a keyword (`class` or `def`), followed by a body of code indented one level in the text.

A class header contains the name of the class, conventionally capitalized in Python, followed by a parenthesized list of one or more parent classes. The body of a class definition, nested one tab under the header, consists of one or more method definitions, which may appear in any order.

A method header looks very much like a function header, but a method always has at least one parameter, in the first position, named `self`. At call time, the PVM automatically assigns to this parameter a reference to the object on which the method is called; thus, you do not pass this object as an explicit argument at call time. For example, given the method header

```
def someMethod(self):
```

the method call

```
anObject.someMethod()
```

automatically assigns the object `anObject` to the `self` parameter for this method. The parameter `self` is used within class and method definitions to call other methods on the same object, or to access that object's instance variables or data, as will be explained shortly.

Subclassing and Inheritance as Abstraction Mechanisms

Our first example program defined a new class named `LabelDemo`. This class was defined as a **subclass** of the class `breezypythongui.EasyFrame`, which in turn is a subclass of the class `tkinter.Frame`. The subclass relationships among these classes are shown in the **class diagram** of Figure 8-4.

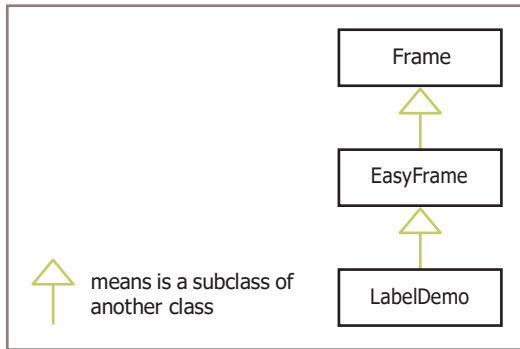


Figure 8-4 A class diagram for the label demo program

Note that the `EasyFrame` class is the **parent** of the `LabelDemo` class, and the `Frame` class is the parent of the `EasyFrame` class. This makes the `Frame` class the **ancestor** of the `LabelDemo` class. When you make a new class a subclass of another class, your new class inherits and thereby acquires the attributes and behavior defined by its parent class, and any of its ancestor classes, for free. Subclassing and inheritance are thus useful abstraction mechanisms, in that you do not have to reinvent the entire wheel when defining a new class of objects, but only customize it a bit. For example, the `EasyFrame` class customizes the `Frame` class with methods to add window components to a window; the `LabelDemo` class customizes the `EasyFrame` method `__init__` to set up a window with a specific window component.

As a rule of thumb, when you are defining a new class of objects, you should look around for a class that already supports some of the structure and behavior of such objects, and then subclass that class to provide exactly the service that you need.

Windows and Window Components

In this section, you will explore the details of windows and window components. In the process, you will learn how to choose appropriate classes of GUI objects, to access and modify their attributes, and to organize them to cooperate to perform the task at hand.

Windows and Their Attributes

A window has several attributes. The most important ones are its

- title (an empty string by default)
- width and height in pixels
- resizability (true by default)
- background color (white by default)

With the exception of the window's title, the attributes of our label demo program's window have the default values. The background color is white and the window is resizable. The window's initial dimensions are automatically established by shrink-wrapping the window around the label contained in it. We can override the window's default title, an empty string, by supplying another string as an optional `title` argument to the `EasyFrame`

`metflod__init__`. Otfler options are to provide a custom initial width and height in pixels. Note that whenever we supply arguments to a `metflod` call, we use the corresponding keywords for clarity in the code. For example, you might override the dimensions and title of our first program's window as follows:

```
EasyFrame.__init__(self, width = 300, height = 200,
                  title = "Label Demo")
```

Another way to change a window's attributes is to reset them in the window's **attribute dictionary**. Each window or window component maintains a dictionary of its attributes and their values. To access or modify an attribute, the programmer uses the standard subscript notation with the attribute name as a dictionary key. For example, later in the label demo's

`__init__` method, the window's background color can be set to yellow with the following statement:

```
self["background"] = "yellow"
```

Note that `self` in this case refers to the window itself.

The final way to change a window's attributes is to run a method included in the `EasyFrame` class. This class includes the four methods listed in Table 8-1.

EasyFrame Method	What It Does
<code>setBackground(color)</code>	Sets the window's background color to <code>color</code> .
<code>setResizable(aBoolean)</code>	Makes the window resizable (<code>True</code>) or not (<code>False</code>).
<code>setSize(width, height)</code>	Sets the window's width and height in pixels.
<code>setTitle(title)</code>	Sets the window's title to <code>title</code> .

Table 8-1 Methods to change a window's attributes

For example, later in the `LabelDemo` class's `__init__` method, the window's size can be permanently frozen with the following statement:

```
self.setResizable(False)
```

Window Layout

Window components are laid out in the window's two-dimensional **grid**. The grid's rows and columns are numbered from the position (0, 0) in the upper left corner of the window. A window component's row and column position in the grid is specified when the component is added to the window. For example, the next program (`layoutdemo.py`) labels the four quadrants of the window shown in Figure 8-5:

```
class LayoutDemo(EasyFrame):
    """Displays labels in the quadrants."""
```

```

def __init__(self):
    """Sets up the window and the labels."""
    EasyFrame.__init__(self)
    self.addLabel(text = "(0, 0)", row = 0, column = 0)
    self.addLabel(text = "(0, 1)", row = 0, column = 1)
    self.addLabel(text = "(1, 0)", row = 1, column = 0)
    self.addLabel(text = "(1, 1)", row = 1, column = 1)

```

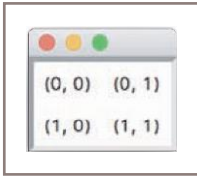


Figure 8-5 Laying out labels in the window's grid

Because the window is `sfrink`-wrapped around the four labels, they appear to be centered in their rows and columns. However, when the user stretches this window, the labels stick to the upper left or northwest corners of their grid positions.

Each type of window component has a default alignment within its grid position. Because labels frequently appear to the left of data entry fields, their default alignment is northwest. The programmer can override the default alignment by including the `sticky` attribute as a keyword argument when the label is added to the window. The values of `sticky` are the strings "N," "S," "E," and "W," or any combination thereof. The next code segment centers the four labels in their grid positions:

```

self.addLabel(text = "(0, 0)", row = 0, column = 0,
              sticky = "NSEW")
self.addLabel(text = "(0, 1)", row = 0, column = 1,
              sticky = "NSEW")
self.addLabel(text = "(1, 0)", row = 1, column = 0,
              sticky = "NSEW")
self.addLabel(text = "(1, 1)", row = 1, column = 1,
              sticky = "NSEW")

```

Now, when the user expands the window, the labels retain their alignments in the exact center of their grid positions.

One final aspect of window layout involves the spanning of a window component across several grid positions. For example, when a window has two components in the first row and only one component in the second row, the latter component might be centered in its row, thus occupying two grid positions. The programmer can force a horizontal and/or vertical spanning of grid positions by supplying the `rowspan` and `columnspan` keyword arguments when adding a component (like merging cells in a table or spreadsheet). The spanning does not take effect unless the alignment of the component is centered along that dimension, however. The next code segment adds the three labels shown in Figure 8-6. The window's grid cells are outlined in the figure.

```

self.addLabel(text = "(0, 0)", row = 0, column = 0,
              sticky = "NSEW")

```

```

self.addLabel(text = "(0, 1)", row = 0, column = 1,
              sticky = "NSEW")
self.addLabel(text = "(1, 0 and 1)", row = 1, column = 0,
              sticky = "NSEW", colspan = 2)

```

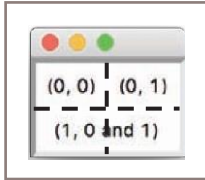


Figure 8-6 Labels with center alignment and a column span of 2

Types of Window Components and Their Attributes

GUI programs use several types of window components, or widgets as they are commonly called. These include labels, entry fields, text areas, command buttons, drop-down menus, sliding scales, scrolling list boxes, canvases, and many others. The `breezypythongui` module includes methods for adding each type of window component to a window. Each such method uses the form

```
self.addComponentType(<arguments>)
```

When this method is called, `breezypythongui`

- Creates an instance of the requested type of window component
- Initializes the component's attributes with default values or any values provided by the programmer
- Places the component in its grid position (the row and column are required arguments)
- Returns a reference to the component

The window components supported by `breezypythongui` are either of the standard `tkinter` types, such as `Label`, `Button`, and `Scale`, or subclasses thereof, such as `FloatField`, `TextArea`, and `EasyCanvas`. A complete list is shown in Table 8-2. Parent classes are shown in parentheses.

Type of Window Component	Purpose
<code>Label</code>	Displays text or an image in the window.
<code>IntegerField(Entry)</code>	A box for input or output of integers.
<code>FloatField(Entry)</code>	A box for input or output of floating-point numbers.
<code>TextField(Entry)</code>	A box for input or output of a single line of text.
<code>TextArea(Text)</code>	A scrollable box for input or output of multiple lines of text.
<code>EasyListbox(Listbox)</code>	A scrollable box for the display and selection of a list of items.

Type of Window Component	Purpose
<code>Button</code>	A clickable command area.
<code>EasyCheckbutton (Checkbutton)</code>	A labeled checkbox.
<code>Radiobutton</code>	A labeled disc that, when selected, deselects related radio buttons.
<code>EasyRadiobuttonGroup (Frame)</code>	Organizes a set of radio buttons, allowing only one at a time to be selected.
<code>EasyMenuBar (Frame)</code>	Organizes a set of menus.
<code>EasyMenubutton (Menubutton)</code>	A menu of drop-down command options.
<code>EasyMenuItem</code>	An option in a drop-down menu.
<code>Scale</code>	A labeled slider bar for selecting a value from a range of values.
<code>EasyCanvas (Canvas)</code>	A rectangular area for drawing shapes or images.
<code>EasyPanel (Frame)</code>	A rectangular area with its own grid for organizing window components.
<code>EasyDialog (simpleDialog.Dialog)</code>	A resource for defining special-purpose popup windows.

Table 8-2 Window components in `breezypythongui`

As with windows, some of a window component's attributes can be set when the component is created, or can be reset by accessing its attribute dictionary at a later time.

Displaying Images

To illustrate the use of attribute options for a label component, let's examine a program (`imagedemo.py`) that displays an image with a caption. The program's window is shown in Figure 8-7.

This program adds two labels to the window. One label displays the image and the other label displays the caption. Unlike earlier examples, the program now keeps variable references to both labels for further processing.

The image label is first added to the window with an empty text string. The program then creates a `PhotoImage` object from an image file and sets the `image` attribute of the image label to this object. Note that the variable used to hold the reference to the image must be an instance variable (prefixed by `self`), rather than a temporary variable. The image file must be



Figure 8-7 Displaying a captioned image

in GIF format. Lastly, the program creates a `Font` object with a non-standard font and resets the text label's `font` and `foreground` attributes to obtain the caption shown in Figure 8-7. The window is `shrink-wrapped` around the two labels and its dimensions are fixed.

Here is the code for the program:

```
from breezypythongui import EasyFrame
from tkinter import PhotoImage
from tkinter.font import Font

class ImageDemo(EasyFrame):
    """Displays an image and a caption."""

    def __init__(self):
        """Sets up the window and the widgets."""
        EasyFrame.__init__(self, title = "Image Demo")
        self.setResizable(False);
        imageLabel = self.addLabel(text = "",
                                   row = 0, column = 0,
                                   sticky = "NSEW")
        textLabel = self.addLabel(text = "Smokey the cat",
                                   row = 1, column = 0,
                                   sticky = "NSEW")

        # Load the image and associate it with the image label.
        self.image = PhotoImage(file = "smokey.gif")
        imageLabel["image"] = self.image

        # Set the font and color of the caption.
        font = Font(family = "Verdana", size = 20,
                   slant = "italic")
        textLabel["font"] = font
        textLabel["foreground"] = "blue"
```

Table 8-3 summarizes the `tkinter.Label` attributes used in this book.

Label Attribute	Type of Value
<code>image</code>	A <code>PhotoImage</code> object (imported from <code>tkinter.PhotoImage</code>). Must be loaded from a GIF file.
<code>text</code>	A string.
<code>background</code>	A color. A label's background is the color of the rectangular area enclosing the text of the label.
<code>foreground</code>	A color. A label's foreground is the color of its text.
<code>font</code>	A <code>Font</code> object (imported from <code>tkinter.Font</code>).

Table 8-3 The `tkinter.Label` attributes

You are encouraged to browse the `breezypythongui` documentation for information on the different types of window components and their attributes. Python also has excellent documentation on the window components at <https://docs.python.org/3/library/tkinter.html#module-tkinter>. For an overview of fonts, see <https://en.wikipedia.org/wiki/Font>. Learning which fonts are available on your system requires some geekery with `tkinter`. A demo program, `fontdemo.py`, that lets you view these fonts is available in the example programs for this book.

In the next section, we will learn how to make GUI programs interactive by responding to user events.

Command Buttons and Responding to Events

A command button is added to a window just like a label, by specifying its text and position in the grid. A button is centered in its grid position by default. The method `addButton` accomplishes all this and returns an object of type `tkinter.Button`. Like a label, a button can display an image, usually a small icon, instead of a string. A button also has a `state` attribute, which can be set to “normal” to enable the button (its default state) or “disabled” to disable it.

GUI programmers often lay out a window and run the application to check its look and feel, before adding the code to respond to user events. Let's adopt this strategy for our next example. This fanciful program (`buttondemo.py`) displays a single label and two command buttons. The buttons allow the user to clear or restore the label. When the user clicks **Clear**, the label is erased, the **Clear** button is disabled, and the **Restore** button is enabled. When the user clicks **Restore**, the label is redisplayed, the **Restore** button is disabled, and the **Clear** button is enabled.

Figure 8-8 shows these two states of the window, followed by the code for the initial version of the program.

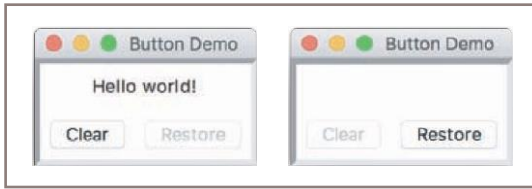


Figure 8-8 Using command buttons

```
class ButtonDemo(EasyFrame):
    """Illustrates command buttons and user events."""

    def __init__(self):
        """Sets up the window, label, and buttons."""
        EasyFrame.__init__(self)

        # A single label in the first row.
        self.label = self.addLabel(text = "Hello world!",
                                   row = 0, column = 0,
                                   columnspan = 2,
                                   sticky = "NSEW")

        # Two command buttons in the second row.
        self.clearBtn = self.addButton(text = "Clear",
                                       row = 1, column = 0)
        self.restoreBtn = self.addButton(text = "Restore",
                                       row = 1, column = 1,
                                       state = "disabled")
```

Note that the **Restore** button, which appears in gray in the window on the left, is initially disabled. When running the first version of the program, the user can click the **Clear** button, but to no effect.

To allow a program to respond to a button click, the programmer must set the button's `command` attribute. There are two ways to do this: either by supplying a keyword argument when the button is added to the window or, later, by assignment to the button's attribute dictionary. The value of the `command` attribute should be a method of no arguments, defined in the program's window class. The default value of this attribute is a method that does nothing.

The completed version of the example program supplies two methods, which are commonly called **event handlers**, for the program's two buttons. Each of these methods resets the label to the appropriate string and then enables and disables the relevant buttons.

```
class ButtonDemo(EasyFrame):
    """Illustrates command buttons and user events."""

    def __init__(self):
        """Sets up the window, label, and buttons."""
```

```

EasyFrame._init_(self)

# A single label in the first row.
self.label = self.addLabel(text = "Hello world!",
                           row = 0, column = 0,
                           colspan = 2,
                           sticky = "NSEW")

# Two command buttons in the second row, with event
# handler methods supplied.
self.clearBtn = self.addButton(text = "Clear",
                               row = 1, column = 0,
                               command = self.clear)
self.restoreBtn = self.addButton(text = "Restore",
                                  row = 1, column = 1,
                                  state = "disabled",
                                  command = self.restore)

# Methods to handle user events.
def clear(self):
    """Resets the label to the empty string and updates
    the button states."""
    self.label["text"] = ""
    self.clearBtn["state"] = "disabled"
    self.restoreBtn["state"] = "normal"

def restore(self):
    """Resets the label to 'Hello world!' and updates
    the button states."""
    self.label["text"] = "Hello world!"
    self.clearBtn["state"] = "normal"
    self.restoreBtn["state"] = "disabled"

```

Now, when the user clicks the **Clear** button, Python automatically runs the `clear` method on the window. Likewise, when the programmer clicks the **Restore** button, Python automatically runs the `restore` method on the window.

Input and Output with Entry Fields

An entry field is a box in which the user can position the mouse cursor and enter a number or a single line of text. This section explores the use of entry fields to allow a GUI program to take input text or numbers from a user and display text or numbers as output.

Text Fields

A text field is appropriate for entering or displaying a single-line string of characters. The programmer uses the method `addTextField` to add a text field to a window. The method returns an object of type `TextField`, which is a subclass of `tkinter.Entry`. Required arguments to `addTextField` are `text` (the string to be initially displayed), `row`, and `column`. Optional arguments are `rowspan`, `colspan`, `sticky`, `width`, and `state`.

A text field is aligned by default to the northwest of its grid cell. A text field has a default width of 20 characters. This represents the maximum number of characters viewable in the box, but the user can continue typing or viewing them by moving the cursor key to the right.

The programmer can set a text field's `state` attribute to "readonly" to prevent the user from editing an output field.

The `TextField` method `getText` returns the string currently contained in a text field. Thus, it serves as an input operation. The method `setText` outputs its string argument to a text field.

Our example program (`textfielddemo.py`) converts a string to uppercase. The user enters text into the input field, clicks the **Convert** button, and views the result in the output field. The output field is read only, to prevent editing the result. Figure 8-9 shows an interaction with the program's window, and the code follows.

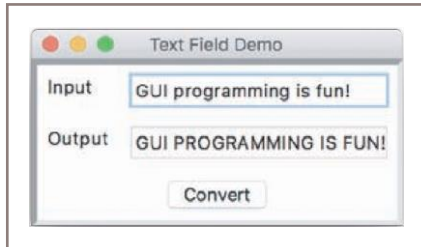


Figure 8-9 Using text fields for input and output

```
class TextFieldDemo(EasyFrame):
    """Converts an input string to uppercase and displays
    the result."""

    def __init__(self):
        """Sets up the window and widgets."""
        EasyFrame.__init__(self, title = "Text Field Demo")

        # Label and field for the input
        self.addLabel(text = "Input", row = 0, column = 0)
        self.inputField = self.addTextField(text = "",
                                           row = 0,
                                           column = 1)

        # Label and field for the output
        self.addLabel(text = "Output", row = 1, column = 0)
        self.outputField = self.addTextField(text = "",
                                           row = 1,
                                           column = 1,
                                           state = "readonly")

        # The command button
        self.addButton(text = "Convert", row = 2, column = 0,
                      colspan = 2, command = self.convert)

        # The event handling method for the button
    def convert(self):
        """Inputs the string, converts it to uppercase,
        and outputs the result."""
        text = self.inputField.getText()
```

```

result = text.upper()
self.outputField.setText(result)

```

Note that the `tflat tfl_``init` method contains about 80% of the program's code. This method is concerned with setting up the window components. The actual program logic is just a few lines of code in the `convert` method. This logic, which takes input data, computes a result, and outputs this result, is similar to the logic of the following, ridiculously simple, terminal-based program:

```

text = input("Input: ")
result = text.upper()
print("Output:", result)

```

Integer and Float Fields for Numeric Data

Although the programmer can use a text field for the input and output of numbers, the data must be converted to strings after input and before output. To simplify the programmer's task, `breezypythongui` includes two types of data fields, called `IntegerField` and `FloatField`, for the input and output of integers and floating-point numbers, respectively.

The methods `addIntegerField` and `addFloatField` are similar in usage to the method `addTextField` discussed earlier. However, instead of an initial `text` attribute, the programmer supplies a `value` attribute. This value must be an integer for an integer field, but can be either an integer or a floating-point number for a float field. The default width of an integer field is 10 characters, whereas the default width of a float field is 20 characters.

The method `addFloatField` allows an optional `precision` argument. Its value is an integer that specifies the precision of the number displayed in the field.

The methods `getNumber` and `setNumber` are used for the input and output of numbers with integer and float fields. The conversion between numbers and strings is performed automatically.

Our example program takes an input integer from a field, computes the square root of this value, and outputs the result, rounded to the nearest hundredth, to a second field. Figure 8-10 shows an interaction with this program (`numberfielddemo.py`), and the code follows.

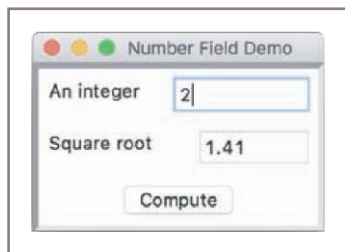


Figure 8-10 Using an integer field and a float field for input and output

```

class NumberFieldDemo(EasyFrame):
    """Computes and displays the square root of an
    input number."""

    def __init__(self):
        """Sets up the window and widgets."""

```

```

EasyFrame.__init__(self, title = "Number Field Demo")

# Label and field for the input
self.addLabel(text = "An integer",
              row = 0, column = 0)
self.inputField = self.addIntegerField(value = 0,
                                       row = 0,
                                       column = 1,
                                       width = 10)

# Label and field for the output
self.addLabel(text = "Square root",
              row = 1, column = 0)
self.outputField = self.addFloatField(value = 0.0,
                                       row = 1,
                                       column = 1,
                                       width = 8,
                                       precision = 2,
                                       state = "readonly")

# The command button
self.addButton(text = "Compute", row = 2, column = 0,
               colspan = 2,
               command = self.computeSqrt)

# The event handling method for the button
def computeSqrt(self):
    """Inputs the integer, computes the square root,
    and outputs the result."""
    number = self.inputField.getNumber()
    result = math.sqrt(number)
    self.outputField.setNumber(result)

```

The program as written will run correctly if the inputs are integers, and these integers are greater than or equal to 0. If the input text is not an integer or is a negative integer, Python raises an exception and, if the program is terminal based, it crashes (you learned about exceptions, like dividing by zero and using an index out of range, in earlier chapters). However, when a GUI-based program raises an exception, the GUI stays alive, allowing the user to edit the input and continue, but a stack trace appears in the terminal window. We next examine how to trap such errors and respond gracefully with error messages.

Using Pop-Up Message Boxes

When errors arise in a GUI-based program, the program often responds by popping up a dialog window with an error message. Such errors are usually the result of invalid input data. The program detects the error, pops up the dialog to inform the user, and, when the user closes the dialog, continues to accept and check input data. In a terminal-based program, this process usually requires an explicit loop structure. In a GUI-based program, Python's implicit event-driven loop continues the process automatically. In this section, we modify an earlier program example to show how this works.

You have seen examples of errors caused by attempting to divide by zero or using a list index that is out of bounds. Python raises an exception or runtime error when these events occur. The square root program raises an exception of type `ValueError` if the input datum is not an integer or is a negative integer. To recover gracefully from this event, we can modify the code of the program's `computeSqrt` method by embedding it in Python's `try-except` statement. The syntax of this statement is a bit like that of the `if-else` statement:

```
try:
    <statements that might raise an exception>
except <exception type>:
    <statements to recover from the exception>
```

In the `try` clause, our program attempts to input the data, compute the result, and output the result, as before. If an exception is raised anywhere in this process, control shifts immediately to the `except` clause. Here, in our example, the program pops up a message box with the appropriate error message. Figure 8-11 shows an interaction with the program, and the modified code follows.

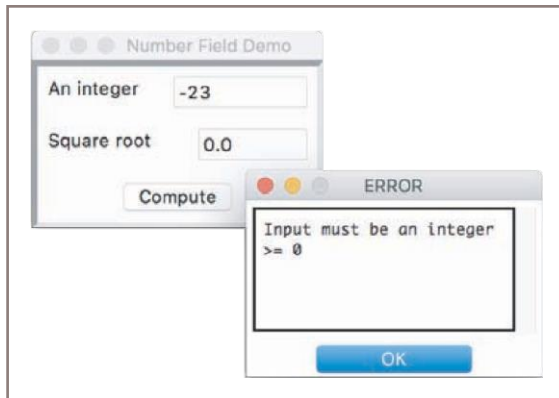


Figure 8-11 Responding to an input error with a message box

```
# The event handling method for the button
def computeSqrt(self):
    """Inputs the integer, computes the square root,
    and outputs the result. Handles input errors
    by displaying a message box."""
    try:
        number = self.inputField.getNumber()
        result = math.sqrt(number)
        self.outputField.setNumber(result)
    except ValueError:
        self.messageBox(title = "ERROR",
                        message = "Input must be an integer >= 0")
```

Python will raise the `ValueError` in the `getNumber` method, if the datum is not an integer, or in the `math.sqrt` function, if the integer is negative. In either case, the `except` clause traps

tfile exception and allows tfile user to correct tfile input after closing tfile message box. A message box is a useful way to alert tfile user to any special event, even if it is not an input error.

Defining and Using Instance Variables

Earlier we said tflat metflods use tfile parameter `self` to call otfler metflods in an object's class or to access tflat object's instance variables. An **instance variable** is used to store data belonging to an individual object. Together, tfile values of an object's instance variables make up its **state**. Tfile state of a given window, for example, includes its title, background color, and dimensions, among otfler tflings. You flave seen tflat a dictionary maintains tfilese data witflin tfile window object. Tfile window class's `__init__` metflod establisfls tfile initial state of a window object wflen it is created, and otfler metflods witflin tflat class are run to access or modify tflis state (to make tfile window larger, cflange its title, or respond to an event). Tfilese basic elements of a window's state are defined and managed in tfile classes

`breezypythongui.EasyFrame` and `tkinter.Frame`.

Wflen you customize an existing class, you can add to tfile state of its objects by including new instance variables. You define tfilese new variables, wflicfl must begin witfl tfile name `self`, witflin tfile class's `__init__` metflod. Tfiley tflen become visible to otfler metflods tflrougflout tfile class definition. A simple example will make tflis clear. A simple counter application is sflown in Figure 8-12.

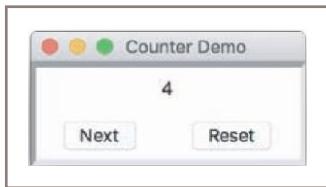


Figure 8-12 The GUI for a counter application

At start-up, tfile window displays a label of 0 and two buttons named **Next** and **Reset**. Wflen tfile user clicks **Next**, tfile window increments tfile number in tfile label; wflen tfile user clicks **Reset**, tfile window resets tfile label to 0.

Clearly, tfile program must flave some way to track tfile value of tfile counter, as it cflanges states after button clicks. We accomplisfl tflis by adding an instance variable to tfile window class in tfile `__init__` metflod and updating tflis variable in tfile event-flandling metflods for tfile buttons. Here is tfile code for tfile `CounterDemo` class:

```
class CounterDemo(EasyFrame):
    """Illustrates the use of a counter with an
    instance variable."""

    def __init__(self):
        """Sets up the window, label, and buttons."""
        EasyFrame.__init__(self, title = "Counter Demo")
        self.setSize(200, 75)

        # Instance variable to track the count.
        self.count = 0
```

```

# A label to display the count in the first row.
(continued)self.label = self.addLabel(text = "0",
                                     row = 0, column = 0,
                                     sticky = "NSEW",
                                     colspan = 2)

# Two command buttons.
self.addButton(text = "Next",
               row = 1, column = 0,
               command = self.next)

self.addButton(text = "Reset",
               row = 1, column = 1,
               command = self.reset)

# Methods to handle user events.
def next(self):
    """Increments the count and updates the display."""
    self.count += 1
    self.label["text"] = str(self.count)

def reset(self):
    """Resets the count to 0 and updates the display."""
    self.count = 0
    self.label["text"] = str(self.count)

```

The separation of the code for setting up and managing the user interface from the code for computation and managing the data is a common design pattern seen in many GUI-based programs. We will explore this design pattern in more detail later in the book.

CASE STUDY: The Guessing Game Revisited

We now pause our survey of GUI components to develop a GUI for a significant application. Chapter 3 presented a guessing game with a terminal-based user interface. We now revise that program to replace the user interface with a GUI.

Request

Replace the terminal-based interface of the guessing game program with a GUI.

Analysis

The program retains the same functions but presents the user with a different look and feel. Figure 8-13 shows a sequence of user interactions with the main window.

Figure 8-13 The GUI for a guessing game

As you can see, the GUI includes a labeled entry field for the user's input guesses, a label for the computer's greeting and responses to the user, and two buttons, one for submitting a guess and another for obtaining a new game. The user plays the game as before, but she enters guesses into the entry field and presses the **Next** button to move the game forward. When the game ends, that button is disabled, and the user can either click the **New game** button to start a new game or close the window to quit.

The program requires one new class, named `GuessingGame`, which extends the `EasyFrame` class.

Laying out the GUI

As in many GUI applications, it's possible to write the code to lay out the user interface before designing the logic (in this case, the game logic) of the application. You can think of this step as part of analysis, in which you create a working **prototype** without any real functionality to get an idea of the application's look and feel. Therefore, here is the code for this part of the process (**guessversion1.py**), which can run without supporting any user interaction:

```
"""
```

```
File: guessversion1.py
```

```
A prototype that lays out the user interface for a GUI-based  
guessing game.
```

```

"""
(continued)
import random
from breezypythongui import EasyFrame

class GuessingGame(EasyFrame):
    """Plays a guessing game with the user."""

    def __init__(self):
        """Sets up the window, widgets, and data."""
        EasyFrame.__init__(self, title = "Guessing Game")
        # Initialize the instance variables for the data
        self.myNumber = random.randint(1, 100)
        self.count = 0
        # Create and add widgets to the window
        greeting = "Guess a number between 1 and 100."
        self.hintLabel = self.addLabel(text = greeting,
                                       row = 0, column = 0,
                                       sticky = "NSEW",
                                       colspan = 2)

        self.addLabel(text = "Your guess", row = 1, column = 0)
        self.guessField = self.addIntegerField(0, row = 1, column = 1)
        # Buttons have no command attributes yet
        self.nextButton = self.addButton(text = "Next", row = 2,
                                         column = 0)

        self.newButton = self.addButton(text = "New game",
                                         row = 2, column = 1)

    def main():
        """Instantiate and pop up the window."""
        GuessingGame().mainloop()

```

(continued)

```

if __name__ == "__main__":
    main()

```

Note that the buttons are added without `command` attributes. Thus, when the user clicks on these buttons, no responses will be triggered. You will develop this functionality in the design phase of the process.

Design

The logic of the guessing game program is to display the computer's greeting and then take user guesses as inputs and respond with hints if the guesses are incorrect. If the user guesses correctly, the process halts with a confirmation message and the number of guesses made. Here is a pseudocode algorithm for the game logic:

```
While True
    count += 1
    Input a guess
    If guess == myNumber
        Output "You've guessed it in", count, "attempts"
        Break
    Else if guess < myNumber
        Output "Sorry, too small"
    Else
        Output "Sorry, too large"
```

As you can see, there is a main loop in which the user's inputs and the computer's hints drive the process forward, until the user guesses correctly. These events will also drive the process forward in a GUI application, but the loop becomes the window's event-driven loop. That is, you will not need an explicit loop in your code; instead, you will embed the logic of the loop's body in an event-handling method. The pseudocode for this method follows:

```
Method nextGuess
    count += 1
    Input a guess
    If guess == myNumber
        Output "You've guessed it in", count, "attempts!"
        Disable the Next button
    Else if guess < myNumber
        Output "Sorry, too small!"
    Else
        Output "Sorry, too large!"
```

This method is triggered whenever the user clicks the **Next** button in the GUI. The inputs now come from the input field, and the outputs go to a label, both also in the GUI. Note that we disable the **Next** button to prevent further user input when a game has finished. The `break` statement is no longer necessary.

The other event in play occurs when the user clicks the **New game** button. In this case, a method is triggered to reset the contents of the GUI to their original state. Here is the pseudocode for this method:

```
Method newGame
    myNumber = a random number between 1 and 100
```

```
count = 0
Hint label = "Guess a number between 1 and 100."
Guess field = 0
Enable the Next button
```

Implementation

The prototype already has most of the code for laying out the GUI. You just have to add the code for the definitions of the two event-handling methods, and set the `command` attributes of the two buttons to these methods when they are added to the window. Here is the code for the two new methods:

```
def nextGuess(self):
    """Processes the user's next guess."""
    self.count += 1
    guess = self.guessField.getNumber()
    if guess == self.myNumber:
        self.hintLabel["text"] = "You've guessed it in " + \
            str(self.count) + " attempts!"
        self.nextButton["state"] = "disabled"
    elif guess < self.myNumber:
        self.hintLabel["text"] = "Sorry, too small!"
    else:
        self.hintLabel["text"] = "Sorry, too large!"

def newGame(self):
    """Resets the data and GUI to their original states."""
    self.myNumber = random.randint(1, 100)
    self.count = 0
    greeting = "Guess a number between 1 and 100."
    self.hintLabel["text"] = greeting
    self.guessField.setNumber(0)
    self.nextButton["state"] = "normal"
```

Note the use of the temporary variables `guess` and `greeting` in these two methods. Because its use is restricted to the method in which it appears, a temporary variable should not begin with the prefix `self`. By contrast, variables that begin with the

prefix `self`, such as `self.count`, `self.hintLabel`, and `self.guessField`, are instance variables. Their purpose is to retain the state of an object (here the instance of `GuessingGame`) between calls of methods. Put metaphorically, the window object does not have to remember the user's guess and the computer's greeting between method calls, but it does have to remember the count, the label, and the entry field. In general, you should try to minimize the use of instance variables, relying on temporaries or parameter names in your methods wherever possible.

Other Useful GUI Resources

Many simple GUI-based applications rely on tfile resources tflat we flave presented tflus far in tflis

clapter. However, as applications become more complex and, in fact, begin to look like tfe ones we use on a daily basis, otfer resources must come into play. Tfe layout of GUI components can be specified in more detail, and groups of components can be nested in multiple frames in a window. Paragrapfls of text can be displayed in scrolling text boxes. Lists of information can be presented for selection in scrolling list boxes, as cfeck boxes, and as radio buttons. Finally, GUI-based programs can be configured to respond to various keyboard and mouse events.

In tflis section, we provide a brief overview of some of tthese advanced resources, so tflat you may use tthem to solve problems in tthe programming projects.

Using Nested Frames to Organize Components

Suppose tflat a GUI requires a row of tthree command buttons beneathfl two columns of labels and text fields, as sflown in Figure 8-14.

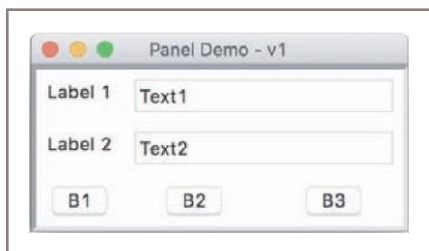


Figure 8-14 Widgets in uneven columns

Tflis grid appears to flave two columns in two rows and tthree columns in a tthird row. Tfe layout is not ragged, but if you look closely, tthe buttons in tthe bottom row are unevenly spaced. Because all of tthe widgets lie in tthe same grid, tthere is no way to center eacfl button in its own column.

A more natural design decomposes tthe window into two nested frames, sometimes called **panels**. Eacfl panel contains its own independent grid. Tfe top panel contains a 2-by-2 grid of labels and entry fields, wflereas tthe bottom panel contains a 1-by-3 grid of buttons. Tfe `brezypythongui` metflod `addPanel` adds a panel to tthe window at a given row and column in tthe window's grid. Tflis metflod returns an instance of tthe `EasyPanel` class, so you can add widgets to it just as if it were a top-level window. Because `EasyPanel` is a descendant of tthe `tkinter.Frame` class, and flas almost tthe same interface as tthe `EasyFrame` class, you can run many of tthe same metflods on a panel object tflat you run on a top-level window object. Tfe user interface for a new version of tthe program tflat organizes tthe widgets in two panels is sflown in Figure 8-15. Note tflat we flave added background colors gray and black to tthe panels for empflasis.

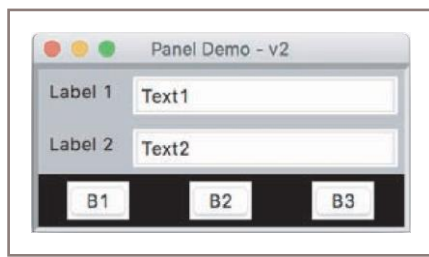


Figure 8-15 Using panels to organize widgets evenly

Here is the code for laying out the GUI shown in Figure 8-15:

```
class PanelDemo(EasyFrame):

    def __init__(self):

        # Create the main frame
        EasyFrame.__init__(self, "Panel Demo - v2")

        # Create the nested frame for the data panel
        dataPanel = self.addPanel(row = 0, column = 0,
                                   background = "gray")

        # Create and add widgets to the data panel
        dataPanel.addLabel(text = "Label 1", row = 0, column = 0,
                            background = "gray")
        dataPanel.addTextField(text = "Text1", row = 0, column = 1)
        dataPanel.addLabel(text = "Label 2", row = 1, column = 0,
                            background = "gray")
        dataPanel.addTextField(text = "Text2", row = 1, column = 1)

        # Create the nested frame for the button panel
        buttonPanel = self.addPanel(row = 1, column = 0,
                                     background = "black")

        # Create and add buttons to the button panel
        buttonPanel.addButton(text = "B1", row = 0, column = 0)
        buttonPanel.addButton(text = "B2", row = 0, column = 1)
        buttonPanel.addButton(text = "B3", row = 0, column = 2)
```

As you can see from this code, the grids of the two panels are independent, as multiple widgets appear to be placed in the same rows and columns. When you design a complex interface like this one, be sure to draw a sketch of the panels with their grids, so you can determine the positions of the widgets and eliminate some guesswork.

Multi-Line Text Areas

Although text fields are useful for entering and displaying single lines of text, some applications need to display larger chunks of text with multiple lines. For instance, the message box introduced earlier displays a multi-line message in a scrolling text area. In a manner similar to the editing window of a word processor, a text area widget allows the program to output and the user to input and edit multiple lines of text.

The method `addTextArea` adds a text area to the window. The required arguments are the initial text to display, the row, and the column. Optional arguments include a width and height in columns (characters) and rows (lines), with defaults of 80 and 5, respectively. The final optional argument is called `wrap`. This argument tells the text area whether to do a line of text when it reaches the right border of the viewable area. The default value of `wrap` is "none," which causes a line of text to continue invisibly beyond the right border. The other values are "word" and "char," which break a line at a word or a character, and continue the text on the next line.

The `addTextArea` method returns an object of type `TextArea`, a subclass of `tkinter.Text`. This object recognizes three important methods: `getText`, `setText`, and `appendText`. The first two methods have the same effect as they do with a text field. The `appendText` method does not replace the text in the text area with its string argument, but instead appends this string to the end of the string currently displayed there. A text area can be disabled to prevent editing, but this disables its input and output methods as well. Therefore, before text is input or output, a disabled text area must be re-enabled.

You can use a text area to recast the user interface of the investment calculator program of Chapter 3. As shown in Figure 8-16, the GUI inputs the initial balance, the number of years, and the interest rate via entry fields. When the user clicks the **Compute** button, the program displays the table of results in a text area.

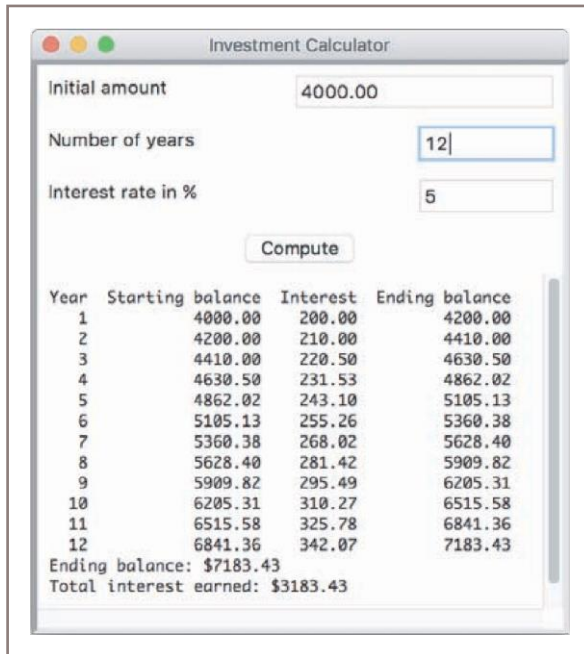


Figure 8-16 Displaying data in a multi-line text area

Here is the code for the window class:

```
class TextAreaDemo(EasyFrame):
    """An investment calculator demonstrates the use of a
    multi-line text area."""

    def __init__(self):
        """Sets up the window and widgets."""
        EasyFrame.__init__(self, "Investment Calculator")
        self.addLabel(text = "Initial amount", row = 0, column = 0)
        self.addLabel(text = "Number of years", row = 1, column = 0)
        self.addLabel(text = "Interest rate in %", row = 2, column = 0)
        self.amount = self.addFloatField(value = 0.0, row = 0, column = 1)
        self.period = self.addIntegerField(value = 0, row = 1, column = 1)
        self.rate = self.addIntegerField(value = 0, row = 2, column = 1)
```

```

self.outputArea = self.addTextArea("", row = 4, column = 0,
                                   columnspan = 2,
                                   width = 50, height = 15)

self.compute = self.addButton(text = "Compute", row = 3, column = 0,
                              columnspan = 2,
                              command = self.compute)

# Event handling method.
def compute(self):
    """Computes the investment schedule based on the inputs
    and outputs the schedule."""
    # Obtain and validate the inputs
    startBalance = self.amount.getNumber()
    rate = self.rate.getNumber() / 100
    years = self.period.getNumber()
    if startBalance == 0 or rate == 0 or years == 0:
        return

    # Set the header for the table
    result = "%4s%18s%10s%16s\n" % ("Year",
                                   "Starting balance",
                                   "Interest",
                                   "Ending balance")

    # Compute and append the results for each year
    totalInterest = 0.0
    for year in range(1, years + 1):
        interest = startBalance * rate
        endBalance = startBalance + interest
        result += "%4d%18.2f%10.2f%16.2f\n" % \
            (year, startBalance, interest, endBalance)
        startBalance = endBalance
        totalInterest += interest

    # Append the totals for the period
    result += "Ending balance: $%0.2f\n" % endBalance
    result += "Total interest earned: $%0.2f\n" % totalInterest

    # Output the result while preserving read-only status
    self.outputArea["state"] = "normal"
    self.outputArea.setText(result)
    self.outputArea["state"] = "disabled"

```

File Dialogs

As anyone who has opened or saved a file on a modern computer knows, GUI-based programs allow the user to browse the computer's file system with **file dialogs**. Figure 8-17 shows a file dialog asking for an input file on my computer.

Python's `tkinter.filedialog` module includes two functions, `askopenfilename` and `asksaveasfilename`, to support file access in a GUI-based program. Each function pops up the standard file dialog for the user's particular computer system. If the user selects the dialog's **Cancel** button, the function returns the empty string. Otherwise, when the user selects the

Open or **Save** button, the file function returns the full pathname of the file selected (opening or saving) or entered as input (saving only) in the dialog. The program can then use the filename to open the file for input or output in the usual manner.

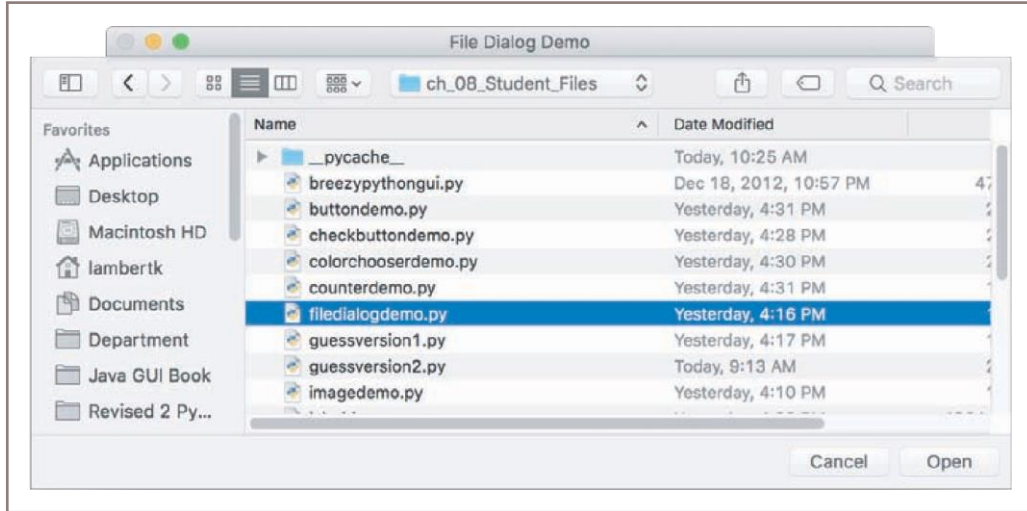


Figure 8-17 A file dialog

For purposes of this book, we use the following syntax with these two functions:

```
fList = [("Python files", "*.py"), ("Text files", "*.txt")]
filename = tkinter.filedialog.askopenfilename(parent = self,
                                             filetype = fList)
```

```
filename = tkinter.filedialog.asksaveasfilename(parent = self)
```

Note that you can use the optional `filetypes` argument to mask the types of files available for input. In our example, we want the user to be able to open files with a `.py` or `.txt` extension, and no others. Table 8-4 lists all of the optional arguments one can supply to the two file dialog functions.

Argument	Value
<code>defaultextension</code>	The extension to add to the filename, if not given by the user (ignored by the open dialog).
<code>filetypes</code>	A sequence of (label, pattern) tuples. Specifies the file types available for input.
<code>initialdir</code>	A string representing the directory in which to open the dialog.
<code>initialfile</code>	A string representing the filename to display in the save dialog name field.
<code>parent</code>	The dialog's parent window.
<code>title</code>	A string to display in the dialog's title bar.

Table 8-4 The optional arguments to the file dialog methods

You can use a file dialog and a text area to create a simple browser that allows the user to view text files. As shown in Figure 8-18, when the user clicks the **Open** button and chooses a file from the file dialog, the text of the file is input and displayed in the window's text area.

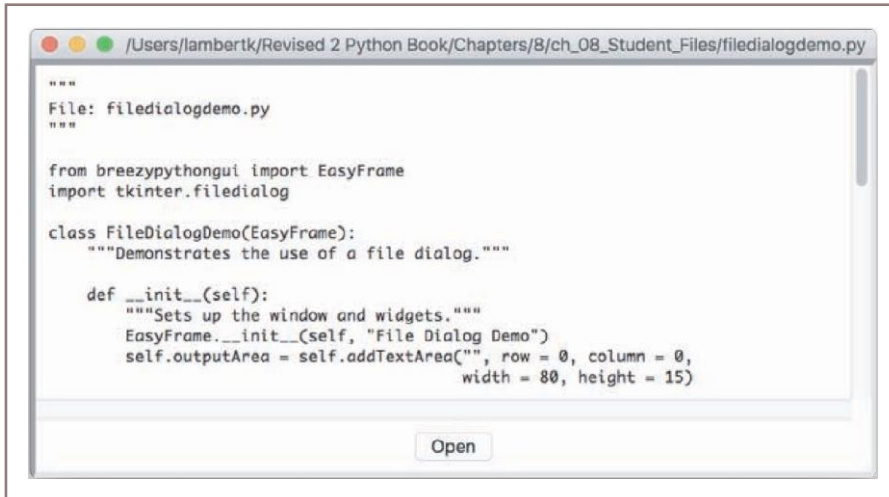


Figure 8-18 A simple file browser

Here is the code for the window class:

```
from breezypythongui import EasyFrame
import tkinter.filedialog

class FileDialogDemo(EasyFrame):
    """Demonstrates the use of a file dialog."""

    def __init__(self):
        """Sets up the window and widgets."""
        EasyFrame.__init__(self, "File Dialog Demo")
        self.outputArea = self.addTextArea("", row = 0,
                                           column = 0,
                                           width = 80,
                                           height = 15)

        self.addButton(text = "Open", row = 1, column = 0,
                       command = self.openFile)

    # Event handling method.
    def openFile(self):
        """Pops up an open file dialog, and if a file is
        selected, displays its text in the text area and
        its pathname in the title bar."""
        fList = [("Python files", "*.py"),
                 ("Text files", "*.txt")]
```



```

self.addButton(text = "Username", row = 1, column = 0,
               command = self.getUserName)

def getUserName(self):
    text = self.prompterBox(title = "Input Dialog",
                           promptString = "Your username:")
    self.label["text"] = "Hi " + name + "!"

```

Check Buttons

A **check button** consists of a label and a box that a user can select or deselect with the mouse. Check buttons often represent a group of several options, any number of which may be selected at the same time. The application program can either respond immediately when a check button is manipulated, or examine the state of the button at a later point in time.

As a simple example, let's assume that a restaurant serves chicken dinners with a standard set of sides. These include French fries, green beans, and applesauce. A customer can omit any of the sides from her order, and vegetarians will want to omit the chicken. The user selects these options via check buttons and clicks the **Place order** button to place her order. A message box then pops up with a summary of the order. Figure 8-20 shows the user interface for the program (**checkbuttondemo.py**).

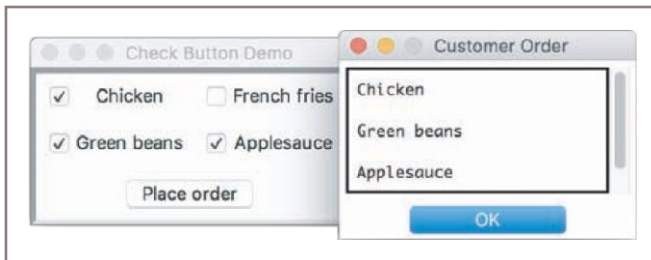


Figure 8-20 Using check buttons

The method `addCheckbutton` expects a `text` argument (the button's label) and an optional `command` argument (a method to be triggered when the user checks or unchecks the button), and returns an object of type `EasyCheckbutton`. The `EasyCheckbutton` method `isChecked` returns `True` if the button is checked, or `False` otherwise. Here is the code for the demo program:

```

class CheckbuttonDemo(EasyFrame):
    """Allows the user to place a restaurant order from a set
    of options."""

    def __init__(self):
        """Sets up the window and widgets."""
        EasyFrame.__init__(self, "Check Button Demo")

```

```

# Add four check buttons
self.chickCB = self.addCheckbutton(text = "Chicken",
                                   row = 0, column = 0)

self.taterCB = self.addCheckbutton(text = "French fries",
                                   row = 0, column = 1)

self.beanCB = self.addCheckbutton(text = "Green beans",
                                   row = 1, column = 0)

self.sauceCB = self.addCheckbutton(text = "Applesauce",
                                   row = 1, column = 1)

# Add the command button
self.addButton(text = "Place order", row = 2, column = 0,
               colspan = 2, command = self.placeOrder)

# Event handling method.
def placeOrder(self):
    """Display a message box with the order information."""
    message = ""
    if self.chickCB.isChecked():
        message += "Chicken\n\n"
    if self.taterCB.isChecked():
        message += "French fries\n\n"
    if self.beanCB.isChecked():
        message += "Green beans\n\n"
    if self.sauceCB.isChecked():
        message += "Applesauce\n"
    if message == "": message = "No food ordered!"
    self.messageBox(title = "Customer Order",
                   message = message)

```

Radio Buttons

Click buttons allow a user to select multiple options in any combination. When the user must be restricted to one selection only, the set of options can be presented as a group of **radio buttons**. Like a click button, a radio button consists of a label and a control widget. One of the buttons is normally selected by default at program start-up. When the user selects a different button in the same group, the previously selected button automatically deselects.

To illustrate the use of radio buttons, consider another restaurant scenario, where a customer has two choices of meats, potatoes, and vegetables, and must choose exactly one of each food type (our apologies to vegetarians). Three radio button groups can be set up to take this order, as shown in the program's user interface (**radiobuttondemo.py**) in Figure 8-21. The default options are chicken, French fries, and applesauce.

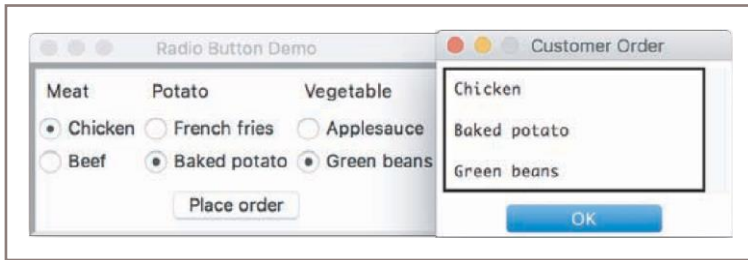


Figure 8-21 Using radio buttons

To add radio buttons to a window, the programmer first adds the radio button group to which these buttons will belong. The method `addRadiobuttonGroup` expects the grid coordinates as required arguments. Optional arguments are `orient` (whose default is “vertical”), `rowspan`, and `columnspan`. In the case of a vertically aligned button group, `rowspan` should be set to the number of buttons, and `columnspan` should be likewise set for a horizontally aligned group. The method returns an object of type `EasyRadiobuttonGroup`, which is a subclass of `tkinter.Frame`. This allows the programmer to place a custom background color in the region of the button group.

The `EasyRadiobuttonGroup` method `getSelectedButton` returns the currently selected radio button in a radio button group. The method `setSelectedButton` selects a radio button under program control. Once a radio button group is created, the programmer can add radio buttons to it with the `EasyRadiobuttonGroup` method `addRadiobutton`. This method expects a `text` argument (the button’s label) and an optional `command` argument (a zero-argument method to be triggered when the button is selected). The method returns an object of type `tkinter.Radiobutton`.

Here is the code for the main window of the radio button demo program:

```
class RadiobuttonDemo(EasyFrame):
    """Allows the user to place a restaurant order from a set
    of options."""

    def __init__(self):
        """Sets up the window and widgets."""
        EasyFrame.__init__(self, "Radio Button Demo")

        # Add the label, button group, and buttons for meats
        self.addLabel(text = "Meat", row = 0, column = 0)
        self.meatGroup = self.addRadiobuttonGroup(row = 1,
                                                    column = 0,
                                                    rowspan = 2)
        defaultRB = self.meatGroup.addRadiobutton(text = "Chicken")
        self.meatGroup.setSelectedButton(defaultRB)
        self.meatGroup.addRadiobutton(text = "Beef")

        # Add the label, button group, and buttons for potatoes
        self.addLabel(text = "Potato", row = 0, column = 1)
```


You cannot use ttle `computeSqrt` method directly as ttle event flandler, because `computeSqrt` does not flave a parameter for ttle event. Instead, you package a call of `computeSqrt` witflin a `lambda` function tflat accepts ttle event as an argument and ignores it. You can set event flandlers for ttle keyboard return event for otfler fllds in a similar manner.

Working with Colors

You flave seen tflat you can set ttle background color of a window and most widgets using ttle string values of common colors, such as “red” and “blue.” However, in Chapter 7, you learned tflat there are millions of colors available to ttle programmer wfllo uses ttle RGB scheme. You saw (in Chapter 7) tflat Turtle graphics and image processing use a triple witfl ttle form (R, G, B) to represent a color in ttle scheme. Each integer in ttle triple represents ttle saturation level of red, green, and blue in ttle given color. To work witfl colors in a GUI-based application, you must be aware of two otfler ways of representing RGB values in Python. Python represents an RGB value as a string containing a six-digit hexadecimal number, of ttle form “0xRRGGBB” wflere ttle pairs of digits indicate ttle values of red, green, and blue in hex. Ttle `tkinter` module also accepts ttle simpler representation “#RRGGBB” for hexadecimal values. We call ttle representation a **hex string**. Table 8-5 lists some basic Python color values in ordinary, RGB triple, and hex string notations.

Ordinary Value	RGB Triple	Hex String
"black"	(0, 0, 0)	"#000000"
"red"	(255, 0, 0)	"#ff0000"
"green"	(0, 255, 0)	"#00ff00"
"blue"	(0, 0, 255)	"#0000ff"
"gray"	(127, 127, 127)	"#7f7f7f"
"white"	(255, 255, 255)	"#ffffff"

Table 8-5 Some basic colors and their RGB values

For example, to set ttle background color of a window to a less intense shade of red tthan ttle maximum value denoted by “red,” you migfl run ttle statement

```
self["background"] = "#DD0000"
```

Now suppose you want to use a random color in a GUI. You must find a way to map a triple of tthree random integers, (R, G, B) , to a hex string. Note tflat each integer in ttle (R, G, B)

notation maps to two hex digits in the corresponding hex string. You could use one of the conversion algorithms discussed in Chapter 4 to perform these conversions, but Python's built-in `hex` function already does that:

```
>>> hex(255)
'0xff'
>>> hex(8)
'0x8'
```

To obtain just the hex digits, you would slice away the `'0x'` prefix as follows:

```
>>> hex(255)[2:]
'ff'
>>> hex(8)[2:]
'8'
```

To handle the case of a single digit, you would pad the string to the left by prepending a `'0'`, as follows:

```
>>> hexDigits = hex(8)[2:]
>>> if len(digits) == 1:
    hexDigits = '0' + hexDigits
>>> hexDigits
'08'
```

Because such conversions might occur frequently, let's define a function, named `rgbToHexString`, that expects a triple of integers as arguments and returns the corresponding hex string. Here is the code (in `rgb.py`):

```
def rgbToHexString(rgbTriple):
    """Converts the rgbTriple (R, G, B) to a hex string
    of the form #RRGGBB."""
    hexString = ""
    for i in rgbTriple: # Iterate through the triple
        twoDigits = hex(i)[2:]
        if len(twoDigits) == 1:
            twoDigits = '0' + twoDigits
        hexString += twoDigits
    return '#' + hexString
```

You are now in a position to easily create colors from RGB triples, including random ones, for a GUI application, as follows:

```
>>> rgbToHexString((255, 255, 255))
'ffffff'
>>> rgbToHexString((10, 8, 32))
'#0a0820'
>>> from random import randint
>>> triple = (randint(0, 255), randint(0, 255), randint(0, 255))
>>> triple
(107, 104, 145)
>>> rgbToHexString(triple)
'#6b6891'
```

Using a Color Chooser

Most graphics software packages allow the user to pick a color with a standard color chooser. This is a dialog that presents a color wheel from which the user can choose a color with the mouse. Python's `tkinter.colorchooser` module includes an `askcolor` function for this purpose. Figure 8-22 shows screenshots of a demo program (`colorchooserdemo.py`) that uses this resource. The window displays the current color in a **canvas** widget (a rectangular area that supports graphics operations). When the user clicks the **Choose color** button in the main window, a color chooser dialog pops up. When the user clicks **OK** to close the dialog, the main window updates its fields and canvas with the information about the chosen color.

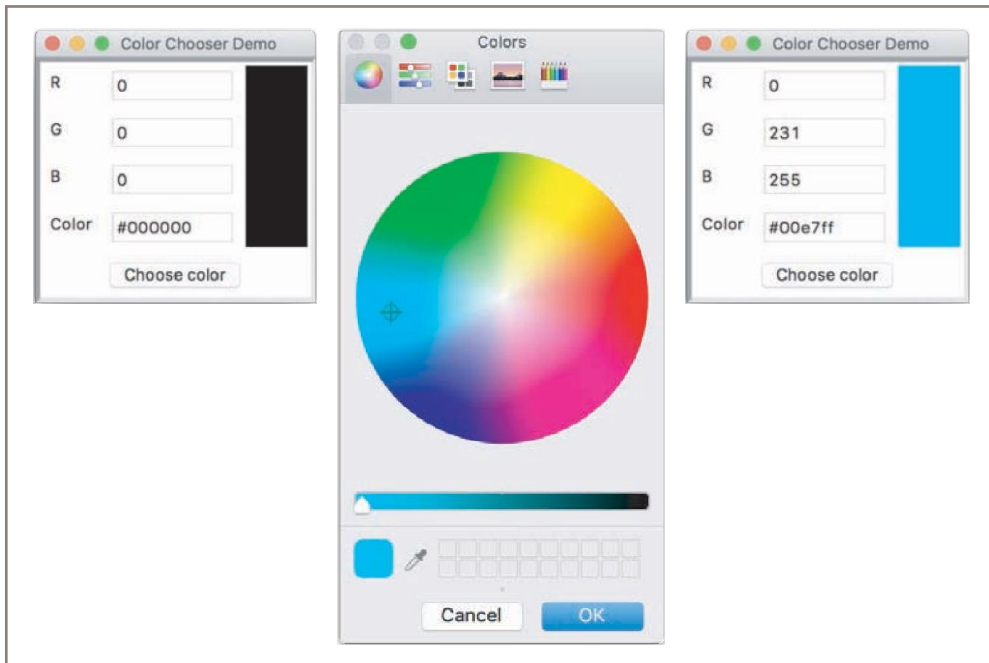


Figure 8-22 Using a color chooser

The `tkinter.colorchooser.askcolor` function returns a tuple of two elements. If the user has clicked **OK** in the dialog, the first element in the tuple is a nested tuple containing the three RGB values, and the second element is the hex string value of the color. If the user has clicked **Cancel** in the dialog, both elements in the tuple are `None`. Because the RGB values are returned as floating-point numbers, the demo program converts them to integers for display. Here is the code for the main window:

```
import tkinter.colorchooser

class ColorPicker(EasyFrame):
    """Displays the results of picking a color."""
```

```

def __init__(self):
    """Sets up the window and widgets."""
    EasyFrame.__init__(self,
                       title = "Color Chooser Demo")

    # Labels and output fields
    self.addLabel('R', row = 0, column = 0)
    self.addLabel('G', row = 1, column = 0)
    self.addLabel('B', row = 2, column = 0)
    self.addLabel("Color", row = 3, column = 0)
    self.r = self.addIntegerField(value = 0,
                                  row = 0, column = 1)
    self.g = self.addIntegerField(value = 0,
                                  row = 1, column = 1)
    self.b = self.addIntegerField(value = 0,
                                  row = 2, column = 1)
    self.hex = self.addTextField(text = "#000000",
                                  row = 3, column = 1,
                                  width = 10)

    # Canvas with an initial black background
    self.canvas = self.addCanvas(row = 0, column = 2,
                                  rowspan = 4,
                                  width = 50,
                                  background = "#000000")

    # Command button
    self.addButton(text = "Choose color", row = 4,
                   column = 0, colspan = 3,
                   command = self.chooseColor)

    # Event handling method
    def chooseColor(self):
        """Pops up a color chooser and outputs the results."""
        colorTuple = tkinter.colorchooser.askcolor()
        if not colorTuple[0]: return
        (r, g, b), hexString = colorTuple
        self.r.setNumber(int(r))
        self.g.setNumber(int(g))
        self.b.setNumber(int(b))
        self.hex.setText(hexString)
        self.canvas["background"] = hexString

```

This concludes our introduction to GUI programming. You are now ready to program applications like the ones you use on a daily basis. Although it might seem like we have covered many features of GUIs, we have only scratched the surface. For a discussion on the use of other window components, such as canvases for graphics, sliding scales, and scrolling list boxes, as well as responding to different types of mouse events, consult the [breezypythongui](http://home.wlu.edu/~lambertk/breezypythongui/) website at <http://home.wlu.edu/~lambertk/breezypythongui/>.