Introduction to Python

Topics

- Introduction and Environment Setup
- Keywords, Variables and Constants
- Data Types, Input and Output
- Operators
- Branching and Looping
- Functions and Modules
- Sequential and Non-Sequential Data Structures
- File Handling
- Exception Handling
- Date Time Functions
- Database Connectivity
- Object Oriented Programming

Introduction

What is Python?
Python is an interpreted, high-level and general-purpose programming language.
Python is programming language as well as scripting language.

History
Invented in the Netherlands, early 90s by Guido van Rossum.
Python was conceived in the late 1980s and its implementation was started in December 1989.
Guido Van Rossum is fan of ‘Monty Python’s Flying Circus’, this is a famous TV show in Netherlands.
Named after Monty Python.
Open sourced from the beginning.
Python 2.0 was released on 16 October 2000.
Python 3.0 was released on 3 December 2008.
Python 2.7's end-of-life date was initially set at 2015 then postponed to 2020.

Environment Setup
For this course, we will be using Jupyter Notebook from Anaconda
You can download the anaconda installer from https://www.anaconda.com/products/individual
Install Anaconda to a directory path that does not contain spaces or unicode characters.
Do not install as Administrator unless admin privileges are required.
If you are behind a company proxy, you may need to do some additional set up. See how to set up your proxy.
Reference: https://docs.anaconda.com/anaconda/install/

Verifying your installation
Windows: Click Start, search or select Anaconda Navigator from the menu.
Windows: Click Start, search, or select Anaconda Prompt from the menu.
Reference: https://docs.anaconda.com/anaconda/install/verify-install/

Who uses python today…
Python is being applied in real revenue-generating products by real companies. For instance:
-Google makes extensive use of Python in its web search system and employs Python's creator.
-Intel, Cisco, Hewlett-Packard, Seagate, Qualcomm, and IBM use Python for hardware testing.
-ESRI uses Python as an end-user customization tool for its popular GIS mapping products.
-The YouTube video sharing service is largely written in Python.

What can we do with Python?
-Data Science
-Machine Learning
-Deep Learning
-Artificial Intelligence
-Web Designing
-Web Deployment
-System programming
-Component Integration
-Database Programming
-Gaming, Robotics and more

Keywords, Variables and Constants

Keywords: These are reserved words, and you cannot use them as constant or variable or any other identifier names. Some of the Python keywords are: if, else, elif, and, or, not, print, return, is, in, while, for, try, except, finally

Variables: A variable is a named location used to store data in the memory. This means that when you create a variable you reserve some space in memory. Python variables do not need explicit declaration to reserve memory space. The declaration happens automatically when you assign a value to a variable.

```
a=10
name=“Peter”
```

Constants: A constant is a type of variable whose value cannot be changed. It is helpful to think of constants as containers that hold information which cannot be changed later.

```
PI = 3.14
GRAVITY = 9.8
import constant
print(constant.PI)
print(constant.GRAVITY)
```

Data Types, Input and Output

Every value in Python has a datatype. 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.
-Numbers
-String
-List
-Tuple
-Dictionary
-Set

a = ‘python’
print(a, "is of type", type(a)) #python is of type <class 'str'>

We have two built-in functions print() and input() to perform I/O task.

Operators

Operators are special symbols in Python that carry out arithmetic or logical computation.

Arithmetic Operators: x = 10 and y = 4

Operator	Meaning	Example
+	Add two operands or unary plus	x + y+ 2 = 16
-	Subtract right operand from the left or unary minus	x - y- 2 = 4
*	Multiply two operands	x * y = 40
/	Divide left operand by the right one (always results into float)	x / y = 2.5
%	Modulus - remainder of the division of left operand by the right	x % y (remainder of x/y) = 2
//	Floor division - division that results into whole number adjusted to the left in the number line	x // y = 2
**	Exponent - left operand raised to the power of right	x**y (x to the power y) = 1000

Comparison Operators: x = 10 and y = 4

Operator	Meaning	Example
>	Greater than - True if left operand is greater than the right	x > y = True
<	Less than - True if left operand is less than the right	x < y = False
==	Equal to - True if both operands are equal	x == y = False
!=	Not equal to - True if operands are not equal	x != y = True
>=	Greater than or equal to - True if left operand is greater than or equal to the right	x >= y = True
<=	Less than or equal to - True if left operand is less than or equal to the right	x <= y = False

Logical Operators: Let x = True and y = False

Operator	Meaning	Example
and	True if both the operands are true	x and y = False
or	True if either of the operands is true	x or y = True
not	True if operand is false (complements the operand)	not x = False

Bitwise Operators: Let x = 10 (0000 1010 in binary) and y = 4 (0000 0100 in binary)

Operator	Meaning	Example
&	Bitwise AND	x & y = 0 (0000 0000)
|	Bitwise OR	x | y = 14 (0000 1110)
~	Bitwise NOT	~x = -11 (1111 0101)
^	Bitwise XOR	x ^ y = 14 (0000 1110)
>>	Bitwise right shift	x >> 2 = 2 (0000 0010)
<<	Bitwise left shift	x << 2 = 40 (0010 1000)

Assignment Operators:

Operator	Example	Equivalent to
=	x = 5	x = 5
+=	x += 5	x = x + 5
-=	x -= 5	x = x - 5
*=	x *= 5	x = x * 5
/=	x /= 5	x = x / 5
%=	x %= 5	x = x % 5
//=	x //= 5	x = x // 5
**=	x **= 5	x = x ** 5
&=	x &= 5	x = x & 5
|=	x |= 5	x = x | 5
^=	x ^= 5	x = x ^ 5
>>=	x >>= 5	x = x >> 5
<<=	x <<= 5	x = x << 5

Identity Operators:

Operator	Meaning	Example
is	True if the operands are identical (refer to the same object)	x is True
is not	True if the operands are not identical (do not refer to the same object)	x is not True

Membership Operators:

Operator	Meaning	Example
in	True if value/variable is found in the sequence	5 in x
not in	True if value/variable is not found in the sequence	5 not in x

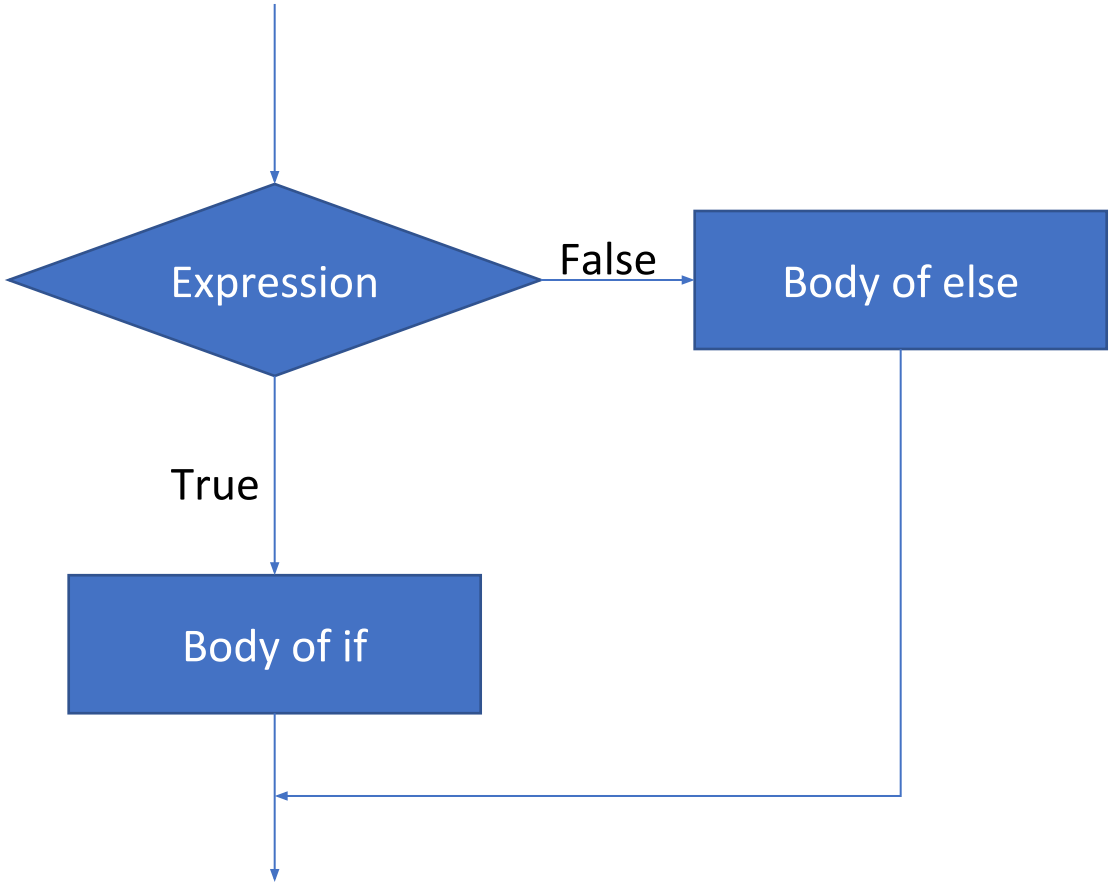
Branching and Looping

if… else statement

Syntax

```
if test expression:
    Body of if
else:
    Body of else
```

Flowchart

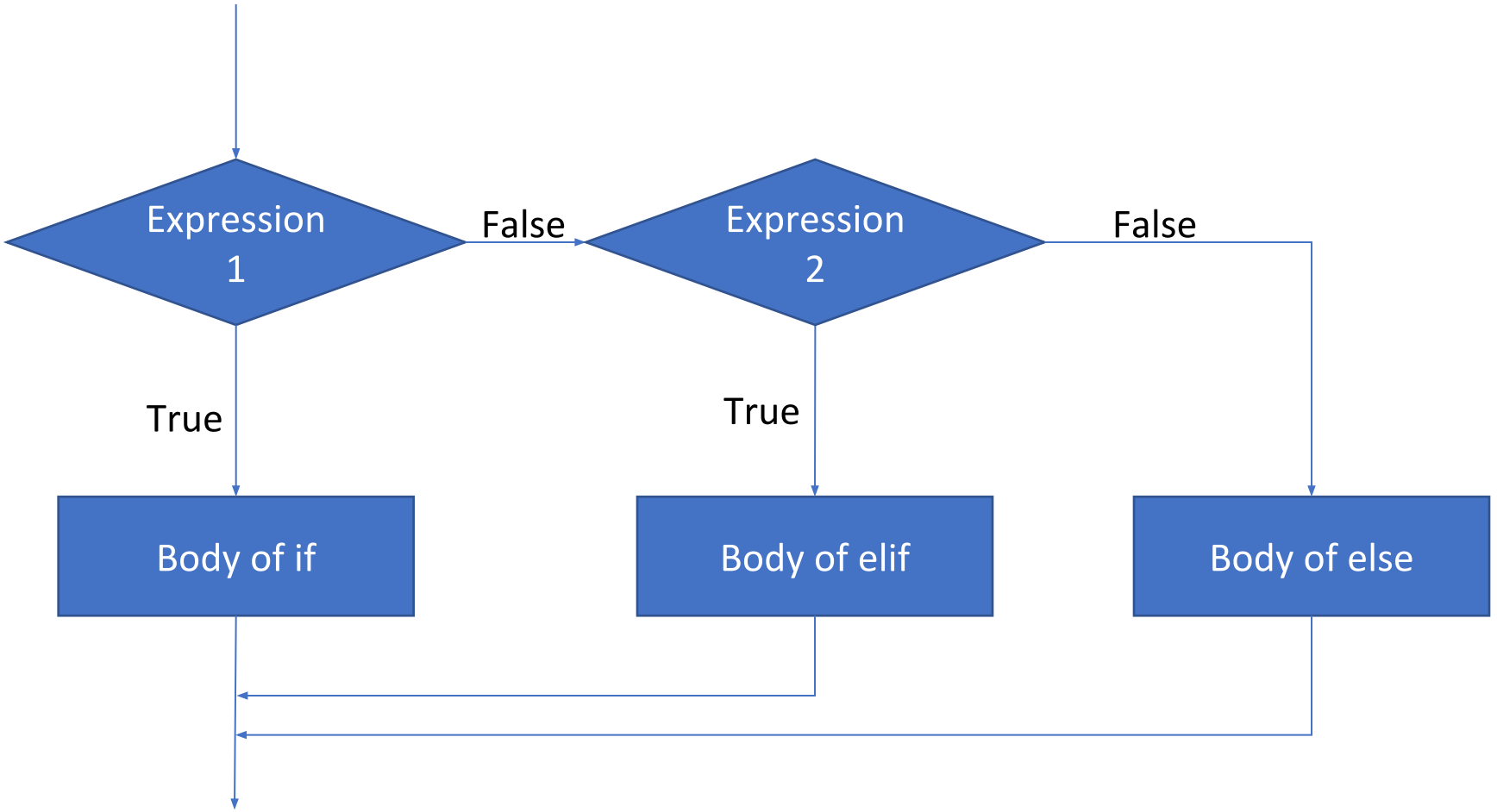


if… elif… else statement

Syntax

```
if test expression1:
    Body of if
elif test expression2:
    Body of elif
else:
    Body of else
```

Flowchart



Nested if statement

```
if test expression1:
    if test expression2:(Body of if)
        Body of nested if
    else:
        Body of nested else
else:
    Body of else
```

Looping

For Loop

Syntax

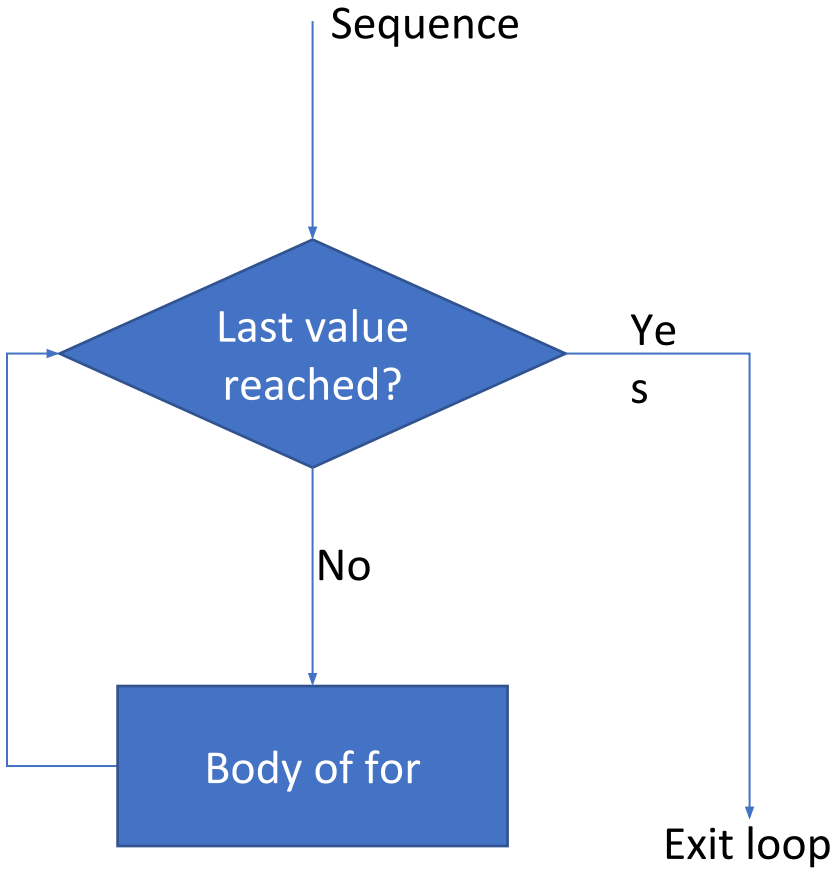
```
sequence = [1,2,3]
for i in sequence:
    print(i)

for i in range(0,len(sequence)):
    print(sequence[i])
```

Output:

1
2
3

Flowchart



Looping

While Loop

Syntax

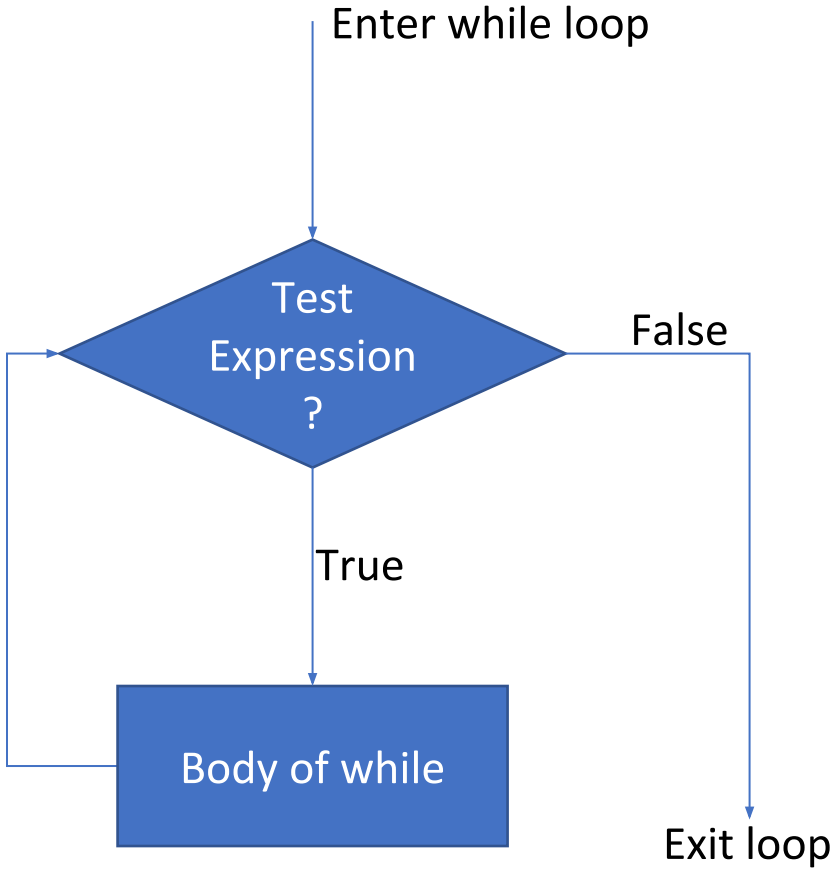
```
while test_expression:
    Body of while
```

```
start = 0
while start<4:
    print(start)
    start +=1
```

Output:

1
2
3

Flowchart



Break, continue and pass

The break statement terminates the loop containing it.

The continue statement is used to skip the rest of the code inside a loop for the current iteration only.

Nothing happens when the pass is executed.

Function

A function is a group of related statements that performs a specific task.

Function can be of two types:

i. Built-in functions
ii. User define functions

Built-in functions: Python has several functions that are readily available for use. These functions are called built-in functions. Example: print(), input(), chr(), ord(), list(), int()

User define functions: Functions that we define ourselves to do certain specific task are referred as user-defined functions. It also helps in reducing the repeated code.

Syntax:

```
def function_name(parameters):
    statement(s)
```

Function Arguments

You can call a function by using the following types of formal arguments:

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

Required arguments

```
def name_func(name)
   print(name)
name_function(name = 'Sam')
name_function(name = 'Tom')
```

Keyword arguments

```
def name_func(name, age)
   print(name,age)
name_function(age = 26, name = 'Sam')
name_function(name = 'Romeo', age = 30)
```

Default arguments

```
def name_func(name, age = 26)
   print(name)
name_function(name = 'Sam')
name_function(name = 'Victor', age= 30)
```

Variable-length arguments

```
def name_func(name, *vartuple)
   print(name)
   for i in vartuple:
      print(i)
name_function('Sam')
name_function('Sam','Peter','Harry'
)
```

Recursive function

When a function call itself within its body statement then these functions are known are recursive function and the phenomena is known as recursion.
A recursive function should always have terminating function to prevent it from going into RecursionError
The maximum depth of recursion is 1000. If limit is crossed, it results in RecursionError

Example: factorial of a number with recursion

```
def factorial(x):
    if x == 1:
        return 1
    else:
        return (x * factorial(x-1))

num = 5
print("The factorial of", num, "is", factorial(num))
```

Example: factorial of a number without recursion

```
def factorial(x):
    result=1
    for i in range(1,x+1):
        result *= i
    return (result)

num = 5
print("The factorial of", num, "is", factorial(num))
```

Lambda function

Lambda forms can take any number of arguments but return just one value in the form of an expression. They cannot contain commands or multiple expressions.

Syntax:

lambda [arg1 [,arg2,.....argn]]:expression

Example:

```
sum_no = lambda arg1, arg2: arg1+arg2
print(sum_no(10,20)) #30
```

Global vs. Local variables

Variables that are defined inside a function body have a local scope, and those defined outside have a global scope.

```
total = 0
def sum_no( arg1, arg2 ):
  total = arg1 + arg2
  print("Inside the function local total : ", total) #30
  return total;

sum_no( 10, 20 )
print("Outside the function global total : ", total) #0
```

```
my_list = [1, 5, 4, 6, 8, 11, 3, 12]
new_list = list(map(lambda x: x * 2 , my_list))
print(new_list) #[2, 10, 8, 12, 16, 22, 6, 24]
```

Modules

A file containing a set of functions you want to include in your application.

To import module:
import module_name

Example: import pandas

To import with renaming
import pandas as pd

Importing function(s) from a module
from math import pi,sqrt

Sequential and Non-Sequential Data Structures

Sequential Data Structures: Data structures which follow a sequence to store the data in it are known as sequential data structure. Example: String, List, Tuple

String

Concatenation of two or more string.

```
str_1='Hello'
str_2= 'Python'
print(str_1+str_2) #HelloPython
print(str_1*3) #HelloHelloHello
print(str_ex[2:-2] #th
```

Escape Sequence	Description
\newline	Backslash and newline ignored
\\	Backslash
\'	Single quote
\"	Double quote
\a	ASCII Bell
\b	ASCII Backspace
\f	ASCII Formfeed
\n	ASCII Linefeed
\r	ASCII Carriage Return
\t	ASCII Horizontal Tab
\v	ASCII Vertical Tab
\ooo	Character with octal value ooo
\xHH	Character with hexadecimal value HH

Method	Description
capitalize()	Converts the first character to upper case
casefold()	Converts string into lower case
center()	Returns a centered string
count()	Returns the number of times a specified value occurs in a string
encode()	Returns an encoded version of the string
endswith()	Returns true if the string ends with the specified value
expandtabs()	Sets the tab size of the string
find()	Searches the string for a specified value and returns the position of where it was found
format()	Formats specified values in a string
format_map()	Formats specified values in a string
index()	Searches the string for a specified value and returns the position of where it was found
isalnum()	Returns True if all characters in the string are alphanumeric
isalpha()	Returns True if all characters in the string are in the alphabet
isdecimal()	Returns True if all characters in the string are decimals
isdigit()	Returns True if all characters in the string are digits
isidentifier()	Returns True if the string is an identifier
islower()	Returns True if all characters in the string are lower case
isnumeric()	Returns True if all characters in the string are numeric
isprintable()	Returns True if all characters in the string are printable
isspace()	Returns True if all characters in the string are whitespaces
istitle()	Returns True if the string follows the rules of a title

isupper()	Returns True if all characters in the string are upper case
join()	Joins the elements of an iterable to the end of the string
ljust()	Returns a left justified version of the string
lower()	Converts a string into lower case
lstrip()	Returns a left trim version of the string
maketrans()	Returns a translation table to be used in translations
partition()	Returns a tuple where the string is parted into three parts
replace()	Returns a string where a specified value is replaced with a specified value
rfind()	Searches the string for a specified value and returns the last position of where it was found
rindex()	Searches the string for a specified value and returns the last position of where it was found
rjust()	Returns a right justified version of the string
rpartition()	Returns a tuple where the string is parted into three parts
rsplit()	Splits the string at the specified separator, and returns a list
rstrip()	Returns a right trim version of the string
split()	Splits the string at the specified separator, and returns a list
splitlines()	Splits the string at line breaks and returns a list
startswith()	Returns true if the string starts with the specified value
strip()	Returns a trimmed version of the string
swapcase()	Swaps cases, lower case becomes upper case and vice versa
title()	Converts the first character of each word to upper case
translate()	Returns a translated string
upper()	Converts a string into upper case

List

List comprehension

```
list1 = [1,2,3,4,5]
list2 = [ x*2 for i in list1]
print(list2)
```

Output:
[2,4,6,8,10]

Method	Description
append()	Add an element to the end of the list
extend()	Add all elements of a list to the another list
insert()	Insert an item at the defined index
remove()	Removes an item from the list
pop()	Removes and returns an element at the given index
clear()	Removes all items from the list
index()	Returns the index of the first matched item
count()	Returns the count of the number of items passed as an argument
sort()	Sort items in a list in ascending order
reverse()	Reverse the order of items in the list
copy()	Returns a shallow copy of the list

Tuple

Since tuples are quite similar to lists, both are used in similar situations. However, there are certain advantages of implementing a tuple over a list. Below listed are some of the main advantages:

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

Method	Description
count()	Returns the number of times a specified value occurs in a tuple
index()	Searches the tuple for a specified value and returns the position of where it was found

Non-Sequential Data Structures

Non-Sequential Data Structures: Data structures which does not follow a sequence to store the data in it are known as non-sequential data structure. Example: Set, Dictionary

Set

A set is an unordered collection of items.
Every set element is unique.
Set cannot have mutable items.
Set is mutable.
set() for empty set.

Function	Description
all()	Returns True if all elements of the set are true (or if the set is empty).
any()	Returns True if any element of the set is true. If the set is empty, returns False.
enumerate()	Returns an enumerate object. It contains the index and value for all the items of the set as a pair.
len()	Returns the length (the number of items) in the set.
max()	Returns the largest item in the set.
min()	Returns the smallest item in the set.
sorted()	Returns a new sorted list from elements in the set(does not sort the set itself).
sum()	Returns the sum of all elements in the set.

Method	Description
add()	Adds an element to the set
clear()	Removes all elements from the set
copy()	Returns a copy of the set
difference()	Returns the difference of two or more sets as a new set
difference_update()	Removes all elements of another set from this set
discard()	Removes an element from the set if it is a member. (Do nothing if the element is not in set)
intersection()	Returns the intersection of two sets as a new set
intersection_update()	Updates the set with the intersection of itself and another
isdisjoint()	Returns True if two sets have a null intersection
issubset()	Returns True if another set contains this set
issuperset()	Returns True if this set contains another set
pop()	Removes and returns an arbitrary set element. Raises KeyError if the set is empty
remove()	Removes an element from the set. If the element is not a member, raises a KeyError
symmetric_difference()	Returns the symmetric difference of two sets as a new set
symmetric_difference_update()	Updates a set with the symmetric difference of itself and another
union()	Returns the union of sets in a new set
update()	Updates the set with the union of itself and others

Dictionary

Method	Description
clear()	Removes all items from the dictionary.
copy()	Returns a shallow copy of the dictionary.
fromkeys(seq[, v])	Returns a new dictionary with keys from `seq` and value equal to `v` (defaults to `None`).
get(key[,d])	Returns the value of the `key`. If the `key` does not exist, returns `d` (defaults to `None`).
items()	Return a new object of the dictionary's items in (key, value) format.
keys()	Returns a new object of the dictionary's keys.
pop(key[,d])	Removes the item with the `key` and returns its value or `d` if `key` is not found. If `d` is not provided and the `key` is not found, it raises `KeyError`.
popitem()	Removes and returns an arbitrary item (key, value). Raises `KeyError` if the dictionary is empty.
setdefault(key[,d])	Returns the corresponding value if the `key` is in the dictionary. If not, inserts the `key` with a value of `d` and returns `d` (defaults to `None`).
update([other])	Updates the dictionary with the key/value pairs from `other`, overwriting existing keys.
values()	Returns a new object of the dictionary's values

Function	Description
all()	Return `True` if all keys of the dictionary are True (or if the dictionary is empty).
any()	Return `True` if any key of the dictionary is true. If the dictionary is empty, return `False`.
len()	Return the length (the number of items) in the dictionary.
sorted()	Return a new sorted list of keys in the dictionary.

File Handling

In Python, a file operation takes place in the following order:

Open a file

Read or write (perform operation)

Close the file

```
f = open("test.txt")    # open file in current directory
f.close()           # closing the file

with open("test.txt", encoding = 'utf-8') as f:
  # perform file operations, no need to explicitly close the file

with open("test.txt",'w',encoding = 'utf-8') as f:
  f.write("my first file\n")
  f.write("This file\n\n")
  f.write("contains three lines\n")

f = open("test.txt",'r',encoding = 'utf-8')
f.read(4)    # read the first 4 data 'This'
f.read(4)    # read the next 4 data 'is  '
```

Mode	Description
r	Opens a file for reading. (default)
w	Opens a file for writing. Creates a new file if it does not exist or truncates the file if it exists.
x	Opens a file for exclusive creation. If the file already exists, the operation fails.
a	Opens a file for appending at the end of the file without truncating it. Creates a new file if it does not exist.
t	Opens in text mode. (default)
b	Opens in binary mode.
+	Opens a file for updating (reading and writing)

```
f.read()     # read in the rest till end of file 'my first file\nThis
file\ncontains three lines\n'
f.seek(0)   # bring file cursor to initial position

for line in f:
    print(line, end = '')

f.readline() #'This is my first file\n'
f.readline() #'This file\n'
f.readline() #'contains three lines\n'
f.readline() #''

f.readlines() #['This is my first file\n', 'This file\n', 'contains three
lines\n']
```

Method	Description
close()	Closes an opened file. It has no effect if the file is already closed.
detach()	Separates the underlying binary buffer from the TextIOBase and returns it.
fileno()	Returns an integer number (file descriptor) of the file.
flush()	Flushes the write buffer of the file stream.
isatty()	Returns True if the file stream is interactive.
read(n)	Reads at most n characters from the file. Reads till end of file if it is negative or None.
readable()	Returns True if the file stream can be read from.
readline(n=-1)	Reads and returns one line from the file. Reads in at most n bytes if specified.
readlines(n=-1)	Reads and returns a list of lines from the file. Reads in at most n bytes/characters if specified.
seek(offset,from=SEEK_SET)	Changes the file position to offset bytes, in reference to from (start, current, end).
seekable()	Returns True if the file stream supports random access.
tell()	Returns the current file location.
truncate(size=None)	Resizes the file stream to size bytes. If size is not specified, resizes to current location.
writable()	Returns True if the file stream can be written to.
write(s)	Writes the string s to the file and returns the number of characters written.
writelines(lines)	Writes a list of lines to the file.

Exception

While we write program, we can make certain mistakes that leads to error when we try to run it. When an error occurs, or exception as we call it, Python will normally stop and generate an error message.
Error can be of 2 types:

i. Syntax Error : As the name suggest, it is caused by incorrect syntax.
ii. Logical Error (Exception): This is caused when the code passes syntax error and there is some flaw in our logic.

We can handle these logical error by using keywords like try and except and this process is known as exception handling.

Syntax:

```
try:
    #code block to execute
except:
    #error handling
```

Types of Built-in-Exception

Exception	Cause of Error
AssertionError	Raised when an assert statement fails.
AttributeError	Raised when attribute assignment or reference fails.
EOFError	Raised when the input() function hits end-of-file condition.
FloatingPointError	Raised when a floating point operation fails.
GeneratorExit	Raise when a generator's close() method is called.
ImportError	Raised when the imported module is not found.
IndexError	Raised when the index of a sequence is out of range.
KeyError	Raised when a key is not found in a dictionary.
KeyboardInterrupt	Raised when the user hits the interrupt key (Ctrl+C or Delete).
MemoryError	Raised when an operation runs out of memory.
NameError	Raised when a variable is not found in local or global scope.
NotImplementedError	Raised by abstract methods.
OSError	Raised when system operation causes system related error.
OverflowError	Raised when the result of an arithmetic operation is too large to be represented.
ReferenceError	Raised when a weak reference proxy is used to access a garbage collected referent.
RuntimeError	Raised when an error does not fall under any other category.
StopIteration	Raised by next() function to indicate that there is no further item to be returned by iterator.
SyntaxError	Raised by parser when syntax error is encountered.
IndentationError	Raised when there is incorrect indentation.
TabError	Raised when indentation consists of inconsistent tabs and spaces.
SystemError	Raised when interpreter detects internal error.
SystemExit	Raised by sys.exit() function.
TypeError	Raised when a function or operation is applied to an object of incorrect type.
UnboundLocalError	Raised when a reference is made to a local variable in a function or method, but no value has been bound to that variable.
UnicodeError	Raised when a Unicode-related encoding or decoding error occurs.
UnicodeEncodeError	Raised when a Unicode-related error occurs during encoding.
UnicodeDecodeError	Raised when a Unicode-related error occurs during decoding.
UnicodeTranslateError	Raised when a Unicode-related error occurs during translating.
ValueError	Raised when a function gets an argument of correct type but improper value.
ZeroDivisionError	Raised when the second operand of division or modulo operation is zero.

User defined exception

```
class Error(Exception):
    """Base class for other exceptions"""
    pass


class ValueTooSmallError(Error):
    """Raised when the input value is too small"""
    pass


class ValueTooLargeError(Error):
    """Raised when the input value is too large"""
    pass


# you need to guess this number
number = 10

# user guesses a number until he/she gets it right
while True:
    try:
        i_num = int(input("Enter a number: "))
        if i_num < number:
            raise ValueTooSmallError
        elif i_num > number:
            raise ValueTooLargeError
        break
    except ValueTooSmallError:
        print("This value is too small, try again!")
        print()
    except ValueTooLargeError:
        print("This value is too large, try again!")
        print()

print("Congratulations! You guessed it correctly”)
```

Try… Except…

Finally…

The try statement in Python can have an optional finally clause. This clause is executed no matter what, and is generally used to release external resources.

The finally block is a place to put any code that must execute, whether the try-block raised an exception or not.

Syntax:

```
try:
   #code block to execute
except:
   #error handling
finally:
   #must execute codes
```

Date Time Functions

Python has a module named datetime to work with dates and times

Commonly used classes in the datetime module are:

- date Class
- time Class
- datetime Class
- timedelta Class

```
#Get Current Date and Time
import datetime

datetime_object = datetime.datetime.now()
print(datetime_object)

#Get Current Date
date_object = datetime.date.today()
print(date_object)
```

```
#Working with date class
#Date object to represent a date
d = datetime.date(2020, 12, 13)
print(d)

#alternative way
from datetime import date
a = date(2020, 12, 13)
print(a)

#Get current date
today = date.today()
print("Current date =", today)

#Get date from a timestamp
timestamp = date.fromtimestamp(1326244364)
print("Date =", timestamp)

#Print today's year, month and day
today = date.today()

print("Current year:", today.year)
print("Current month:", today.month)
print("Current day:", today.day)
```

```
#Working with time class
from datetime import time

# time(hour = 0, minute = 0, second = 0)
a = time()
print("a =", a)

# time(hour, minute and second)
b = time(11, 34, 56)
print("b =", b)

# time(hour, minute and second)
c = time(hour = 11, minute = 34, second = 56)
print("c =", c)

# time(hour, minute, second, microsecond)
d = time(11, 34, 56, 234566)
print("d =", d)

a = time(11, 34, 56)
print("hour =", a.hour)
print("minute =", a.minute)
print("second =", a.second)
print("microsecond =", a.microsecond)
```

```
#Working with datetime class
from datetime import datetime

#datetime(year, month, day)
a = datetime(2018, 11, 28)
print(a)

# datetime(year, month, day, hour, minute,
second, microsecond)
b = datetime(2017, 11, 28, 23, 55, 59, 342380)
print(b)

a = datetime(2017, 11, 28, 23, 55, 59, 342380)
print("year =", a.year)
print("month =", a.month)
print("hour =", a.hour)
print("minute =", a.minute)
print("timestamp =", a.timestamp())
```

```
#Working with timedelta class
from datetime import datetime, date

t1 = date(year = 2018, month = 7, day = 12)
t2 = date(year = 2017, month = 12, day = 23)
t3 = t1 - t2
print("t3 =", t3)

t4 = datetime(year = 2018, month = 7, day = 12, hour = 7, minute = 9, second = 33)
t5 = datetime(year = 2019, month = 6, day = 10, hour = 5, minute = 55, second = 13)
t6 = t4 - t5
print("t6 =", t6)

print("type of t3 =", type(t3))
print("type of t6 =", type(t6))

from datetime import timedelta

t1 = timedelta(weeks = 2, days = 5, hours = 1, seconds = 33)
t2 = timedelta(days = 4, hours = 11, minutes = 4, seconds = 54)
t3 = t1 - t2

print("t3 =", t3)
```

```
#represents the difference between two dates or times.

from datetime import timedelta

t1 = timedelta(seconds = 33)
t2 = timedelta(seconds = 54)
t3 = t1 - t2

print("t3 =", t3) #negative timedelta
print("t3 =", abs(t3))
```

```
#Time duration in seconds
from datetime import timedelta

t = timedelta(days = 5, hours = 1, seconds = 33,
microseconds = 233423)
print("total seconds =", t.total_seconds())
```

Format datetime
Python has strftime() and strptime() methods to handle this.

The strftime() method is defined under classes date, datetime and time. The method creates a formatted string from a given date, datetime or time object.

```
#Format date using strftime()
from datetime import datetime
now = datetime.now()# current date and time

t = now.strftime("%H:%M:%S")
print("time:", t)

s1 = now.strftime("%m/%d/%Y, %H:%M:%S")
# mm/dd/YY H:M:S format
print("s1:", s1)

s2 = now.strftime("%d/%m/%Y, %H:%M:%S")
# dd/mm/YY H:M:S format
print("s2:", s2)
```

The strptime() method creates a datetime object from a given string (representing date and time).

```
from datetime import datetime

date_string = "21 June, 2018"
print("date_string =", date_string)

date_object = datetime.strptime(date_string, "%d %B, %Y")
print("date_object =", date_object)
```

The table below shows all the codes that you can pass to the strftime() method.

Directive	Meaning	Example
%a	Abbreviated weekday name.	Sun, Mon, ...
%A	Full weekday name.	Sunday, Monday, ...
%w	Weekday as a decimal number.	0, 1, ..., 6
%d	Day of the month as a zero-padded decimal.	01, 02, ..., 31
%-d	Day of the month as a decimal number.	1, 2, ..., 30
%b	Abbreviated month name.	Jan, Feb, ..., Dec
%B	Full month name.	January, February, ...
%m	Month as a zero-padded decimal number.	01, 02, ..., 12
%-m	Month as a decimal number.	1, 2, ..., 12
%y	Year without century as a zero-padded decimal number.	00, 01, ..., 99
%-y	Year without century as a decimal number.	0, 1, ..., 99
%Y	Year with century as a decimal number.	2013, 2019 etc.
%H	Hour (24-hour clock) as a zero-padded decimal number.	00, 01, ..., 23
%-H	Hour (24-hour clock) as a decimal number.	0, 1, ..., 23
%I	Hour (12-hour clock) as a zero-padded decimal number.	01, 02, ..., 12
%-I	Hour (12-hour clock) as a decimal number.	1, 2, ... 12
%p	Locale’s AM or PM.	AM, PM
%M	Minute as a zero-padded decimal number.	00, 01, ..., 59
%-M	Minute as a decimal number.	0, 1, ..., 59
%S	Second as a zero-padded decimal number.	00, 01, ..., 59
%-S	Second as a decimal number.	0, 1, ..., 59
%f	Microsecond as a decimal number, zero-padded on the left.	000000 - 999999
%z	UTC offset in the form +HHMM or -HHMM.	
%Z	Time zone name.	
%j	Day of the year as a zero-padded decimal number.	001, 002, ..., 366
%-j	Day of the year as a decimal number.	1, 2, ..., 366
%U	Week number of the year (Sunday as the first day of the week). All days in a new year preceding the first Sunday are considered to be in week 0.	00, 01, ..., 53
%W	Week number of the year (Monday as the first day of the week). All days in a new year preceding the first Monday are considered to be in week 0.	00, 01, ..., 53
%c	Locale’s appropriate date and time representation.	Mon Sep 30 07:06:05 2013
%x	Locale’s appropriate date representation.	09/30/13
%X	Locale’s appropriate time representation.	0.295891204
%%	A literal '%' character.	%

Database Connectivity

```
#Create Connection
import mysql.connector
mydb = mysql.connector.connect(
  host="localhost",
  user="testuser",
  password="test123"
)

print(mydb)

mycursor = mydb.cursor()

mycursor.execute("CREATE DATABASE mydatabase")
#creating database

mycursor.execute("SHOW DATABASES") #query to list
database

for x in mycursor:
  print(x) #print database
```

```
#Connect to database
import mysql.connector
mydb = mysql.connector.connect(
  host="localhost",
  user="testuser",
  password="test123“,
  database="mydatabase"
)

mycursor.execute("SHOW TABLES") #check table
for x in mycursor:
  print(x)
mycursor = mydb.cursor()
mycursor.execute("CREATE TABLE customers (name
VARCHAR(255), address VARCHAR(255))")

mycursor.execute("DROP TABLE customers") #drop
table
sql = "DROP TABLE IF EXISTS customers"
mycursor.execute(sql)
```

Insert

```
mycursor.execute("CREATE TABLE customers (id INT
AUTO_INCREMENT PRIMARY KEY, name VARCHAR(255), address
VARCHAR(255))")

#if table exist than use alter table
mycursor.execute("ALTER TABLE customers ADD COLUMN id INT
AUTO_INCREMENT PRIMARY KEY")

#insert record
sql = "INSERT INTO customers (name, address) VALUES (%s, %s)"
val = ("John", "Highway 21")
mycursor.execute(sql, val)
mydb.commit()
print(mycursor.rowcount, "record inserted.")

#if last row id required after inserting record
sql = "INSERT INTO customers (name, address) VALUES (%s, %s)"
val = ("Michelle", "Blue Village")
mycursor.execute(sql, val)
mydb.commit()
print("1 record inserted, ID:", mycursor.lastrowid)
```

```
#insert multiple records
sql = "INSERT INTO customers (name, address)
VALUES (%s, %s)"
val = [
  ('Peter', 'Lowstreet 4'),
  ('Amy', 'Apple st 652'),
  ('Hannah', 'Mountain 21'),
  ('Michael', 'Valley 345'),
  ('Sandy', 'Ocean blvd 2'),
  ('Betty', 'Green Grass 1'),
  ('Richard', 'Sky st 331'),
  ('Susan', 'One way 98'),
  ('Vicky', 'Yellow Garden 2'),
  ('Ben', 'Park Lane 38'),
  ('William', 'Central st 954'),
  ('Chuck', 'Main Road 989'),
  ('Viola', 'Sideway 1633')
]

mycursor.executemany(sql, val)
mydb.commit()
print(mycursor.rowcount, "was inserted.")
```

Select

```
mycursor.execute("SELECT * FROM customers") #selecting all
records

myresult = mycursor.fetchall()
for x in myresult:
  print(x)

mycursor.execute("SELECT name, address FROM customers")
#selecting columns

myresult = mycursor.fetchall()
for x in myresult:
  print(x)

mycursor.execute("SELECT * FROM customers") #only one row
is returned

myresult = mycursor.fetchone()
print(myresult)
```

Where

```
sql = "SELECT * FROM customers WHERE address ='Park Lane 38’” #exact match

mycursor.execute(sql)
myresult = mycursor.fetchall()
for x in myresult:
  print(x)

sql = "SELECT * FROM customers WHERE address LIKE '%way%’” #wildcard character % used
for startswith, endswith or include

mycursor.execute(sql)
myresult = mycursor.fetchall()
for x in myresult:
  print(x)

sql = "SELECT * FROM customers WHERE address = %s"
adr = ("Yellow Garden 2", )

mycursor.execute(sql, adr)
myresult = mycursor.fetchall()
for x in myresult:
  print(x)
```

Order By

```
sql = "SELECT * FROM customers ORDER BY name“          #Accending order

mycursor.execute(sql)
myresult = mycursor.fetchall()
for x in myresult:
  print(x)

sql = "SELECT * FROM customers ORDER BY name DESC“          #Descending order

mycursor.execute(sql)
myresult = mycursor.fetchall()
for x in myresult:
  print(x)
```

Delete

```
sql = "DELETE FROM customers WHERE address = 'Mountain 21'"

mycursor.execute(sql)
mydb.commit()
print(mycursor.rowcount, "record(s) deleted")
```

Update

```
sql = "UPDATE customers SET address = 'Canyon 123' WHERE address = 'Valley 345'"

mycursor.execute(sql)
mydb.commit()
print(mycursor.rowcount, "record(s) affected")
```

Limit

```
mycursor.execute("SELECT * FROM customers LIMIT 5")

myresult = mycursor.fetchall()
for x in myresult:
  print(x)

mycursor.execute("SELECT * FROM customers LIMIT 5 OFFSET 2")

myresult = mycursor.fetchall()
for x in myresult:
  print(x)
```

Object Oriented Programming

Python is an object-oriented programming language.
Almost everything in Python is an object, with its properties and methods.
A Class is like an object constructor, or a "blueprint" for creating objects.
An object (instance) is an instantiation of a class.

An object has two characteristics:

- attributes
- behavior

Concepts of object-oriented programing are important like:

- Inheritance
- Encapsulation
- Polymorphism