Python Language
&
Library

By

Srikanth Pragada

COPYRIGHT

ABOUT THE AUTHOR

Srikanth Pragada is the director of Srikanth Technologies, a software training company. He started programming in early 90s and worked with more than 15 different programming languages.

Srikanth Pragada holds the following certifications:

- Sun Certified Java Programmer
- Sun Certified Web Component Developer
- Sun Certified Business Component Developer
- Oracle Database SQL Certified Expert
- Oracle PL/SQL Developer Certified Associate
- Microsoft Certified Technology Specialist for .NET 4.0 (Web Applications)

He currently conducts online, classroom and onsite training on C, Java, Oracle, Microsoft.NET, Python, Data Science, Angular, AWS and React.

His website www.srikanthtechnologies.com provides online examinations, programs, projects, articles and his blog.

He provides a lot of example programs of various languages and technologies using https://github.com/srikanthpragada.

When he is not teaching or learning, he would like to visit new places, read books, play sports and listen to music.

He can be reached through his email address srikanthpragada@gmail.com.

HOW TO USE THIS MATERIAL

You are suggested to read relevant content before and after attending the class.

Use picture and text to grasp the concept. Programs are to illustrate how to implement the concepts. Try the programs given in this material in your system.

REQUEST FOR FEEDBACK

We have taken considerable effort to ensure accuracy of the contents of this material. However, if you come across any mistakes or have any suggestions to improve the quality of this material, please take a few minutes of your valuable time to send an email to me at **srikanthpragada@gmail.com.**

Alternatively, you can visit my website **http://www.srikanthtechnologies.com/feedback.aspx** and provide feedback there.

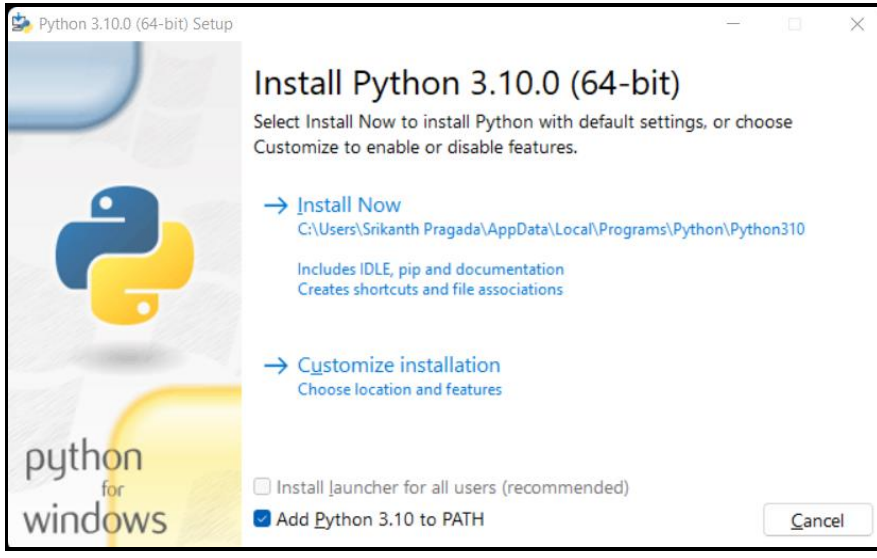
TABLE OF CONTENT

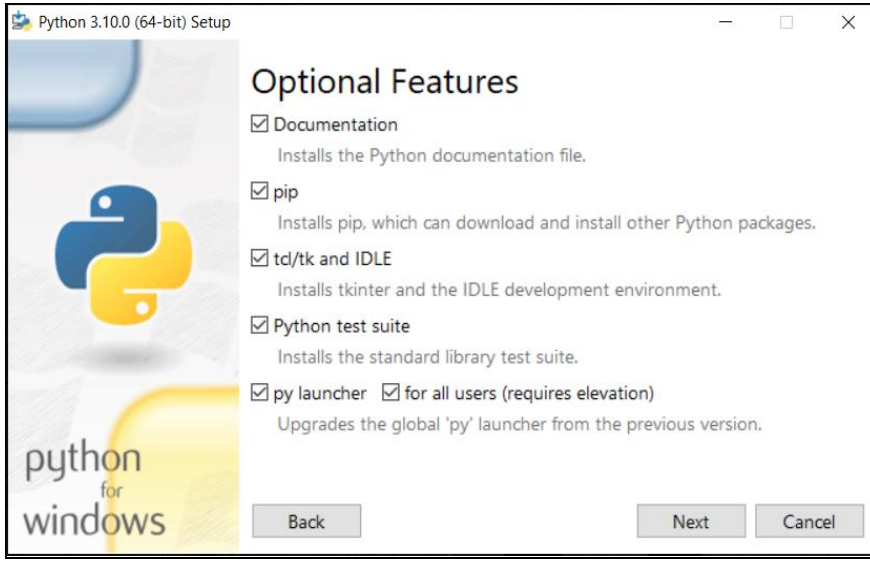
PYTHON LANGUAGE

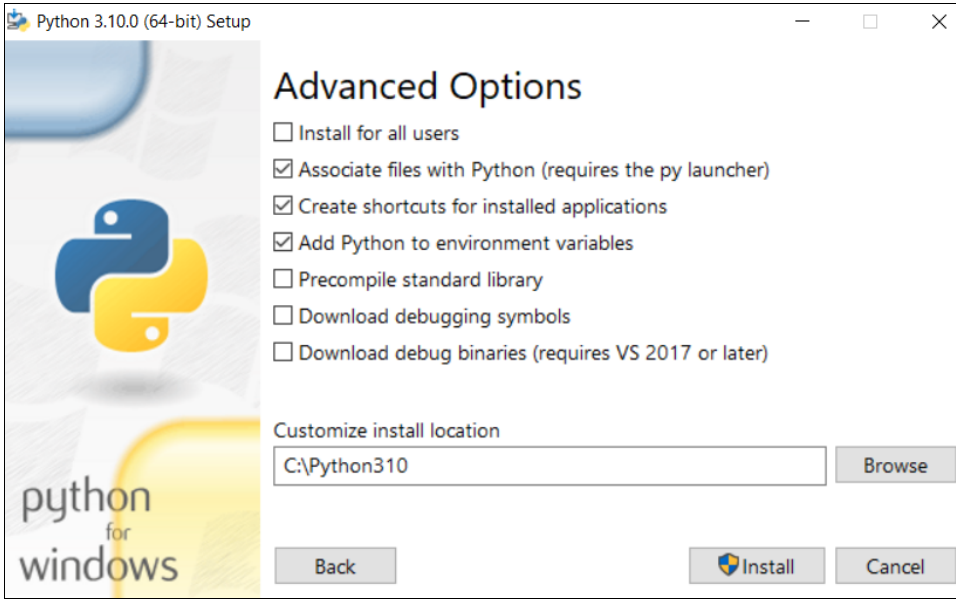
- Easy and powerful language.
- Supports different programming paradigms like *Structured programming* and *Object-oriented programming*.
- Is an interpreted language.
- Ideal for scripting and rapid application development.
- Supports high-level data structures like List, Set, Dictionary and Tuple.
- Python has a design philosophy that emphasizes code readability, and a syntax that allows programmers to express concepts in fewer lines of code.
- Created by ***Guido van Rossum*** and first released in 1991.
- Python features a dynamic type system and automatic memory management.
- Python 2.0 was released on 16th October 2000.
- Python 3.0 (initially called Python 3000 or py3k) was released on 3rd December 2008.
- Python 3.10 was released on October, 2021.

INSTALLATION OF PYTHON

1. Go to python.org (https://www.python.org/downloads).
2. Click on Downloads menu and select your platform.
3. It will take you to related downloads page. For example, for Windows it takes you to https://www.python.org/downloads/windows/.
4. Select Windows x86-64 executable installer and download the installer (python-3.10.0-amd64.exe).
5. Run installer and opt for *Custom installation*.
6. Change directory into which installer installs Python to something like c:\python.
7. Also make sure you select *Add Python 3.10 to PATH* option in installation window.
8. Installer installs all required files into selected folder. Installer automatically sets python installation folder in system path.


Python 3.10.0 (64-bit) Setup
Install Python 3.10.0 (64-bit)
Select Install Now to install Python with default settings, or choose
Customize to enable or disable features.
Install Now
C:\Users\Srikanth Pragada\AppData\Local\Programs\Python\Python310
Includes IDLE, pip and documentation
Creates shortcuts and file associations
Customize installation
Choose location and features
python
for
windows
Install launcher for all users (recommended)
Add Python 3.10 to PATH
Cancel


Python 3.10.0 (64-bit) Setup
Optional Features
Documentation
Installs the Python documentation file.
pip
Installs pip, which can download and install other Python packages.
tcl/tk and IDLE
Installs tkinter and the IDLE development environment.
Python test suite
Installs the standard library test suite.
py launcher
for all users (requires elevation)
Upgrades the global 'py' launcher from the previous version.
python
for
windows
Back
Next
Cancel


Python 3.10.0 (64-bit) Setup
Advanced Options
Install for all users
Associate files with Python (requires the py launcher)
Create shortcuts for installed applications
Add Python to environment variables
Precompile standard library
Download debugging symbols
Download debug binaries (requires VS 2017 or later)
Customize install location
C:\Python310
Browse
python
for
windows
Back
Install
Cancel

USING PYTHON INTERPRETER - REPL

- Go to Command Prompt.
- Make sure system PATH is set to folder where Python was installed. If that is not the case then you need to be in the folder into which you installed Python (for example, c:\python).
- Run **python.exe** to start interpreter. It is also known as *Read Evaluate Print Loop* (REPL).

```
c:\python>python
Python 3.10.0 (tags/v3.10.0:b494f59, Oct  4 2021, 19:00:18)
[MSC v.1929 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license" for more
information.
>>>
```

- Use CTRL-Z or exit() to end interpreter and come back to command prompt.
- The interpreter's line-editing features include interactive editing, history substitution and code completion on systems that support reading line.

INTERACTIVE MODE

- When commands are read from keyboard, the interpreter is said to be in *interactive mode.*
- It prompts for the next command with the *primary prompt*, usually three greater-than signs (**>>>**); for continuation lines it prompts with the *secondary prompt*, by default three dots (**...**).
- In the interactive interpreter, the output string is enclosed in quotes and special characters are escaped with backslashes.
- The *print*() function produces a more readable output, by omitting the enclosing quotes and by printing escape and special characters.
- Two or more *string literals* (i.e. the ones enclosed between quotes) next to each other are automatically concatenated.

VARIABLES

- Python is a dynamic language where variable is created by directly assigning value to it.
- Based on the value assigned to a variable, its datatype is determined.
- Built-in function **type** () can be used to find out the type of a variable.

```
>>> a = 10
>>> type(a)
<class 'int'>
>>> b = "Python"
>>> type(b)
<class 'str'>
>>>
```

NOTE: We can find out data type of any variable using type () built-in function.

RULES FOR IDENTIFIER

While creating names for variables, function and other identifiers, we need to follow the rules given below:

- Can contain 0-9, a-z, A-Z and underscore
- Cannot start with a digit
- Length is unlimited
- Case is significant

OPERATORS

The following are different types of operators available in Python.

Assignment operator (=)

- Assignment operator is used to assign a value to a variable.
- It creates variables if not already present otherwise it changes its current value.
- It is possible to assign multiple values to multiple variables in single assignment statement.

```
>>> # create variables
>>> a = 10
>>> b = 20.50
>>>
>>> # Assign multiple values to multiple variables
>>> a, b = 0, 1
>>>
>>> # swap two variables
>>> a, b = b, a
>>>
>>> # Assign 10 to a, b and c
>>> a = b = c = 10
```

Arithmetic Operators

The following are available arithmetic operators:

Operator	Meaning
+	Addition
-	Subtraction
*	Multiplication
**	Exponentiation
/	Division
//	Integer Division
%	Modulus

```
>>> a, b = 10, 4
>>> a / b, a // b
(2.5, 2)
>>> a ** b
10000
>>> a % 4
2
```

Relational Operators

The following relational operators are available to compare values:

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

```
>>> a = 10
>>> b = 20
>>> a < b
True
>>> a == b
False
>>> s1 = "Abc"
>>> s2 = "ABC"
>>> s1 > s2
True
>>> s1 == s2
False
```

Logical Operators

The following are logical operators used to combine conditions:

Operator	Meaning
and	Anding
or	Oring
not	Negates condition

```
>>> a, b, c = 10, 20, 30
>>> a < b and a < c
True
>>> a < b < c    # same as a < b and b < c
True
>>> a != b or b != c
True
```

NOTE: The Boolean operators **and** & **or** are known as *short-circuit operators* - their arguments are evaluated from left to right, and evaluation stops as soon as the outcome is determined.

BUILT-IN DATA TYPES

The following are built-in data types in Python.

Data Type	Meaning
None	There is a single object with this value. This object is accessed through the built-in name ***None***. It is used to signify the absence of a value in many situations, e.g., it is returned from functions that don't explicitly return anything. Its truth value is false.
NotImplemented	There is a single object with this value. This object is accessed through the built-in name ***NotImplemented***. Numeric methods and rich comparison methods should return this value if they do not implement the operation for the operands provided. Its truth value is true.
Integers (int)	These represent numbers in an unlimited range, subject to available (virtual) memory only.
Booleans (bool)	These represent the truth values *False* and *True*. The two objects representing the values False and True are the only Boolean objects. The Boolean type is a subtype of the integer type, and Boolean values behave like the values 0 (False) and 1 (True), respectively, in almost all contexts, the exception being that when converted to a string, the strings "False" or "True" are returned, respectively.
Real (float)	These represent machine-level double precision floating point numbers. You are at the mercy of the underlying machine architecture for the accepted range and handling of overflow. Python does not support single-precision floating point numbers.

Complex (complex)	These represent complex numbers as a pair of machine-level double precision floating point numbers.
String (str)	A string is a sequence of values that represent Unicode codes.
Tuple (tuple)	The items of a tuple are arbitrary Python objects. Tuples of two or more items are formed by comma-separated lists of expressions.
Bytes (bytes)	A bytes object is an immutable array. The items are 8-bit bytes, represented by integers in the range 0 <= x < 256.
List (list)	The items of a list are arbitrary Python objects. Lists are formed by placing a comma-separated list of expressions in square brackets.
Byte Array (bytearray)	A bytearray object is a mutable array. They are created by the built-in bytearray() constructor.
Set (set)	These represent a mutable set of unique values created by set() constructor.
Frozen set (frozenset)	These represent an immutable set created by built-in frozenset() constructor.
Dictionary (dict)	These represent finite sets of objects indexed by nearly arbitrary values.

```
>>> v1 = 10
>>> v2 = 10.50
>>> v3 = "Python"
>>> v4 = True
>>> type(v1)
<class 'int'>
>>> type(v2)
<class 'float'>
>>> type(v3)
<class 'str'>
>>> type(v4)
<class 'bool'>
```

KEYWORDS

The following are important keywords in Python.

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

BUILT-IN FUNCTIONS

The following are built-in functions in Python.

Function	Meaning
abs(x)	Returns the absolute value of a number.
all(iterable)	Returns True if all elements of the iterable are true (or if the iterable is empty).
any(iterable)	Returns True if any element of the iterable is true. If the iterable is empty, returns False.
bin(x)	Converts an integer number to a binary string prefixed with “0b”.
chr(i)	Returns the string representing a character whose Unicode code point is the integer i.
dir([object])	Without arguments, returns the list of names in the current local scope. With an argument, attempts to return a list of valid attributes for that object.
filter(function, iterable)	Constructs an iterator from those elements of iterable for which function returns true.
format(value[, format_spec])	Converts a value to a “formatted” representation, as controlled by format_spec.
getattr(object, name[, default])	Returns the value of the named attribute of object.
hex(x)	Converts an integer number to a lowercase hexadecimal string prefixed with “0x”.
id(object)	Returns the “identity” of an object. This is an integer which is guaranteed to be unique and constant for this object during its lifetime.
len(s)	Returns the length (the number of items) of an object.

max(iterable) max(arg1, arg2,*args)	Returns the largest item in an iterable or the largest of two or more arguments.
input([prompt])	If the prompt argument is present, it is written to standard output without a trailing newline. The function then reads a line from input, converts it to a string (stripping a trailing newline), and returns that. When EOF is read, EOFError is raised.
min(iterable) min(arg1, arg2, *args)	Returns the smallest item in an iterable or the smallest of two or more arguments.
oct(x)	Converts an integer number to an octal string prefixed with “0o”.
ord(c)	Given a string representing one Unicode character, returns an integer representing the Unicode code point of that character.
print(*objects, sep=' ', end='\n', file=sys.stdout, flush=False)	Prints objects to the text stream file, separated by sep and followed by end. sep, end, file and flush, if present, must be given as keyword arguments.
reversed(seq)	Returns a reverse iterator.
round(number[, ndigits])	Returns number rounded to ndigits precision after the decimal point.
sorted(iterable, *, key=None, reverse=False)	Returns a new sorted list from the items in iterable.
sum(iterable[, start])	Sums start and the items of an iterable from left to right and returns the total. start defaults to 0.
type(object)	With one argument, returns the type of an object.

```
>>> abs(-10)
10
>>> bin(10)
'0b1010'
>>> chr(65)         # char for ascii code
'A'
>>> a = 10
>>> id(a)           # Address of the object
140728047310784
>>> max(10, 20, 30)
30
>>> ord('a')        # Ordinal value (ascii code)
97
>>> round(10.566)
11
>>> type(True)
<class 'bool'>
>>>
```

FUNCTION INPUT()

- Built-in function **input**() is used to take input from user.
- It always returns a string, so we need to convert it to required type using other built-in functions like **int**().

```
>>> num = int(input("Enter a number :"))
Enter a number :25
>>> print("Square of", num, "is", num * num)
Square of 25 is 625
>>>
```

Using print() function

- Built-in function print() is used to print one or more values.
- It is possible to specify separator and end characters.

```
>>> print(1, 2, 3)
1 2 3
>>> print(1, 2, 3, sep = '-')
1-2-3
>>> print(1, 2, 3, end = '\n\n')
1 2 3

>>> print("Python", 3.10, sep= " - ", end = "\n\n")
Python - 3.10

>>>
```

Formatted output

- It is possible to print formatted output using % with conversion characters like %d and %s.
- Method **format**() of string can be used to format output. String on which this method is called can contain literal text or replacement fields delimited by braces {}.
- Each replacement field contains either the numeric index of a positional argument, or the name of a keyword argument. Returns a copy of the string where each replacement field is replaced with the string value of the corresponding argument.

```
str % (values)
```

```
str.format(*args, **kwargs)
```

```
>>> name = "Srikanth"
>>> mobile = "9059057000"
>>> print("%s %s" % (name, mobile))
Srikanth 9059057000
>>> print("Name = {0}".format(name))
Name = Srikanth
>>> print("Name = {}, Mobile = {}".format(name, mobile))
Name = Srikanth, Mobile = 9059057000
>>> print("{name} {ver}".format(name="Python", ver=3.10))
Python 3.10
>>>
```

The f-string

- F-string is a string prefixed with f and inserts values of variables when variables are enclosed in {} inside the string.
- New feature of Python **3.6**.

```
>>> name = "Srikanth"
>>> lang = "Python"
>>> f"{name} is a {lang} trainer"
'Srikanth is a Python trainer'
```

It is possible to format values as follows:

```
>>> mode = "Cash"
>>> amount = 251.567
>>> print(f"{mode:10} {amount:8.2f} {amount*0.20:5.2f}")
Cash         251.57 50.31
>>> f"{10000.00:8.2}"
'   1e+04'
```

NOTE: In format specifier, 8.2 means; total columns including decimal point is 8 and 2 digits after decimal point. Char f means fixed format, otherwise it uses E format.

THE IF STATEMENT

- It is used for conditional execution.
- It selects exactly one of the suites by evaluating the expressions one by one until one is found to be true.
- There can be zero or more elif parts, and the else part is optional.

```
if boolean_expression:
      statements
[elif boolean_expression:
      statements] ...
[else:
      statements]
```

```
if a > b:
   print(a)
else:
   print(b)
```

```
if a > 0:
   print("Positive")
elif a < 0:
   print("Negative")
else:
   print("Zero")
```

CONDITIONAL EXPRESSION

- It returns either true value or false value depending on the condition.
- If *condition* is true then it returns *true_value* otherwise it returns *false_value*.

```
true_value if condition else false_value
```

```
>>> a = 10
>>> b = 20
>>> a if a > b else b
20
```

THE WHILE LOOP

The **while** statement is used for repeated execution as long as the boolean expression is true.

```
while boolean_expression:
     statements
[else:
     statements]
```

NOTE: The **else** part of while is executed only when loop is terminated normally, i.e. without **break** statement.

```
01 # Program to print numbers from 1 to 10
02 i = 1
03 while i <= 10:
04     print(i)
05     i += 1
06
```

```
01 # Program to print digits in a number in reverse order
02 num = int(input("Enter a number :"))
03 while num > 0:
04     digit = num % 10    # Take rightmost digit
05     print(digit)
06     num = num // 10     # Remove rightmost digit
07
```

THE RANGE() FUNCTION

- We can use range() function to generate numbers between the given start and end (exclusive).
- If you do need to iterate over a sequence of numbers, the built-in function range() comes in handy. It generates arithmetic progressions.
- In many ways the object returned by range() behaves as if it is a list, but in fact it isn't. It is an object which returns the successive items of the desired sequence when you iterate over it, but it doesn't really make the list, thus saving space.

```
range([start,] end [, step])
```

If start is not given then 0 is taken, if step is not given then 1 is taken.

```
range (10)                  # will produce 0 to 9
range (1, 10, 2)            # will produce 1,3,5,7,9
```

THE PASS STATEMENT

The pass statement does nothing. It can be used when a statement is required syntactically but the program requires no action.

THE FOR STATEMENT

Executes given statements until list is exhausted.

```
for target in expression_list:
     statements
[else:
     statements]
```

The expression_list is evaluated once; it should yield an iterable object. An iterator is created for the result of the expression_list. The set of statements is then executed once for each item provided by the iterator, in the order returned by the iterator.

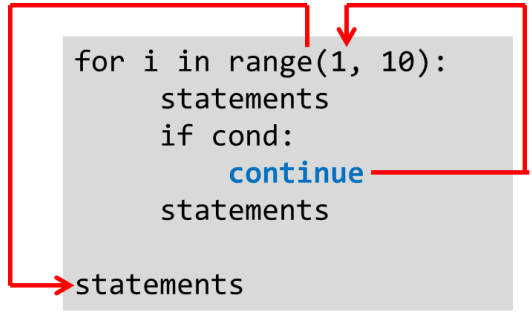
When the items are exhausted (which is immediately when the sequence is empty or an iterator raises a *StopIteration* exception), the statements in the **else** clause, if present, are executed, and the loop terminates.

```
01 # Print numbers from 1 to 10 across
02 for n in range(1, 11):
03     print(n, end = ' ')
```

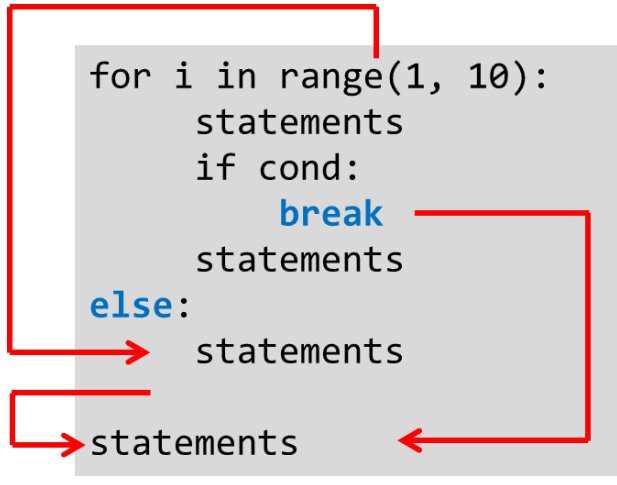
```
01 # program to take a number and display its factorial
02 num = int(input("Enter a number :"))
03 fact = 1
04 for i in range(2, num + 1):
05     fact *= i
06
07 print(f"Factorial of {num} is {fact}")
```

BREAK, CONTINUE AND ELSE

- The **break** statement, like in C, breaks out of the innermost enclosing **for** or **while** loop.
- Loop statements may have an **else** clause; it is executed when the loop terminates through exhaustion of the list (with for) or when the condition becomes false (with while), but not when the loop is terminated by a **break** statement.
- The **continue** statement, also borrowed from C, continues with the next iteration of the loop.



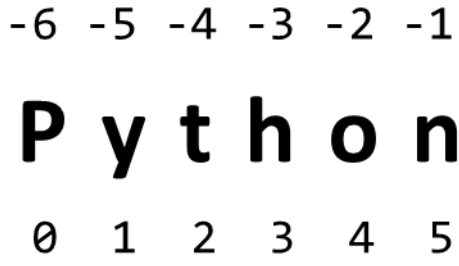

```
01 # Check whether the number is prime
02 num = int(input("Enter a number :"))
03 for i in range(2, num//2 + 1):
04     if num % i == 0:
05         print('Not a prime number')
06         break
07 else:
08     print('Prime number')
```




```
01 # program to print prime numbers from 1 to 100
02 for num in range(1, 101):
03     for i in range(2, num//2 + 1):
04         if num % i == 0:
05             break
06     else:
07         print(num)
```

STRINGS

- Strings can be enclosed either in single quotes or double quotes.
- Python strings cannot be changed — they are immutable.
- Built-in **len**() function returns length of the string.
- Strings can be *indexed* (subscripted), with the first character having index 0.
- There is no separate character type; a character is simply a string of size one.
- Indices may also be negative numbers, to start counting from the right.
- In addition to indexing, *slicing* is also supported. While indexing is used to obtain individual characters, *slicing* allows you to obtain substring.
- Slice indices have useful defaults; an omitted first index defaults to zero, an omitted second index defaults to the size of the string being sliced.



The following are examples for indexing and slicing:

```
>>>name="Python"
>>>name[0]
'P'
>>>name[-1]         # Last char
'n'
>>>name[-3:]        # Take chars from 3rd char from end
'hon'
>>>name[0:2]        # Take from 0 to 1
'Py'
>>>name[4:]         # Take chars from 4th position
'on'
>>> name[::-1]     # Take chars in reverse
'nohtyP'
>>> name[-2:-5:-1] # Take char from -2 to -5 in reverse
'oht'
```

Method	Description
capitalize()	Returns a copy of the string with its first character capitalized and the rest lowercased.
count(sub[, start[, end]])	Returns the number of non-overlapping occurrences of substring **sub** in the range [start, end].
endswith(suffix[, start[, end]])	Returns True if the string ends with the specified **suffix**, otherwise returns False.
find(sub[, start[, end]])	Returns the lowest index in the string where substring **sub** is found within the slice s[start:end]. Returns -1 if sub is not found.
format(*args, **kwargs)	Performs a string formatting operation.
index(sub[, start[, end]])	Like find(), but raises ValueError when the substring is not found.
isalnum()	Returns true if all characters in the string are alphanumeric and there is at least one character, false otherwise.
isalpha()	Returns true if all characters in the string are alphabetic and there is at least one character, false otherwise.
isdecimal()	Returns true if all characters in the string are decimal characters and there is at least one character, false otherwise.
isdigit()	Returns true if all characters in the string are digits and there is at least one character, false otherwise.
islower()	Returns true if all characters in the string are lowercase and there is at least one cased character, false otherwise.

isupper()	Returns true if all characters in the string are uppercase and there is at least one cased character, false otherwise.
join(iterable)	Returns a string which is the concatenation of the strings in iterable.
lower()	Returns a copy of the string with all the characters converted to lowercase.
partition(sep)	Splits the string at the first occurrence of **sep**, and returns a 3-tuple containing the part before the separator, the separator itself, and the part after the separator.
replace (old, new[, count])	Returns a copy of the string with all occurrences of substring **old** replaced by **new**. If the optional argument count is given, only the first **count** occurrences are replaced.
split(sep=None, maxsplit=-1)	Returns a list of the words in the string, using **sep** as the delimiter string. If **maxsplit** is given, at most maxsplit splits are done (thus, the list will have at most maxsplit+1 elements). If maxsplit is not specified or -1, then there is no limit on the number of splits (all possible splits are made).
startswith (prefix[, start[, end]])	Returns True if string starts with the **prefix**, otherwise returns False.
strip([chars])	Returns a copy of the string with the leading and trailing characters removed.
upper()	Returns a copy of the string with all the characters converted to uppercase.
zfill(width)	Returns a copy of the string left filled with ASCII '0' digits to make a string of length width.

```
>>> name = "Srikanth Technologies"
>>> print(name.upper())
SRIKANTH TECHNOLOGIES
>>> print(name.count("i"))
2
>>> name.find('Tech')
9
>>> name.replace(' ', '-')
'Srikanth-Technologies'
>>> name.startswith('S')
True
>>> for c in name:
...     print(c,end =  ' ')
...
S r i k a n t h   T e c h n o l o g i e s
>>> for c in name[:-4:-1]:  # Last 3 chars in reverse
...     print(c)
...
s
e
i
```

THE LIST DATA STRUCTURE

- Represents a list of values.
- Supports duplicates and maintains order of the elements.
- List can be modified (Mutable).
- Elements can be accessed using index.

Method	Meaning
append(x)	Adds an item to the end of the list. Equivalent to a[len(a):] = [x].
extend(iterable)	Extends the list by appending all the items from the iterable. Equivalent to a[len(a):] = iterable.
insert(i, x)	Inserts an item at a given position. The first argument is the index of the element before which to insert, so a.insert(0, x) inserts at the front of the list, and a.insert(len(a), x) is equivalent to a.append(x).
remove(x)	Removes the first item from the list whose value is **x**. It is an error if there is no such item.
pop([i])	Removes the item at the given position in the list, and returns it. If no index is specified, a.pop() removes and returns the last item in the list. The square brackets around the i in the method signature denote that the parameter is optional, not that you should type square brackets at that position.
clear()	Removes all items from the list. Equivalent to del a[:].
index(x[, start [, end]])	Returns zero-based index in the list of the first item whose value is **x**. Raises a ValueError if there is no such item. Optional arguments **start** and **end** are interpreted as in the slice notation and are used to limit the search to a particular subsequence of the list. The returned index is computed relative to the beginning of the full sequence rather than the start argument.

count(x)	Returns the number of times **x** appears in the list.
sort(key=None, reverse=False)	Sorts the items of the list in place (the arguments can be used for sort customization).
reverse()	Reverses the elements of the list in place.
copy()	Returns a shallow copy of the list. Equivalent to a[:].

```
>>> fruits = ['orange', 'apple', 'grape',
...           'banana', 'mango', 'apple']
>>> print(fruits.count('apple'))   # count of apple
2
>>> print(fruits.index('banana'))  # index of banana
3
>>> print(fruits.index('apple', 3)) # search from 3rd index
5
>>> fruits.reverse()
>>> print(fruits)
['apple', 'mango', 'banana', 'grape', 'apple', 'orange']
>>> fruits.append('kiwi')
>>> fruits
['apple', 'mango', 'banana', 'grape', 'apple', 'orange',
'kiwi']
>>> fruits.sort()
>>> fruits
['apple', 'apple', 'banana', 'grape', 'kiwi', 'mango',
'orange']
>>> print(fruits.pop())
orange
```

List Comprehension

- List comprehensions provide a concise way to create lists.
- A list comprehension consists of brackets containing an expression followed by a **for** clause, then zero or more **for** or **if** clauses.

```
[value for v in iterable [if condition]]
```

```
>>> [n * n for n in range(1,10)]
[1, 4, 9, 16, 25, 36, 49, 64, 81]
```

```
>>> nums = [5, 3, 4, 7, 8]
>>> [n * n for n in nums if n % 2 == 0]
[16, 64]
```

THE DEL STATEMENT

Removes one or more items from the list.

```
del item
```

```
>>> a = [10, 20, 30, 40, 50]
>>> del a[0]
>>> a
[20, 30, 40, 50]
>>> del a[1:3]
>>> a
[20, 50]
>>> del a[:]
>>> a
[]
>>> del a
>>> a      # Throws error
```

OPERATIONS RELATED TO SEQUENCE TYPES

Sequences like str, list and tuple support the following common operations:

Operation	Result
x in s	True if an item of s is equal to x, else False
x not in s	False if an item of s is equal to x, else True
s + t	the concatenation of s and t
s * n or n * s	equivalent to adding s to itself n times
s[i]	i^{th} item of s, origin 0
s[i:j]	slice of s from i to j
s[i:j:k]	slice of s from i to j with step k
len(s)	length of s
min(s)	smallest item of s
max(s)	largest item of s

THE TUPLE DATA STRUCTURE

- A tuple consists of a number of values separated by commas.
- Tuples are immutable, and usually contain a heterogeneous sequence of elements that are accessed via unpacking or indexing.
- It is not possible to assign to the individual items of a tuple, however it is possible to create tuples which contain mutable objects, such as lists.
- Membership operator **in** and **not in** can be used to check whether an object is member of tuple.
- A function can return multiple values using a tuple.
- Empty tuples are constructed by an empty pair of parentheses; a tuple with one item is constructed by following a value with a comma (it is not sufficient to enclose a single value in parentheses).

```
>>> t1 = ()                                  # Empty tuple
>>> t2 = (10, )                              # single value tuple
>>> t3 = ("Python", "Language", 1991) # Tuple with 3 values
>>> t4 = ((1, 2), (10, 20))                  # Nested tuple
>>> t1
()
>>> t2
(10,)
>>> t3
('Python', 'Language', 1991)
>>> t3[0]                                    # prints first element
Python
>>> n, t, y = t3                             # Unpacking of tuple
>>> print(n, t, y)
Python Language 1991
>>>
```

FUNCTION ZIP() AND ENUMERATE()

- When looping through a sequence, the position index and corresponding value can be retrieved at the same time using the **enumerate()** function.
- To loop over two or more sequences at the same time, the entries can be paired with the **zip()** function.

```
enumerate(iterable, start=0)
```

```
01 l1 = [10, 20, 30]
02 for i, n in enumerate(l1, start = 1):
03     print(i, n)
```

Output

```
1 10
2 20
3 30
```

```
zip(*iterables, strict=False)
```

If **strict** is true then all iterables must be of same size, otherwise it throws error. By default, **strict** is false and it considers the smallest length of the sequences.

```
01 l1 = [10, 20, 30]
02 l2 = ['abc', 'xyz, 'pqr']
03 for fv, sv in zip(l1, l2):
04     print(fv, sv)
```

Output

```
10 abc
20 xyz
30 pqr
```

SORTED() AND REVERSED()

Function **sorted**() is used to return a new sorted list of values for the given iterable.

```
sorted(iterable, *, key=None, reverse=False)
```

NOTE: Argument *key* is used to specify a function whose return value is considered for comparison of values. More on passing function as parameter later.

```
>>> l = [4, 3, 5, 1, 2]
>>> sorted(l)
[1, 2, 3, 4, 5]
```

Function **reversed**() is used to provide the iterable in reverse order.

```
reversed(iterable)
```

```
>>> l = [4, 3, 5, 1, 2]
>>> for n in reversed(l):
...   print(n)
...
2
1
5
3
4
```

THE SET DATA STRUCTURE

- A set is an *unordered* collection with *no duplicate* elements.
- Set objects also support mathematical operations like union, intersection, difference, and symmetric difference.
- Curly braces or the **set**() function can be used to create sets.
- To create an empty set you have to use set(), not {}; the latter creates an empty dictionary.
- Items cannot be accessed using index, i.e., not subscriptable.

Method	Meaning
isdisjoint(other)	Returns True if the set has no elements in common with other.
issubset(other) or set <= other	Tests whether every element in the set is in other.
set < other	Tests whether the set is a proper subset of other, that is, set <= other and set != other.
issuperset(other) or set >= other	Tests whether every element in other is in the set.
set > other	Tests whether the set is a proper superset of other, that is, set >= other and set != other.
union(*others) or set | other | ...	Returns a new set with elements from the set and all others.
intersection(*others) or set & other & ...	Returns a new set with elements common to the set and all others.
difference(*others) or set - other - ...	Returns a new set with elements in the set that are not in the others.
symmetric_difference (other) or set ^ other	Returns a new set with elements in either the set or other but not both.
update(*others) or set |= other	Updates the set, adding elements from all others.

add(elem)	Adds element **elem** to the set.
remove(elem)	Removes element **elem** from the set. Raises KeyError if **elem** is not contained in the set.
discard(elem)	Removes element **elem** from the set if it is present.
pop()	Removes and returns an arbitrary element from the set. Raises KeyError if the set is empty.
clear()	Removes all elements from the set.

```
>>> langs = {"Python", "Java", "C#", "C", "Pascal"}
>>> old_langs = {"C", "Pascal", "COBOL"}
>>> print("Java" in langs)
True
>>> letters = set("Python")   # Convert str to set
>>> print(letters)
{'t', 'o', 'h', 'y', 'P', 'n'}
>>> print("Union:", langs | old_langs)
Union: {'C#', 'COBOL', 'Python', 'C', 'Java', 'Pascal'}
>>> print("Minus:", langs - old_langs)
Minus: {'C#', 'Java', 'Python'}
>>> print("Intersection:", langs & old_langs)
Intersection: {'C', 'Pascal'}
>>> print("Exclusive Or:", langs ^ old_langs)
Exclusive Or: {'C#', 'COBOL', 'Python', 'Java'}
```

Set Comprehension

It is used to create a set from the given iterable, optionally based on condition.

```
{value for v in iterable [if condition]}
```

```
>>> st = "abc123acdef456"
>>> {c for c in st if c.isalpha() }
{'c', 'd', 'b', 'f', 'e', 'a'}
```

LIST VS. SET VS. TUPLE

The following table compares features of different data structures.

Feature	List	Set	Tuple
Supports Duplicates	Yes	No	Yes
Mutable	Yes	Yes	No
Indexable	Yes	No	Yes
Insertion Order Maintained	Yes	No	Yes
Iterable	Yes	Yes	Yes

THE DICTIONARY DATA STRUCTURE

- Dictionaries are indexed by keys, which can be any immutable type; strings and numbers can always be keys.
- It is best to think of a dictionary as an unordered set of *key: value* pairs, with the requirement that the keys are unique (within one dictionary).
- Placing a comma-separated list of *key:value* pairs within the braces adds initial *key:value* pairs to the dictionary.
- It is an error to extract a value using a non-existent key.
- The **dict**() constructor builds dictionaries directly from sequences of key-value pairs.
- When looping through dictionaries, the key and corresponding value can be retrieved at the same time using the **items**() method.

```
>>> dict1 = {"k1": "Value1", "k2": "Value2"}
>>> print(dict1)
{'k1': 'Value1', 'k2': 'Value2'}
>>> print(dict1.keys())
dict_keys(['k1', 'k2'])
>>> for k in dict1.keys():
...     print( k, ':', dict1[k])
...
k1 : Value1
k2 : Value2
>>> print('k3' in dict1)
False
>>> # Unpack tuple returned by items()
>>> for k, v in dict1.items():
...     print( k, ':', v)
...
k1 : Value1
k2 : Value2
```

Method	Meaning
d[key]	Returns the item of d with key *key*. Raises a KeyError if key is not in the map.
d[key] = value	Sets d[key] to value.
del d[key]	Removes d[key] from d. Raises a KeyError if key is not in the map.
key in d	Returns True if d has a key *key*, else False.
key not in d	Equivalent to not key in d.
iter(d)	Returns an iterator over the keys of the dictionary. This is a shortcut for iter(d.keys()).
clear()	Removes all items from the dictionary.
copy()	Returns a shallow copy of the dictionary.
get(key[, default])	Returns the value for key if key is in the dictionary, else default. If default is not given, it defaults to None, so that this method never raises a KeyError.
items()	Returns a new view of the dictionary's items ((key, value) pairs).
keys()	Returns a new view of the dictionary's keys.
pop(key[, default])	If key is in the dictionary, removes it and returns its value, else returns default. If default is not given and key is not in the dictionary, a KeyError is raised.
setdefault (key[, default])	If key is in the dictionary, returns its value. If not, inserts key with a value of default and returns default. The *default* defaults to None.
update([other])	Updates the dictionary with the key/value pairs from other, overwriting existing keys. Returns None.
values()	Returns a new view of the dictionary's values.

Dictionary Comprehension

It is possible to create a dictionary by taking values from an iterable.

The following is general syntax for dictionary comprehension.

```
{key:value for v in iterable}
```

```
>>> # Create a dictionary from a list
>>> nums = [10, 4, 55, 23, 9]
>>> squares = {n:n*n for n in nums}
>>> print(squares)
{10: 100, 4: 16, 55: 3025, 23: 529, 9: 81}
>>> # Create a dictionary from a string
>>> name = "Srikanth"
>>> codes = {ch:ord(ch) for ch in name}
>>> print(codes)
{'S': 83, 'r': 114, 'i': 105, 'k': 107, 'a': 97, 'n': 110,
't': 116, 'h': 104}
>>>
```

STRUCTURAL PATTERN MATCHING

- Structural pattern matching has been added in the form of a **match statement** and **case statements** of patterns with associated actions.
- A match statement takes an **expression** and compares its value to successive **patterns** given as one or more case blocks.
- Patterns consist of sequences, mappings, primitive data types as well as class instances.
- This feature was introduced in Python **3.10**.

```
match expression:
    case <pattern_1>:
        <action_1>
    case <pattern_2>:
        <action_2>
    case <pattern_3>:
        <action_3>
    case _:
        <action_wildcard>
```

```
01 match code:
02     case 1:
03         discount = 10
04     case 2:
05         discount = 20
06     case 3:
07         discount = 25
08     case _:
09         discount = 5
```

It is possible to capture values into variables while pattern matching.

```
01 point = (0, 10)  # row, col
02 match point:
03     case (0, 0):
04         print("First row, First Col")
05     case (0, c):
06         print(f"First Row, {c} col")
07     case (r, 0):
08         print("First Col, {r} row")
09     case (r, c):
10         print(f"{r} row, {c} col")
```

It is also possible to use multiple literals in case statement as follows:

```
01 match month:
02     case 2:
03         nodays = 28
04     case 4 | 6 | 9 | 11:
05         nodays = 30
06     case _:
07         nodays = 31
```

When expression is a dictionary, it is possible to match keys and capture values into variables.

```
01 # Assume d may have any of the two structures
02 d = {'name': 'Jack', 'email': 'jack@gmail.com'}
03 d = {'firstname': 'Scott'}
04
05 match d:
06     case {'name': user}:
07         pass
08     case {'firstname': user}:
09         pass
10     case _:
11         user = 'Unknown'
12
13 print(user)
```

FUNCTIONS

- The keyword **def** introduces a function *definition*. It must be followed by the function name and the parenthesized list of formal parameters. The statements that form the body of the function start at the next line, and must be indented.
- The first statement of the function body can optionally be a string literal; this string literal is the function's documentation string, or *docstring.*
- Variable references first look in the local symbol table, then in the local symbol tables of enclosing functions, then in the global symbol table, and finally in the table of built-in names.
- Arguments are passed using *call by value* (where the *value* is always an object *reference*, not the value of the object).
- In fact, even functions without a return statement do return a value – None.

```
[decorators] "def" funcname "(" [parameter_list] ")"
["->" expression] ":" suite
```

```
01 # Function to return sum of two numbers
02 def add(a,b):
03     return a + b
```

```
01 # Function to return factorial of the given number
02 def factorial(num):
03    fact=1
04    for i in range(1,num + 1):
05       fact *= i
06
07    return fact
```

DEFAULT ARGUMENT VALUES

- It is possible to specify default value for one or more parameters.
- The default values are evaluated at the point of function definition in the *defining* scope and not at the time of running it.

```
01 def print_line(len=10, ch='-'):
02       for i in range(len):
03           print(ch, end='')
04       else:
05           print()  # come to next line at the end
```

```
01 print_line(30, '*')  # Passing values by position
02 print_line(20)       # Draw a line using hyphens(-)
03 print_line()         # Draws line using default values
04 print_line(ch='*')   # Passing value by keyword
05
06 # Passing values using keywords
07 print_line(ch='*', len=15)
```

NOTE: The default value is evaluated only once. This makes a difference when the default is a mutable object such as a list, dictionary, or instances of most classes.

VARYING ARGUMENTS

- A function can take any number of arguments by defining formal parameter with prefix *****.
- When a function has a varying formal parameter then it can take any number of actual parameters.
- A function can mix varying parameters with normal parameters.
- However, normal parameters can be passed values by name or they should appear before varying argument.

```
01 def print_message(*names, message = "Hi"):
02     for n in names:
03         print(message, n)
04
05 #Call function
06 print_message("Bill", "Larry", "Tim")
07 print_message("Steve", "Jeff", message="Good Morning")
```

NOTE: Argument names is of type tuple.

KEYWORD ARGUMENTS

- A function can be defined to take arbitrary sequence of keyword arguments by defining a parameter with **** as prefix.
- Function treats this parameter as a **dictionary** and provides all keyword arguments as keys in dictionary.
- A function can be called with keyword arguments using **kwarg=value**, where kwarg is keyword and value is value.
- In a function call, keyword arguments must follow positional arguments, if any are present.

```
01 def show(**kwargs):
02     for k, v in kwargs.items():
03         print(k, v)
04
05 def showall(*args, **kwargs):
06     print(args)
07     for k, v in kwargs.items():
08         print(k, v)
09
10 show(a=10, b=20, c=20, msg="Hello")
11 showall(10, 20, 30, x=1, y=20)
```

The above program will display:

```
a 10
b 20
c 20
msg Hello
(10, 20, 30)
x 1
y 20
```

KEYWORD-ONLY ARGUMENTS

- It is possible to define parameters as keyword only parameters by giving an * before them.
- All parameters after * must be passed values only by using keywords and not by position.

```
01 # Parameters name & age can be passed only as keyword
02 # arguments and NOT as positional
03
04 def details(*, name, age=30):
05      print(name)
06      print(age)
07
08 # Call details()
09 details(name="Bill", age=60)
10 details(age=50, name="Scott")
11 details(name="Tom")
12 details("Bill")                  # Error
```

When you try to call ***details("Bill")*** with positional argument, Python throws error as follows:

```
TypeError: details() takes 0 positional arguments but 1 was
given
```

POSITIONAL-ONLY ARGUMENTS

- Starting from Python 3.8, it is possible to create a function that takes parameters only by position and not by keywords.
- Give a / (slash) after all parameters that are to be positional only.

```
01 def add(n1, n2, /):
02     return n1 + n2
03
04 print(add(10, 20)) # parameters are passed by position
05 print(add(n1=10, n2=20))  # Can't use keyword args
```

```
30
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
TypeError: add() got some positional-only arguments passed
as keyword arguments: 'n1, n2'
```

PASSING FUNCTION AS A PARAMETER

- It is possible for a function to receive another function as a parameter.
- This is possible because a function is treated as an object in Python, so just like any other object, even a function can be passed as parameter to another function.

```
01 def add(n1, n2):
02     return n1 + n2
03
04 def mul(n1, n2):
05     return n1 * n2
06
07 def math_operation(a, b, operation):
08     return operation(a, b)
09
10 print(math_operation(10, 20, add))
11 print(math_operation(10, 20, mul))
```

USING FILTER, SORTED AND MAP FUNCTIONS

The following examples show how to use a function as a parameter with built-in functions filter, sorted and map.

Filter function

Function **filter** is used to select a set of elements from an iterable for which function returns true. The given function must take a value and return true or false. When function returns true, value is selected, otherwise value is ignored.

```
filter(function, iterable)
```

NOTE: Function passed to filter should take a single value and return bool.

The following example selects all even numbers from the given list of numbers.

```
01 def iseven(n):
02     return n % 2 == 0
03
04 nums = [1, 4, 3, 5, 7, 8, 9, 2]
05
06 # filter calls iseven and selects even numbers
07 for n in filter (iseven, nums):
08     print(n)
```

NOTE: filter() returns an object of type filter, which is iterable, and not list.

Sorted function

Sorts the given iterable and returns a list with sorted values.

```
sorted(iterable, *, key=None, reverse=False)
```

Key parameter represents a function that returns a value.

The following code sorts names by length of the name and not by characters.

Built-in function **sorted()** uses function passed to **key** argument to extract values for elements in the collection and uses them for comparison. If no **key** is passed, it uses elements directly.

```
01 names = ["Php", "Java", "C", "Python", "SQL", "C#"]
02 # Sort names based on length
03 for n in sorted(names, key=len):
04     print(n)
```

```
C
C#
Php
SQL
Java
Python
```

Map function

Returns an iterator that applies the given function to each element in iterable, yielding new values.

```
map (function, iterable, ...)
```

Function given as first parameter must return a value.

The following example shows how to use map() function to return next even number for the given value.

```
01 def next_even(n):
02     return n + 2 if n % 2 == 0 else n + 1
03
04 nums = [10, 11, 15, 20, 25]
05 for n in map(next_even,nums):
06     print(n)
```

LAMBDA EXPRESSION

- Lambda expression refers to an anonymous function.
- Where a function is needed, we can use lambda expression.
- Keyword *lambda* is used to create lambda expressions.

```
lambda parameters: expression
```

Parameters are separated by comma (,) and they represent parameters of the function in question. Expression given after colon (:) represents the required action.

The following example shows how we can use lambda in conjunction with **filter()** function, which returns a list of values that are selected by the given function from the given list.

```
01 nums = [10, 11, 33, 45, 44]
02
03 # with lambda, get all odd numbers
04 for n in filter (lambda v: v % 2 == 1, nums):
05    print(n)
```

The following example sorts all names by stripping all whitespaces and then converting them to lowercase (for case insensitivity) using lambda expression passed to key parameter of sorted () function.

```
01 names = [" PHP", " JAVA", "c", "Python ",
02               " JavaScript", "C#  ", "cobol"]
03 #Sort after stripping spaces and converting to lowercase
04 for n in sorted(names,
05                  key=lambda s: s.strip().lower()):
06 print(n)
```

PASSING ARGUMENTS - PASS BY VALUE AND REFERENCE

- In Python everything is an object.
- An *int* is an object, *string* is an object and *list* is an object.
- Some objects are mutable, some are immutable.
- All objects are passed by reference (address) to a function. That means we pass reference of the object and not the object itself.
- But whether the function can modify the value of the object depends on the mutability of the object.
- So, if you pass a string, it behaves like pass by value as we can't change actual parameter with formal parameter.
- If you pass a list (mutable object) then it behaves like pass by reference as we can use formal parameter to change actual parameter.

```
01 # Effectively pass by reference
02 def add_num(v, num):
03      v.append(num)     # modify list
04
05 nums = []
06 # Call function with a list and value to add to list
07 add_num(nums, 10)
08 add_num(nums, 20)
09 print(nums)
```

```
[10, 20]
```

```
01 # Effectively pass by value
02 def swap(n1, n2):
03     n1, n2 = n2, n1
04     print('Inside swap() :','id(n1)', id(n1),
05                 'id(n2)', id(n2))
06     print('Values : ', n1, n2)
07
08 # call swap with two int variables
09 a = 10
10 b = 20
11 print('Original Ids:', 'id(a)', id(a), 'id(b)', id(b))
12 swap(a, b)
13 print('Values after swap :', a, b)
```

Output:

```
Original Ids  : id(a) 503960768 id(b) 503960928
Inside swap() : id(n1) 503960928 id(n2) 503960768
Values :  20 10
Values after swap :  10 20
```

LOCAL FUNCTIONS

- Functions defined inside another function are called local functions.
- Local functions are local to function in which they are defined.
- They are defined each time the enclosing function is called.
- They are governed by same **LEGB** (Local, Enclosing, Global, Built-in) rule.
- They can access variables that are in enclosing scope.
- Cannot be called from outside outer function using notation *outerfunction. Localfunction.*
- They can contain multiple statements whereas lambdas can have only one statement.
- Local function can be returned from outer function and then can be called from outside.
- Local function can refer to variables in global namespace using **global** keyword and enclosing namespace using **nonlocal** keyword.

```
01 gv = 100  # Global variable
02 def f1():
03     v = 200  # enclosing variable
04
05     # Local function f2
06     def f2():
07         lv = 300  # local variable
08         # Local, Enclosing, Global, Built-in name
09         print(lv, v, gv, True)
10
11
12     f2()  #call local fun from enclosing fun
13
14
15 f1()
```

VARIABLE'S SCOPE

- Variables that are defined outside all functions in a module are called global variables and can be accessed from anywhere in the module.
- Variables created inside a function can be used only inside the function.
- Keyword **global** is used to access a global variable from a function so that Python doesn't create a local variable with the same name when you assign a value to a variable.
- Python looks in the order – local, enclosing, global and built-in (LEGB) variables.

```
01 sum = 0   # Global variable
02 def add_square(value):
03     square = value * value # Square is local variable
04     global sum             # Refer to global variable
05     sum += square          # Add square to global sum
06
07 def outerfun():
08     a = 10
09     def innerfun():
10         nonlocal a
11         a = a + 1          # Increment nonlocal a
12
13     innerfun()
14     print(a)
15
16 add_square(4)
17 add_square(5)
18 print(sum)
19 outerfun()
```

MODULES

- A module is a file containing Python definitions (functions and classes) and statements.
- It can be used in a script (another module) or in an interactive instance of the interpreter.
- A module can be *imported* into other modules or run as a script.
- The file name is the module name with the suffix .py appended. Within a module, the module's name (as a string) is available as the value of the global variable *__name__*.
- A module can contain executable statements as well as function and class definitions. These statements are intended to initialize the module. They are executed only the *first* time the module name is encountered in an import statement.

num_funs.py

```
01 def is_even(n):
02     return  n % 2 == 0
03
04 def is_odd(n):
05     return  n % 2 == 1
06
07 def is_positive(n):
08     return n > 0
```

use_num_funs.py

```
01 import num_funs                  # import module
02
03 print(num_funs.__name__)
04 print(num_funs.is_even(10))
```

THE IMPORT STATEMENT

- In order to make use of classes and functions in a module, we must first import module using **import** statement.
- The system maintains a table of modules that have been initialized, indexed by module name. This table is accessible as **sys**.**modules**.
- If no matching file is found, **ImportError** is raised. If a file is found, it is parsed, yielding an executable code block. If a syntax error occurs, **SyntaxError** is raised.
- Whenever module is imported, code in module (not classes and functions) is executed.

```
import module [as name] (, module [as name])*
from module import identifier [as name]
            (, identifier [as name])*
from module import *
```

The following are examples of import statement:

```
01 # import num_funs and refer to it by alias nf
02 import num_funs as nf
03
04 print(nf.is_even(10))
```

```
01 # import all definitions of module num_funs
02 from num_funs import *
03
04 print(is_even(10)
```

```
01 # import specific function from num_funs module
02 from num_funs import is_even, is_odd
03
04 print(is_odd(10))
```

THE DIR() FUNCTION

It is used to get members of the module. It returns a sorted list of strings.

The following shows members of num_funs module.

```
import num_funs
print(dir(num_funs))
```

Output:

```
['__builtins__', '__cached__', '__doc__', '__file__',
'__loader__', '__name__', '__package__', '__spec__',
'is_even', 'is_odd', 'is_positive']
```

NOTE: When **dir()** and **__name__** are used in a module they refer to current module.

THE HELP() FUNCTION

- It invokes the built-in help system.
- If no argument is given, the interactive help system starts on the interpreter console.
- If the argument is any other kind of object, a help page on the object is generated.

```
help([object])
```

Use SPACE key to go to next page and Q to quit the help system.

MODULE SEARCH PATH

When a module is imported, the interpreter first searches for a built-in module with that name. If not found, it then searches for a file named modulename.py in a list of directories given by the variable **sys.path**.

The **sys.path** is initialized from the following locations:

- ❑ The directory containing the input script (or the current directory when no file is specified).
- ❑ PYTHONPATH (a list of directory names, with the same syntax as the shell variable PATH).
- ❑ The installation-dependent default.

```
>>> import sys
>>> sys.path
['', 'C:\\python\\python38.zip', 'C:\\python\\DLLs',
'C:\\python\\lib', 'C:\\python', 'C:\\python\\lib\\site-
packages']
>>>
```

Setting PYTHONPATH

The following example sets PYTHONPATH to a few directories so that they are added to module search path.

```
c:\python>set PYTHONPATH=c:\dev\python;c:\dev\projects
```

NOTE: It is possible to add entries to **sys.path** as it is a list. Use **r** as prefix to indicate it is raw string so that \ (backslash) is treated as a normal character and not special character.

```
sys.path.append(r'c:\dev\python\projects')
```

EXECUTING MODULE AS SCRIPT

- A module can contain executable statements as well as function and class definitions. These statements are intended to initialize the module.
- Executable statements in a module are executed whenever you run module as a script and when you import module into another module using import statement.
- When you run a Python module using ***python filename.py*** then the code in the module will be executed, but with the __name__ set to *__main__*.
- But if we want to execute code only when module is run as script then we need to check whether name of the module is set to __main__.

module1.py

```
01 # this is simple module
02 def print_info():
03     print("I am in module1.print_info()")
04
05 # code executed when imported or when run as script
06 print("In Module1")
07
08 # code executed only when run as script
09 if __name__ == "__main__":
10     print("Running as script")
```

When you run the above code as script (**python.exe module1.py**) the following output is generated:

```
In Module1
Running as script
```

But when you import this module into another file as shown below then the output shown below is generated.

use_module1.py

```
01 import module1
02
03 module1.print_info()
```

Output when module is just imported and function is called:

```
In Module1
I am in module1.print_info()
```

USING COMMAND LINE ARGUMENTS

- It is possible to pass command line arguments while invoking a module from command line.
- Command line arguments are placed in **argv** list, which is present in **sys** module.
- First element in **sys.argv** is always name of the module that is being executed.

argv_demo.py

```
01 import sys
02 print("No. of arguments:", len(sys.argv))
03 print("File: ", sys.argv[0])
04 for v in sys.argv[1:]:
05     print(v)
```

Run **argv_demo.py** as script using python as follows:

```
C:\python>python argv_demo.py first second
No. of arguments: 3
File: argv_demo.py
first
second
```

DOCUMENTATION

- By convention, every function must be documented using documentation conventions.
- Documentation is provided between three double quotes (""").
- The first line should always be a short, concise summary of the object's purpose. This line should begin with a capital letter and end with a period.

```
01 def add(n1,n2):
02     """Adds two numbers and returns the result.
03
04     Args:
05         n1(int) : first number.
06         n2(int) : second number.
07
08     Returns:
09         int : Sum of the given two numbers.
10     """
11
12     return n1 + n2
13
14 help(add)           # prints documentation for add()
15 print(add.__doc__)  # prints documentation
```

PACKAGES

- Package is a collection of modules.
- When importing the package, Python searches through the directories on ***sys.path*** looking for the package subdirectory.
- Generally, **__init__.py** file is used to make Python treat the directory as package; this is done to prevent directories with a common name, such as string, from unintentionally hiding valid modules that occur later on the module search path. However, **__init__.py** is optional.
- File **__init__.py** can just be an empty file, but it can also execute initialization code for the package or set the **__all__** variable.
- Users of the package can import individual modules from the package.

Folder Structure

```
use_st_lib.py
stlib
  __init__.py
  str_funs.py
  num_funs.py
  mis_funs.py
```

stlib\str_funs.py

```
01 def has_upper(st):
02       # code
03 def has_digit(st):
04       # code
```

use_st_lib.py

```
01 # Import module from package
02 import stlib.str_funs
03
04 # call a function in module
05 print(stlib.str_funs.has_upper("Python"))
06
07 # import a function from a module in a package
08 from stlib.str_funs import has_digit
09
10 # call function after it is imported
11 print(has_digit("Python 3.10"))
```

Importing with *

- In order to import specific modules when * is used for module with package, we must define variable **__all__** in package's **__init__.py** to list modules that are to be imported.
- If variable __all__ is not defined in __init__.py then Python ensures that the package has been imported (running initialization code in __init__.py) and then imports whatever names are defined in the package but no modules are imported.

stlib__init__.py

```
__all__ = ["num_funs", "str_funs"]
```

PIP AND PYPI

- PyPI (Python Package Index) is a repository of python packages.
- URL **https://pypi.org/** lists all python packages that we can download and use.
- PIP is a general-purpose installation tool for Python packages.
- Run **pip.exe** from **python\scripts** folder.

The following are some of the important options available with PIP:

```
>pip install requests       # Installs requests package
>pip uninstall requests     # Uninstalls requests package
>pip list                   # Lists all installed packages
```

```
>pip show requests
Name: requests
Version: 2.27.1
Summary: Python HTTP for Humans.
Home-page: https://requests.readthedocs.io
Author: Kenneth Reitz
Author-email: me@kennethreitz.org
License: Apache 2.0
Location: c:\python\lib\site-packages
Requires: certifi, charset-normalizer, idna, urllib3
Required-by:
```

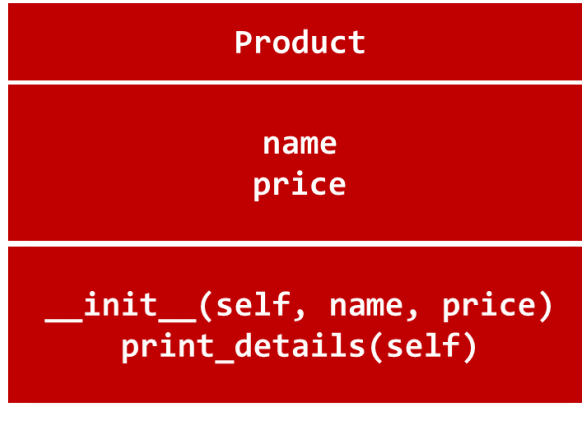
CLASSES

- A class contains data (data attributes) and code (methods) encapsulated.
- Creating a new class creates a new *type*, allowing new *instances* of that type to be made.
- Data attributes need not be declared; like local variables, they spring into existence when they are first assigned a value.
- Class *instantiation* uses function notation. Just pretend that the class object is a parameter-less function that returns a new instance of the class.
- The special thing about methods is that the instance object is passed as the first argument (called self) of the function.

```
class className:
   definition
```

__init__ method

- When a class defines an __init__() method, class instantiation automatically invokes __init__() for the newly-created class instance.
- Arguments given to the class instantiation operator are passed on to __init__().



```
01 class Product:
02     def __init__(self, name, price):
03         # Object attributes
04         self.name = name
05         self.price = price
06 
07     def print_details(self):
08         print("Name  : ", self.name)
09         print("Price : ", self.price)
10 
11 p = Product("Dell XPS Laptop",80000)  # create object
12 p.print_details()
```

Private members (Name Mangling)

- Python doesn't have private variables concept.
- However, a convention followed by most Python programmers is, an attribute prefixed with a single underscore (_attribute) or double underscore (__attribute) should be treated as non-public (protected and private respectively) part of the API.
- Any identifier of the form **__attribute** (two leading underscores) is textually replaced with **_classname__attribute**, where classname is the current class name. This process is called as **name mangling**.
- For example, *__salary* in *Employee* class becomes *_Employee__salary*.
- However, it is still possible to access or modify a variable that is considered private from outside the class.

Name	Notation	Behaviour
name	Public	Can be accessed from inside and outside.
_name	Protected	Like a public member, but they shouldn't be directly accessed from outside.
__name	Private	Can't be seen and accessed from outside.

```
01 class Product:
02     def __init__(self, name, price):
03         self.__name = name
04         self.__price = price
05
06     def print_details(self):
07         print("Name  : ", self.__name)
08         print("Price : ", self.__price)
```

As attributes name and price are prefixed with __ (double underscore) they are to be treated as private members of the class. Python will prefix classname to those attributes.

The following code fails to access *__name* attribute because its name is prefixed with class name due to ***name mangling***.

```
p = Product("Dell XPS Laptop", 80000)
print(p.__name)       # will throw error
```

```
AttributeError : 'Product' object has no attribute '__name'
```

However, you can access private attributes from outside if you use _classname as prefix as shown below:

```
p = Product("Dell XPS Laptop", 80000)
print(p._Product__name)
```

NOTE: Object attribute ***__dict__*** returns a dictionary of attributes, where keys are attribute names and values are attribute values.

STATIC METHODS AND VARIABLES

- Any variable declared in the class is called class variable. It is also known as static variable and class attribute.
- Any method created with **@staticmethod** decorator becomes static method.
- Static methods are not passed any parameter by default.
- Static methods are called with class name.
- Static methods perform operations that are related to class such as manipulating static variables.

```
01 class Point:
02      # Static attributes
03      max_x = 100
04      max_y = 50
05      def __init__(self, x, y):
06          self.x = x
07          self.y = y
08
09      @staticmethod
10      def isvalid(x,y):
11         return x <= Point.max_x and y <= Point.max_y
```

In order to call a static method, we need to use classname as follows:

```
print(Point.isvalid(10,20))
```

CLASS METHODS

- When a method in the class is decorated with **@classmethod**, it is called as a class method.
- Class methods are used as **factory methods** to create and return objects of class.
- They are always passed the class that is invoking them, as first parameter.

```
01 class Time:
02      @classmethod
03      def create(cls):
04          return cls(0,0,0)
05
06      def __init__(self,h,m,s):
07          self.h = h
08          self.m = m
09          self.s = s
10
11 # create an object
12 t = Time.create()   # Time is passed to create()
```

Comparison of methods

Here is a table listing different types of methods that can be created in a class and their characteristics.

	Instance Method	Static Method	Class Method
Decorator used	None	@staticmethod	@classmethod
Invoked by	Object	Classname	Classname
Must return	None	None	An object
Mandatory param	Self	None	Class

BUILT-IN FUNCTIONS RELATED TO ATTRIBUTES

- It is possible to create new attributes any time by just assigning value to attribute using an object.
- If class name is used with attribute, it becomes class attribute.
- If object is used with attribute, it becomes object attribute.
- We can also use the following predefined methods to manipulate attributes of class or object.

Function	Meaning
getattr (object, name [, default])	Returns the value of the named attribute of object If attribute is found otherwise returns default value, if given, else raises error.
hasattr (object, name)	Returns True if object has the attribute.
setattr (object, name, value)	Creates or modifies an attribute with the given value.
delattr (object, name)	Deletes the specified attribute from the given object.

```
01 class Product:
02     tax = 10
03     def __init__(self,name):
04         self.name = name
```

```
>>> p = Product("iPad Air 2")
>>> getattr(p,'qoh',0)
0
>>> setattr(p,'price',45000)
>>> hasattr(p,'price')
True
>>> delattr(p,'price')
>>> hasattr(p,'price')
False
```

BUILT-IN CLASS ATTRIBUTES

Every Python class has the following built-in attributes.

Attribute	Description
__dict__	Dictionary containing the members.
__doc__	Class documentation string or none, if undefined.
__name__	Class name.
__module__	Module name in which the class is defined. This attribute is "__main__" when module is run as a script.
__bases__	A tuple containing the base classes, in the order of their occurrence in the base class.

```
>>> Product.__module__
'__main__'
>>> Product.__bases__
(<class 'object'>,)
>>> Product.__dict__
mappingproxy({'__module__': '__main__', 'tax': 10,
'__init__': <function Product.__init__ at
0x00000269FF186280>, '__dict__': <attribute '__dict__' of
'Product' objects>, '__weakref__': <attribute '__weakref__'
of 'Product' objects>, '__doc__': None})
```

SPECIAL METHODS

- Python allows us to overload different operators and operations related to our class by implementing special methods.
- Objects related to operation are passed as parameters to function.

Relational operators

The following special methods represent relational operators. By implementing these methods, we provide support for those operators in our user-defined class.

Operator	Method
<	object.__lt__(self, other)
<=	object.__le__(self, other)
==	object.__eq__(self, other)
!=	object.__ne__(self, other)
>=	object.__ge__(self, other)
>	object.__gt__(self, other)

Unary operators

The following are special methods for unary operators.

Operator	Method
-	object.__neg__(self)
+	object.__pos__(self)
abs()	object.__abs__(self)
~	object.__invert__(self)
complex()	object.__complex__(self)
int()	object.__int__(self)
float()	object.__float__(self)
oct()	object.__oct__(self)
hex()	object.__hex__(self

Binary operators

The following are special methods related to binary operators.

Operator	Method
+	object.__add__(self, other)
-	object.__sub__(self, other)
*	object.__mul__(self, other)
//	object.__floordiv__(self, other)
/	object.__truediv__(self, other)
%	object.__mod__(self, other)
**	object.__pow__(self, other[, modulo])
<<	object.__lshift__(self, other)
>>	object.__rshift__(self, other)
&	object.__and__(self, other)
^	object.__xor__(self, other)
|	object.__or__(self, other)

Extended assignments

Here are special methods related to extended operators.

Operator	Method
+=	object.__iadd__(self, other)
-=	object.__isub__(self, other)
*=	object.__imul__(self, other)
/=	object.__idiv__(self, other)
//=	object.__ifloordiv__(self, other)
%=	object.__imod__(self, other)
**=	object.__ipow__(self, other[, modulo])
<<=	object.__ilshift__(self, other)
>>=	object.__irshift__(self, other)
&=	object.__iand__(self, other)
^=	object.__ixor__(self, other)
|=	object.__ior__(self, other)

The following program shows how to implement special methods.

```
01 class Time:
02      def __init__(self, h=0, m=0, s=0):
03          """ Initializes hours, mins and seconds """
04          self.h = h
05          self.m = m
06          self.s = s
07
08      def total_seconds(self):
09          """Returns total no. of seconds """
10          return self.h * 3600 + self.m * 60 + self.s
11
12      def __eq__(self, other):
13          return self.total_seconds() ==   \
14                          other.total_seconds()
15
16      def __str__(self):
17         return f"{self.h:02}:{self.m:02}:{self.s:02}"
18
19
20      def __bool__(self):
21       """Returns false if hours, mins and seconds
22           are 0 otherwise true
23        """
24          return self.h != 0 or self.m != 0   \
25                                or self.s != 0
26
27      def __gt__(self,other):
28          return self.total_seconds() >     \
29                          other.total_seconds()
30
```

```
31      def __add__(self, other):
32          return Time(self.h + other.h,
33                      self.m + other.m, self.s + other.s)
34
35 t1 = Time(1, 20, 30)
36 t2 = Time(10, 20, 30)
37 print(t1)
38 print(t1 == t2)
39
40 t3 = Time()     # h,m,s are set to zeros
41 if t3:
42      print("True")
43 else:
44      print("False")
45
46 print (t1 < t2)
47
48 t4 = t1 + t2
49 print(t4)
```

```
01:20:30
False
False
True
11:40:60
```

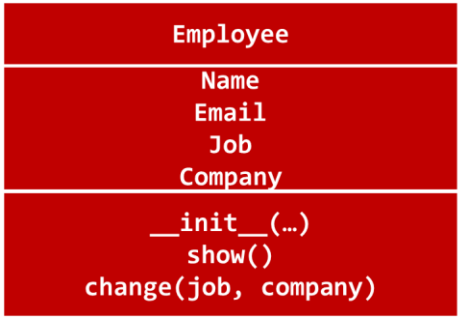
PROPERTIES

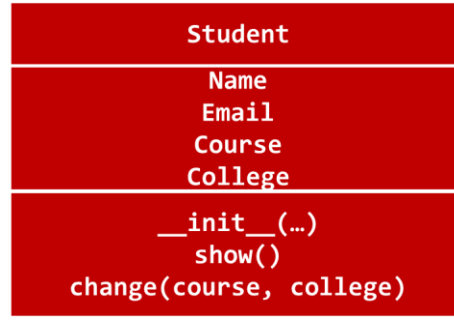
- It is possible to create a property in Python using two decorators - **@property** and **@setter**.
- A property is used like an attribute, but it is internally implemented by two methods – one to get value (getter) and one to set value (setter).
- Properties provide advantages like validation, abstraction and lazy loading.

```
01 class Person:
02     def __init__(self, first='', last=''):
03         self.__first = first
04         self.__last = last
05 
06     @property    # Getter
07     def name(self):
08         return self.__first + " " + self.__last
09 
10     @name.setter   # Setter
11     def name(self, value):
12         self.__first, self.__last = value.split(" ")
13 
14 
15 p = Person("Srikanth", "Pragada")
16 print(p.name)   # Calls @property getter method
17 p.name="Pragada Srikanth" #Calls @name.setter method
```

INHERITANCE

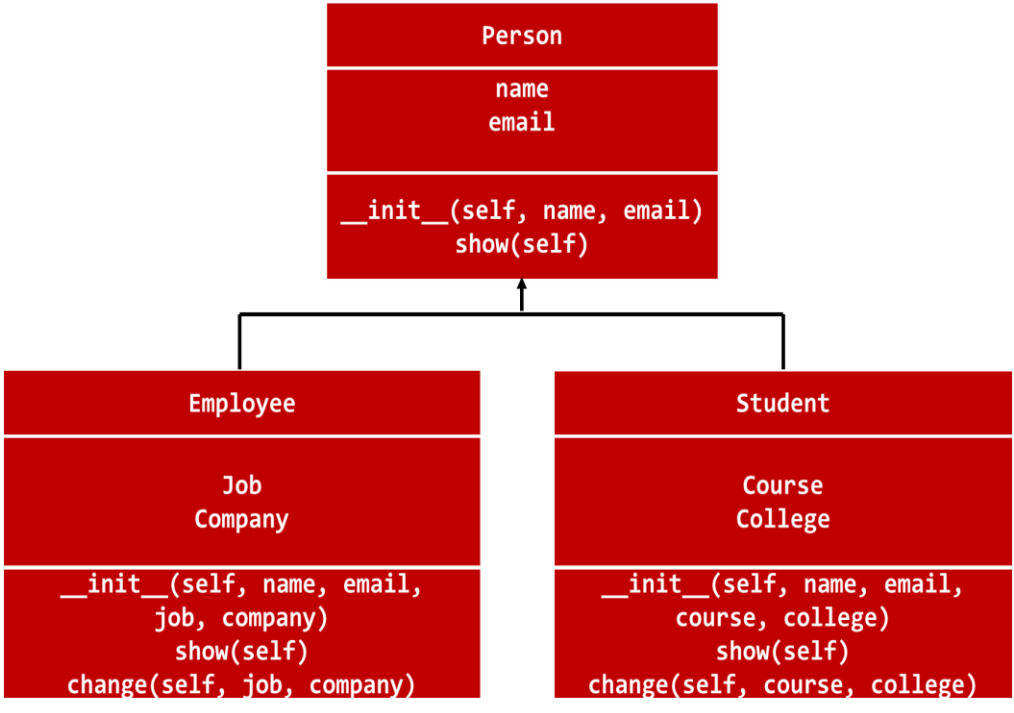
- When a new class is created from an existing class, it is called as inheritance. It enables us to reuse existing classes while creating new classes.
- A new class can be created from *one or more* existing classes.
- If a requested attribute is not found in the class, the search proceeds to look in the base class. This rule is applied recursively if the base class itself is derived from some other class.
- New class is called **subclass** and the class being inherited is called **superclass**.
- Subclass can **override** a method of superclass to enhance or change functionality of superclass method.
- Function **super**() is used to access superclass from subclass.
- It is possible to call methods of superclass using super() function – *super().methodname(arguments).*
- It is also possible to call superclass method directly - *superclassname.methodname(self, arguments)*. We must send self as first argument.
- Inheritance is also known as *generalization* as we start with most generic class (superclass) and create more specific classes (subclasses) later.





```
class SubclassName (superclass [, superclass]…):
<statement-1>
…
<statement-N>
```

NOTE: Every class that is not a subclass of another class is implicitly inheriting **object** class.



```
class Employee:
    def __init__(self,name, salary):
        self.__name = name
        self.__salary = salary
    def print(self):
        print(self.__name)
        print(self.__salary)
    def get_salary(self):
        return self.__salary
```

```
class Manager(Employee):
    def __init__(self,name, salary, hra):
        super().__init__(name,salary)
        self.__hra = hra
    def print(self):    # Overrides print()
        super().print()
        print(self.__hra)
    def get_salary(self):  # Overrides get_salary()
        return super().get_salary() + self.__hra


e = Employee("Scott",100000)
m = Manager("Mike",150000,50000)
e.print()
print("Net Salary : ", e.get_salary())
m.print()
print("Net Salary : ", m.get_salary())
```

Output:

```
Scott
100000
Net Salary :  100000
Mike
150000
50000
Net Salary :  200000
```

Overriding

- When a method in subclass is created with same name as a method in superclass, it is called as **overriding**.
- Subclass method is said to override method in superclass.
- Overriding is done to change the behavior of inherited method of superclass by creating a new version in subclass.

Functions isinstance() and issubclass()

- Function **isinstance**() checks whether an object is an instance of a class.
- Function **issubclass**() checks whether a class is a subclass of another class.

```
isinstance(object, class)
issubclass(class, class)
```

```
e = Employee(…)
print("Employee ??  ", isinstance(e, Employee)) # True
print("Manager subclass of Employee ?? ",
    issubclass(Manager, Employee))             # True
```

MULTIPLE INHERITANCE

Python supports a form of multiple inheritance as well. A class definition with multiple super classes is as follows:

```
class SubclassName(superclass1,superclass2,…):
. . .
```

Python searches for attributes in Subclass first. If not found it searches in superclass1, then (recursively) in the super classes of superclass1, and if it was not found there, it searches in superclass2, and so on.

In the following example, as method process() is not found in class C, Python calls process() method in class A as it is the first superclass.

```
01 class A:
02     def process(self):
03         print('A process()')
04
05
06 class B:
07     def process(self):
08         print('B process()')
09
10 class C(A, B):
11     pass
12
13 obj = C()
14 obj.process()  # will call process() of A
```

Python always considers subclass version, if one is present. In the following example, process() from class C is called because Python considers subclass version ahead of superclass version. So, it will not consider process() method in A as class A is superclass of C and method is present in class C.

```
01 class A:
02     def process(self):
03         print('A process()')
04
05
06 class B(A):
07     pass
08
09
10 class C(A):
11     def process(self):
12         print('C process()')
13
14
15 class D(B, C):
16     pass
17
18
19 obj = D()
20 obj.process()     # Calls method from class C
```

Method Resolution Order (MRO)

MRO is the order in which Python searches for a method in the hierarchy of classes.

For the above example, calling method **mro**() on class D will return the following:

```
[<class '__main__.D'>, <class '__main__.B'>, <class
'__main__.C'>, <class '__main__.A'>, <class 'object'>]
```

Please refer to the following additional resources on this topic:

- My blog at: http://www.srikanthtechnologies.com/blog/python/mro.aspx
- My video tutorial at: https://youtu.be/tViLEZXUO3U

ABSTRACT CLASS AND METHODS

- Support for abstract class and method is provided by module **abc**.
- Methods marked with @**abstractmethod** decorator of **abc** module are made abstract.
- An abstract method is a method that must be implemented by subclass.
- When an abstract method is present in a class then the class must be declared as abstract.
- A class that is to be abstract must extend class **ABC** (Abstract Base Class) of abc module.
- No instances of abstract class can be created.

```
01 from abc import ABC, abstractmethod
02 class Student(ABC):
03     @abstractmethod
04     def getsubject(self):
05         pass
06
07 class PythonStudent(Student):
08     def getsubject(self):
09         return "Python"
10
11 s = Student()           # throws error as shown below
12 ps = PythonStudent()
```

```
TypeError: Can't instantiate abstract class Student with
abstract methods getsubject
```

EXCEPTION HANDLING

- Errors detected during execution are called *exceptions.*
- Exceptions come in different types, and the type is printed as part of the message: the types are ZeroDivisionError, KeyError and AttributeError.
- **BaseException** is base class for all built-in exceptions.
- **Exception** is base class for all built-in, non-system-exiting exceptions. All user-defined exceptions should also be derived from this class.
- After try block, at least one except block or finally block must be given.

```
try:
   Statements
[except (exception [as identifier] [, exception] …)] … :
   Statements]
[else:
   Statements]
[finally:
   Statements]
```

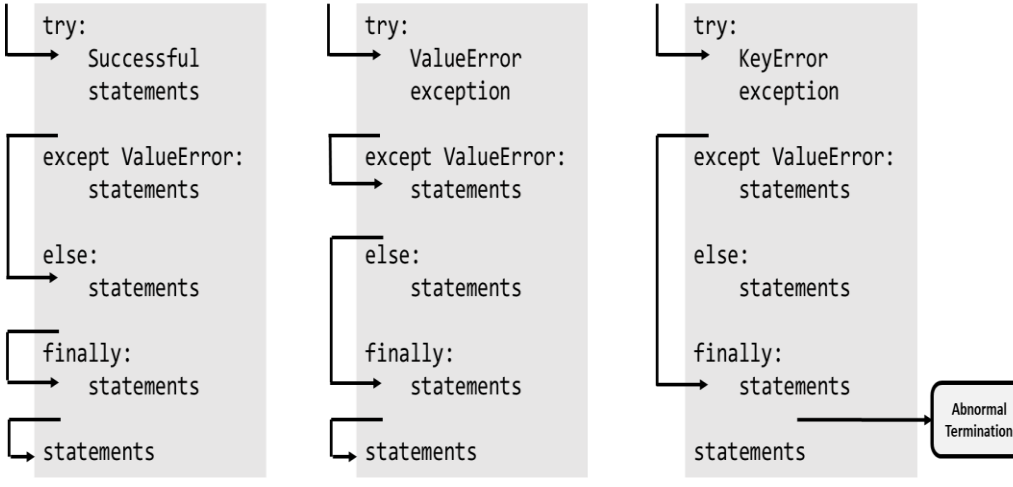
Clause	Meaning
try	Specifies exception handlers and/or cleanup code for a group of statements.
except	Specifies one or more exception handlers. It is possible to have multiple except statements for a single try statement. Each except can specify one or more exceptions that it handles.
else	Executed when try exits successfully.
finally	Executed at the end of try whether try succeeds or fails.

NOTE: After try, one *except* block or *finally* block must be given.

```
01 a = 10
02 b = 20
03
04 try:
05     c = a / b
06     print(c)
07 except:
08     print("Error")
09 else:
10     print("Job Done!")
11 finally:
12     print("The End!")
```

Output:

```
0.5
Job Done!
The End!
```



The following example produces a different result as value of **b** is 0. We are catching exception and referring to it using **ex** in **except** block. As **ex** contains error message, printing ex will produce error message. The **else** block is not executed as try failed with error.

```
01 a = 10
02 b = 0
03
04 try:
05     c = a / b
06     print(c)
07 except Exception as ex:
08     print("Error :", ex)
09 else:
10     print("Job Done!")
11 finally:
12     print("The End!")
```

Output:

```
Error : division by zero
The End!
```

The following program takes numbers from user until 0 is given and then displays sum of given numbers.

```
01 total = 0
02 while True:
03     num = int(input("Enter number [0 to stop]: "))
04     if num == 0:
05         break
06
07     total += num
08
09 print(f"Total = {total}")
```

But the program is fragile as any invalid input will crash the program as shown in output below:

```
Enter number [0 to stop]: 10
Enter number [0 to stop]: abc
Traceback (most recent call last):
  File "C:/dev/python/sum_demo.py", line 3, in <module>
    num = int(input("Enter number [0 to stop] :"))
ValueError: invalid literal for int() with base 10: 'abc'
```

In order to make program more robust so that it can continue in spite of invalid input from user, we need to enclose sensitive part of the program in try block and continue after displaying error message regarding invalid input.

```
01 total = 0
02 while True:
03     try:
04         num = int(input("Enter number [0 to stop]: "))
05         if num == 0:
06             break
07         total += num
08     except ValueError:
09         print("Invalid Number!")
10
11 print(f"Total = {total}")
```

In the output below, whenever invalid input is given, an error is displayed and program continues till end.

```
Enter number [0 to stop]: 10
Enter number [0 to stop]: abc
Invalid Number!
Enter number [0 to stop]: 30
Enter number [0 to stop]: xyz
Invalid Number!
Enter number [0 to stop]: 50
Enter number [0 to stop]: 0
Total = 90
```

Predefined Exceptions

The following are predefined exceptions in Python.

```
BaseException
 +-- SystemExit
 +-- KeyboardInterrupt
 +-- GeneratorExit
 +-- Exception
      +-- StopIteration
      +-- StopAsyncIteration
      +-- ArithmeticError
      |    +-- FloatingPointError
      |    +-- OverflowError
      |    +-- ZeroDivisionError
      +-- AssertionError
      +-- AttributeError
      +-- BufferError
      +-- EOFError
      +-- ImportError
      |    +-- ModuleNotFoundError
      +-- LookupError
      |    +-- IndexError
      |    +-- KeyError
      +-- MemoryError
      +-- NameError
      |    +-- UnboundLocalError
```

```
+-- OSError
|   +-- BlockingIOError
|   +-- ChildProcessError
|   +-- ConnectionError
|   |   +-- BrokenPipeError
|   |   +-- ConnectionAbortedError
|   |   +-- ConnectionRefusedError
|   |   +-- ConnectionResetError
|   +-- FileExistsError
|   +-- FileNotFoundError
|   +-- InterruptedError
|   +-- IsADirectoryError
|   +-- NotADirectoryError
|   +-- PermissionError
|   +-- ProcessLookupError
|   +-- TimeoutError
+-- ReferenceError
+-- RuntimeError
|   +-- NotImplementedError
|   +-- RecursionError
+-- SyntaxError
|   +-- IndentationError
|       +-- TabError
+-- SystemError
+-- TypeError
+-- ValueError
|   +-- UnicodeError
|       +-- UnicodeDecodeError
|       +-- UnicodeEncodeError
|       +-- UnicodeTranslateError
```

The raise statement

- Used to raise an exception.
- If no exception is given, it re-raises the exception that is active in the current scope.

```
raise [expression]
```

The given expression must be an object of a class that is a subclass of BaseException.

User-defined exception and raise statement

- A user-defined exception is an exception created by our program.
- Use raise statement to raise an exception, which is an object of exception class.
- User-defined exception class must be a subclass of Exception class.
- Subclass of Exception class can create its own attributes and methods.

```
01 # User-defined exception
02 class AmountError(Exception):
03     def __init__(self, message):
04         self.message = message
05     def __str__(self):
06         return self.message
```

```
01 try:
02   if amount < 1000:
03     raise AmountError("Minimum amount is 1000")
04 except Exception as ex:
05   print("Error : ", ex)
```

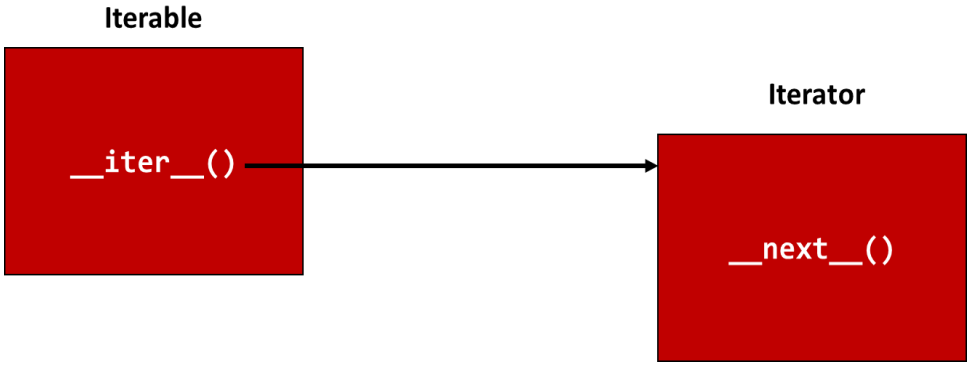
THE ITERATOR

- An iterator is an object representing a stream of data; this object returns the data one element at a time.
- Several of Python's built-in data types support iteration, the most common being lists and dictionaries.
- An object is called **iterable** if you can get an **iterator** for it.
- Method **iter**() of an object returns iterator object that defines **__next__()** method.
- Method **__next__()** is used to return next element and raises **StopIteration** exception when there are no more elements to return.

The following example shows how **list** class provides **list_iterator** to iterate over elements of list.

```
>>> l = [1, 2, 3]
>>> li = iter(l)
>>> type(l), type(li)
(<class 'list'>, <class 'list_iterator'>)
>>> next(li)
1
>>> next(li)
2
>>> next(li)
3
>>> next(li)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
>>>
```

An **iterable** class is the one that provides ***__iter__()*** method, which returns an object **iterator** class that provides ***__next__()*** method to return one element at a time.



In the following example **Marks** is an iterable class that provides **Marks_Iterator** class to iterate over it.

```
01 class Marks_Iterator:
02     def __init__(self, marks):
03         self.marks = marks
04         self.pos = 0
05
06     def __next__(self):
07         if self.pos == len(self.marks):
08             raise StopIteration
09         else:
10             value = self.marks[self.pos]
11             self.pos += 1       # move to next element
12             return value
13
```

```
14 class Marks:
15     def __init__(self):
16       self.marks = [20, 30, 40, 25, 66]
17
18     def __iter__(self):
19       return Marks_Iterator(self.marks)
```

It is possible to use an object of Marks as an iterable with for loop.

```
01 m = Marks()
02 for v in m:
03    print(v)
```

THE GENERATOR

- Generator is a simple and powerful tool for creating iterators.
- They are written like regular functions but use the **yield** statement whenever they want to return data.
- Each time **next**() is called on it, the generator resumes where it left off.
- Generator can be thought of as resumable function.
- Generator can be used where iterable is needed.

```
01 def leap_years(start, end):
02    for year in range(start, end + 1):
03      if year % 4 == 0 and year % 100 != 0 \
04                       or year % 400 == 0:
05           yield year
06
07 def alphabets(st):
08    for ch in st:
09         if ch.isalpha():
10             yield ch
11
12 # Get all leap years in the range  2000 and 2050
13 for year in leap_years(2000, 2050):
14     print(year)
15
16 # Get all alphabets from the given string
17 for ch in alphabets ("Python 3.10 version"):
18     print (ch, end=' ')
```

Generator Expression

Generator expression creates a generator object that returns a value at a time.

```
(expression for expr in sequence if condition)
```

```
>>> st = "Python 3.10"
>>> digits = (ch for ch in st if ch.isdigit())
>>> digits
<generator object <genexpr> at 0x0000026347C520B0>
>>> for d in digits:
...   print(d)
...
3
1
0
>>>
```

FILE HANDLING

The following are important functions related to file handling.

Function open()

Opens the specified file in the given mode and returns file object. If the file cannot be opened, an OSError is raised.

```
open(file, mode='r')
```

Mode	Meaning
'r'	Open for reading (default).
'w'	Open for writing, truncating the file first.
'x'	Open for exclusive creation, failing if the file already exists.
'a'	Open for writing, appending to the end of the file if it exists.
'b'	Binary mode.
't'	Text mode (default).
'+'	Open a disk file for updating (reading and writing).

NOTE: Python doesn't depend on the underlying operating system's notion of text files; all the processing is done by Python itself, and is therefore platform-independent.

NOTE: When a string is prefixed with r, it is called as raw string. In raw string, even character like \ is treated as simple character and not escape sequence character.

```
01 f = open(r"c:\python\names.txt","wt")
02 names = ["Python", "C#", "Java", "JavaScript", "C++"]
03
04 for name in names:
05     f.write(name + "\n")
06 f.close()
```

The with statement (Context Manager)

It is a good practice to use the **with** keyword when dealing with file objects. The advantage is that the file is properly closed after its suite finishes, even if an exception is raised at some point.

```
with expression [as variable]:
    with-block
```

By using parentheses, we can have multiple lines in context managers.

```
with
(  open("a.txt", "r") as file1,
   open("b.txt", "r") as file2
):
   print("Do something!")
```

File object

File object represents an open file. Built-in function **open**() returns File object on success.

The following are important attributes of file object.

Attribute	Meaning
file.closed	Returns true if file is closed, false otherwise.
file.mode	Returns access mode with which file was opened.
file.name	Returns name of the file.

The following are important methods of File object.

Method	Meaning
read([count])	Reads everything until end of file unless count is specified, otherwise only count number of chars.
readline()	Reads a single line. Returns empty string on EOF.
readlines()	Reads all lines and returns a list of lines.
close()	Closes and flushes content to file.
write(value)	Writes the given content to file.
tell()	Returns current position of file.
seek(offset, base)	Takes file pointer to the required location from the given base.

The following program displays all lines along with line numbers.

```
01 with open(r"c:\python\names.txt","r") as f:
02   for idx, name in enumerate(f.readlines(), start = 1):
03       print(f"{idx:03} : {name.strip()}")
```

NOTE: While reading lines from file, **readlines**() reads line along with new line (\n) character at the end of the line.

Use **strip**() function of str class to get rid of it, if necessary.

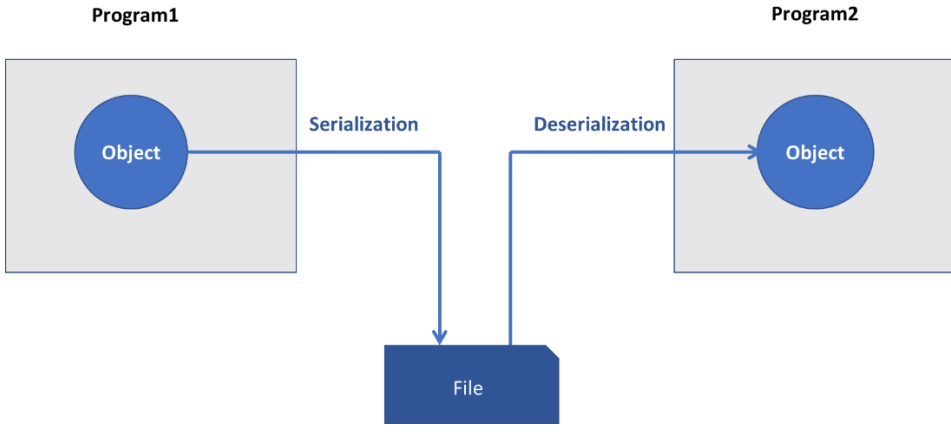
The following program displays all customer names and phone numbers in the sorted order of customer name by reading data from **phones.txt**, which is given below:

```
Steve,9339933390
Jason,3939101911
Ben,2939991113
George,3939999999
Larry
Ellison,39393999393
```

```
01 f = open("phones.txt","rt")
02 phones = {}      # Empty dictionary
03 for line in f:
04     # Split line into two parts – name and phone
05     parts = line.split(",")
06
07     # Ignore line if it doesn't contain 2 parts
08     if len(parts) != 2:
09         continue
10
11     # Add entry to dictionary
12     phones[parts[0]]= parts[1].strip()
13
14 # sort by names and print along with phone number
15 for name,phone in sorted(phones.items()):
16     print(f"{name:20} - {phone}")
```

PICKLE – PYTHON OBJECT SERIALIZATION

- The pickle module is used to serialize and de-serialize Python objects.
- It provides methods like **dump**() for pickling and **load**() for unpickling.
- It uses binary protocol.



The following are functions in pickle module.

Function	Meaning
dump(obj, file)	Writes a pickled representation of obj to the open file object *file*.
dumps(obj)	Returns the pickled representation of the object as a bytes object, instead of writing it to a file.
load(file)	Reads a pickled object representation from the open file object *file* and returns the reconstituted object hierarchy specified therein.
loads(bytes_object)	Reads a pickled object hierarchy from a bytes object and returns the reconstituted object hierarchy specified therein.

```
01 import pickle
02 class Person:
03     def __init__(self, name, email):
04         self.name = name
05         self.email = email
06     def __str__(self):
07         return f"{self.name}-{self.email}"
08
09
10  f = open("person.dat", "wb")
11  p1 = Person("Srikanth", "srikanthpragada@yahoo.com")
12  pickle.dump(p1, f)                # pickle object
13  print("Dumped object to file!")
14  f.close()
```

The following program opens person.dat and retrieves (unpickles) object pickled to it.

```
01 # Unpickle object from person.dat
02 f = open("person.dat", "rb")
03 p1 = pickle.load(f)
04 print(p1)
```

JSON MODULE

- JSON module allows serializing and deserializing object to and from JSON format.
- JSON stands for **J**ava**S**cript **O**bject **N**otation.
- It converts a dict in Python to JSON object.
- It converts a JSON object back to dict object in Python.
- JSON array is converted to Python list, which contains dict as elements.
- Any Python iterable is converted to JSON array.

The following are methods available in **json** module:

```
dump(object, file)
dumps(object)
load(file)
loads(str)
```

```
01 import json
02
03 class Contact:
04     def __init__(self, name, phone, email):
05         self.name = name
06         self.phone = phone
07         self.email = email
08
09 c = Contact("Srikanth",
10             "9059057000",
11             "contact@srikanthtechnologies.com")
12 print(json.dumps(c.__dict__))
```

The above program will generate the following JSON:

```
{"name": "Srikanth", "phone": "9059057000", "email":
"contact@srikanthtechnologies.com"}
```

The following code converts each Contact object to dict object using map() function and then converts the list of dict to an array of JSON objects.

```
contacts = (Contact("A","8888899999","a@gmail.com"),
            Contact("B","9999988888","b@yahoo.com"))
# convert each Contact object to dict using map()
clist = list(map(lambda c : c.__dict__, contacts))
print(json.dumps(clist))
```

Function map() is used to convert each Contact object to a dict. As map() function returns only a map object, we need to convert that to list using list().

Eventually we give a list of dict objects to dumps() to generate an array of JSON objects as follows:

```
[{"name": "A", "phone": "8888899999", "email":
"a@gmail.com"}, {"name": "B", "phone": "9999988888",
"email": "b@yahoo.com"}]
```

THE SYS MODULE

- This module provides access to some variables used or maintained by the interpreter and to functions that interact strongly with the interpreter.
- It provides members to access command line arguments, exit program etc.

Member	Meaning
argv	The list of command line arguments passed to a Python script. argv[0] is the script name.
exc_info()	This function returns a tuple of three values that give information about the exception that is currently being handled.
exit([arg])	Exits from Python.
getsizeof (object [, default])	Returns the size of an object in bytes.
modules	This is a dictionary that maps module names to modules which have already been loaded.
path	A list of strings that specifies the search path for modules. Initialized from the environment variable PYTHONPATH, plus an installation-dependent default.
platform	This string contains a platform identifier.
stdin, stdout, stderr	File objects used by the interpreter for standard input, output and errors.
version	A string containing the version number of the Python interpreter plus additional information on the build number and compiler used.

THE OS MODULE

- This module provides a portable way of using operating system dependent functionality.
- All functions in this module raise OSError in case of invalid or inaccessible file names and paths, or other arguments that have the correct type, but are not accepted by the operating system.

Function	Meaning
chdir(path)	Changes current directory.
getcwd()	Returns current directory.
getenv(key, default=None)	Returns the value of the environment variable key if it exists, or default if it doesn't. key, default and the result are str.
putenv (key, value)	Sets the environment variable named key to the string value.
listdir(path='.')	Returns a list containing the names of the entries in the directory given by path.
mkdir(path)	Creates a directory named path.
remove(path)	Removes (deletes) the file path.
removedirs(name)	Removes directories recursively.
rename(src, dst)	Renames the file or directory src to dst.
rmdir(path)	Removes directory.
walk(top)	Generates the file names in a directory tree by walking the tree either top-down or bottom-up.

```
01 import os
02 # get all files from given folder
03 files = os.listdir(r"c:\python")
04 for file in files:
05     print(file)    # print filename
```

```
01 import os
02
03 # Get all files and folders from the given path
04 allfiles = os.walk(r"c:\dev\python")
05
06 for  (dirname , directories , files) in allfiles:
07      # print directory name
08      print("Directory : ", dirname)
09      print("=============" + "=" *  len(dirname))
10
11      # print files in that directory
12      for file in files:
13          print(file)
```

USING RE (REGULAR EXPRESSION) MODULE

- Module **re** provides methods that use regular expressions.
- A regular expression (or RE) is a string with special characters that specifies which strings would match it.
- Module **re** provides functions to search for a partial and full match for a regular expression in the given string. It provides functions to split strings and extract strings using regular expression.

The following are special characters (metacharacters) used in regular expressions:

Character	Description
[]	A set of characters
\	Signals a special sequence (can also be used to escape special characters)
.	Any character (except newline character)
^	Starts with
$	Ends with
*	Zero or more occurrences
+	One or more occurrences
{}	Exactly the specified number of occurrences
|	Either or
()	Represents a group

The following are special sequences that have special meaning in a regular expression.

Character	Description
\d	Returns a match where the string contains a digit (numbers from 0-9).
\D	Returns a match where the string contains a non-digit.
\s	Returns a match where the string contains a whitespace character.
\S	Returns a match where the string contains a non-whitespace character.
\w	Returns a match where the string contains any word character (characters from a to Z, digits from 0-9, and the underscore _ character).
\W	Returns a match where the string contains a non-word character.

The following are functions provided by **re** module.

Function	Meaning
compile(pattern, flags=0)	Compiles a regular expression pattern into a regular expression object, which can be used for matching using its match(), search() and other methods.
search(pattern, string,flags=0)	Scans through string looking for the first location where the regular expression pattern produces a match, and returns a corresponding match object. Returns None if no position in the string matches the pattern.
match(pattern, string,flags=0)	If zero or more characters at the beginning of string match the regular expression pattern, returns a corresponding match object. Returns None if the string does not match the pattern.
fullmatch(pattern, string,flags=0)	If the whole string matches the regular expression pattern, returns a corresponding match object. Returns None if the string does not match the pattern.
split(pattern,string, maxsplit=0, flags=0)	Splits string by the occurrences of pattern. If capturing parentheses are used in pattern, then the text of all groups in the pattern are also returned as part of the resulting list. If maxsplit is nonzero, at most maxsplit splits occur, and the remainder of the string is returned as the final element of the list.
findall(pattern, string, flags=0)	Returns all non-overlapping matches of pattern in string, as a list of strings. The string is scanned left-to-right, and matches are returned in the order found.
sub(pattern, replace, string)	Replaces string that matches pattern with the given string.

```
>>> import re
>>> st ="abc 123 xyz pqr 456"
>>> re.match(r'\w+', st)   # Looks only at start of string
<re.Match object; span=(0, 3), match='abc'>
>>> re.match(r'\d+',st)    # Returns None
```

```
>>> re.search(r'\d+', st)
<re.Match object; span=(4, 7), match='123'>
>>>
```

```
>>> re.findall(r'\d+', st)
['123', '456']
>>> re.split(r'\W+', st)
['abc', '123', 'xyz', 'pqr', '456']
```

```
>>> re.sub(r'[0-9]','.', st)
'abc ... xyz pqr ...'
```

Match Object

Match object is returned by match() and search() and fullmatch() functions of **re** module.

Function	Meaning
group ([group1, ...])	Returns one or more subgroups of the match. If there is a single argument, the result is a single string; if there are multiple arguments, the result is a tuple with one item per argument. Without arguments, group1 defaults to zero (the whole match is returned). If the regular expression uses the (?P<name>...) syntax, the groupN arguments may also be strings identifying groups by their group name.
groups()	Returns a tuple containing all the subgroups of the match, from 1 up to however many groups are in the pattern. The default argument is used for groups that did not participate in the match; it defaults to None.
groupdict (default=None)	Returns a dictionary containing all the named subgroups of the match, keyed by the subgroup name.
start([group]), end([group])	Returns the indices of the start and end of the substring matched by group.
span([group])	For a match m, returns the 2-tuple (m.start(group), m.end(group)).
pos	Returns the value of pos which was passed to the search() or match() method of a regex object.
endpos	Returns the value of endpos which was passed to the search() or match() method of a regex object.
lastindex	Returns the integer index of the last matched capturing group, or None if no group was matched at all.

lastgroup	Returns the name of the last matched capturing group, or None if the group didn't have a name or if no group was matched at all.
re	Returns the regular expression object whose match() or search() method produced this match instance.
string	Returns the string passed to match() or search().

The following example uses grouping concept to extract name and phone number from the given string.

```
>>> st = "Srikanth Pragada 9059057000"
>>> m = re.fullmatch(r'([A-Za-z ]+)(\d+)',st)
>>> m.group(1)
'Srikanth Pragada '
>>> m.group(2)
'9059057000'
>>> m.span(2)
(17, 27)
>>> m.groups()
('Srikanth Pragada ', '9059057000')
```

THE DATETIME MODULE

- The datetime module supplies classes for manipulating dates and times in both simple and complex ways.
- Provides classes like **date**, **time**, **datetime** and **timedelta.**
- A timedelta object represents a duration, the difference between two dates or times.
- Modules **calendar** and **time** provide additional functionality related to dates and times.
- Type **date** contains year, month and day.
- Type **time** contains hour, minute, second and microsecond.
- Type **datetime** is a composite of date and time.
- Type **timedelta** contains days, seconds, microseconds.
- They are all immutable.

The date type

A date object represents a date (year, month and day).

```
datetime.date(year, month, day)
```

All arguments are required. Arguments are integers, in the following ranges:

- MINYEAR <= year <= MAXYEAR
- 1 <= month <= 12
- 1 <= day <= number of days in the given month and year

The class method **today**() returns the current local date.

Class attributes **date.min** and **date.max** contain the earliest and latest representable dates - date(MINYEAR, 1, 1) and date(MAXYEAR, 12, 31).

Operation	Result
date1 + timedelta	Adds timedelta.days to date1.
date1 - timedelta	Subtracts timedelta.days from date1.
date1 - date2	Subtracts date2 from date1 and returns timedelta to represent period between dates.
date1 < date2	Returns true if date1 is less than date2.

Attribute	Meaning
year	Between MINYEAR and MAXYEAR inclusive.
month	Between 1 and 12 inclusive.
day	Between 1 and the number of days in the given month of the given year.

Instance Method	Meaning
replace (year, month, day)	Returns a date with the same value, except for those parameters given new values by whichever keyword arguments are specified.
weekday()	Returns the day of the week as an integer, where Monday is 0 and Sunday is 6.
isoweekday()	Returns the day of the week as an integer, where Monday is 1 and Sunday is 7.
isocalendar()	Returns a 3-tuple, (ISO year, ISO week number, ISO weekday).
isoformat()	Returns a string representing the date in ISO 8601 format, 'YYYY-MM-DD'.
ctime()	Returns a string representing the date.
strftime(format)	Returns a string representing the date, controlled by an explicit format string.

The time type

A time object represents (local) time of day.

```
datetime.time(hour=0, minute=0, second=0, microsecond=0)
```

Attribute	Meaning
hour	Hours between 0 to 23
minute	Minutes between 0 to 59
second	Seconds between 0 to 59
microsecond	Microseconds between 0 and 999999

Instance Method	Meaning
replace(hour, minute, second, microsecond)	Returns a time with the same value, except for those attributes given new values by whichever keyword arguments are specified.
isoformat (timespec='auto')	Returns a string representing the time in ISO 8601 format, HH:MM:SS.mmmmmm or, if microsecond is 0, HH:MM:SS.
strftime(format)	Returns a string representing the time, controlled by an explicit format string.

The datetime type

A datetime object represents both date and time.

```
datetime.datetime (year, month, day, hour=0, minute=0,
second=0, microsecond=0, tzinfo=None, *, fold=0)
```

Class Method	Meaning
today()	Returns the current local datetime, with tzinfo None.
now()	Returns the current local date and time.
utcnow()	Returns the current UTC date and time
combine (date, time)	Returns a new datetime object combining date and time.
strptime (date_string, format)	Returns a datetime corresponding to date_string, parsed according to format.

Attribute	Meaning
year	Between MINYEAR and MAXYEAR inclusive.
month	Between 1 and 12 inclusive.
day	Between 1 and the number of days in the given month of the given year.
hour	In range(24).
minute	In range(60).
second	In range(60).
microsecond	In range(1000000).

Instance Method	Meaning
date()	Returns date object with same year, month and day.
time()	Returns time object with same hour, minute, second, microsecond and fold.
replace(year, month, day, hour, minute, second, microsecond)	Returns a datetime with the same attributes, except for those attributes given new values by whichever keyword arguments are specified.
weekday()	Returns the day of the week as an integer, where Monday is 0 and Sunday is 6. The same as self.date().weekday(). See also isoweekday().
isoweekday()	Returns the day of the week as an integer, where Monday is 1 and Sunday is 7.
isocalendar()	Returns a named tuple with three components - year, week and weekday.
isoformat()	Returns a string representing the date and time in ISO 8601 format
ctime()	Returns a string representing the date and time.
strftime(format)	Returns a string representing the date and time, controlled by an explicit format string.

The timedelta type

A timedelta object represents a duration, the difference between two dates or times.

```
timedelta(days=0, seconds=0, microseconds=0,
          milliseconds=0, minutes=0, hours=0, weeks=0)
```

Only *days*, *seconds* and *microseconds* are stored internally. Arguments are converted to those units:

- A millisecond is converted to 1000 microseconds
- A minute is converted to 60 seconds
- An hour is converted to 3600 seconds
- A week is converted to 7 days

The following are instance attributes of timedelta type:

Attribute	Value
days	Between -999999999 and 999999999 inclusive
seconds	Between 0 and 86399 inclusive
microseconds	Between 0 and 999999 inclusive

Format Codes

The following are format codes used in strftime() and strptime() functions.

Directive	Meaning	Example
%a	Weekday as locale's abbreviated name.	Sun, Mon, …, Sat
%w	Weekday as a decimal number, where 0 is Sunday and 6 is Saturday.	0, 1, …, 6
%d	Day of the month as a zero-padded decimal number.	01, 02, …, 31
%b	Month as locale's abbreviated name.	Jan, Feb, …, Dec
%m	Month as a zero-padded decimal number.	01, 02, …, 12
%y	Year without century as a zero-padded decimal number.	00, 01, …, 99
%Y	Year with century as a decimal number.	2018
%H	Hour (24-hour clock) as a zero-padded decimal number.	00, 01, …, 23
%I	Hour (12-hour clock) as a zero-padded decimal number.	01, 02, …, 12
%p	Locale's equivalent of either AM or PM.	AM, PM
%M	Minute as a zero-padded decimal number.	00, 01, …, 59
%S	Second as a zero-padded decimal number.	00, 01, …, 59
%j	Day of the year as a zero-padded decimal number.	001, 002, …, 366
%c	Locale's appropriate date and time representation.	Tue Aug 16 21:30:00 1988 (en_US)

```
>>> from datetime import *
>>> date.today()
datetime.date(2022, 4, 29)
>>> dob = date(1998, 10, 24)
>>> cd = date.today()
>>> cd - dob
datetime.timedelta(days=8588)
>>> cd + timedelta(days=10)
datetime.date(2022, 5, 9)
```

```
# Convert string to datetime
>>> datetime.strptime("24-oct-1998", "%d-%b-%Y")
datetime.datetime(1998, 10, 24, 0, 0)
```

```
>>> cd = datetime.now()
>>> cd.strftime("%d-%m-%Y %H:%M:%S")
'29-04-2022 11:03:07'
```

The following program takes date of birth from user and displays age in years, months and days.

```
01 from datetime import *
02
03 dobstr = input("Enter your date of birth (yyyymmdd) : ")
04 dob = datetime.strptime(dobstr, "%Y%m%d")
05 now = datetime.now()
06 diff = now – dob
07 years = diff.days // 365
08 months = diff.days % 365 // 30
09 days = diff.days - (years * 365 + months * 30)
10 print(f"{years} y {months} m {days} d")
```

```
Enter your date of birth (yyyymmdd) : 19981024
23 y 6 m 13 d
```

MULTITHREADING

- Threading is a technique for decoupling tasks which are not sequentially dependent.
- Threads can be used to improve the responsiveness of applications that accept user input while other tasks run in the background.
- Use **threading** module and **Thread** class to implement multi-threading.
- In order to create a new thread, extend **Thread** class and provide required code in **run**() method.
- Subclass of Thread class must call Thread.__init__() from subclass's __init__() if it overrides it in subclass.

```
01 #Create a new thread to print numbers from 1 to 10
02
03 from threading import Thread
04
05 class PrintThread(Thread):
06     def run(self):
07         for i in range(1,11):
08             print(i)
09
10 t1 = PrintThread()
11 t1.start()
```

Functions in threading module

The following are functions provided in multithreading module.

Function	Meaning
active_count	Returns the number of Thread objects currently alive.
current_thread	Returns the current Thread object, corresponding to the caller's thread of control.
main_thread	Returns the main Thread object. In normal conditions, the main thread is the thread from which the Python interpreter was started.
enumerate	Returns a list of all Thread objects currently alive.

Thread Class

Thread object represents a thread. The following are important methods of Thread class.

Method	Meaning
start()	Starts the thread's activity.
run()	Method representing the thread's activity.
join()	Waits until the thread terminates.
getName()	Returns thread's name.
setName()	Sets thread's name.
is_alive()	Returns whether the thread is alive.

The following program creates a thread to check whether the given number is prime or not.

```
01 def isprime(num):
02     for n in range(2, math.floor(math.sqrt(num)) + 1):
03         if num % n == 0:
04             print(f"{num} is not a prime number!")
05             break
06     else:
07         print(f"{num} is a prime number!")
08
09 nums = [393939393, 12121212121, 29292939327,
10         38433828281, 62551414124111]
11
12 for n in nums:
13    t = Thread(target=isprime, args=(n,))
14    t.start()
```

```
393939393 is not a prime number!
29292939327 is not a prime number!
12121212121 is a prime number!
62551414124111 is not a prime number!
38433828281 is a prime number!
```

REQUESTS MODULE

Requests is an elegant and simple HTTP library for Python, built for human beings.

It is to be installed using **pip** as follows:

```
pip install requests
```

The following are important methods of **requests** module:

```
request(method, url, **kwargs)
get(url, params=None, **kwargs)
post(url, data=None, json=None, **kwargs)
put(url, data=None, **kwargs)
delete(url, **kwargs)
```

The requests.Response object

The Response object contains server's response for an HTTP request. When a request is made using requests module, it returns an object of Response class.

Property	Meaning
content	Content of the response, in bytes.
cookies	A CookieJar of Cookies the server sent back.
headers	Case-insensitive Dictionary of Response Headers. For example, headers['content-encoding'] will return the value of a 'Content-Encoding' response header.
json(**kwargs)	Returns the json-encoded content of a response, if any.
reason	Textual reason of responded HTTP Status, e.g. "Not Found" or "OK".
request	The PreparedRequest object to which this is a response.

status_code	Integer Code of responded HTTP Status, e.g. 404 or 200.
text	Content of the response, in unicode.
url	URL location of Response.

The following program retrieves information about countries from **restcountries.com** and displays country name, capital, borders and population.

```
01 import requests
02
03 code = input("Enter country code :")
04 resp = requests.get
05      (f"https://restcountries.com/v3.1/alpha/{code}")
06 if resp.status_code == 404:
07     print("Sorry! Country code not found!")
08
09 elif resp.status_code != 200:
10     print("Sorry! Could not get country details!")
11 else:
12     # Take first elements in the list of dictionaries
13     details = resp.json()[0]
14     print("Country Information");
15     print("Name    : " + details["name"]["common"])
16     print("Capital : " + details["capital"][0])
17     print("Population   : " +
18           str(details["population"]))
19     print("Sharing borders with :")
20     for c in details["borders"]:
21         print(c)
```

BEAUTIFUL SOUP LIBRARY

- Beautiful Soup is a Python package for parsing HTML and XML documents.
- It creates a parse tree for parsed pages that can be used to extract data from HTML and XML, which is useful for web scraping.

Install beautiful soup using pip as follows:

```
pip install beautifulsoup4
```

To process XML document, install lxml package as follows and use xml as the parser.

```
pip install lxml
```

Creating an object of BeautifulSoup

- We need to create a BeautifulSoup object with required content.
- The content may be XML or HTML. BeautifulSoup creates a parse tree.
- Methods like find_all() search in parse tree and return matching tags.

```
BeautifulSoup(content, type)
```

Type of the content specifies what type of content is being parsed and which parse is to be used. Available options are:

Type	Meaning
html.parser	Uses Python's HTML Parser
lxml	Uses lxml's HTML parser
lxml-xml or xml	Uses lxml's XML parser

```
01 from bs4 import BeautifulSoup
02
03 with open("schedule.html") as f:
04      soup = BeautifulSoup(f.read(), "html.parser")
```

The following program displays titles from RSS document at *www.srikanthtechnologies.com/rss.xml*.

```
01 from bs4 import BeautifulSoup
02 import requests
03
04 resp = requests.get
05 ("http://www.srikanthtechnologies.com/rss.xml")
06
07 # must install lxml to use xml
08 soup = BeautifulSoup(resp.text, "xml")
09 for item in soup.find_all("item"):
10     print(item.find("title").text.strip())
```

Tag Object

Tag object corresponds to an XML or HTML tag in document.

Property	Meaning
name	Name of the tag
text	Text of the tag
[attribute]	Provides value for the given attribute in []
contents	Provides all children of the tag
children	Allows iteration over tag's children
descendants	Provides all descendants of the tag
parent	Provides parent tag for the tag

parents	Provides all parents from tag to top
next_sibling	Provides next sibling
previous_sibling	Provides previous sibling
next_element	Returns next element
previous_element	Returns previous element
attrs	Provides all attributes of the tag

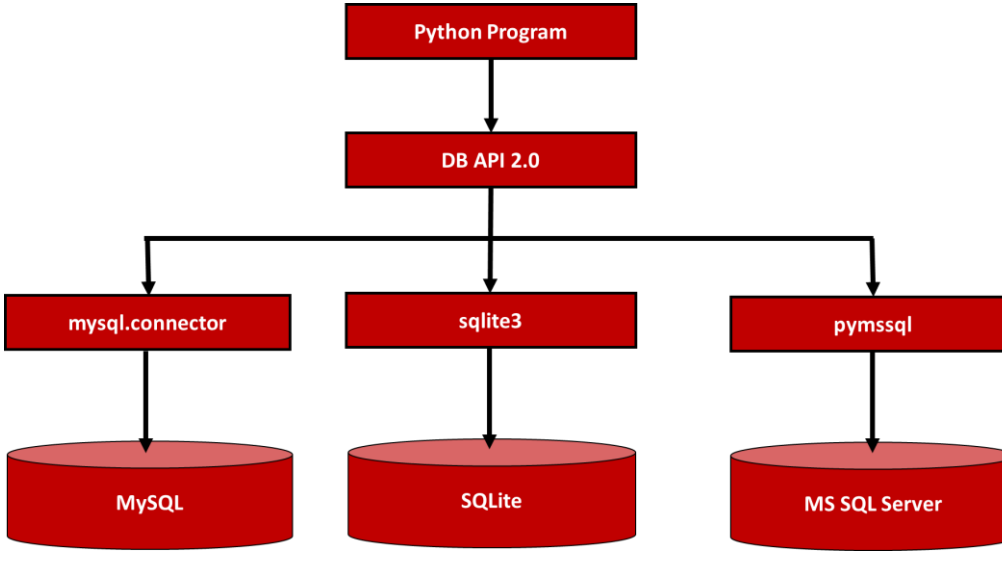
Methods find() and find_all()

Parameter for these methods can be any of the following:

```
# A simple string
soup.find_all('b')

# A regular expression
soup.find_all(re.compile("^b"))

# Look for any value in list
soup.find_all(["a", "b"])

# Looking for id with link2
soup.find_all(id='link2')

# Looking for CSS class bright
soup.find_all("a", class_="bright")
```

DATABASE PROGRAMMING

- Python supports different databases.
- Python Database API Specification has been defined to provide similarity between modules to access different databases.
- Database API is known as Python DB-API 2.0 with PEP 249 at https://www.python.org/dev/peps/pep-0249
- It enables code that is generally more portable across databases as all database modules provide the same API.
- Modules required to access database are to be downloaded.
- Module sqlite3, which is used to access SQLite database, is provided along with Python.



SQLite3 Database

- SQLite is a C library that provides a lightweight disk-based database that doesn't require a separate server process and allows accessing the database using a nonstandard variant of the SQL query language.
- It is possible to prototype an application using SQLite and then port the code to a larger database such as Oracle.
- Python ships with SQLite Database.

Data types in SQLite3

The following are available data types in SQLite Database.

Datatype	Meaning	Python Type
NULL	The value is a NULL value.	None
INTEGER	The value is a signed integer, stored in 1, 2, 3, 4, 6, or 8 bytes depending on the magnitude of the value.	int
REAL	The value is a floating-point value, stored as an 8-byte IEEE floating point number.	float
TEXT	The value is a text string, stored using the database encoding (UTF-8, UTF-16BE or UTF-16LE).	str
BLOB	The value is a blob of data, stored exactly as it was input.	bytes

MODULE SQLITE3

- It is the interface for SQLite database.
- This module is part of Python standard library.
- It implements DB API 2.0 specifications (PEP 249).

Method connect()

- It is used to establish a connection to database with given parameters.
- Parameter is name of the database to connect to. If database is not present, it is created.
- It returns Connection object.

```
connect(databasename)
```

Connection object

Connection object represents a connection to database.

Method	Meaning
close()	Closes connection.
commit()	Commits pending changes in transaction to database.
rollback()	Causes the database to roll back to the start of any pending transaction. Closing a connection without committing the changes first will cause an implicit rollback to be performed.
cursor()	Returns a cursor object using this connection.

Cursor Object

- Cursor represents a database cursor, which is used to manage the context of a fetch operation.
- Cursors created from the same connection are not isolated, i.e., any changes done to the database by a cursor are immediately visible to other cursors.

Method	Meaning
close()	Closes cursor.
execute(operation [, parameters])	Prepare and execute a database operation.
Executemany (*operation*, *seq_of_parameters*)	Prepare a database operation (query or command) and then execute it against all parameter sequences or mappings found in the sequence *seq_of_parameters*.
fetchone()	Fetch the next row of a query result set, returning a single sequence, or None when no more data is available.
fetchmany ([size=cursor.arraysize])	Fetch the next set of rows of a query result, returning a sequence of sequences (e.g. a list of tuples). An empty sequence is returned when no more rows are available.
fetchall()	Fetch all (remaining) rows of a query result, returning them as a sequence of sequences (e.g. a list of tuples). Note that the cursor's arraysize attribute can affect the performance of this operation.

Attribute	Meaning
rowcount	This read-only attribute specifies the number of rows that the last execute() method retrieved or affected.
lastrowid	Read-only attribute provides the rowid of the last modified row.
arraysize	Read/write attribute that controls the number of rows returned by fetchmany(). The default value is 1, which means a single row would be fetched per call.
connection	This read-only attribute provides the SQLite database Connection used by the Cursor object.

NOTE: Use SQLite Studio, which is a free GUI tool, to manage SQLite database. Download it from https://sqlitestudio.pl/index.rvt

The following program shows how to connect to a database and close connection.

```
01 import sqlite3
02 con = sqlite3.connect("test.db")
03 print("Connected to test database!")
04 con.close()
```

Create the following table in **test.db** database:

```
create table expenses
(
  id integer primary key autoincrement,
  description text (30),
  amount real
)
```

Inserting row into table

The following program shows how to insert a row into EXPENSES table by taking data from user.

```
01 import sqlite3
02 con = sqlite3.connect("test.db")
03 cur = con.cursor()
04
05 # insert a row into EXPENSES table
06 try:
07    # take data from user
08    des = input("Enter Description :")
09    amt = input("Enter Amount      :")
10    row = (des, amt)
11    cur.execute(
12    "insert into expenses(description,amount)values(?,?)"
13    , row)
14    con.commit()
15    print("Added successfully!")
16 except Exception as ex:
17    print("Sorry! Error: ", ex)
18 finally:
19    con.close()
```

Retrieving rows from table

The following program shows how to list all rows from EXPENSES table.

```
01 import sqlite3
02
03 con = sqlite3.connect("test.db")
04 cur = con.cursor()
05
06 # List rows from EXPENSES table
07 try:
08     cur.execute("select * from expenses order by id")
09     for row in cur.fetchall():
10        print(f"{row[0]:3d} {row[1]:30s} {row[2]:10.2f}")
11
12     cur.close()
13 except Exception as ex:
14     print("Error : ", ex)
15 finally:
16     con.close()
```

Updating row in table

The following program updates an existing row in EXPENSES table.

```
01 import sqlite3
02
03 con = sqlite3.connect(r"c:\dev\python\test.db")
04 cur = con.cursor()
05
06 # Update EXPENSES table
07 try:
08     # take data from user
09     id = input("Enter Id     :")
10     amount = input("Enter Amount :")
11     cur.execute
12      ("update expenses set amount=? where id = ?",
13      (amount, id))
14     if cur.rowcount == 1:
15         con.commit()
16         print("Updated successfully!")
17     else:
18         print('Sorry! Id not found!')
19 except Exception as ex:
20     print("Sorry! Error: ", ex)
21 finally:
22     con.close()
```

Deleting row from table

The following program deletes an existing row in EXPENSES table.

```
01 import sqlite3
02
03 con = sqlite3.connect("test.db")
04 cur = con.cursor()
05
06 # Delete row from EXPENSES table
07 try:
08     # take data from user
09     id = input("Enter Id     :")
10     cur.execute
11          ("delete from expenses where id = ?", (id, ))
12     if cur.rowcount == 1:
13        con.commit()
14        print("Deleted successfully!")
15     else:
16         print('Sorry! Id not found!')
17 except Exception as ex:
18     print("Sorry! Error: ", ex)
19 finally:
20     con.close()
```

WORKING WITH OTHER DATABASES

Here are my blogs about how to access other database systems from Python.

Access Oracle from Python

http://www.srikanthtechnologies.com/blog/python/using_cx_oracle.aspx

Access MySQL from Python

http://www.srikanthtechnologies.com/blog/python/using_mysql.aspx