Contents

Datatypes

List

The list is the most versatile datatype available in Python, which can be written as a list of comma-separated values (items) between square brackets. Important thing about a list is that the items in a list need not be of the same type.

Lists

S.No.	Methods & Description
1	**list.append(obj)** Appends object obj to list
2	**list.count(obj)** Returns count of how many times obj occurs in list
3	**list.extend(seq)** Appends the contents of seq to list
4	**list.index(obj)** Returns the lowest index in list that obj appears
5	**list.insert(index, obj)** Inserts object obj into list at offset index
6	**list.pop(obj = list[-1])** Removes and returns last object or obj from list
7	**list.remove(obj)** Removes object obj from list
8	**list.reverse()** Reverses objects of list in place
9	**list.sort([func])** Sorts objects of list, use compare func if given

The del statement

There is a way to remove an item from a list given its index instead of its value: the del statement. This differs from the pop() method which returns a value. The del statement can also be used to remove slices from a list or clear the entire list (which we did earlier by assignment of an empty list to the slice). For example:

CODE:

```
>>> a = [-1, 1, 66.25, 333, 333, 1234.5]
>>> del a[0]
>>> a


>>> del a[2:4]
>>> a

>>> del a[:]
>>> a


>>> del a

>>>print(a)
```

Tuple

Advantages of Tuple over List

Since, tuples are quite similar to lists, both of them are used in similar situations as well.

However, there are certain advantages of implementing a tuple over a list. Below listed are some of the main advantages:

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

1. Indexing

We can use the index operator [] to access an item in a tuple where the index starts from 0. So, a tuple having 6 elements will have index from 0 to 5. Trying to access an element other that (6, 7,...) will raise an IndexError. The index must be an integer, so we cannot use float or other types. This will result into TypeError.

Code::

```
>>>my_tuple = ('p','e','r','m','i','t')


>>>print(my_tuple[0])


>>>print(my_tuple[5])


# nested tuple
>>>n_tuple = ("mouse", [8, 4, 6], (1, 2, 3))

>>>print(n_tuple[0][3])


>>>print(n_tuple[1][1])
```

Negative Indexing

Python allows negative indexing for its sequences.

The index of -1 refers to the last item, -2 to the second last item and so on.

```
>>>my_tuple = ('p','e','r','m','i','t')
>>>print(my_tuple[-1])
```

Slicing

We can access a range of items in a tuple by using the slicing operator - colon ":".

```
>>>my_tuple = ('p','r','o','g','r','a','m','i','z')

>>>print(my_tuple[1:4])


>>>print(my_tuple[:-7])


>>>print(my_tuple[7:])


>>>print(my_tuple[:])


>>>my_tuple = (4, 2, 3, [6, 5])

# we cannot change an element
# you will get an error:
# TypeError: 'tuple' object does not support item assignment

>>>my_tuple[1] = 9

# but item of mutable element can be changed
>>>my_tuple[3][0] = 9


# tuples can be reassigned
>>>my_tuple = ('p','r','o','g','r','a','m','i','z')
>>>print(my_tuple)
```

We can use + operator to combine two tuples. This is also called **concatenation**.
We can also **repeat** the elements in a tuple for a given number of times using the * operator.
Both + and * operations result into a new tuple.

```
>>>print((1, 2, 3) + (4, 5, 6))


>>>print(("Repeat",) * 3)
```

Deleting a Tuple

As discussed above, we cannot change the elements in a tuple. That also means we cannot delete or remove items from a tuple.But deleting a tuple entirely is possible using the keyword del.

```
>>>my_tuple = ('p','r','o','g','r','a','m','i','z')


>>>del my_tuple[3]

>>>del my_tuple
>>>my_tuple
```

Function	Description
all()	Return True if all elements of the tuple are true (or if the tuple is empty).
any()	Return True if any element of the tuple is true. If the tuple is empty, return False.
enumerate()	Return an enumerate object. It contains the index and value of all the items of tuple as pairs.
len()	Return the length (the number of items) in the tuple.
max()	Return the largest item in the tuple.
min()	Return the smallest item in the tuple
sorted()	Take elements in the tuple and return a new sorted list (does not sort the tuple itself).
sum()	Retrun the sum of all elements in the tuple.
tuple()	Convert an iterable (list, string, set, dictionary) to a tuple.

Dictionary

Each key is separated from its value by a colon (:), the items are separated by commas, and the whole thing is enclosed in curly braces. An empty dictionary without any items is written with just two curly braces, like this: {}.
Keys are unique within a dictionary while values may not be. The values of a dictionary can be of any type, but the keys must be of an immutable data type such as strings, numbers, or tuples.

Accessing Values in Dictionary

To access dictionary elements, you can use the familiar square brackets along with the key to obtain its value. Following is a simple example −

```
>>>dict = {'Name': 'Zara', 'Age': 7, 'Class': 'First'}
>>>print ("dict['Name']: ", dict['Name'])
>>>print ("dict['Age']: ", dict['Age'])
```

If we attempt to access a data item with a key, which is not a part of the dictionary, we get an error as follows −

```
>>>dict = {'Name': 'Zara', 'Age': 7, 'Class': 'First'};
>>>print "dict['Alice']: ", dict['Alice']
```

Updating Dictionary

You can update a dictionary by adding a new entry or a key-value pair, modifying an existing entry, or deleting an existing entry as shown in a simple example given below.

```
dict = {'Name': 'Zara', 'Age': 7, 'Class': 'First'}
dict['Age'] = 8; # update existing entry
dict['School'] = "DPS School" # Add new entry

print ("dict['Age']: ", dict['Age'])
print ("dict['School']: ", dict['School'])
```

When the above code is executed, it produces the following result −

```
dict['Age']:  8
dict['School']:  DPS School
```

Delete Dictionary Elements

You can either remove individual dictionary elements or clear the entire contents of a dictionary. You can also delete entire dictionary in a single operation.To explicitly remove an entire dictionary, just use the **del** statement. Following is a simple example

```
>>>dict = {'Name': 'Zara', 'Age': 7, 'Class': 'First'}

>>>del dict['Name'] # remove entry with key 'Name'
```

```
>>>dict.clear()     # remove all entries in dict
>>>del dict         # delete entire dictionary

>>>print ("dict['Age']: ", dict['Age'])


>>>print ("dict['School']: ", dict['School'])
```

Properties of Dictionary Keys

Dictionary values have no restrictions. They can be any arbitrary Python object, either standard objects or user-defined objects. However, same is not true for the keys.
There are two important points to remember about dictionary keys −
(a) More than one entry per key is not allowed. This means no duplicate key is allowed. When duplicate keys are encountered during assignment, the last assignment wins. For example −

```
>>>dict = {'Name': 'Zara', 'Age': 7, 'Name': 'Manni'}
>>>print ("dict['Name']: ", dict['Name'])
```

(b) Keys must be immutable. This means you can use strings, numbers or tuples as dictionary keys but something like ['key'] is not allowed. Following is a simple example

```
>>>dict = {['Name']: 'Zara', 'Age': 7}
print ("dict['Name']: ", dict['Name'])
```

Strings

Strings are amongst the most popular types in Python. We can create them simply by enclosing characters in quotes. Python treats single quotes the same as double quotes.

String Special Operators

Assume string variable **a** holds 'Hello' and variable **b** holds 'Python', then −

Operator	Description	Example
+	Concatenation - Adds values on either side of the operator	a + b will give HelloPython
*	Repetition - Creates new strings, concatenating multiple copies of the same string	a*2 will give -HelloHello
[]	Slice - Gives the character from the given index	a[1] will give e
[:]	Range Slice - Gives the characters from the given range	a[1:4] will give ell
in	Membership - Returns true if a character exists in the given string	H in a will give 1
not in	Membership - Returns true if a character does not exist in the given string	M not in a will give 1
r/R	Raw String - Suppresses actual meaning of Escape characters. The syntax for raw strings is exactly the same as for normal strings with the exception of the raw string operator, the letter "r," which precedes the quotation marks. The "r" can be lowercase (r) or uppercase (R) and must be placed immediately preceding the first quote mark.	print r'\n' prints \n and print R'\n'prints \n
%	Format - Performs String formatting	See at next section

String Formatting Operator

One of Python's coolest features is the string format operator %. This operator is unique to strings and makes up for the pack of having functions from C's printf() family. Following is a simple example −

```
>>>print ("My name is %s and weight is %d kg!" % ('Zara', 21))
```

S.No.	Format Symbol & Conversion
1	**%c** character
2	**%s** string conversion via str() prior to formatting
3	**%i** signed decimal integer
4	**%d** signed decimal integer
5	**%u** unsigned decimal integer
6	**%o** octal integer
7	**%x** hexadecimal integer (lowercase letters)
8	**%X** hexadecimal integer (UPPERcase letters)
9	**%e** exponential notation (with lowercase 'e')
10	**%E** exponential notation (with UPPERcase 'E')
11	**%f** floating point real number
12	**%g** the shorter of %f and %e
13	**%G** the shorter of %f and %E

Other supported symbols and functionality are listed in the following table −

S.No.	Symbol & Functionality
1	***** argument specifies width or precision
2	**-** left justification
3	**+** display the sign
4	**<sp>** leave a blank space before a positive number
5	**#** add the octal leading zero ('0') or hexadecimal leading '0x' or '0X', depending on whether 'x' or 'X' were used.
6	**0** pad from left with zeros (instead of spaces)
7	**%** '%%' leaves you with a single literal '%'
8	**(var)** mapping variable (dictionary arguments)
9	**m.n.** m is the minimum total width and n is the number of digits to display after the decimal point (if appl.)

Escape Characters

Following table is a list of escape or non-printable characters that can be represented with backslash notation.An escape character gets interpreted; in a single quoted as well as double quoted strings.

Backslash notation	Hexadecimal character	Description
\a	0x07	Bell or alert
\b	0x08	Backspace
\cx		Control-x
\C-x		Control-x
\e	0x1b	Escape
\f	0x0c	Formfeed
\M-\C-x		Meta-Control-x
\n	0x0a	Newline
\nnn		Octal notation, where n is in the range 0.7
\r	0x0d	Carriage return
\s	0x20	Space
\t	0x09	Tab
\v	0x0b	Vertical tab
\x		Character x
\xnn		Hexadecimal notation, where n is in the range 0.9, a.f, or A.F

String methods

S.No.	Methods & Description
1	**capitalize()** Capitalizes first letter of string
2	**center(width, fillchar)** Returns a string padded with *fillchar* with the original string centered to a total of *width* columns.
3	**count(str, beg = 0,end = len(string))** Counts how many times str occurs in string or in a substring of string if starting index beg and ending index end are given.
4	**decode(encoding = 'UTF-8',errors = 'strict')** Decodes the string using the codec registered for encoding. encoding defaults to the default string encoding.
5	**encode(encoding = 'UTF-8',errors = 'strict')** Returns encoded string version of string; on error, default is to raise a ValueError unless errors is given with 'ignore' or 'replace'.
6	**endswith(suffix, beg = 0, end = len(string))** Determines if string or a substring of string (if starting index beg and ending index end are given) ends with suffix; returns true if so and false otherwise.
7	**expandtabs(tabsize = 8)** Expands tabs in string to multiple spaces; defaults to 8 spaces per tab if tabsize not provided.
8	**find(str, beg = 0 end = len(string))** Determine if str occurs in string or in a substring of string if starting index beg and ending index end are given returns index if found and -1 otherwise.
9	**index(str, beg = 0, end = len(string))** Same as find(), but raises an exception if str not found.
10	**isalnum()** Returns true if string has at least 1 character and all characters are alphanumeric and false otherwise.
11	**isalpha()**

	Returns true if string has at least 1 character and all characters are alphabetic and false otherwise.
12	**isdigit()** Returns true if string contains only digits and false otherwise.
13	**islower()** Returns true if string has at least 1 cased character and all cased characters are in lowercase and false otherwise.
14	**isnumeric()** Returns true if a unicode string contains only numeric characters and false otherwise.
15	**isspace()** Returns true if string contains only whitespace characters and false otherwise.
16	**istitle()** Returns true if string is properly "titlecased" and false otherwise.
17	**isupper()** Returns true if string has at least one cased character and all cased characters are in uppercase and false otherwise.
18	**join(seq)** Merges (concatenates) the string representations of elements in sequence seq into a string, with separator string.
19	**len(string)** Returns the length of the string
20	**ljust(width[, fillchar])** Returns a space-padded string with the original string left-justified to a total of width columns.
21	**lower()** Converts all uppercase letters in string to lowercase.
22	**lstrip()** Removes all leading whitespace in string.
23	**maketrans()** Returns a translation table to be used in translate function.
24	**max(str)** Returns the max alphabetical character from the string str.

25	**min(str)** Returns the min alphabetical character from the string str.
26	**replace(old, new [, max])** Replaces all occurrences of old in string with new or at most max occurrences if max given.
27	**rfind(str, beg = 0,end = len(string))** Same as find(), but search backwards in string.
28	**rindex(str, beg = 0, end = len(string))** Same as index(), but search backwards in string.
29	**rjust(width,[, fillchar])** Returns a space-padded string with the original string right-justified to a total of width columns.
30	**rstrip()** Removes all trailing whitespace of string.
31	**split(str="", num=string.count(str))** Splits string according to delimiter str (space if not provided) and returns list of substrings; split into at most num substrings if given.
32	**splitlines(num=string.count('\n'))** Splits string at all (or num) NEWLINEs and returns a list of each line with NEWLINEs removed.
33	**startswith(str, beg=0,end=len(string))** Determines if string or a substring of string (if starting index beg and ending index end are given) starts with substring str; returns true if so and false otherwise.
34	**strip([chars])** Performs both lstrip() and rstrip() on string

35	**swapcase()** Inverts case for all letters in string.
36	**title()** Returns "titlecased" version of string, that is, all words begin with uppercase and the rest are lowercase.
37	**translate(table, deletechars="")** Translates string according to translation table str(256 chars), removing those in the del string.
38	**upper()** Converts lowercase letters in string to uppercase.
39	**zfill (width)** Returns original string leftpadded with zeros to a total of width characters; intended for numbers, zfill() retains any sign given (less one zero).
40	**isdecimal()** Returns true if a unicode string contains only decimal characters and false otherwise.

Standard Exceptions

Here is a list of Standard Exceptions available in Python. −

S.No.	Exception Name & Description
1	**Exception** Base class for all exceptions
2	**StopIteration** Raised when the next() method of an iterator does not point to any object.
3	**SystemExit** Raised by the sys.exit() function.
4	**StandardError** Base class for all built-in exceptions except StopIteration and SystemExit.
5	**ArithmeticError** Base class for all errors that occur for numeric calculation.
6	**OverflowError** Raised when a calculation exceeds maximum limit for a numeric type.
7	**FloatingPointError** Raised when a floating point calculation fails.
8	**ZeroDivisonError** Raised when division or modulo by zero takes place for all numeric types.
9	**AssertionError** Raised in case of failure of the Assert statement.
10	**AttributeError** Raised in case of failure of attribute reference or assignment.
11	**EOFError** Raised when there is no input from either the raw_input() or input() function and the end of file is reached.
12	**ImportError** Raised when an import statement fails.
13	**KeyboardInterrupt** Raised when the user interrupts program execution, usually by pressing Ctrl+c.

14	**LookupError** Base class for all lookup errors.
15	**IndexError** Raised when an index is not found in a sequence.
16	**KeyError** Raised when the specified key is not found in the dictionary.
17	**NameError** Raised when an identifier is not found in the local or global namespace.
18	**UnboundLocalError** Raised when trying to access a local variable in a function or method but no value has been assigned to it.
19	**EnvironmentError** Base class for all exceptions that occur outside the Python environment.
20	**IOError** Raised when an input/ output operation fails, such as the print statement or the open() function when trying to open a file that does not exist.
21	**OSError** Raised for operating system-related errors.
22	**SyntaxError** Raised when there is an error in Python syntax.
23	**IndentationError** Raised when indentation is not specified properly.
24	**SystemError** Raised when the interpreter finds an internal problem, but when this error is encountered the Python interpreter does not exit.
25	**SystemExit** Raised when Python interpreter is quit by using the sys.exit() function. If not handled in the code, causes the interpreter to exit.
26	**TypeError** Raised when an operation or function is attempted that is invalid for the specified data type.
27	**ValueError** Raised when the built-in function for a data type has the valid type of arguments, but the arguments have invalid values specified.

28	**RuntimeError** Raised when a generated error does not fall into any category.
29	**NotImplementedError** Raised when an abstract method that needs to be implemented in an inherited class is not actually implemented.

FILE METHODS

1	**file.close()** Close the file. A closed file cannot be read or written any more.
2	**file.flush()** Flush the internal buffer, like stdio's fflush. This may be a no-op on some file-like objects.
3	**file.fileno()** Returns the integer file descriptor that is used by the underlying implementation to request I/O operations from the operating system.
4	**file.isatty()** Returns True if the file is connected to a tty(-like) device, else False.
5	**next(file)** Returns the next line from the file each time it is being called.
6	**file.read([size])** Reads at most size bytes from the file (less if the read hits EOF before obtaining size bytes).
7	**file.readline([size])** Reads one entire line from the file. A trailing newline character is kept in the string.
8	**file.readlines([sizehint])** Reads until EOF using readline() and return a list containing the lines. If the optional sizehint argument is present, instead of reading up to EOF, whole lines totalling approximately sizehint bytes (possibly after rounding up to an internal buffer size) are read.
9	**file.seek(offset[, whence])** Sets the file's current position
10	**file.tell()** Returns the file's current position
11	**file.truncate([size])** Truncates the file's size. If the optional size argument is present, the file is truncated to (at most) that size.

12	**file.write(str)** Writes a string to the file. There is no return value.
13	**file.writelines(sequence)** Writes a sequence of strings to the file. The sequence can be any iterable object producing strings, typically a list of strings.

OS METHODS

S.No.	Methods & Description
1	**os.access(path, mode)** Use the real uid/gid to test for access to path.
2	**os.chdir(path)** Change the current working directory to path.
3	**os.chflags(path, flags)** Set the flags of path to the numeric flags.
4	**os.chmod(path, mode)** Change the mode of path to the numeric mode.
5	**os.chown(path, uid, gid)** Change the owner and group id of path to the numeric uid and gid.
6	**os.chroot(path)** Change the root directory of the current process to path.
7	**os.close(fd)** Close file descriptor fd.
8	**os.closerange(fd_low, fd_high)** Close all file descriptors from fd_low (inclusive) to fd_high (exclusive), ignoring errors.
9	**os.dup(fd)** Return a duplicate of file descriptor fd.
10	**os.dup2(fd, fd2)** Duplicate file descriptor fd to fd2, closing the latter first if necessary.
11	**os.fchdir(fd)**

	Change the current working directory to the directory represented by the file descriptor fd.
12	**os.fchmod(fd, mode)** Change the mode of the file given by fd to the numeric mode.
13	**os.fchown(fd, uid, gid)** Change the owner and group id of the file given by fd to the numeric uid and gid.
14	**os.fdatasync(fd)** Force write of file with filedescriptor fd to disk.
15	**os.fdopen(fd[, mode[, bufsize]])** Return an open file object connected to the file descriptor fd.
16	**os.fpathconf(fd, name)** Return system configuration information relevant to an open file. name specifies the configuration value to retrieve.
17	**os.fstat(fd)** Return status for file descriptor fd, like stat().
18	**os.fstatvfs(fd)** Return information about the filesystem containing the file associated with file descriptor fd, like statvfs().
19	**os.fsync(fd)** Force write of file with filedescriptor fd to disk.
20	**os.ftruncate(fd, length)** Truncate the file corresponding to file descriptor fd, so that it is at most length bytes in size.
21	**os.getcwd()** Return a string representing the current working directory.
22	**os.getcwdu()** Return a Unicode object representing the current working directory.
23	**os.isatty(fd)** Return True if the file descriptor fd is open and connected to a tty(-like) device, else False.
24	**os.lchflags(path, flags)** Set the flags of path to the numeric flags, like chflags(), but do not follow symbolic links.

25	**os.lchmod(path, mode)** Change the mode of path to the numeric mode.
26	**os.lchown(path, uid, gid)** Change the owner and group id of path to the numeric uid and gid. This function will not follow symbolic links.
27	**os.link(src, dst)** Create a hard link pointing to src named dst.
28	**os.listdir(path)** Return a list containing the names of the entries in the directory given by path.
29	**os.lseek(fd, pos, how)** Set the current position of file descriptor fd to position pos, modified by how.
30	**os.lstat(path)** Like stat(), but do not follow symbolic links.
31	**os.major(device)** Extract the device major number from a raw device number.
32	**os.makedev(major, minor)** Compose a raw device number from the major and minor device numbers.
33	**os.makedirs(path[, mode])** Recursive directory creation function.
34	**os.minor(device)** Extract the device minor number from a raw device number.
35	**os.mkdir(path[, mode])** Create a directory named path with numeric mode mode.
36	**os.mkfifo(path[, mode])** Create a FIFO (a named pipe) named path with numeric mode mode. The default mode is 0666 (octal).
37	**os.mknod(filename[, mode = 0600, device])** Create a filesystem node (file, device special file or named pipe) named filename.
38	**os.open(file, flags[, mode])** Open the file file and set various flags according to flags and possibly its mode according to mode.

39	**os.openpty()** Open a new pseudo-terminal pair. Return a pair of file descriptors (master, slave) for the pty and the tty, respectively.
40	**os.pathconf(path, name)** Return system configuration information relevant to a named file.
41	**os.pipe()** Create a pipe. Return a pair of file descriptors (r, w) usable for reading and writing, respectively.
42	**os.popen(command[, mode[, bufsize]])** Open a pipe to or from command.
43	**os.read(fd, n)** Read at most n bytes from file descriptor fd. Return a string containing the bytes read. If the end of the file referred to by fd has been reached, an empty string is returned.
44	**os.readlink(path)** Return a string representing the path to which the symbolic link points.
45	**os.remove(path)** Remove the file path.
46	**os.removedirs(path)** Remove directories recursively.
47	**os.rename(src, dst)** Rename the file or directory src to dst.
48	**os.renames(old, new)** Recursive directory or file renaming function.
49	**os.rmdir(path)** Remove the directory path
50	**os.stat(path)** Perform a stat system call on the given path.
51	**os.stat_float_times([newvalue])** Determine whether stat_result represents time stamps as float objects.
52	**os.statvfs(path)** Perform a statvfs system call on the given path.
53	**os.symlink(src, dst)**

	Create a symbolic link pointing to src named dst.
54	**os.tcgetpgrp(fd)** Return the process group associated with the terminal given by fd (an open file descriptor as returned by open()).
55	**os.tcsetpgrp(fd, pg)** Set the process group associated with the terminal given by fd (an open file descriptor as returned by open()) to pg.
56	**os.tempnam([dir[, prefix]])** Return a unique path name that is reasonable for creating a temporary file.
57	**os.tmpfile()** Return a new file object opened in update mode (w+b).
58	**os.tmpnam()** Return a unique path name that is reasonable for creating a temporary file.
59	**os.ttyname(fd)** Return a string which specifies the terminal device associated with file descriptor fd. If fd is not associated with a terminal device, an exception is raised.
60	**os.unlink(path)** Remove the file path.
61	**os.utime(path, times)** Set the access and modified times of the file specified by path.
62	**os.walk(top[, topdown = True[, onerror = None[, followlinks = False]]])** Generate the file names in a directory tree by walking the tree either top-down or bottom-up.
63	**os.write(fd, str)** Write the string str to file descriptor fd. Return the number of bytes actually written.

DATABASE HANDLING

Python Database API supports a wide range of database servers such as GadFly

- mSQL
- MySQL
- PostgreSQL
- Microsoft SQL Server 2000
- Informix
- Interbase
- Oracle
- Sybase

programming interface.

Benefits of Python for database programming

There are many good reasons to use Python for programming database applications:

- Programming in Python is arguably more efficient and faster compared to other languages.
- Python is famous for its portability.
- It is platform independent.
- Python supports SQL cursors.
- In many programming languages, the application developer needs to take care of the open and closed connections of the database, to avoid further exceptions and errors. In Python, these connections are taken care of.
- Python supports relational database systems.
- Python database APIs are compatible with various databases, so it is very easy to migrate and port database application interfaces.

OPTIONS FOR CONNECTIVITY

MySQLdb?

MySQLdb is an interface for connecting to a MySQL database server from Python. It implements the Python Database API v2.0 and is built on top of the MySQL C API. Old style.

New is : MySQL Connector/Python

MySQL Connector/Python enables Python programs to access MySQL databases, using an API that is compliant with thePython Database API Specification v2.0 (PEP 249). It is written in pure Python and does not have any dependencies except for the Python Standard Library.

MySQL Connector/Python includes support for:

- Almost all features provided by MySQL Server up to and including MySQL Server version 5.7.Connector/Python 8.0 also supports X DevAPI.
- Converting parameter values back and forth between Python and MySQL data types, for example Python datetime and MySQL DATETIME. You can turn automatic conversion on for convenience, or off for optimal performance.
- All MySQL extensions to standard SQL syntax.
- Protocol compression, which enables compressing the data stream between the client and server.
- Connections using TCP/IP sockets and on Unix using Unix sockets.
- Secure TCP/IP connections using SSL.
- Self-contained driver. Connector/Python does not require the MySQL client library or any Python modules outside the standard library.

Connecting to MySQL Using Connector/Python

The connect() constructor creates a connection to the MySQL server and returns a MySQLConnection object.
The following example shows how to connect to the MySQL server:

```
 import mysql.connector
cnx = mysql.connector.connect(user='scott', password='password',
                              host='127.0.0.1',
                              database='employees')
cnx.close()
```

To handle connection errors, use the try statement and catch all errors using the errors.Error exception:

```
import mysql.connector
from mysql.connector import errorcode

try:
  cnx = mysql.connector.connect(user='scott',
                                database='testt')
except mysql.connector.Error as err:
  if err.errno == errorcode.ER_ACCESS_DENIED_ERROR:
    print("Something is wrong with your user name or password")
  elif err.errno == errorcode.ER_BAD_DB_ERROR:
    print("Database does not exist")
  else:
    print(err)
else:
  cnx.close()
```

Performing Transactions

Transactions are a mechanism that ensures data consistency.
Transactions have the following four properties –

- **Atomicity** – Either a transaction completes or nothing happens at all.
- **Consistency** – A transaction must start in a consistent state and leave the system in a consistent state.
- **Isolation** – Intermediate results of a transaction are not visible outside the current transaction.
- **Durability** – Once a transaction was committed, the effects are persistent, even after a system failure.

The Python DB API 2.0 provides two methods to either *commit* or *rollback* a transaction.

READ Operation

READ Operation on any database means to fetch some useful information from the database.

Once our database connection is established, you are ready to make a query into this database. You can use either **fetchone()** method to fetch single record or **fetchall()** method to fetech multiple values from a database table.

- **fetchone()** – It fetches the next row of a query result set. A result set is an object that is returned when a cursor object is used to query a table.
- **fetchall()** – It fetches all the rows in a result set. If some rows have already been extracted from the result set, then it retrieves the remaining rows from the result set.

- **rowcount** – This is a read-only attribute and returns the number of rows that were affected by an execute() method.

Creating Database Table

Once a database connection is established, we are ready to create tables or records into the database tables using **execute** method of the created cursor.

```
import mysql.connector
from mysql.connector import errorcode

db = mysql.connector.connect(user='root',
password='',host='localhost',database='medical')

# prepare a cursor object using cursor() method
cursor = db.cursor()

# Drop table if it already exist using execute() method.
cursor.execute("DROP TABLE IF EXISTS EMPLOYEE")

# Create table as per requirement
sql = """CREATE TABLE EMPLOYEE (
        FIRST_NAME  CHAR(20) NOT NULL,
        LAST_NAME  CHAR(20),
        AGE INT,
        SEX CHAR(1),
        INCOME FLOAT )"""

cursor.execute(sql)

# disconnect from server
db.close()
```

INSERT Operation

It is required when you want to create your records into a database table.

```
import mysql.connector
from mysql.connector import errorcode

db = mysql.connector.connect(user='root',
password='',host='localhost',database='medical')

# prepare a cursor object using cursor() method
cursor = db.cursor()

sql = """INSERT INTO EMPLOYEE(FIRST_NAME,
         LAST_NAME, AGE, SEX, INCOME)
         VALUES ('Mac', 'Mohan', 20, 'M', 2000)"""
try:
   # Execute the SQL command
   cursor.execute(sql)
   # Commit your changes in the database
   db.commit()
except:
   # Rollback in case there is any error
   db.rollback()

# disconnect from server
db.close()
```

Update

```
import mysql.connector
from mysql.connector import errorcode
try:
  db = mysql.connector.connect(user='root',
password='',host='localhost',database='medical')
#prepare a cursor object using cursor() method
  cursor = db.cursor()

#Prepare SQL query to UPDATE required records
  sql="UPDATE EMPLOYEE SET AGE = AGE + 1 WHERE SEX ='M'"

   # Execute the SQL command
  cursor.execute(sql)
   # Commit your changes in the database
  db.commit()
except:
   # Rollback in case there is any error
   db.rollback()

# disconnect from server
db.close()
```

Deletion

```
import mysql.connector
from mysql.connector import errorcode
```

```
try:
  db = mysql.connector.connect(user='root',
password='',host='localhost',database='medical')
# prepare a cursor object using cursor() method
  cursor = db.cursor()

# Prepare SQL query to UPDATE required records
  sql = "DELETE FROM EMPLOYEE WHERE AGE >= 30"

# Execute the SQL command
  cursor.execute(sql)
   # Commit your changes in the database
  db.commit()
except:
   # Rollback in case there is any error
  db.rollback()

# disconnect from server
db.close()
```

Handling Errors

There are many sources of errors. A few examples are a syntax error in an executed SQL statement, a connection failure, or calling the fetch method for an already canceled or finished statement handle.

The DB API defines a number of errors that must exist in each database module. The following table lists these exceptions.

Sr.No.	Exception & Description
1	**Warning** Used for non-fatal issues. Must subclass StandardError.
2	**Error** Base class for errors. Must subclass StandardError.
3	**InterfaceError** Used for errors in the database module, not the database itself. Must subclass Error.
4	**DatabaseError** Used for errors in the database. Must subclass Error.
5	**DataError** Subclass of DatabaseError that refers to errors in the data.
6	**OperationalError** Subclass of DatabaseError that refers to errors such as the loss of a connection to the database. These errors are generally outside of the control of the Python scripter.
7	**IntegrityError** Subclass of DatabaseError for situations that would damage the relational integrity, such as uniqueness constraints or foreign keys.
8	**InternalError** Subclass of DatabaseError that refers to errors internal to the database module, such as a cursor no longer being active.
9	**ProgrammingError**

	Subclass of DatabaseError that refers to errors such as a bad table name and other things that can safely be blamed on you.
10	**NotSupportedError** Subclass of DatabaseError that refers to trying to call unsupported functionality.