

Customised for **Collins Aerospace**

Python Programming



June 10, 2019

Trainer: SHOBHIT NIGAM

Email: Mr.Nigam@gmail.com; Ph: 9949999239

This document, codes and exercises discussed in the sessions can also be downloaded from GitHub: https://github.com/shobhit-nigam/training/tree/master/collins/10June19

Table of Contents

1. Fun with Python 4

Comments: 4

Numbers 5

variables 6

Multiple Assignment 7

Strings 7

string index 10

slicing 11

List 13

Nesting of lists 15

Dynamic data types 16

object and id() 16

conversion functions 17

del 17

None 18

Bool and Logical Operator 18

Comparison operators 19

Logical operators 19

Membership operators 19

Identity operators 20

Bitwise operators 20

2. Flow control 21

while 21

if 22

for 22

range 23

break, continue, else 25

pass 27

3. Functions 28

function nesting 30

symbol table 31

Default arguments 32

Documentation strings 33

4. Data Structures 34

Lists, extended 34

List Comprehensions 35

Tuple 36

Sets 37

Dictionary 39

5. Modules 41

Custom Modules 42

6. Object Oriented Programming

\_\_init\_\_() 46

Inheritance 47

private data 48

operator overloading 50

Python magic methods 52

7. Exception Handling 54

try 55

Multiple Except blocks 56

generic except block 57

else 57

finally 58

nesting of try-except 59

8. Files 61

open 61

write 62

read 62

9. Database connection 65

Database 65

SQLite programming 65

Create SQLite3 Connection 66

SQLite3 Cursor 66

Create Table in SQLite3 66

Insert in Table (SQLite3) 68

Update Table (SQLite3) 69

Select statement (SQLite3) 70

Fetch all data (SQLite3) 70

SQLite3 rowcount 72

List tables (SQLite3) 72

Check if table exists or not (SQLite3) 73

Drop Table (SQLite3) 75

SQLite3 Executemany (Bulk insert) 75

Close Connection (SQLite3) 76

Appendix A: Executing python files on console 77

Spyder 77

idle 77

Appendix B: Getting Help 78

Getting help 78

# 1. Fun with Python

Let us get started with writing codes in the IDE. The examples discussed here have been executed in anaconda’s spyder console, however we will get similar output in any python based IDE/interpreter/framework.

Python prompt as a calculator:

In the cell of spyder console, type the following:

4 + 3

After pressing shift+enter, the output is visible as follows:

7

## Comments:

Comments in Python are single line comments and start with the hash character.

# this is a comment

A comment may appear at the start of a line or following whitespace or code, but not within a string literal. A hash character within a string literal is just a hash character. Since comments are to clarify code and are not interpreted by Python, they may be omitted when typing in examples.

Try out the following examples to get a feel of python language:

Type the following in a input cell of console, and press shift+enter to see the output:

input:

a = 10

print(a)

output:

10

print() is a function used to print value of any variable

It can also be used to print a string

input:

print("hello")

output:

hello

Let us further explore strings & numbers subsequently.

## 

## Numbers

Integer numbers (e.g. 8, 10, 22) have type **int**.

Floating point types (fractions, e.g. 5.4, 1.98) have type **float**.

We will see more about numeric data types later.

Operators +, -, \* and / work just like in most other languages with selective differences

Some examples:

input:

20 + 10 \* 2

output:

40

input:

20/6 #division with float result

output:

3.3333333333333335

input:

20//6 #division with int result

#this is called floor division

output:

3

input:

17 % 3

# the % operator returns the remainder of the division

output:

2

With Python, it is possible to use the \*\* operator to calculate powers

input:

9 \*\* 2 # 9 squared

output:

81

## 

## variables

Variables in python can be created anywhere.

Data types are not required to be mentioned.

Data type is picked up based on value assigned to it.

input:

var\_x = 20 #data type becomes int

var\_y = "hyderabad" #data type becomes string

var\_z = 4.56 #data type becomes float

can check the inherent data type by using the type() function

input:

type(var\_y)

output:

str

Variable names can start with an underscore or a letter.

Variable names can have an underscore, letter or a number.

Undefined variables (unassigned) will throw an error when used.

‘\_’ is a name of a variable which stores the latest output (whatever has been displayed as output last)

input:

var\_x + var\_z

output:

24.56

input:

\_

output:

24.56

if ‘\_’ is overwritten (assigned a value explicitly) then it loses its default behaviour and becomes a named variable in the current context.

## 

## Multiple Assignment

Multiple variables can be assigned different values in a single statement.

input:

a, b, c = 4, 5, 7

print(a)

print(b)

print(c)

output:

4

5

7

Multiple variables can be assigned same values in a single statement.

input:

d = e = f = 10

print(d)

print(e)

print(f)

output:

10

10

10

## Strings

Strings can be enclosed in single quotes ('...') or double quotes ("...") with the same result. Use either single or double quotes, the examples below will give a further clarity.

\ can be used to escape sequences like in other languages

The following are valid examples:

input:

'Hyderabad'

output:

'Hyderabad'

input:

"Secunderabad"

output:

'Secunderabad'

input:

"hyderabad doesn't have a port"

#single quote within double quote

output:

"hyderabad doesn't have a port"

input:

'hyderabad doesn\'t have a port' #see the usage of \

output:

"hyderabad doesn't have a port"

input:

'"vizag" is a port city'

output:

'"vizag" is a port city'

input:

"hyderabad has a \"metro\" service though"

output:

'hyderabad has a "metro" service though

input:

'hyd' "era" 'bad' #auto concatenation

output:

'hyderabad'

As seen in above example, strings written adjacent to each other gets concatenated.

‘+’ can be used to concatenate explicitly and ‘\*’ can be used to multiply

input:

str\_a = 'hyder'

str\_b = 'abad'

input:

str\_a + str\_b

output:

'hyderabad'

input:

str\_a + str\_b \* 3

output:

'hyderabadabadabad'

**print()** function reads & prints string appropriately

It also attaches a newline at the end by default.

input:

vijayawada = 'hot city \n but beautiful'

input:

vijayawada

output:

'hot city \n but beautiful'

input:

print(vijayawada)

output:

hot city

but beautiful

If you don’t want characters prefaced by \ to be interpreted as special characters, you can use *raw strings* by adding an r before the first quote:

input:

print("\telangana\nizamabad")

output:

elangana

izamabad

input:

print(r"\telangana\nizamabad")

output:

\telengana\nizamabad

**docstrings**

String literals can span multiple lines by using triple-quotes:

"""...""" or '''...'''

internally its stored using \n

input:

docstring1 = """this line

spans

multiple lines """

input:

docstring1

output:

'this line \nspans \nmultiple lines '

input:

print(docstring1)

output:

this line

spans

multiple lines

Docstrings are used for adding descriptions about classes & packages

Some programmers also tend to use it like multi line comments

## string index

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.

input:

var\_c = "collins"

input:

var\_c[0] # character in position 0

output:

'c'

input:

var\_c[1] # character in position 1

output:

'o'

input:

var\_c[9] # character in position 7

#this should throw us an error

output:

---------------------------------------------------------------------------

Traceback (most recent call last):

File "<ipython-input-7-4446c6562fdf>", line 1, in <module>

var\_c[9]

IndexError: string index out of range

Indices may also be negative numbers, it counts from right.

(Note that since -0 is the same as 0, negative indices start from -1)

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **c** | **o** | **l** | **l** | **i** | **n** | **s** |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 |
| -7 | -6 | -5 | -4 | -3 | -2 | -1 |

input:

var\_c[-1]

output:

's'

input:

var\_c[-4]

output:

'l'

## slicing

In addition to indexing, *slicing* is also supported. While indexing is used to obtain individual characters, *slicing* allows you to obtain substring:

input:

var\_c[0:2]

# characters from position 0 (included) to 2 (excluded)

output:

'co'

input:

var\_c[2:5]

# characters from position 2 (included) to 5 (excluded)

output:

'lli'

input:

var\_c[:2]

# character from the beginning to position 2 (excluded)

output:

'co'

input:

var\_c[4:]

# characters from position 4 (included) to the end

output:

'ins'

input:

var\_c[-2:]

# characters from the second-last (included) to the end

output:

'ns'

Earlier we observed that attempting to use an index that is too large resulted in an error;

However, out of range slice indexes are handled gracefully when used for slicing:

input:

var\_c[2:50]

output:

'llins'

input:

var\_c[-2:-24]

output:

''

input:

var\_c[10:14]

output:

''

Python strings cannot be changed — they are **immutable**. It means that individual letters of a string can not be replaced.

Therefore, assigning to an indexed position in the string results in an error:

input:

var\_c[2] = 't'

output:

---------------------------------------------------------------------------

TypeError Traceback (most recent call last)

<ipython-input-76-78a8ecf8cb2e> in <module>

----> 1 var\_c[2] = 't'

TypeError: 'str' object does not support item assignment

The built-in function len() returns the length of a string:

input:

len(var\_c)

output:

6

## 

## List

There are a number of *compound* data types or data structures in Python, which are used to group together other values. The most versatile is the *list*, which can be written as a list of comma-separated values (items) between square brackets. Lists might contain items of different types, but usually the items all have the same type.

input:

list\_a = [34, 6, 9, 12, 65, 23, 78]

list\_a

output:

[34, 6, 9, 12, 65, 23, 78]

input:

print(list\_a)

output:

[34, 6, 9, 12, 65, 23, 78]

Like strings, lists can be indexed and sliced.

The data type of list is list, where as the type of individual elements may be different

All slice operations return a new list (shallow copy) containing the requested elements

input:

type(list\_a)

output:

list

input:

type(list\_a[2])

output:

int

input:

list\_a[3]

output:

12

input:

list\_a[2:4]

output:

[9, 12]

input:

list\_a[-4:]

output:

[12, 65, 23, 78]

Like strings, lists also support operations like concatenation:

input:

list\_b = list\_a + [11,22,33]

list\_b

output:

[34, 6, 9, 12, 65, 23, 78, 11, 22, 33]

Unlike strings, which are immutable, lists are a **mutable** type, i.e. it is possible to change their content:

input:

list\_c = ['hi', 'hello', 'nice','dp']

input:

list\_c[3] = 'display pic'

list\_c

output:

['hi', 'hello', 'nice', 'display pic']

We can also add new items at the end of the list, by using the append() *method*.

(we will see more about methods later)

The built-in function len() also applies to lists.

input:

list\_c.append('goodday')

list\_c

output:

['hi', 'hello', 'nice', 'display pic', 'goodday']

Assignment to slices is also possible, and this can even change the size of the list or clear it entirely:

input:

list\_d = ['h', 'y', 'd', 'e', 'r', 'a', 'b', 'a', 'd']

list\_d

output:

['h', 'y', 'd', 'e', 'r', 'a', 'b', 'a', 'd']

input:

list\_d[2:5] = ['D', 'E', 'R']

list\_d

output:

['h', 'y', 'D', 'E', 'R', 'a', 'b', 'a', 'd']

input:

list\_d[2:5] = []

list\_d

output:

['h', 'y', 'a', 'b', 'a', 'd']

## Nesting of lists

Lists can have different data types including lists.

In other words lists can be nested too.

input:

list\_e = [49 , 'centuries']

list\_f = [44.8 , 'average']

player = ['sachin', list\_e, list\_f]

player

output:

['sachin', [49, 'centuries'], [44.8, 'average']]

input:

player[1]

output:

[49, 'centuries']

input:

player[1][1]

output:

'centuries'

input:

player[1][1][1]

output:

'e'

## 

## Dynamic data types

The type of data in python is dynamic in nature and can be changed based on the value that is assigned to it.

input:

var\_a = 'hello'

type(var\_a)

output:

str

input:

var\_a = 34

type(var\_a)

output:

int

input:

var\_a = [4,7]

type(var\_a)

output:

list

So here we have seen var\_a taking different data types based on what is stored in it.

So the question here is, does the value keep getting updated in the same location?

No it doesn't, different values (‘hello’, 34, [4,7]) are stored at different locations, however var\_a represents different locations based on its assignment.

Unlike C/C++ memory locations can not be accessed & printed, so the above concept can be understood with the id() function.

## object and id()

Everything in python is internally treated as an object. Integers that we create are actually objects of the class int. Strings are objects of the class str. Even functions are treated as objects, belonging to an inbuilt class ‘function’.

Each of these objects are stored at unique locations, but the memory addresses can’t be displayed. Hence id() function helps us look at the unique identity attached to any object. objects which same value get same id.

input:

var\_a = 10

id(var\_a)

output:

4371596688

input:

id(10)

output:

4371596688

input:

var\_b = 20

id(var\_b)

output:

4371597008

As expected var\_b has a different id as it's a different variable with a different value.

However when var\_b is equated to same value as var\_a (value 10), var\_b gets the same id as var\_a as shown below:

input:

var\_b = 10

id(var\_b)

output:

4371596688

So all the objects with the value 10 will have same id in the system.

## conversion functions

int() function converts any data to integer

input:

var\_i = '5678923'

type(var\_i)

output:

str

input:

var\_k = int(var\_i)

type(var\_k)

output:

int

Likewise str() converts data to string, float() converts data to float.  
list() will generate a list.

## del

the ***del*** statement is used to delete an object(variable, list, string, function, class…. etc.)

del var\_a

will delete the variable named var\_a if it exists.

An entire list, or an element in a list can be deleted using ***del***

## None

A value exists in the Python interpreter as ***None*** . It means nothing and is similar to void values in C/C++ programming.

When a variable is equated to None, it means that the variable exists but has nothing in it yet. It is different from ***del*** (which deletes the variable’s identity and clears the memory for the variable)

Functions which return no values are equivalent to saying functions return ***None***.

## Bool and Logical Operator

A boolean expression (or logical expression) evaluates to one of two states true or false. A bool data type exists in python having two values:

**True**

**False**

Many functions and operations returns boolean objects. Logical operators can be used anywhere in the code.

input:

var\_a = 0

var\_b = (var\_a == 0)

print(var\_b)

output:

True

input:

type(var\_b)

output:

bool

input:

type(True)

output:

bool

The not keyword can also be used to inverse a boolean type

The following elements are false:

***None***

***False***

0 (whatever type from integer, float to complex)

Empty data structures: “”, (), [], {}

Objects from classes that have the special method \_\_nonzero\_\_

## Comparison operators

The <, <=, >, >=, ==, != operators compare the values of 2 objects and returns True or False. Comparison depends on the type of the objects.

Comparison operators can be chained

input:

var\_x = 20

1 < var\_x < 40

output:

True

input:

20 == var\_x < 50

output:

True

input:

10 < var\_x < 20

output:

False

## Logical operators

and

or

not

## Membership operators

**in** evaluates to True if it finds a variable in a specified sequence and false otherwise.

**not in** evaluates to False if it finds a variable in a sequence, True otherwise.

input:

'h' in 'hyderabad'

output:

True

input:

'h' not in 'hyderabad'

output:

False

## 

## Identity operators

**is** evaluates to True if the variables on either side of the operator point to the same object and False otherwise

**is not** evaluates to False if the variables on either side of the operator point to the same object and True otherwise

## Bitwise operators

Bitwise operators are used to work on integers in their binary formats.

For example, the and operation between 2 the values 5 and 4 is actually the and operations between 0101 and 0100 binaries. It is therefore equal to 0100

input:

4 & 5

output:

4

>> bitwise left shift

<< bitwise right shift

& bitwise and

^ bitwise xor

| bitwise or

~ bitwise not

# 2. Flow control

Like many other languages, python has features for controlling the flow of a program and to implement algorithms. Some of these features are:

while

if elif

for

break and continue

else clause on loops

In Python, like in C, any non-zero integer value is true; zero is false. The condition may also be a string or list value, in fact any sequence; anything with a non-zero length is true, empty sequences are false.

The standard comparison operators are written the same as in C: < (less than), > (greater than), == (equal to), <= (less than or equal to), >= (greater than or equal to) and != (not equal to).

## while

The *while loop* is similar in behaviour as in other programming languages.

Consider the following example:

input:

var\_a = 'india'

i = 0

while i < 5:

print(var\_a[i])

i = i+1

output:

i

n

d

i

a

The test condition used in the example is a simple comparison (i < 5)

The body of the loop is *indented*: indentation is Python’s way of grouping statements.

The indentation was done automatically after a colon (:)

When a compound statement is entered interactively, it must be followed by a blank line to indicate completion (since the parser cannot guess when you have typed the last line). Note that each line within a basic block must be indented by the same amount.

As discussed earlier, print() function adds a new line by default. The keyword argument *end* can be used to avoid the newline after the output. In the following example, newline is replaced by a comma operator using *end* keyword.

input:

var\_a = 'india'

i = 0

while i < 5:

print(var\_a[i], end=',')

i = i+1

output:

i,n,d,i,a,

## if

if statements are plaintively simple

input:

var\_a = 30

if var\_a == 20:

print('value is 20')

elif var\_a < 20:

print('value is greater than 20')

elif var\_a > 20:

print('value is greater than 20')

else:

print('something is not right')

output:

value is greater than 20

There can be zero or more elif parts, and the else part is optional. The keyword ‘elif’ is short for ‘else if’, and is useful to avoid excessive indentation.

An if … elif … elif … sequence is a substitute for the switch or case statements found in other languages.

## for

The for statement in Python differs a bit from what you may be used to in other languages. Rather than always iterating over an arithmetic progression of numbers, or giving the user the ability to define both the iteration step and halting condition, Python’s for statement iterates over the items of any sequence (a list or a string), in the order that they appear in the sequence.

input:

cities = ['hyderabad', 'delhi', 'mumbai', 'bengaluru', 'chennai']

for var\_c in cities:

print(var\_c, len(var\_c))

output:

hyderabad 9

delhi 5

mumbai 6

bengaluru 9

chennai 7

The for loop has iterated over the entire list, equating var\_c to each item during the iteration.

Slicing can also be used to iterate from a particular start point to a particular end point.

input:

for var\_c in cities[1:4]:

print(var\_c, len(var\_c))

output:

delhi 5

mumbai 6

bengaluru 9

## range

If you do need to iterate over a sequence of numbers, the built-in function range() comes in handy. It generates arithmetic progressions.

input:

for i in range(5):

print(i)

output:

0

1

2

3

4

Range can have a start & end point. The given end point is never part of the generated sequence. It is possible to let the range start at another number, or to specify a different increment (even negative; sometimes this is called the ‘step’):

input:

for i in range(3,9):

print(i)

output:

3

4

5

6

7

8

input:

for i in range(3,9,2):

print(i)

output:

3

5

7

input:

for i in range(-1,-10,-3):

print(i)

output:

-1

-4

-7

To iterate over the indices of a sequence, you can combine range() and len() as follows:

input:

cities = ['hyderabad', 'delhi', 'mumbai', 'bengaluru',

'chennai']

for i in range(len(cities)):

print(i, cities[i])

output:

0 hyderabad

1 delhi

2 mumbai

3 bengaluru

4 chennai

When looping through a sequence, the position index and corresponding value can be retrieved at the same time using the enumerate() function.

input:

cities = ['hyderabad', 'delhi', 'mumbai', 'bengaluru',

'chennai']

for i, var\_d in enumerate(cities):

print(i, var\_d)

output:

0 hyderabad

1 delhi

2 mumbai

3 bengaluru

4 chennai

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.

**iterable**

We say such an object is iterable, that is, suitable as a target for functions and constructs that expect something from which they can obtain successive items until the supply is exhausted. We have seen that the *for* statement is such an *iterator*.

The function list() as we say previously is another such iterator; it creates lists from iterables.

input:

list\_i = list(range(2,6))

list\_i

output:

[2, 3, 4, 5]

## break, continue, else

The break statement, like in C, breaks out of the innermost enclosing for or while loop.

input:

for i in range(0,3):

for j in range(0,3):

if(i==j):

print("tea break")

break

else:

print("no tea break")

output:

tea break

no tea break

tea break

no tea break

no tea break

tea break

Unlike other languages, 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.

This is exemplified by the following loop, which searches for prime numbers:

input:

for n in range(2, 10):

for x in range(2, n):

if n % x == 0:

print(n, 'equals', x, '\*', n//x)

break

else:

# loop completed without finding a factor

print(n, 'is a prime number')

output:

2 is a prime number

3 is a prime number

4 equals 2 \* 2

5 is a prime number

6 equals 2 \* 3

7 is a prime number

8 equals 2 \* 4

9 equals 3 \* 3

The ***else*** clause here belongs to the ***for*** loop, not the ***if*** statement

## pass

The ***pass*** statement does nothing. It can be used when a statement is required syntactically but the program requires no action. For example in below code, ***pass*** does nothing:

input:

var\_a = 30

if var\_a == 20:

pass

elif var\_a < 20:

pass

elif var\_a > 20:

pass

else:

print('something is not right')

pass is used more for readability rather than any algorithm.

It can be used as a placeholder for a function, class or conditional body when you are working on new code, allowing you to keep thinking at a more abstract level.

# 3. 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.

input:

def func\_a():

print('hello world')

Now func\_a() can be called anywhere in the program

input:

func\_a()

output:

hello world

Functions can also return values using keyword return. Unlike most other languages, multiple values can also be returned from a function. ***None***is returned when nothing is mentioned.

input:

def func\_b():

print('returns a float value')

return 44.55

input:

var\_b = func\_b()

print(var\_b)

output:

returns a float value

44.55

input:

def func\_c():

print('returns multiple values')

return 3, 5, 9

input:

var\_a, var\_b, var\_c = func\_c()

print(var\_b)

output:

returns multiple values

5

The name of the function itself is an object type.

input:

def funch():

print('hello world')

input:

funch():

output:

hello world

input:

funch

output:

<function funch at 0x10f56c158>

input:

type (funch)

output:

<class 'function'>

input:

f=funch

f()

output:

hello world

input:

id(f)

output:

4552311128

input:

id(funch)

output:

4552311128

## 

## function nesting

Function definitions can be enclosed within another function. The enclosed function can not be called outside but only within the enclosing function.

input:

def func\_o():

print("outer 1st statement")

#some code here

#-----------------

def func\_i():

print("in inner function")

#-----------------

print("outer 2nd statement")

#some more code here

#

#

#

func\_i()

print("outer 3rd statement")

input:

func\_o()

output:

outer 1st statement

outer 2nd statement

in inner function

outer 3rd statement

input:

func\_i()

output:

---------------------------------------------------------------------------

NameError Traceback (most recent call last)

<ipython-input-32-a5da79d35acf> in <module>

----> 1 func\_i()

NameError: name 'func\_i' is not defined

## 

## symbol table

The execution of a function introduces a new symbol table used for the local variables of the function. More precisely, all variable assignments in a function store the value in the local symbol table; whereas 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. Thus, global variables cannot be directly assigned a value within a function (unless named in a ***global*** statement), although they may be referenced.

input:

x = 'global'

def outer():

x = "local"

def inner():

# nonlocal x

# global x

x = "nonlocal"

print("inner:", x)

inner()

print("outer:", x)

outer()

print(x)

output:

inner: nonlocal

outer: local

global

Remove the comments alternatively in the above code to see the behaviour of global, local & nonlocal.

## 

## Default arguments

A function can take values from the user, and also have default values for the arguments if the user fails to give any values.

input:

def default(var\_a = 20, var\_b = 30):

print('var\_a = ', var\_a)

print('var\_b = ', var\_b)

input:

default()

output:

var\_a = 20

var\_b = 30

input:

default(11)

output:

var\_a = 11

var\_b = 30

input:

default(13,14)

output:

var\_a = 13

var\_b = 14

Selective arguments can be given values while calling, these are called as keyword arguments. For example var\_b becomes the keyword argument here

input:

default(var\_b = 100)

output:

var\_a = 20

var\_b = 100

Keyword arguments can be given in any order provided all the keyword arguments passed must match one of the arguments accepted by the function

For example

**default(var\_b = 100, var\_a = 200)** and

**default(var\_a = 200, var\_b = 100)** are same.

Please note that positional argument can not follow keyword argument, the following will throw an error

input:

default(var\_b = 100, 30)

output:

File "<ipython-input-70-638a5c2ff929>", line 1

default(var\_b = 100, 30)

^

SyntaxError: positional argument follows keyword argument

## Documentation strings

If a string is written just below the function definition then it automatically becomes a doc string (documentation string). Any string can be used but conventionally a string with triple quotes is used and most programmers tend to call only the triple quote string as document string. It can be accessed by an inbuilt variable called \_\_doc\_\_

input:

def func\_d():

"""function is used to do something important

What it does can be documented here

"""

pass

input:

print(func\_d.\_\_doc\_\_)

output:

function is used to do something important

What it does can be documented here

# 4. Data Structures

Python is enriched with multiple data structures (or sequences) in various forms. They are similar to the ‘collections’ of java or ‘STL’ of C++

We already have discussed lists, let us deeper into it and look at some of the functions used with lists.

## Lists, extended

list.append(x)

Add an item to the end of the list. Equivalent to a[len(a):] = [x].

list.extend(iterable)

Extend the list by appending all the items from the iterable.

Equivalent to a[len(a):] =iterable.

list.insert(i, x)

Insert 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).

list.remove(x)

Remove the first item from the list whose value is equal to x.

It raises a ValueError if there is no such item.

list.pop([i])

Remove the item at the given position in the list, and return it.

If no index is specified, a.pop() removes and returns the last item in the list.

list.clear()

Remove all items from the list. Equivalent to del a[:].

list.index(x)

Return zero-based index in the list of the first item whose value is equal to x. Raises a ValueError if there is no such item.

list.count(x)

Return the number of times x appears in the list.

list.sort()

Sort the items of the list in place

list.sort(reverse=True) will sort in reverse order

list.reverse()

Reverse the elements of the list in place.

list.copy()

Return a shallow copy of the list. Equivalent to a[:].

A few of the functions have been demonstrated below:

input:

cities = ['hyderabad', 'delhi', 'mumbai', 'bengaluru', 'chennai']

northeast = ['agartala', 'gangtok', 'guwahati', 'shillong']

input:

cities.append('kolkata')

print(cities)

output:

['hyderabad', 'delhi', 'mumbai', 'bengaluru', 'chennai', 'kolkata']

input:

cities.extend(northeast)

print(cities)

output:

['hyderabad', 'delhi', 'mumbai', 'bengaluru', 'chennai', 'agartala', 'gangtok', 'guwahati', 'shillong']

input:

cities.index('chennai')

output:

4

input:

cities.count('chennai')

output:

1

input:

cities.sort(reverse = True)

print(cities)

output:

['shillong', 'mumbai', 'hyderabad', 'guwahati', 'gangtok', 'delhi', 'chennai', 'bengaluru', 'agartala']

## List Comprehensions

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

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

input:

squares = []

for x in range(10):

squares.append(x\*\*2)

input:

print(squares)

output:

[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]

This can be written in a concise manner using list comprehension:

input:

squares = [x\*\*2 for x in range(10)]

input:

print(squares)

output:

[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]

For example, this listcomp combines the elements of two lists if they are not equal:

input:

[(x, y) for x in [3,4,5] for y in [2,3,4] if x != y]

output:

[(3, 2), (3, 4), (4, 2), (4, 3), (5, 2), (5, 3), (5, 4)]

## Tuple

Tuples are similar to list, barring a few differences like:

* declared using parentheses
* immutable
* faster

input:

telangana =('hyderabad', 'warangal', 'khammam', 'karimnagar', 'nizamabad')

type(telangana)

output:

tuple

Since tuples are immutable, the tuple telangana can’t be appended. The elements in telangana can’t be changed, popped etc.

Tuples can also be declared without using parentheses by using comma operator

The following is a valid declaration:

input:

telangana ='hyderabad', 'warangal', 'khammam', 'karimnagar', 'nizamabad'

However a tuple with a single element can not be created using parentheses, instead has to be created using a trailing comma

input:

t1 = ('one') #this creates a string

type(t1)

output:

str

input:

t2 = 'one', #this creates a tuple

output:

type(t2)

Important functions for a tuple:

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.

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() Return the sum of all elements in the tuple.

tuple() Convert an iterable (list, string, set, dictionary) to a tuple.

count(x) Return the number of items that is equal to x

index(x) Return index of first item that is equal to x

## Sets

A set is an unordered collection of items. Every element is unique (no duplicates) and must be immutable (which cannot be changed). However, the set itself is mutable. We can add or remove items from it.

Sets can be used to perform mathematical set operations like union, intersection, symmetric difference etc.

A set is created by placing all the items (elements) inside curly braces {}, separated by comma

input:

set\_a = {1, 2, 3}

print(set\_a)

output:

{1, 2, 3}

As mentioned earlier sets can have different data types, but cant have mutable type (example it can not have lists)

input:

set\_b = {1, 2, 3, 'some string'}

print(set\_b)

output:

{'some string', 1, 2, 3}

input:

cities = ['hyderabad', 'delhi', 'mumbai', 'bengaluru', 'chennai']

set\_b = {1, 2, 3, cities}

print(set\_b)

output:

---------------------------------------------------------------------------

TypeError Traceback (most recent call last)

<ipython-input-116-f9c88584d41b> in <module>

----> 1 set\_b = {1, 2, 3, cities}

**2** print(set\_b)

TypeError: unhashable type: 'list'

We get an error in the above code cause it is trying to have a list in a set.

We can add single element using the add() method and multiple elements using the update() method. The update() method can take tuples, lists, strings or other sets as its argument. In all cases, duplicates are avoided.

A particular item can be removed from set using methods, discard() and remove().

The only difference between the two is that, while using discard() if the item does not exist in the set, it remains unchanged. But remove() will raise an error in such condition.

Set Operations:

Sets can be used to carry out mathematical set operations like union, intersection, difference and symmetric difference. We can do this with operators or methods.

Union of A and B is a set of all elements from both sets.

Union is performed using | operator or using the method union().

input:

set\_c = {1, 2, 3, 4, 5}

set\_d = {4, 5, 6, 7, 8}

print(set\_c|set\_d)

output:

{1, 2, 3, 4, 5, 6, 7, 8}

Similarly intersection, difference etc can be performed over sets. Following is a list of important functions associated with set.

add() Adds an element to the set

clear() Removes all elements from the set

copy() Returns a copy of the set

difference() Returns the difference of two or more sets as a new set

discard() Removes an element from the set if it is a member.

(Do nothing if the element is not in set)

intersection() Returns the intersection of two sets as a new set

isdisjoint() Returns True if two sets have a null intersection

issubset() Returns True if another set contains this set

issuperset() Returns True if this set contains another set

pop() Removes and returns an arbitrary set element.

Raise KeyError if the set is empty

remove() Removes an element from the set.

If the element is not a member, raise a KeyError

union() Returns the union of sets in a new set

update() Updates the set with the union of itself and others

all() Return True if all elements of the set are true (or if the set is empty).

any() Return True if any element of the set is true.

If the set is empty, return False.

len() Return the length (the number of items) in the set.

max() Return the largest item in the set.

min() Return the smallest item in the set.

sorted() Return a new sorted list from elements in the set

(does not sort the set itself).

sum() Return the sum of all elements in the set.

## Dictionary

Python dictionary is an unordered collection of items. While other compound data types have only value as an element, a dictionary has a key: value pair.

Dictionaries are optimized to retrieve values when the key is known.

An item has a key and the corresponding value expressed as a pair, key: value.

While values can be of any data type and can repeat, keys must be of immutable type (string, number or tuple with immutable elements) and must be unique.

input:

dict\_a = {'thor':'hammer', 'captain':'shield', 'ironman':'suit', 'hawkeye':'arrow'}

input:

print(dict\_a['thor'])

print(dict\_a['hawkeye'])

output:

hammer

arrow

Dictionary are mutable. We can add new items or change the value of existing items using assignment operator.If the key is already present, value gets updated, else a new key: value pair is added to the dictionary.

We can remove a particular item in a dictionary by using the method pop(). This method removes as item with the provided key and returns the value.The method, popitem() can be used to remove and return an arbitrary item (key, value) form the dictionary. All the items can be removed at once using the clear() method.We can also use the del keyword to remove individual items or the entire dictionary itself.

clear() Remove all items from the dictionary.

copy() Return a shallow copy of the dictionary.

get(key[,d]) Return the value of key. If key does not exist, return d (defaults to None).

items() Return a new view of the dictionary's items (key, value).

keys() Return a new view of the dictionary's keys.

pop(key[,d]) Remove the item with key and return its value or d if key is not found.

If d is not provided and key is not found, raises KeyError.

popitem() Remove and return an arbitrary item (key, value).

Raises KeyError if the dictionary is empty.

all() Return True if all keys of the dictionary are true

(or if the dictionary is empty).

any() Return True if any key of the dictionary is true.

If the dictionary is empty, return False.

len() Return the length (the number of items) in the dictionary.

cmp() Compares items of two dictionaries.

sorted() Return a new sorted list of keys in the dictionary.

5. Modules

## Modules

Python has a way to put definitions in a file and use them in a script or in an interactive instance of the interpreter. Such a file is called a *module*; definitions from a module can be *imported* into other modules or into the *main* module

A module is a file containing Python definitions and statements. 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\_\_

Python comes with a library of standard modules; ***sys , os, builtins*** are some of the examples. Functions, data and objects within these modules can be used after importing these modules on to our programming environment

input:

import os

os.getcwd()

output:

'/Users/apple/Desktop/collins'

The above output told me the current directory where anaconda spyder is running

input:

os.mkdir('temp')

This will create a directory named ‘temp’ in the current directory

Some other functions are:

The current directory can be changed using ***os.chdir()***

The ***listdir()*** function returns the content of a directory

You can delete an empty directory with ***rmdir()***

You can remove all directories within a directory (if there are not empty) by using ***os.removedirs()***.

To remove a file, use ***os.remove()***. It raise the OSError exception if the file cannot be removed. Under Linux, you can also use ***os.unlink().***

You can rename a file from an old name to a new one by using ***os.rename()***.

The following example puts the kernel to sleep for 4 seconds

input:

import time

input:

time.sleep(4)

print("hi") #will print after 4 seconds

output:

hi

***sys*** is another such inbuilt module.

***sys.path*** gives us the list of environment path where python interpreter will search for any other modules/files/packages

***sys.version*** displays a string containing the version number of the current Python interpreter.

## Custom Modules

We can create our own modules by saving our files with the .py extension

Any functions or objects defined in the module will be accessible to the other programs which import this module. Enter the following statements in the input cell.

Create a new file in Python IDE named mod\_a

input:

def func\_a():

print("in funcA of module A")

input:

def func\_b():

print("in funcB of module A")

input:

var\_a = 100

var\_b = 200

input:

print("hello from mod A")

output:

hello from mod A

Close this tab and create a new file named mod\_b

There are three ways to import mod\_a in mod\_b

1. import mod\_a

In this case mod\_a.py will run once in the current environment (background) and then will be accessible to codes & functions in mod\_b.

Your code can access the functions & variables of mod\_a in mod\_b in the following way:

mod\_a.func\_a()

mod\_a.var\_a()

The inbuilt variable \_\_name\_\_ (mod\_a.\_\_name\_\_) will display name of the module in string format

1. from mod\_a import func\_a()

In this case mod\_a.py will run once in the current environment (background) and then only func\_a() will be accessible to codes & functions in mod\_b.

No prefix is required and function can be used directly with the name as

func\_a()

1. from mod\_a import \*

In this case mod\_a.py will run once in the current environment (background) and then will be accessible to codes & functions in mod\_b.

Your code can access all the functions & variables of mod\_a in mod\_b in the following way:

func\_a()

var\_a()

(No prefix is required and function can be used directly with the name)

In all the above cases imported module runs once in your environment, so if the imported module has any printable statements, their output may be displayed in your current environment (IDE specific).

If you import the module again, the import statement does not fetch the module again as its already present in the current environment. This creates an issue if the module has been changed during execution and the application needs the newer version of it. To solve this, reload() can be used to run the module again.

For versions before/and Python3.3

input:

import imp

imp.reload(mod\_a)

For Python3.4 versions:

input:

import importlib

importlib.reload(mod\_a)

6. Object Oriented Programming

Object Oriented Programming is achieved in Python by creating classes & objects

The simplest form of class definition looks like this:

class ClassName:

<statement-1>

.

.

.

<statement-N>

Class definitions must be executed before they have any effect.The statements inside a class definition will usually be function definitions, but other statements are also allowed. When a class definition is entered, a new namespace is created, and used as the local scope — thus, all assignments to local variables go into this new namespace. In particular, function definitions bind the name of the new function here.

Class objects support two kinds of operations: attribute references and instantiation.

Attribute references use the standard syntax used for all attribute references in Python: obj.name. Valid attribute names are all the names that were in the class’s namespace when the class object was created. So, if the class definition looked like this:

input:

class bahubali:

areaname = "mahishmati"

def funca(self):

print("jai mahishmati")

then bahubali.areaname and bahubali.funca() are valid attribute references, returning an integer and a function object, respectively. Class attributes can also be assigned to, so you can change the value of bahubali.areaname by assignment.

Class instantiation uses function notation

input:

obj = bahubali()

calling attribute references via object is done in the following way:

print(bahubali.areaname)

obj.funca()

output:

mahishmati

jai mahishmati

understanding the **self** parameter:

obj.funca() translates into amarendra.funca(obj)

so **obj** is received as an object and stored in **self**

‘self’ is an argument

can be named anything, but conventionally the term **self** is used

doctrings used earlier with functions also have an effect here:

input:

class amarendra:

"jai mahishmati"

areaname = "mahishmati"

print(amarendra.\_\_doc\_\_)

output:

jai mahishmati

so the \_\_doc\_\_ here displays the string written after the class name (it can not be written anywhere else) and tells us about what the class does.

## \_\_init\_\_()

Similar to a constructor in other languages a special function \_\_init\_\_() can be used to instantiate class members. When a class defines an \_\_init\_\_() method, class instantiation automatically invokes \_\_init\_\_() for the newly-created class instance. The \_\_init\_\_() method may have arguments for greater flexibility. In that case, arguments given to the class instantiation operator are passed on to \_\_init\_\_()

input:

class amarendra:

"jai mahishmati"

areaname = "mahishmati"

def \_\_init\_\_(self, x=10,y=20):

print("hello")

self.a = x

self.b = y

def funca(self):

print(self.a, self.b)

input:

obja = amarendra()

objb = amarendra(2,3)

obja.funca()

objb.funca()

output:

hello

hello

10 20

2 3

The ‘hello’ printed twice shows us that \_\_init\_\_() was called twice (once for each object). The values for objb indicate how default values were overwritten.

## Inheritance

Classes can be derived out of base classes (like in any other OOP language).

If we have to derive class mahendra from class amarendra, the the syntax would be

class mahendra(amarendra):

pass

When the class object is constructed, the base class is remembered. This is used for resolving attribute references: 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.

Derived classes may override methods of their base classes. Because methods have no special privileges when calling other methods of the same object, a method of a base class that calls another method defined in the same base class may end up calling a method of a derived class that overrides it. (this is more like saying all methods are ‘virtual’ in C++ analogy).

input:

class mahendra(amarendra):

"jai mahishmati part 2"

def \_\_init\_\_(self, x=5,y=15):

print("hello from beta")

self.a = x

self.b = y

def funcb(self):

print(self.a, self.b)

input:

objc = mahendra()

objd = mahendra(7,8)

objc.funcb()

objd.funcb()

output:

hello from beta

hello from beta

5 15

7 8

## private data

For java, c++ or other oops programmers the concept of public and private/protected is pretty core. However python does not support such keywords or explicit access specifiers like ‘public’ or ‘private’.

Instead all data attributes created using \_\_ (double underscore) and not ending with \_\_ (double underscore) automatically becomes inaccessible outside the class.

For example in the following code, the class data **\_\_source** is not accessible (readable) outside the class. However can be updated from within the class.

On the contrary **\_api** and **\_\_internet\_\_**  can be accessed both from within and outside the class.

input:

class windows:

\_api = "api"

\_\_source = "source"

\_\_internet\_\_ = "internet"

def funcw(self):

print(self.\_api, ' ', self.\_\_source, ' ', self.\_\_internet\_\_ )

objectw = windows()

input:

print(objectw.\_api)

output:

api

input:

print(objectw.\_\_internet\_\_)

output:

internet

**\_api** and **\_\_internet\_\_** got printed however we will encounter an error when we try to access **\_\_source**

input:

print(objectw.\_\_source)

output:

Traceback (most recent call last):

File "<ipython-input-51-dd5aaa85bf3f>", line 1, in <module>

print(objectw.\_\_source)

AttributeError: 'windows' object has no attribute '\_\_source'

input:

objectw.funcw()

output:

api source internet

## operator overloading

Polymorphism exists in Python in the form of operator overloading. For example operator + is used to add two integers as well as join two strings and merge two lists. It is achievable because ‘+’ operator is overloaded by int class and str class.

so here the same built-in operator shows different behavior for objects of different classes, this is called *Operator Overloading*.

If we have two objects which are a physical representation of a class and we have to add two objects with binary ‘+’ operator it throws an error, because interpreter doesn't know how to add two objects. So we define a method for an operator and that process is called operator overloading.

We can overload all existing operators but we can’t create a new operator. To perform operator overloading, Python provides some special function or magic function that is automatically invoked when it is associated with that particular operator. For example, when we use + operator, the magic method \_\_add\_\_ is automatically invoked in which the operation for + operator is defined.

Changing the behavior of operator is as simple as changing the behavior of method or function. We define methods in our class and operators work according to that behavior defined in methods.

As mentioned above when we use + operator, the magic method \_\_add\_\_ is automatically invoked in which the operation for + operator is defined. There by changing this magic method’s code, we can give extra meaning to the + operator.

For example we will create a class called time with ‘hours’ and ‘minutes’ attributes.

input:

class time:

def \_\_init\_\_(self, h=0, m=0):

self.h = h

self.m = m

After creating two time objects, we will not be able to add them cause the interpreter fails to understand hot to perform addition on two custom objects.

input:

ob1 = time(3, 30)

ob2 = time(1, 50)

input:

ob3 = ob1 + ob2

output:

Traceback (most recent call last):

File "<ipython-input-21-c5a69b2ec660>", line 1, in <module>

ob3 = ob1 + ob2

TypeError: unsupported operand type(s) for +: 'time' and 'time'

So we resolve this by adding **\_\_add\_\_()** function within the class and then implementing the logic for adding two time objects (corresponding hours & minutes) and giving the result in object form.

input:

class time:

def \_\_init\_\_(self, h=0, m=0):

self.h = h

self.m = m

# adding two objects

def \_\_add\_\_(self, other):

oa = time()

temp1 = self.h \* 60 + self.m

temp2 = other.h \* 60 + other.m

temp3 = temp1 + temp2

oa.h = temp3//60

oa.m = temp3%60

return oa

Now lets recreate objects and try to add them.

input:

ob1 = time(3, 30)

ob2 = time(1, 50)

input:

ob3 = ob1 + ob2

print(ob3.h, "hours and", ob3.m, "minutes")

output:

5 hours and 20 minutes

## Python magic methods

Python magic methods or special functions for operator overloading are as follows:

**Binary Operators**

|  |  |
| --- | --- |
| Operator | magic method |
| + | \_\_add\_\_(self, other) |
| – | \_\_sub\_\_(self, other) |
| \* | \_\_mul\_\_(self, other) |
| / | \_\_truediv\_\_(self, other) |
| // | \_\_floordiv\_\_(self, other) |
| % | \_\_mod\_\_(self, other) |
| \*\* | \_\_pow\_\_(self, other) |

**Unary Operators**

|  |  |
| --- | --- |
| Operator | magic method |
| – | \_\_neg\_\_(self, other) |
| + | \_\_pos\_\_(self, other) |
| ~ | \_\_invert\_\_(self, other) |

**Assignment Operators**

|  |  |
| --- | --- |
| Operator | magic method |
| **-=** | \_\_isub\_\_(self, other) |
| **+=** | \_\_iadd\_\_(self, other) |
| **\*=** | \_\_imul\_\_(self, other) |
| **/=** | \_\_idiv\_\_(self, other) |
| **//=** | \_\_ifloordiv\_\_(self, other) |
| **%=** | \_\_imod\_\_(self, other) |
| **\*\*=** | \_\_ipow\_\_(self, other) |

**Comparison Operators**

|  |  |
| --- | --- |
| operator | magic method |
| < | \_\_lt\_\_(self, other) |
| > | \_\_gt\_\_(self, other) |
| <= | \_\_le\_\_(self, other) |
| >= | \_\_ge\_\_(self, other) |
| == | \_\_eq\_\_(self, other) |
| != | \_\_ne\_\_(self, other) |

## 

# 7. Exception Handling

Python has many built-in exceptions which forces your program to output an error when something in it goes wrong. When these exceptions occur, it causes the current process to stop and passes it to the calling process until it is handled. If not handled, our application may crash.

Some examples of exceptions are:

input:

var\_a = 10

var\_b = 0

input:

var\_c = var\_a/var\_b

output:

Traceback (most recent call last):

File "<ipython-input-60-92bf7458072c>", line 1, in <module>

var\_c = var\_a/var\_b

ZeroDivisionError: division by zero

So here we get a **ZeroDivisionError** exception cause division by zero sp not possible in simple maths.

Another example:

input:

var\_c = var\_a/var\_d

output:

Traceback (most recent call last):

File "<ipython-input-61-71eeb4e9e00b>", line 1, in <module>

var\_c = var\_a/var\_d

NameError: name 'var\_d' is not defined

So here we get a **NameError** exception as var\_d has not been defined in the code yet.

The Exception Handling mechanism in Python constitutes of four important keywords:

**try except finally else**

## try

In Python, exceptions can be handled using a **try** statement.

A critical operation which can raise exception is placed inside the try clause and the code that handles exception is written in except clause.It is up to us, what operations we perform once we have caught the exception.

Here is a simple example.

input:

a\_list =[]

average =1

input:

try:

average = sum(a\_list) / len(a\_list)

print("average is ", average)

except ZeroDivisionError:

print("list is empty")

output:

list is empty

Another example:

We will create a list and try to access an element based on the index given by user.

input:

random = [5, 6, 3, 1]

input:

try:

i= int(input("enter a number"))

value = random[i]

print("value =", random[i])

except IndexError:

print("out of bounds")

If the user enters any 2, we get

output:

enter a number3

3

If the user enters any 7, we get

output:

enter a number7

out of bounds

Another example:

We can write try, except blocks within a function too.

input:

def divide(x, y):

try:

result = x/ y

except ZeroDivisionError:

print("division by zero!")

else:

print("result is", result)

input:

divide(4,3)

output:

result is 1.3333333333333333

input:

divide(4,0)

output:

division by zero!

## Multiple Except blocks

Multiple except blocks can exist for a single try block. Based on the exception encountered first, the most applicable except block will get executed

input:

def divide(x, y):

try:

result = x/ a

except ZeroDivisionError:

print("division by zero!")

except NameError:

print("variable name is wrong!")

Here we see that two except block exists, and we are using undefined variable **a** in the statement. This is bound to give us a NameError (irrespective of Zero Division).

input:

divide(6,7)

output:

variable name is wrong!

input:

divide(6,0)

output:

variable name is wrong!

## generic except block

if we do not mention the name of the exception after the **except** keyword then it automatically becomes a generic except block (similar to catch(...) of C++). This block will execute when none of the exception block catch the exception.

input:

try:

x = 20

lista = [6, 7, 8]

print(lista[5])

#trying to access out-of-range index

except NameError:

print("variable name is wrong")

except:

print("something else went wrong")

output:

something else went wrong

It is important to note that the generic except block can not be written before the named exception blocks.

## else

The **else** block can be used with the **else** keyword to define a block of code to be executed if no errors were raised. This block executes only if there were no exceptions

input:

try:

x = 20

lista = [6, 7, 8]

print(lista[2])

#nothing wrong with this code

except NameError:

print("variable name is wrong")

except:

print("something else went wrong")

else:

print("nothing went wrong")

output:

8

nothing went wrong

## finally

The finally block, if specified, will be executed regardless if the try block raises an error or not.

input:

def divide(x, y):

try:

result = x/ y

except ZeroDivisionError:

print("division by zero!")

else:

print("result is", result)

finally:

print("executing finally clause")

input:

divide(2,0)

output:

division by zero!

executing finally clause

input:

divide(2,2)

output:

result is 1.0

executing finally clause

## nesting of try-except

Nesting of try-except is allowed and if an exception is not caught in the inner block, its caught in the outer block.

Nesting can be achieved in two ways.

One is to write a try-catch within a catch

Another is to call a function within a try and the exception thrown within the function (if not caught) will be caught in the calling function.

Example 1:

input:

try:

try:

varx = 6/0

except NameError:

print("name is wrong")

except ZeroDivisionError:

print("outer except block")

output:

outer except block

Example 2:

input:

def funca():

try:

print(4/0)

except NameError:

print("name of data is wrong")

input:

def funcb():

try:

funca()

except:

print("something went wrong")

So here we have two functions. Calling funca() will throw us an error (cause divide by zero is not being handled). However when we call funca() within funcb(), ZeroDivisionError will be caught by the external catch (of funcb).

input:

funca()

output:

Traceback (most recent call last):

File "<ipython-input-91-cf9529241c1e>", line 1, in <module>

funca()

File "<ipython-input-86-dd7cc3fdd260>", line 3, in funca

print(4/0)

ZeroDivisionError: division by zero

input:

funcb()

output:

something went wrong

# 8. Files

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

1. Open a file
2. Read or write (perform operation)
3. Close the file

## open

Python has a built-in function open() to open a file. This function returns a file object, also called a handle, as it is used to read or modify the file accordingly.

input:

f = open("test.txt")

# open file in current directory

f = open("C:/Python/sample.txt") # specifying full path

#full path varies in linux, mac & windows

we specify whether we want to read 'r', write 'w' or append 'a' to the file. We also specify if we want to open the file in text mode or binary mode.

The default is reading in text mode. In this mode, we get strings when reading from the file.On the other hand, binary mode returns bytes and this is the mode to be used when dealing with non-text files like image or exe files.

'r' Open a file for reading. (default)

'w' Open a file for writing.

Creates a new file if it does not exist or truncates the file if it exists.

'x' Open a file for exclusive creation. If the file already exists, the operation fails.

'a' Open for appending at the end of the file without truncating it.

Creates a new file if it does not exist.

't' Open in text mode. (default)

'b' Open in binary mode.

'+' Open a file for updating (reading and writing)

input:

f = open("test.txt") # equivalent to 'r' or 'rt'

f = open("test.txt",'w') # write in text mode

f = open("img.bmp",'r+b') # read and write in binary mode

The default encoding differs from platform to platform. open function assumes default text formatting. To be on safer side, including the encoding standard while opening the file is a better way to program.

input:

f = open("test.txt",mode = 'r',encoding = 'utf-8')

Closing a file will free up the resources that were tied with the file and is done using Python close() method.

input:

f.close()

## write

In order to write into a file in Python, we need to open it in write 'w', append 'a' or exclusive creation 'x' mode. We need to be careful with the 'w' mode as it will overwrite into the file if it already exists. All previous data are erased.

f = open("test.txt",'w',encoding = 'utf-8')

f.write("This is my first file\n")

f.write("This file\n\n")

f.write("contains three lines\n")

## read

There are various methods available for reading from a file. We can use the read(size) method to read in size number of data. If size parameter is not specified, it reads and returns up to the end of the file.

input:

f = open("test.txt",'r',encoding = 'utf-8')

f.read(4) # read the first 4 data

output:

'This'

input:

f.read(4) # read the next 4 data

output:

' is '

input:

f.read() # read in the rest till end of file

output:

'my first file\nThis file\ncontains three lines\n'

input:

f.read() # further reading returns empty string

output:

''

We can change our current file cursor (position) using the seek() method.

Similarly, the tell() method returns our current position (in number of bytes).

input:

f.tell() # get the current file position

output:

56

input:

f.seek(0) # bring file cursor to initial position

seek will jump certain bytes (*offset*) from certain (*from*)location (start, current, end)

seek(*offset*,*from*=SEEK\_SET)

*from* is understood as:

**0**: means your reference point is the **beginning** of the file

**1**: means your reference point is the **current** file position

**2**: means your reference point is the **end** of the file

input:

print(f.read()) # read the entire file

output:

This is my first file

This file

contains three lines

One can also store the contents of a file into a string.

The readline() function will read only a line of the file.

readlines() method returns a list of remaining lines of the entire file.

input:

f.readlines()

output:

['This is my first file\n', 'This file\n', 'contains three lines\n']

The following are some of the important methods:

close() Close an open file. It has no effect if the file is already closed.

fileno() Return an integer number (file descriptor) of the file.

flush() Flush the write buffer of the file stream.

isatty() Return True if the file stream is interactive.

read(n) Read atmost n characters from the file.

Reads till end of file if it is negative or None.

readable() Returns True if the file stream can be read from.

readline() Read and return one line from the file.

Reads in at most n bytes if specified.

readlines() Read and return a list of lines from the file.

Reads in at most n bytes/characters if specified.

seek(offset,from=SEEK\_SET)

Change the file position to offset bytes, in reference to from

(start, current, end).

tell() Returns the current cursor location in the file.

truncate() Resize the file stream to size bytes.

If size is not specified, resize to current cursor location.

write(s) Write string s to the file and return the number of characters written.

writelines() Write a list of lines to the file.

# 9. Database connection

## Database

Database is essentially a collections of tables connected with each other through columns. Most database systems support SQL, the Structured Query Language, which is used to create, access and manipulate the data. SQL is used to access data, and also to create and exploit the relationships between the stored data.

Nosql databases like MongoDB are also used in modern practice.

The Python programming language has powerful features for database programming. Python supports various databases like

MySQL

Oracle

SQLite

Sybase

PostgreSQL

MongoDB

Microsoft SQLserver etc.

## SQLite programming

We are going to connect SQLite with Python. SQLite is a lightweight database that can provide a relational database management system with zero-configuration because there is no need to configure or setup anything to use it. Python has a native library for SQLite.

**Steps for SQLite programming with python:**

Step 1.

To use SQLite, we must import sqlite3.

Then create a connection using connect() method and pass the name of the database that we want to access. If there is a file with that name, it will open that file. Otherwise, Python will create a file with the given name.

Step 2.

After this, a cursor object is called to be capable to send commands to the SQL. Cursor is a control structure used to traverse and fetch the records of the database. Cursor has a major role in working with Python. All the commands will be executed using cursor object only.

Step 3

To create a table, or insert values in the database, create an object and write the SQL command in it with being commented.

So let's implement this step by step

## Create SQLite3 Connection

To use SQLite3 in Python we will have to import the sqlite3 module and then create a connection object which will connect us to the database and will let us execute the SQL statements.

A connection object is created using the connect() function

input:

import sqlite3

con = sqlite3.connect('sample.db')

A new file called sample.db’ will be created where our database will be stored. This file will be created in the current directory.

## SQLite3 Cursor

To execute SQLite statements in Python, you need a cursor object. You can create it using the cursor() method.

The SQLite3 cursor is a method of the connection object. To execute the SQLite3 statements, a connection is established at first and then an object of the cursor is created using the connection object as follows:

input:

con = sqlite3.connect('sample.db')

cursorObj = con.cursor()

## Create Table in SQLite3

To create a table in SQLite3, you can use the Create Table query in the execute() method. Consider the following steps:

1. The connection object is created
2. Cursor object is created using the connection object
3. Using cursor object, execute method is called with create table query as the parameter

Let’s create employees with the following attributes:

employees (id, name, salary, department, position, hireDate)

input:

cursorObj.execute("CREATE TABLE employees(id integer PRIMARY KEY, name text, salary real, department text, position text, hireDate text)")

output:

<sqlite3.Cursor at 0x11b26c260>

The output here is optional (some IDEs don’t display anything).

Table may or may not be created until commit() is called.

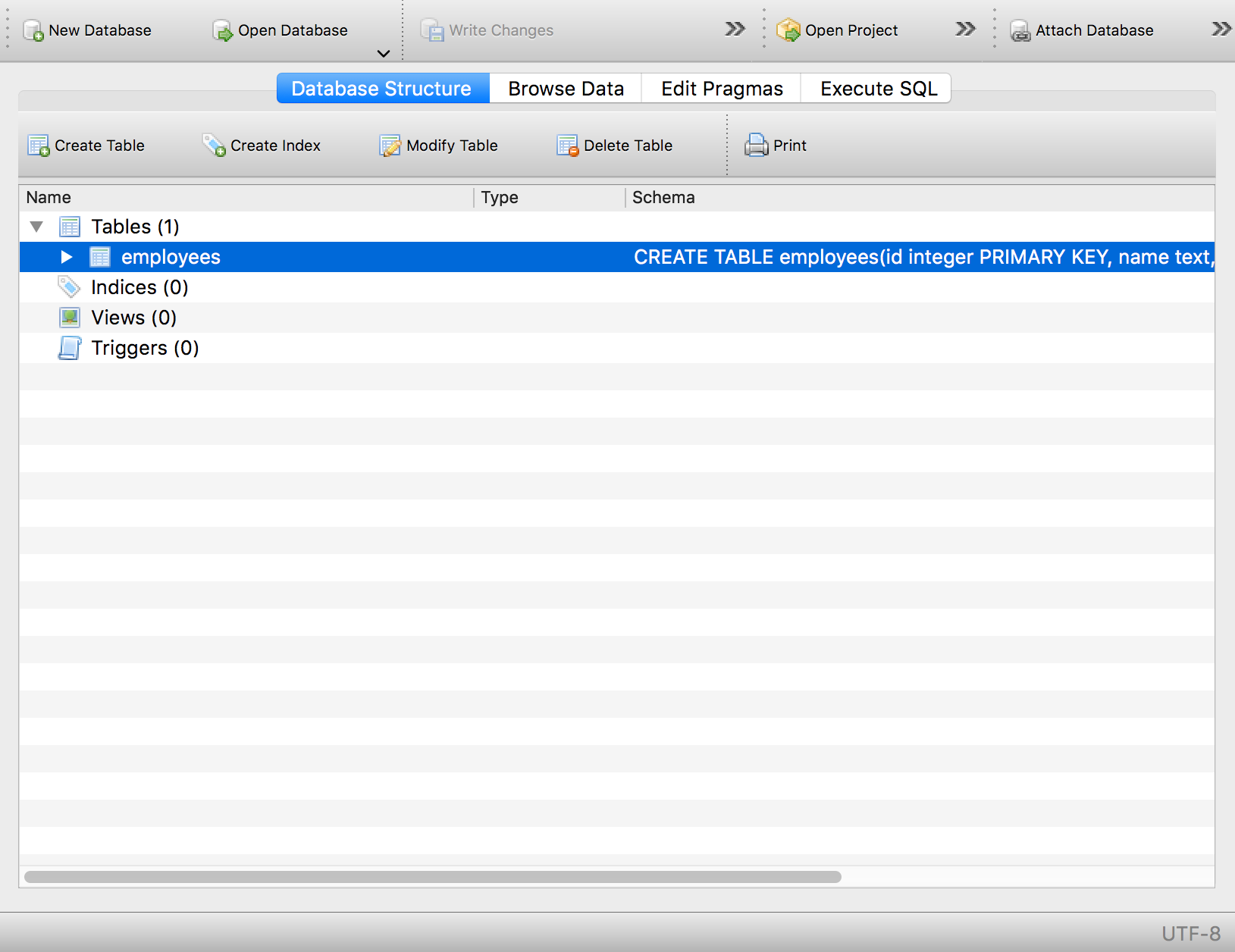
The commit() method saves all the changes we make.

input:

con.commit()

To check if our table is created, we can use any display application for SQLite. I have used the **DB browser** for sqlite to view your table. Its free and available online. You can download it at this link, <https://sqlitebrowser.org/dl/> .

Open sample.db file with the DB browser program and we should see our table:



## Insert in Table (SQLite3)

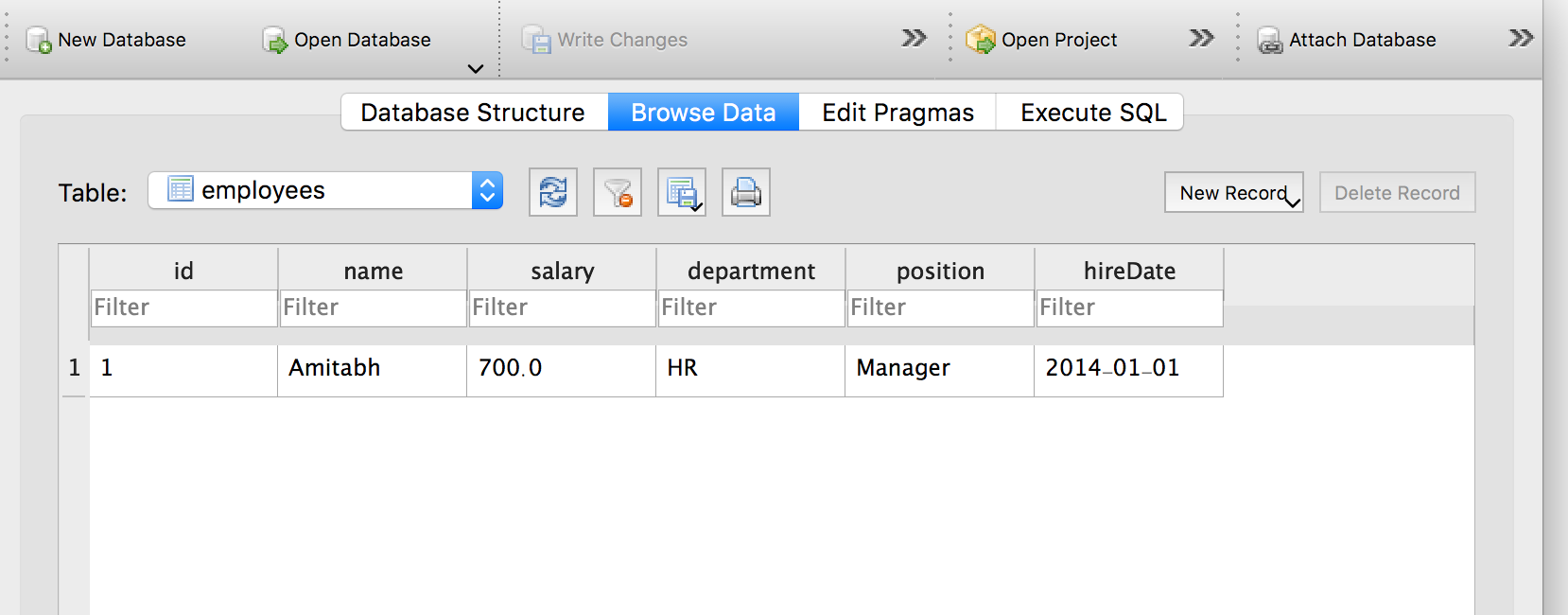
To insert data in a table, we use the INSERT INTO statement. Consider the following line of code:

input:

cursorObj.execute("INSERT INTO employees VALUES(1, 'Amitabh', 700, 'HR', 'Manager', '2014-01-01')")

con.commit()

To check if the data is inserted, click on Browse Data in the DB Browser:



We can also first create a list or a tuple with all the data.

input:

employeeB = (2, 'Rajnikant', 600, 'IT', 'Tech','2016-02-06')

And then pass values/arguments to an INSERT statement in the execute() method. We can use the question mark (?) as a placeholder for each value.

The syntax of the INSERT will be like the following:

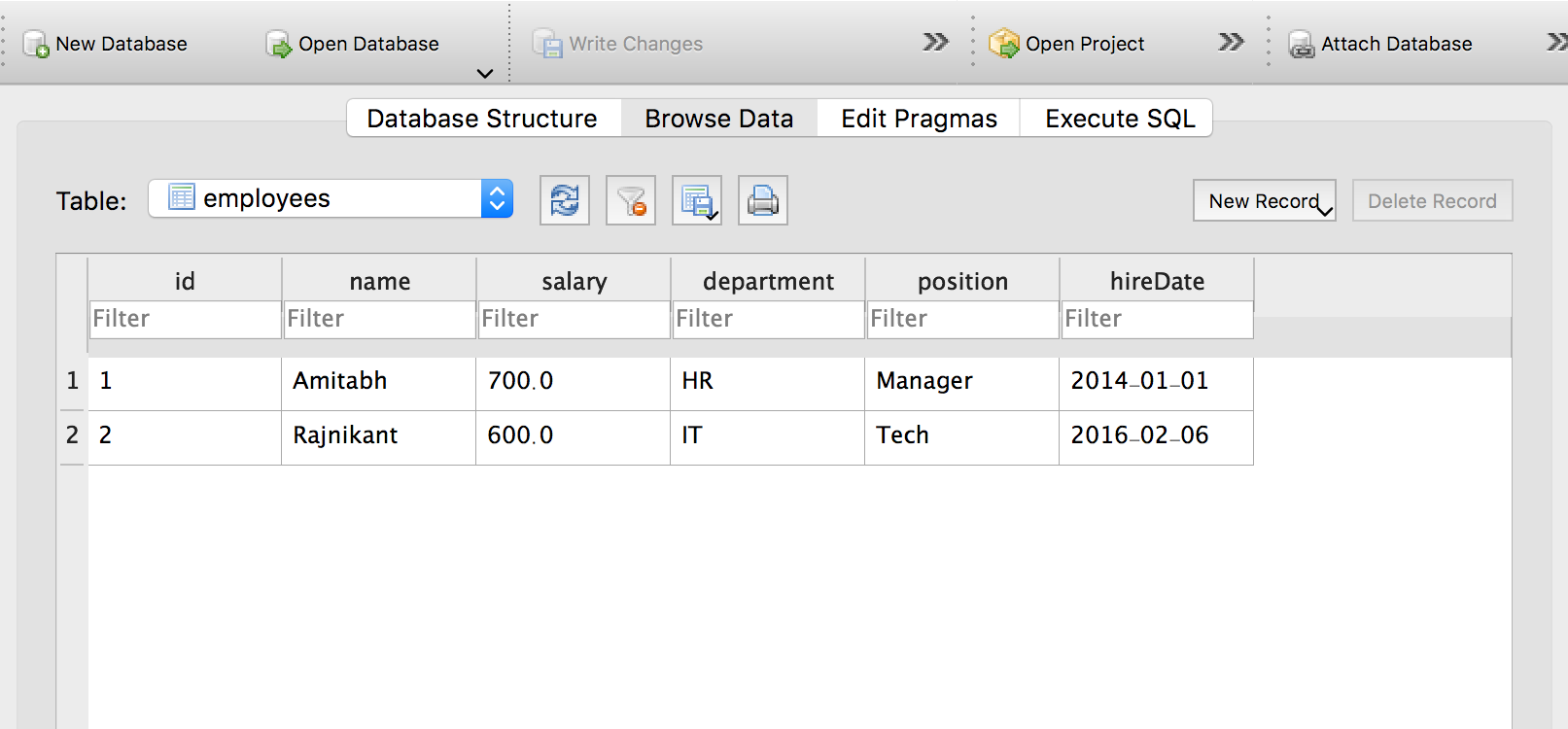
input:

cursorObj.execute('''INSERT INTO employees(id, name, salary, department, position, hireDate) VALUES(?, ?, ?, ?, ?, ?)''', employeeB)

Run commit after insert, to save the changes

input:

con.commit()



## Update Table (SQLite3)

To update the table simply create a connection, then create a cursor object using the connection and finally use the UPDATE statement in the execute() method.

Suppose that we want to update the name of the employee whose id equals 2. For updating, we will use the UPDATE statement and for the employee whose id equals 2. We will use the WHERE clause as a condition to select this employee.

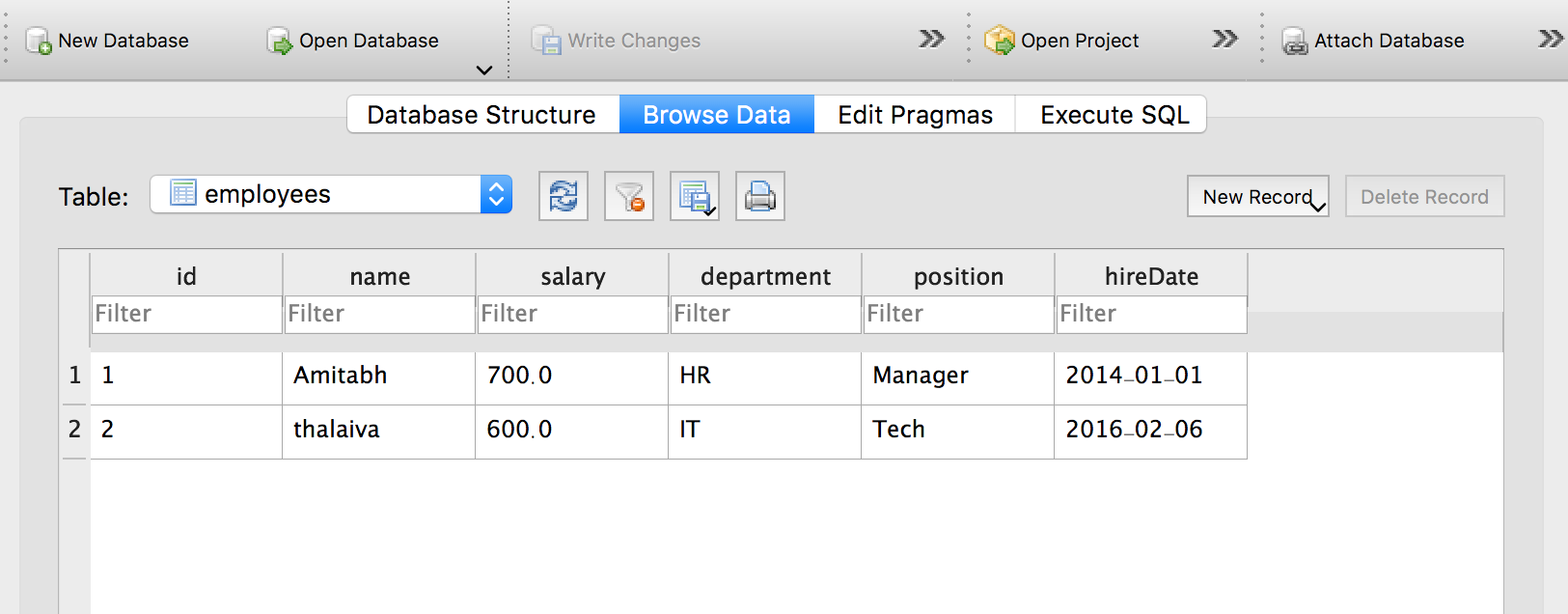
Consider the following code:

input:

cursorObj.execute('UPDATE employees SET name = "thalaiva" where id = 2')

con.commit()

This will change the name from Rajnikant to thalaiva as follows:



## Select statement (SQLite3)

The select statement is used to select data from a particular table. If you want to select all the columns of the data from a table, you can use the asterisk (\*). The SQL syntax for this will be as follows:

select \* from table\_name

In SQLite3, the SELECT statement is executed in the execute method of the cursor object. For example, select all the columns of the employees’ table, run the following code:

cursorObj.execute('SELECT \* FROM employees ')

If you want to select a few columns from a table then specify the columns like the following:

select column1, column2 from tables\_name

For example,

cursorObj.execute('SELECT id, name FROM employees')

The select statement selects the required data from the database table and if you want to fetch the selected data, the fetchall() method of the cursor object is used. This is demonstrated in the next section.

## Fetch all data (SQLite3)

To fetch the data from a database we will execute the SELECT statement and then will use the fetchall() method of the cursor object to store the values into a variable. After that, we will loop through the variable and print all values.

The code will be like this:

input:

cursorObj.execute('SELECT \* FROM employees')

rows = cursorObj.fetchall()

for row in rows:

print(row)

output:

(1, 'Amitabh', 700.0, 'HR', 'Manager', '2014-01-01')

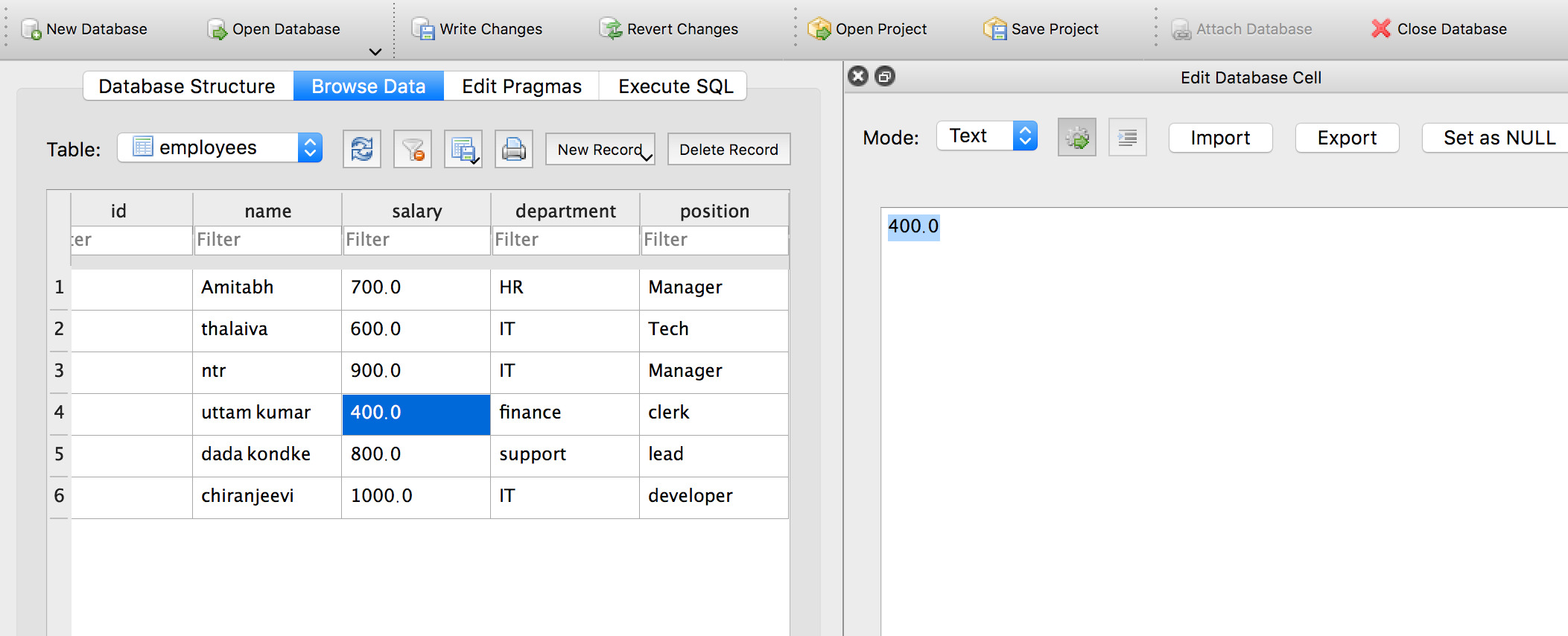
(2, 'thalaiva', 600.0, 'IT', 'Tech', '2016-02-06')

You can also use the fetchall() in one line as follows:

[print(row) for row in cursorObj.fetchall()]

If you want to fetch specific data from the database, you can use the WHERE clause. For example, we want to fetch the ids and names of those employees whose salary is greater than 800.

For this, let’s populate our table with more rows, then execute our query. We can use the insert statement to populate the data or you can enter them manually in the DB browser program.



Let us fetch the id & names of all the employees whose salary is less than 900

input:

cursorObj.execute('SELECT id, name FROM employees WHERE salary < 900.0')

rows = cursorObj.fetchall()

for row in rows:

print(row)

output:

(1, 'Amitabh')

(2, 'thalaiva')

(4, 'uttam kumar')

(5, 'dada kondke')

## SQLite3 rowcount

The SQLite3 rowcount is used to return the number of rows that are affected or selected by the latest executed SQL query.

When we use rowcount with the SELECT statement, -1 will be returned as how many rows are selected is unknown until they are all fetched.

Consider the example below:

input:

print(cursorObj.execute('SELECT \* FROM employees').rowcount)

output:

-1

Therefore, to get the row count, you need to fetch all the data, and then get the length of the result:

input:

rows = cursorObj.fetchall()

print(len(rows))

output:

6

## List tables (SQLite3)

To list all tables in a SQLite3 database, you should query sqlite\_master table and then use the fetchall() to fetch the results from the SELECT statement.

The sqlite\_master is the master table in SQLite3 which stores all tables.

input:

cursorObj.execute('SELECT name from sqlite\_master where type= "table"')

print(cursorObj.fetchall())

output:

[('employees',)]

## Check if table exists or not (SQLite3)

When creating a table, we should make sure that the table is not already existed. Similarly, when removing/ deleting a table, the table should exist.

To check if the table doesn’t already exist we use “if not exists” with the CREATE TABLE statement (SQL) as follows:

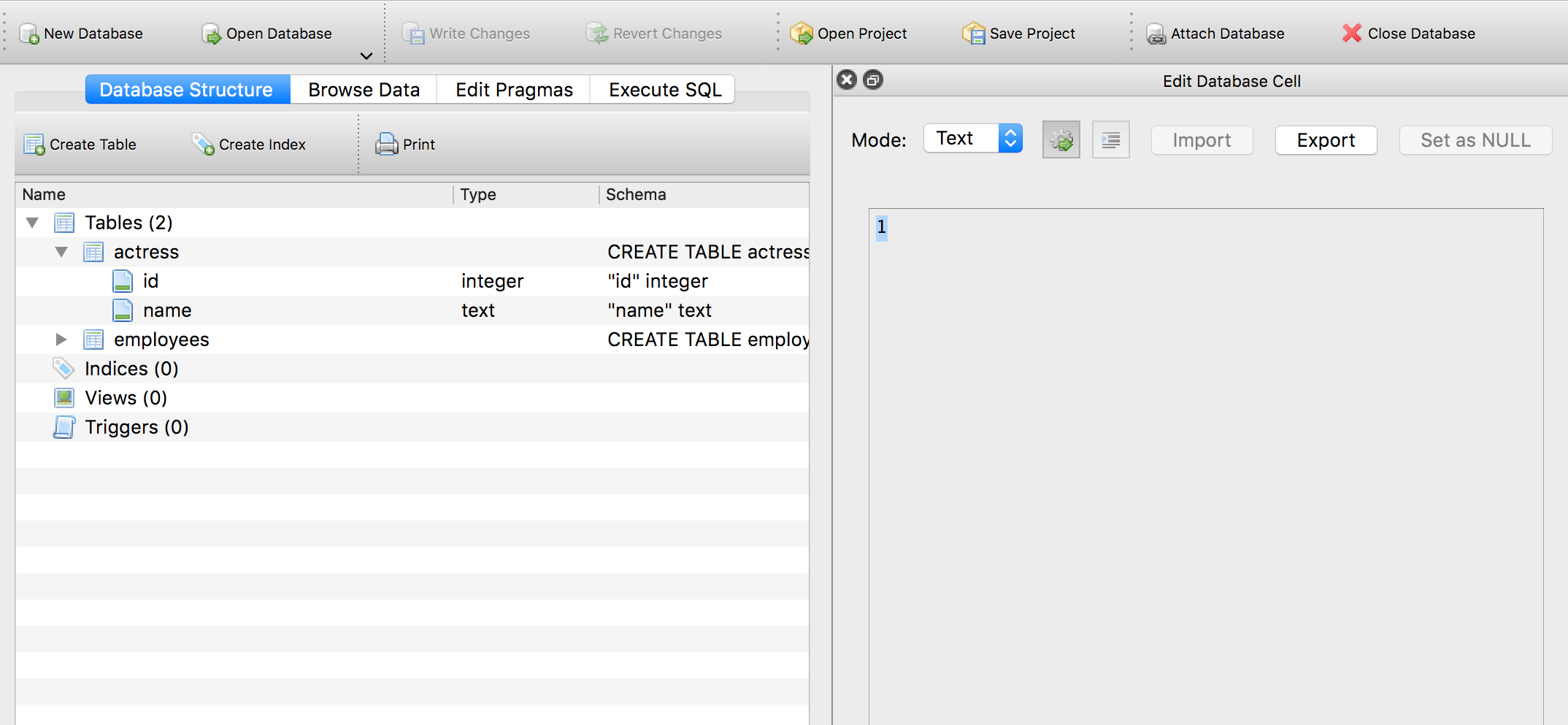
create table if not exists table\_name (column1, column2, …, columnN)

For example let's create a table called as actress

input:

cursorObj.execute('create table if not exists actress(id integer, name text)')

con.commit()

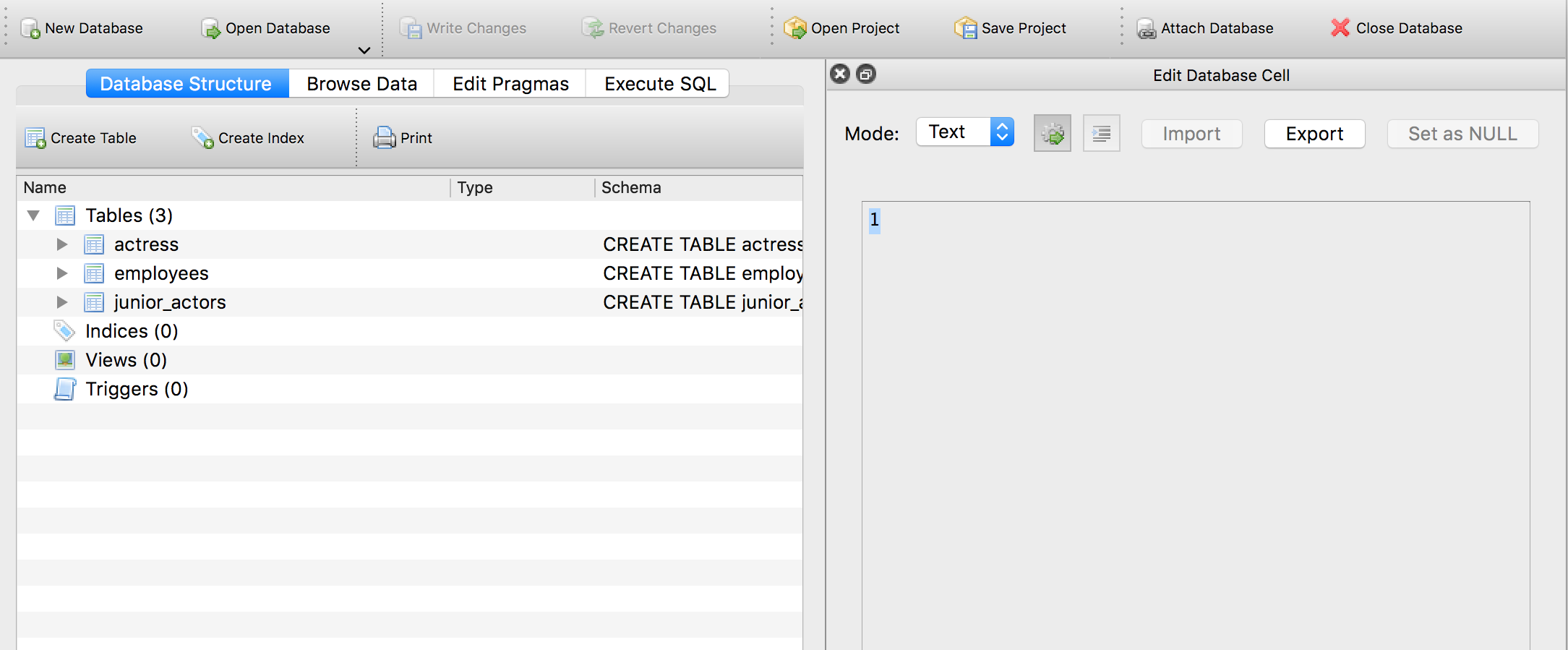


Let us create another table ‘junior\_actors’

input:

cursorObj.execute('create table if not exists actress(id integer, name text)')

con.commit()

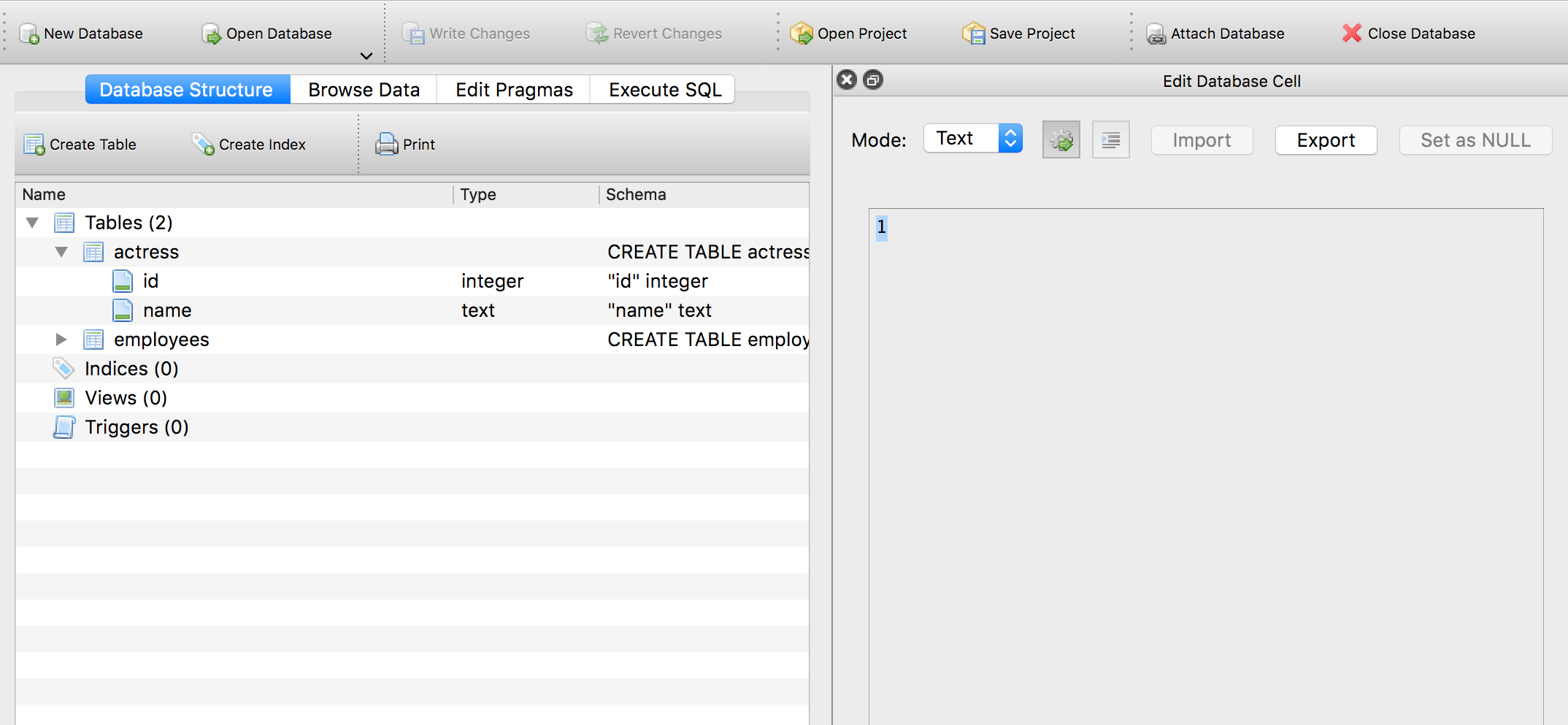


Similarly, to check if the table exists when deleting we use “if exists” with the DROP TABLE statement as follows:

drop table if exists table\_name

For example, to drop the ‘junior\_actors’ table

cursorObj.execute('drop table if exists "junior\_actors"')



We can also check if the table we want to access exists or not by executing the following query:

input:

cursorObj.execute('SELECT name from sqlite\_master WHERE type = "table" AND name = "employees"')

print(cursorObj.fetchall())

If the employees’ table exists, its name will be returned as follows:

output:

[('employees',)]

If the table name we specified doesn’t exist, an empty list will be returned:

input:

cursorObj.execute('SELECT name from sqlite\_master WHERE type = "table" AND name = "junior\_actors"')

print(cursorObj.fetchall())

output:

[]

## Drop Table (SQLite3)

We can drop/delete a table using the DROP statement. The syntax of DROP SQL statement is as follows:

drop table table\_name

To drop a table, the table should exist in the database. Therefore, it is recommended to use “if exists” with the drop statement as follows:

drop table if exists table\_name

cursorObj.execute('DROP table if exists actress')

## SQLite3 Executemany (Bulk insert)

We can use the executemany statement to insert multiple rows at once.

The list called **data** has 5 values, each value is a tuple of size two (two columns).

data = [(1, "nargis"), (2, "sridevi"), (3, "madhuri"), (4, "meena"), (5, "alia")]

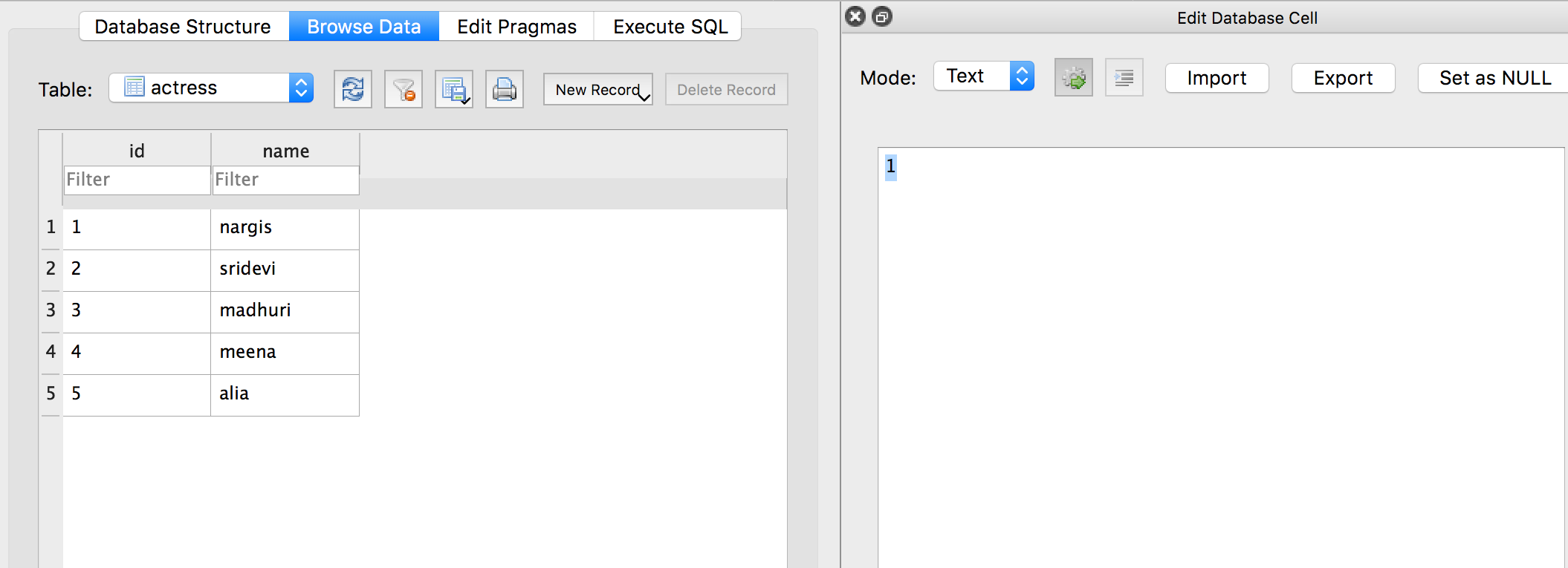
This variable is passed to the executemany() method along with the query.

We will use the placeholder to pass the values.

cursorObj.executemany("INSERT INTO actress VALUES(?, ?)", data)

con.commit()

The above code will generate the following result:



## Close Connection (SQLite3)

Once you are done with your database, it is a good practice to close the connection. The connection can be closed by using the close() method.

To close a connection, use the connection object and call the close() method as follows:

input:

con = sqlite3.connect('sample.db')

#program statements

con.close()

# Appendix A: Executing python files on console

To load an execute a python file, functions & method differs for different IDEs.

## Spyder

input:

runfile(“/path/file.py”)

## idle

**GUI:**

Start IDLE.

Go to File menu and click on Open.

Select your file (name.py)

The file opens in another window

On that window go to Run and click Run Module.

**command based:**

input:

exec(open(“/path/file.py”).read())

or

if it's in the same directory

input:

import file.py

exec(“file.py”)

# Appendix B: Getting Help

## Getting help

Help on python functions can be gained from the console environment by using **help** function. For example to get help on **print** :

input:

help(print)

output:

Help on built-in function print in module builtins:

print(...)

print(value, ..., sep=' ', end='\n', file=sys.stdout,

flush=False)

Prints the values to a stream, or to sys.stdout by

default.

Optional keyword arguments:

file: a file-like object (stream); defaults to the

current sys.stdout.

sep: string inserted between values, default a space.

end: string appended after the last value, default a

newline.

flush: whether to forcibly flush the stream.

dir() will list all the functions within the modules (inbuilt & user defined)

just typing help() will take us to the inbuilt ‘help console’

input:

help()

Welcome to Python 3.7's help utility!

output:

---

---

---

help>

within this console we can type the name of function/identifier or keyword for getting details about it

input:

help> print

Help on built-in function print in module builtins:

print(...)

print(value, ..., sep=' ', end='\n', file=sys.stdout,

flush=False)

Prints the values to a stream, or to sys.stdout by

default.

Optional keyword arguments:

file: a file-like object (stream); defaults to the

current sys.stdout.

sep: string inserted between values, default a space.

end: string appended after the last value, default a

newline.

flush: whether to forcibly flush the stream.

to get help for functions within a module, its required to mention the function name with the module name. For example if we need help for the function ‘getcwd’ from os module then the right way to get help is os.getcwd()

input:

help> getcwd

output:

No Python documentation found for 'getcwd'.

Use help() to get the interactive help utility.

Use help(str) for help on the str class.

input:

help> os.getcwd

Help on built-in function getcwd in os:

output:

os.getcwd = getcwd()

Return a unicode string representing the current working directory.

quit will exit the help console

help> quit

Help can be used to get help for the customised modules created by the programmers too; the **\_\_doc\_\_** string is printed when help is called for