UNIT 5

Python Files I/O

Files and exceptions

While a program is running, its data is in memory. When the program ends, or the computer shuts down, data in memory disappears. To store data permanently, you have to put it in a file. Files are usually stored on a hard drive, floppy drive, or CD-ROM. When there are a large number of files, they are often organized into directories (also called "folders"). Each file is identified by a unique name, or a combination of a file name and a directory name. By reading and writing files, programs can exchange information with each other and generate printable formats like PDF.

Working with files is a lot like working with books. To use a book, you have to open it. When you're done, you have to close it. While the book is open, you can either write in it or read from it. In either case, you know where you are in the book. Most of the time, you read the whole book in its natural order, but you can also skip around. All of this applies to files as well. To open a file, you specify its name and indicate whether you want to read or write.

Opening a file creates a file object. In this example, the variable f refers to the new file object.

```
>>> f = open("test.dat","w")
>>> print f
<open file 'test.dat', mode 'w' at fe820>
```

The open function takes two arguments. The first is the name of the file, and the second is the mode. Mode "w" means that we are opening the file for writing. If there is no file named test.dat, it will be created. If there already is one, it will be replaced by the file we are writing. When we print the file object, we see the name of the file, the mode, and the location of the object.

To put data in the file we invoke the write method on the file object:

```
>>> f.write("Now is the time")
>>> f.write("to close the file")
```

Closing the file tells the system that we are done writing and makes the file available for reading:

```
>>> f.close()
```

Now we can open the file again, this time for reading, and read the contents into a string. This time, the mode argument is "r" for reading:

```
>>> f = open("test.dat","r")
```

If we try to open a file that doesn't exist, we get an error:

```
>>> f = open("test.cat","r")
```

IOError: [Errno 2] No such file or directory: 'test.cat'

Not surprisingly, the read method reads data from the file. With no arguments, it reads the entire contents of the file:

```
>>> text = f.read()
>>> print text
```

Now is the timeto close the file. There is no space between “time” and “to” because we did not write a space between the strings. read can also take an argument that indicates how many characters to read:

```
>>> f = open("test.dat","r")
>>> print f.read(5)
```

Now I If not enough characters are left in the file, read returns the remaining characters. When we get to the end of the file, read returns the empty string:

```
>>> print f.read(1000006)
s the timeto close the file
>>> print f.read()
```

The following function copies a file, reading and writing up to fifty characters at a time. The first argument is the name of the original file; the second is the name of the new file:

```
def copyFile(oldFile, newFile):
f1 = open(oldFile, "r")
f2 = open(newFile, "w")
while True:
text = f1.read(50)
if text == "":
break
f2.write(text)
f1.close()
f2.close()
return
```

The break statement is new. Executing it breaks out of the loop; the flow of execution moves to the first statement after the loop. In this example, the while loop is infinite because the value True is always true. The *only* way to get out of the loop is to execute break, which happens when text is the empty string, which happens when we get to the end of the file.

Text files

A text file is a file that contains printable characters and whitespace, organized into lines separated by newline characters. Since Python is specifically designed to process text files, it provides methods that make the job easy. To demonstrate, we’ll create a text file with three lines of text separated by newlines:

```
>>> f = open("test.dat","w")
>>> f.write("line one\nline two\nline three\n")
>>> f.close()
```

The readline method reads all the characters up to and including the next newline character:

```
>>> f = open("test.dat","r")
>>> print f.readline()
line one

>>>
```

readlines returns all of the remaining lines as a list of strings:

```
>>> print f.readlines()
```

['line two\012', 'line three\012']
In this case, the output is in list format, which means that the strings appear with quotation marks and the newline character appears as the escape sequence 012. At the end of the file, readline returns the empty string and readlines returns the empty list:

```
>>> print f.readline()
>>> print f.readlines()
```

The following is an example of a line-processing program. filterFile makes a copy of oldFile, omitting any lines that begin with #:

```
def filterFile(oldFile, newFile):
f1 = open(oldFile, "r")
f2 = open(newFile, "w")
while True:
text = f1.readline()
if text == "":
break
if text[0] == '#':
continue
f2.write(text)
f1.close()
f2.close()
return
```

The continue statement ends the current iteration of the loop, but continues looping. The flow of execution moves to the top of the loop, checks the condition, and proceeds accordingly. Thus, if text is the empty string, the loop exits. If the first character of text is a hash mark, the flow of execution goes to the top of the loop. Only if both conditions fail do we copy text into the new file.

Writing variables

The argument of write has to be a string, so if we want to put other values in a file, we have to convert them to strings first. The easiest way to do that is with the str function:

```
>>> x = 52
>>> f.write (str(x))
```

An alternative is to use the format operator %. When applied to integers, % is the modulus operator. But when the first operand is a string, % is the format operator. The first operand is the format string, and the second operand is a tuple of expressions. The result is a string that contains the values of the expressions, formatted according to the format string. As a simple example, the format sequence "%d" means that the first expression in the tuple should be formatted as an integer. Here the letter *d* stands for "decimal":

```
>>> cars = 52
>>> "%d" % cars
'52'
```

The result is the string '52', which is not to be confused with the integer value 52. A format sequence can appear anywhere in the format string, so we can embed a value in a sentence:

```
>>> cars = 52
>>> "In July we sold %d cars." % cars
```

’In July we sold 52 cars.’ The format sequence "%f" formats the next item in the tuple as a floating-point number, and "%s" formats the next item as a string:
>>> "In %d days we made %f million %s." % (34,6.1,’dollars’)
’In 34 days we made 6.100000 million dollars.’ By default, the floating-point format prints six decimal places. The number of expressions in the tuple has to match the number of format sequences in the string. Also, the types of the expressions have to match the format sequences:

>>> "%d %d %d" % (1,2)
TypeError: not enough arguments for format string

>>> "%d" % ’dollars’
TypeError: illegal argument type for built-in operation

In the first example, there aren’t enough expressions; in the second, the expression is the wrong type. For more control over the format of numbers, we can specify the number of digits as part of the format sequence:
>>> "%6d" % 62
’ 62’
>>> "%12f" % 6.1
’ 6.100000’

The number after the percent sign is the minimum number of spaces the number will take up. If the value provided takes fewer digits, leading spaces are added. If the number of spaces is negative, trailing spaces are added:
>>> "%-6d" % 62
’62 ’
For floating-point numbers, we can also specify the number of digits after the decimal point:
>>> "%12.2f" % 6.1
’ 6.10’
In this example, the result takes up twelve spaces and includes two digits after the decimal. This format is useful for printing dollar amounts with the decimal points aligned. For example, imagine a dictionary that contains student names as keys and hourly wages as values. Here is a function that prints the contents of the dictionary as a formatted report:
```
def report (wages) :
students = wages.keys()
students.sort()
for student in students :
print "%-20s %12.2f" % (student, wages[student])
```

To test this function, we’ll create a small dictionary and print the contents:
>>> wages = {’mary’: 6.23, ’joe’: 5.45, ’joshua’: 4.25}
>>> report (wages)
joe 5.45
joshua 4.25
mary 6.23

By controlling the width of each value, we guarantee that the columns will line up, as long as the names contain fewer than twenty-one characters and the wages are less than one billion dollars an hour.

Directories

When you create a new file by opening it and writing, the new file goes in the current directory (wherever you were when you ran the program). Similarly, when you open a file for reading, Python looks for it in the current directory. If you want to open a file somewhere else, you have to specify the path to the file, which is the name of the directory (or folder) where the file is located:

```
>>> f = open("/usr/share/dict/words","r")
>>> print f.readline()
Aarhus
```

This example opens a file named words that resides in a directory named dict, which resides in share, which resides in usr, which resides in the top-level directory of the system, called /. You cannot use / as part of a filename; it is reserved as a delimiter between directory and filenames. The file /usr/share/dict/words contain a list of words in alphabetical order, of which the first is the name of a Danish university.

Pickling

In order to put values into a file, you have to convert them to strings. You have already seen how to do that with str:

```
>>> f.write (str(12.3))
>>> f.write (str([1,2,3]))
```

The problem is that when you read the value back, you get a string. The original type information has been lost. In fact, you can't even tell where one value ends and the next begins:

```
>>> f.readline()
'12.3[1, 2, 3]'
```

The solution is pickling, so called because it "preserves" data structures. The pickle module contains the necessary commands. To use it, import pickle and then open the file in the usual way:

```
>>> import pickle
>>> f = open("test.pck","w")
```

To store a data structure, use the dump method and then close the file in the usual way:

```
>>> pickle.dump(12.3, f)
>>> pickle.dump([1,2,3], f)
>>> f.close()
```

Then we can open the file for reading and load the data structures we dumped:

```
>>> f = open("test.pck","r")
>>> x = pickle.load(f)
>>> x
```

12.3
>>> type(x)
<type ’float’>
>>> y = pickle.load(f)
>>> y
[1, 2, 3]
>>> type(y)
<type ’list’>
Each time we invoke load, we get a single value from the file, complete with its original type.

Exceptions

Whenever a runtime error occurs, it creates an exception. Usually, the program stops and Python prints an error message. For example, dividing by zero creates an exception:
>>> print 55/0
ZeroDivisionError: integer division or modulo So does accessing a nonexistent list item:
>>> a = []
>>> print a[5]
IndexError: list index out of range Or accessing a key that isn’t in the dictionary:
>>> b = {}
>>> print b[’what’]
KeyError: what Or trying to open a nonexistent file:
>>> f = open("Idontexist", "r")
IOError: [Errno 2] No such file or directory: ’Idontexist’

In each case, the error message has two parts: the type of error before the colon, and specifics about the error after the colon. Normally Python also prints a traceback of where the program was, but we have omitted that from the examples. Sometimes we want to execute an operation that could cause an exception, but we don’t want the program to stop. We can handle the exception using the try and except statements. For example, we might prompt the user for the name of a file and then try to open it. If the file doesn’t exist, we don’t want the program to crash; we want to handle the exception:

```
filename = raw_input(’Enter a file name: ’)
try:
f = open (filename, "r")
except IOError:
print ’There is no file named’, filename
```

The try statement executes the statements in the first block. If no exceptions occur, it ignores the except statement. If an exception of type IOError occurs, it executes the statements in the except branch and then continues. We can encapsulate this capability in a function: exists takes a filename and returns true if the file exists, false if it doesn’t:

```
def exists(filename):
try:
f = open(filename)
```

f.close()
return True
except IOError:
return False
You can use multiple except blocks to handle different kinds of exceptions. The *Python Reference Manual* has the details. If your program detects an error condition, you can make it raise an exception. Here is an example that gets input from the user and checks for the value 17. Assuming that 17 is not valid input for some reason, we raise an exception.
def inputNumber () :
x = input (’Pick a number: ’)
126 Files and exceptions
if x == 17 :
raise ValueError, ’17 is a bad number’
return x
The raise statement takes two arguments: the exception type and specific information about the error. ValueError is one of the exception types Python provides for a variety of occasions. Other examples include TypeError, KeyError, and my favorite, NotImplementedError. If the function that called inputNumber handles the error, then the program can continue; otherwise, Python prints the error message and exits:
>>> inputNumber ()
Pick a number: 17
ValueError: 17 is a bad number
The error message includes the exception type and the additional information you provided.
As an exercise, write a function that uses input Number *to input a number from the keyboard and that handles the* ValueError *exception.*

Python MySQL Database Access

The Python standard for database interfaces is the Python DB-API. Most Python database interfaces adhere to this standard.

You can choose the right database for your application. Python Database API supports a wide range of database servers such as −

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

Here is the list of available Python database interfaces: Python Database Interfaces and APIs .You must download a separate DB API module for each database you need to access. For example, if you need to access an Oracle database as well as a MySQL database, you must download both the Oracle and the MySQL database modules.

The DB API provides a minimal standard for working with databases using Python structures and syntax wherever possible. This API includes the following:

- Importing the API module.
- Acquiring a connection with the database.
- Issuing SQL statements and stored procedures.
- Closing the connection

We would learn all the concepts using MySQL, so let us talk about MySQLdb module.

What is 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.

How do I Install MySQLdb?

Before proceeding, you make sure you have MySQLdb installed on your machine. Just type the following in your Python script and execute it:

```
#!/usr/bin/python

import MySQLdb
```

If it produces the following result, then it means MySQLdb module is not installed:

```
Traceback (most recent call last):
  File "test.py", line 3, in <module>
    import MySQLdb
ImportError: No module named MySQLdb
```

To install MySQLdb module, download it from MySQLdb Download page and proceed as follows:

```
$ gunzip MySQL-python-1.2.2.tar.gz
$ tar -xvf MySQL-python-1.2.2.tar
$ cd MySQL-python-1.2.2
$ python setup.py build
$ python setup.py install
```

Note: Make sure you have root privilege to install above module.

Database Connection

Before connecting to a MySQL database, make sure of the followings −

- You have created a database TESTDB.
- You have created a table EMPLOYEE in TESTDB.
- This table has fields FIRST_NAME, LAST_NAME, AGE, SEX and INCOME.
- User ID "testuser" and password "test123" are set to access TESTDB.
- Python module MySQLdb is installed properly on your machine.
- You have gone through MySQL tutorial to understand MySQL Basics.

Example

Following is the example of connecting with MySQL database "TESTDB"

```
#!/usr/bin/python

import MySQLdb

# Open database connection
db = MySQLdb.connect("localhost","testuser","test123","TESTDB" )

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

# execute SQL query using execute() method.
cursor.execute("SELECT VERSION()")

# Fetch a single row using fetchone() method.
data = cursor.fetchone()

print "Database version : %s " % data

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

While running this script, it is producing the following result in my Linux machine.

```
Database version : 5.0.45
```

If a connection is established with the datasource, then a Connection Object is returned and saved into **db** for further use, otherwise **db** is set to None. Next, **db** object is used to create a **cursor** object, which in turn is used to execute SQL queries. Finally, before coming out, it ensures that database connection is closed and resources are released.

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.

Example

Let us create Database table EMPLOYEE:

```
#!/usr/bin/python

import MySQLdb

# Open database connection
db = MySQLdb.connect("localhost","testuser","test123","TESTDB" )

# 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.

Example

The following example, executes SQL *INSERT* statement to create a record into EMPLOYEE table −

```
#!/usr/bin/python

import MySQLdb

# Open database connection
db = MySQLdb.connect("localhost","testuser","test123","TESTDB" )

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

# Prepare SQL query to INSERT a record into the database.
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()
```

Above example can be written as follows to create SQL queries dynamically −

```
#!/usr/bin/python

import MySQLdb

# Open database connection
db = MySQLdb.connect("localhost","testuser","test123","TESTDB" )

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

# Prepare SQL query to INSERT a record into the database.
sql = "INSERT INTO EMPLOYEE(FIRST_NAME, \
       LAST_NAME, AGE, SEX, INCOME) \
       VALUES ('%s', '%s', '%d', '%c', '%d' )" % \
       ('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()
```

Example

Following code segment is another form of execution where you can pass parameters directly −

```
.................................
user_id = "test123"
password = "password"

con.execute('insert into Login values("%s", "%s")' % \
            (user_id, password))
.................................
```

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.

Example

The following procedure queries all the records from EMPLOYEE table having salary more than 1000 −

```
#!/usr/bin/python

import MySQLdb

# Open database connection
db = MySQLdb.connect("localhost","testuser","test123","TESTDB" )

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

# Prepare SQL query to INSERT a record into the database.
sql = "SELECT * FROM EMPLOYEE \
       WHERE INCOME > '%d'" % (1000)
try:
   # Execute the SQL command
   cursor.execute(sql)
   # Fetch all the rows in a list of lists.
   results = cursor.fetchall()
   for row in results:
      fname = row[0]
      lname = row[1]
      age = row[2]
      sex = row[3]
      income = row[4]
      # Now print fetched result
      print "fname=%s,lname=%s,age=%d,sex=%s,income=%d" % \
             (fname, lname, age, sex, income )
except:
   print "Error: unable to fecth data"

# disconnect from server
```

```
db.close()
```

This will produce the following result −

```
fname=Mac, lname=Mohan, age=20, sex=M, income=2000
```

Update Operation

UPDATE Operation on any database means to update one or more records, which are already available in the database.

The following procedure updates all the records having SEX as '**M**'. Here, we increase AGE of all the males by one year.

Example

```
#!/usr/bin/python

import MySQLdb

# Open database connection
db = MySQLdb.connect("localhost","testuser","test123","TESTDB" )

# 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 = '%c'" % ('M')
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()
```

DELETE Operation

DELETE operation is required when you want to delete some records from your database. Following is the procedure to delete all the records from EMPLOYEE where AGE is more than 20 −

Example

```
#!/usr/bin/python
```

```
import MySQLdb

# Open database connection
db = MySQLdb.connect("localhost","testuser","test123","TESTDB" )

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

# Prepare SQL query to DELETE required records
sql = "DELETE FROM EMPLOYEE WHERE AGE > '%d'" % (20)
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()
```

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.

Example

You already know how to implement transactions. Here is again similar example −

```
# Prepare SQL query to DELETE required records
sql = "DELETE FROM EMPLOYEE WHERE AGE > '%d'" % (20)
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()
```

COMMIT Operation

Commit is the operation, which gives a green signal to database to finalize the changes, and after this operation, no change can be reverted back.

Here is a simple example to call **commit** method.

```
db.commit()
```

ROLLBACK Operation

If you are not satisfied with one or more of the changes and you want to revert back those changes completely, then use **rollback()** method.

Here is a simple example to call **rollback()** method.

```
db.rollback()
```

Disconnecting Database

To disconnect Database connection, use close() method.

```
db.close()
```

If the connection to a database is closed by the user with the close() method, any outstanding transactions are rolled back by the DB. However, instead of depending on any of DB lower level implementation details, your application would be better off calling commit or rollback explicitly.

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.

Exception	**Description**
Warning	Used for non-fatal issues. Must subclass StandardError.
Error	Base class for errors. Must subclass StandardError.
InterfaceError	Used for errors in the database module, not the database itself. Must subclass

	Error.
DatabaseError	Used for errors in the database. Must subclass Error.
DataError	Subclass of DatabaseError that refers to errors in the data.
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.
IntegrityError	Subclass of DatabaseError for situations that would damage the relational integrity, such as uniqueness constraints or foreign keys.
InternalError	Subclass of DatabaseError that refers to errors internal to the database module, such as a cursor no longer being active.
ProgrammingError	Subclass of DatabaseError that refers to errors such as a bad table name and other things that can safely be blamed on you.
NotSupportedError	Subclass of DatabaseError that refers to trying to call unsupported functionality.

Your Python scripts should handle these errors, but before using any of the above exceptions, make sure your MySQLdb has support for that exception. You can get more information about them by reading the DB API 2.0 specification.