04-External-Programs-and-Databases

November 21, 2019

1 Executing an external program

- `os.system()` is not secure. Shell executes the program.
- `subprocess` module is more secure, it provides pipe functionality
- `Popen([fullpath,arg1, arg2, arg3, ...])` creates a subprocess executing the program. Program `stdin` and `stdout` is the terminal.
- `Popen` has `stdin, stdout,stderr` paramaters to control program input, output and error.

```
In [3]: from subprocess import Popen,PIPE
```

```
In [4]: # output goes to terminal (in Jupyter it is in server, you cannot see it)
        p=Popen(["/bin/ls","-l"])
        p.wait()
```

```
Out[4]: 0
```

1.1 Getting input from a file, output to a file

```
In [6]: # gets input from the file , outputs to another file
        fp = open("/etc/protocols","r")
        ofp = open("testfile.txt","w")
        p=Popen(["/bin/grep","v6"], stdin=fp, stdout=ofp)
        p.wait()
        # check testfile.txt on root of Jupyter.
```

```
Out[6]: 0
```

1.2 Pipes

Pipes are virtual communication channels among the programs. They are used as file objects. If data is written on one end, it can be read from the other end. subprocess.PIPE creates a pipe object when it is used in `stdin, stdout, stderr` parameter.

Unix `man ls | grep modification` is an example of a pipe.

Getting output of a program, giving input to it:

```
In [29]: p = Popen(["/bin/ls", "-l"], stdout=PIPE)
         print(p.stdout.readlines())
         p.wait()
```

```
[b'total 1632\n', b'-rw-r--r--  1 onur onur  13273 Nov  6 23:21 CEng445-oct-10.ipynb\n', b'-rw-r
```

```
Out[29]: 0
```

```
In [30]: p = Popen(["/usr/bin/tr","/a-z/","/A-Z/"], stdin=PIPE, stdout=PIPE)
         p.stdin.write(b'hello /usr/bin/tr\n')
         p.stdin.write(b'please capitilize these\n')
         p.stdin.close()
         for line in p.stdout:
             print(line.decode(),end='')
         p.wait()
```

```
HELLO /USR/BIN/TR
PLEASE CAPITILIZE THESE
```

```
Out[30]: 0
```

1.3 Chain commands

```
In [7]: # man ls | grep modification
        print("-- man ls | grep modification --")

        p = Popen(["/usr/bin/man","ls"], stdout = PIPE)
        q = Popen(["/bin/grep","modification"], stdin = p.stdout, stdout = PIPE)
        for line in q.stdout:
            print(line.decode(), end='')
        p.wait()
        q.wait()

        print("-- man ls | grep modification | cat -n --")

        #man ls | grep modification | cat -n
        p = Popen(["/usr/bin/man","ls"], stdout = PIPE)
        q = Popen(["/bin/grep","modification"], stdin = p.stdout, stdout = PIPE)
        r = Popen(["/bin/cat","-n"], stdin = q.stdout, stdout = PIPE)
        for line in r.stdout:
            print(line.decode(), end='')
        p.wait()
        q.wait()
        r.wait()
```

```
-- man ls | grep modification --
       -c     with -lt: sort by, and show, ctime (time of last modification of
              with -l, show time as WORD instead of default modification time:
       -t     sort by modification time, newest first
-- man ls | grep modification | cat -n --
     1               -c     with -lt: sort by, and show, ctime (time of last modification of
```

```
     2                      with -l, show time as WORD instead of default modification time:
     3              -t      sort by modification time, newest first
```

```
Out[7]: 0
```

Pipe objects is alive until there is still a reader or writer for that pipe. A PIPE created as a `stdin` pamarameter is automatically opened for writing by the current process (one calling Popen). The readers of the pipe (process with `stdin` parameter) gets EOF when the last reader closes the pipe.

Following is a multiple writer example. If current process does not call `p.stdin.close()`, `grep terminal` process will wait until it.

```
In [11]: p = Popen(["/bin/grep", "terminal"], stdin = PIPE, stdout = PIPE)
         q1 = Popen(["/usr/bin/man","bash"], stdout = p.stdin)
         q2 = Popen(["/usr/bin/man","ls"], stdout = p.stdin)
         q3 = Popen(["/usr/bin/man","ssh"], stdout = p.stdin)

         # close the unused input pipes
         p.stdin.close()

         for line in p.stdout:
             print(line.decode(), end='')

         q1.wait()
         q2.wait()
         q3.wait()
         p.wait()
```

```
              'ls' and output is a terminal)
       standard output is connected to a terminal.  The LS_COLORS  environment
       and error are both connected to terminals (as determined by isatty(3)),
              Used by the select compound command to  determine  the  terminal
              coming from a terminal.  In an interactive shell, the  value  is
       -t fd  True if file descriptor fd is open and refers to a terminal.
     -T      Disable pseudo-terminal allocation.
     -t      Force pseudo-terminal allocation.  This can be used to execute
     pseudo-terminal (pty) for interactive sessions when the client has one.
        .....
```

```
Out[11]: 0
```

1.4 Synchronization and deadlocks

Pipes and subprocesses are difficult to control for a complicated task. Make sure unused ends of pipes (especially that you write) are closed. You do not `wait` for a subprocess that is blocked on some other thing (I/O or other process). Otherwise your code will wait forever

2 Serializing, Storing-Saving Objects

- `pickle` module allows conversion of an arbitrary object into a string representation and vice versa.
- `pickle.dumps(object)` and `pickle.loads(string)` methods are used in conversion

Serialization scenarios include: * saving/restoring object state on a file or database * sending an object over network * calling methods of a remote object (parameters and return value serialized) * object and application persistent

```
In [13]: import pickle

         a=[1,2,3,{'a':123,'b':'hello','c':[1]}]
         mystr=pickle.dumps(a)
         print(mystr)
         b=pickle.loads(mystr)
         print(b)
```

```
b'\x80\x03]q\x00(K\x01K\x02K\x03}q\x01(X\x01\x00\x00\x00cq\x02]q\x03K\x01aX\x01\x00\x00\x00aq\x0
[1, 2, 3, {'c': [1], 'a': 123, 'b': 'hello'}]
```

User defined classes can also be serialized with `pickle`. All properly defined methods (no lambda) and members of an object are serialized automatically. If methods make external references (call outside methods or access global variables) they are not serialized. If restoring program does not have those definitions, run time error is generated when invoked.

```
In [14]: class LList:
             '''Linked list implementation. Iterator reuse is fixed'''
             class Node:
                 def __init__(self, v,n):
                     self.val, self.next = v, n
                 def __str__(self):
                     return "( " + str(self.val) + ", " + str(self.next) + " )"

             def __init__(self,vals=[]):
                 self.head = self.last = None
                 for v in vals:
                     self.append(v)
             def append(self,v):
                 if self.last == None:
                     # very first element
                     self.head = self.last = LList.Node(v,None)
                 else:
                     self.last.next = LList.Node(v,None)
                     self.last = self.last.next
             def __getitem__(self,no):
                 count = 0
                 ptr = self.head
```

```
        while count < no:
            if ptr:
                ptr = ptr.next    # next
            else:
                raise IndexError
            count += 1
        if ptr:
            return ptr.val
        else:
            raise IndexError
    def __setitem__(self,no,val):
        count = 0
        ptr = self.head
        while count < no:
            if ptr:
                ptr = ptr.next
            else:
                raise IndexError
            count += 1
        if ptr:
            ptr.val=val
            return ptr.val
        else:
            raise IndexError
    def __delitem__(self,no):
        count = 0
        prev = ptr = self.head
        while count < no:
            if ptr:
                prev = ptr
                ptr = ptr.next
            else:
                raise IndexError
            count += 1
        if ptr:
            if ptr is self.head:
                if self.head is self.last:
                    self.head = self.last = None
                else:
                    self.head = self.head.val
            else:
                if ptr == self.last:
                    self.last = prev
                prev.next = ptr.next
        else:
            raise IndexError
    def __str__(self):
        ret="["
```

```
            ptr = self.head
            while True:
                if ptr:
                    ret += str(ptr.val)
                else:
                    break
                ptr = ptr.next
                if ptr:
                    ret += " -> "
            ret += "]"
            return ret

        def __iter__(self):
            '''return a brand new iterator'''
            return LListIterator(self)

        # yes, nested iterators possible
        class LListIterator:
            def __init__(self,llist):
                self.llist = llist
                self.itptr = llist.head
            def __next__(self):
                if self.itptr == None:
                    raise StopIteration
                else:
                    val=self.itptr[0]
                    self.itptr = self.itptr[1]
                    return val



    a = LList([3,5,8,8,7,6,1])

    apick = pickle.dumps(a)
    print(apick)

    # this scenario even works when apick is send to a different python instance where LLis
    b = pickle.loads(apick)
    print(b)
```

```
b'\x80\x03c__main__\nLList\nq\x00)\x81q\x01}q\x02(X\x04\x00\x00\x00headq\x03cbuiltins\ngetattr\n
[3 -> 5 -> 8 -> 8 -> 7 -> 6 -> 1]
```

3 Database Access

- `sqlite3` is a module providing simple single file SQL library with same name

- `db = sqlite3.connect(filepath)` returns a database connector
- `cursor = db.cursor()` returns a handle to execute queries
- `cursor.execute(querystring)` will execute the query
- `cursor.fetchone()` , `cursor.fetchall()` returns a single row or list of rows respectively
- A query result can also be iterated.

```
In [1]: import sqlite3
        try:
            db=sqlite3.connect("mydb.sql3")
            cur = db.cursor()
        except Exception as e:
            print("SQL error",e)
        try:
            cur.execute("create table student (stid int primary key, name varchar(40), sname var
        except:
            print("error ignored") # ignore this error

        try:
            cur.execute("insert into student values (12341,'yilmaz','yilar'), (54213,'nalan','na
            db.commit()
        except Exception as e:
            print("SQL error",e)
```

```
error ignored
SQL error UNIQUE constraint failed: student.stid
```

```
In [2]: try:
            cur.execute("select * from student")
            for v in cur:
                print(v)
        except Exception as e:
            print("query error",e)
```

```
(12341, 'yilmaz', 'yilar')
(54213, 'nalan', 'nalmayan')
(61231, 'hasan', 'hasmayan')
(63441, 'beren', 'bermeyen')
```

```
In [2]: stid='XXX\' OR  name like \'%'
        stid2='XXX\'; DELETE STUDENT WHERE NAME LIKE \'%'
        try:
            cur.execute("select * from student where stid ='" + stid + "'")
            for v in cur:
                print(v)
            print('-------')
```

```
            cur.execute("select * from student where stid =?", (stid,))
            for v in cur:
                print(v)
            print('-------')
            cur.execute("select * from student where stid ='" + stid2 + "'")
        except Exception as e:
            print("query error",e)
```

```
(12341, 'yilmaz', 'yilar')
(54213, 'nalan', 'nalmayan')
(61231, 'hasan', 'hasmayan')
(63441, 'beren', 'bermeyen')
-------
-------
query error You can only execute one statement at a time.
```