## Subprocess

A running program is called a **process**. Each process has its own system state, which includes memory, lists of open files, a program counter that keeps track of the instruction being executed, and a call stack used to hold the local variables of functions.

Normally, a process executes statements one after the other in a single sequence of control flow, which is sometimes called the main thread of the process. At any given time, the program is only doing one thing.

A program can create new processes using library functions such as those found in the os or subprocess modules such as **os.fork()**, **subprocess.Popen()**, etc. However, these processes, known as **subprocesses**, run as completely independent entities-each with their own private system state and main thread of execution.

Because a subprocess is independent, it executes concurrently with the original process. That is, the process that created the subprocess can go on to work on other things while the subprocess carries out its own work behind the scenes.

## Subprocess Module

The subprocess module allows us to:

1. spawn new processes
2. connect to their input/output/error pipes
3. obtain their return codes

It offers a higher-level interface than some of the other available modules, and is intended to replace the following functions:

1. os.system()
2. os.spawn*()
3. os.popen*()
4. popen2.*()
5. commands.*()

We cannot use UNIX commands in our Python script as if they were Python code. For example, **echo name** is causing a syntax error because **echo** is not a built-in statement or function in Python. So, in Python script, we're using **print name** instead.

To run UNIX commands we need to create a **subprocess** that runs the command. The recommended approach to invoking **subprocesses** is to use the convenience functions for all use cases they can handle. Or we can use the underlying **Popen** interface can be used directly.

## os.system()

The simplest way of running UNIX command is to use

**os.system()**.

```
>>> import os
>>> os.system('echo $HOME')
/user/khong
0

>>> # or we can use
>>> os.system('echo %s' %'$HOME')
/user/khong
0
```

As expected, we got **$HOME** as stdout (to a terminal). Also, we got a return value of 0 which is the result of executing this command, which means there was no error in the execution.

**os.system**('command with args') passes the command and arguments to our system's shell. By using this can actually run multiple commands at once and set up pipes and input/output redirections. :

```
os.system('command_1 < input_file | command_2 > output_fi
```

If we run the code above **os.system('echo $HOME')** in the Python IDLE, we only see the **0** because the stdout means a terminal. To see the command output we should redirect it to a file, and the read from it:

```
>>> import os
>>> os.system('echo $HOME > outfile')
0
>>> f = open('outfile','r')
>>> f.read()
'/user/khong\n'
```

## os.popen()

Open a pipe to or from command. The return value is an open file object connected to the pipe, which can be read or written depending on whether mode is **'r'** (default) or **'w'**. The bufsize argument has the same meaning as the corresponding argument to the built-in **open()** function. The exit status of the command (encoded in the format specified for **wait()**) is available as the return value of the **close()** method of the file object, except that when the exit status is zero (termination without errors), **None** is returned.

```
>>> import os
>>> stream = os.popen('echo $HOME')
>>> stream.read()
'/user/khong\n'
```

**os.popen()** does the same thing as **os.system** except that it gives us a file-like stream object that we can use to access standard input/output for that process. There are 3 other variants of **popen** that all handle the i/o slightly differently.

If we pass everything as a string, then our command is passed to the shell; if we pass them as a list then we don't need to worry about escaping anything.

However, it's been **deprecated** since version 2.6: This function is obsolete. Use the **subprocess** module. docs.python.org (http://docs.python.org/2/library/os.html#os.popen)

## subprocess.call()

This is basically just like the **Popen** class and takes all of the same arguments, but it simply wait until the command completes and gives us the return code.

```
subprocess.call(args, *, stdin=None, stdout=None, stderr=
```

Run the command described by **args**. Wait for command to complete, then return the **returncode** attribute.

```
>>> import os
>>> os.chdir('/')
>>> import subprocess
>>> subprocess.call(['ls','-l'])
total 181
drwxr-xr-x    2 root root  4096 Mar  3  2012 bin
drwxr-xr-x    4 root root  1024 Oct 26  2012 boot
...
```

The command line arguments are passed as a list of strings, which avoids the need for escaping quotes or other special characters that might be interpreted by the shell.

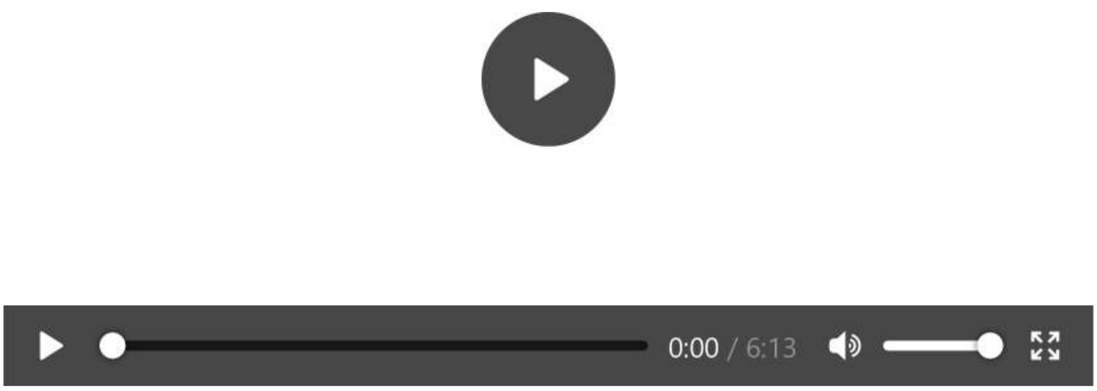
```
>>> import subprocess
>>> subprocess.call('echo $HOME')
Traceback (most recent call last):
...
OSError: [Errno 2] No such file or directory
>>>
>>> subprocess.call('echo $HOME', shell=True)
/user/khong
0
```

Setting the shell argument to a true value causes **subprocess** to spawn an intermediate shell process, and tell it to run the command. In other words, using an intermediate shell means that variables, glob patterns, and other special shell features in the command string are processed **before** the command is run. Here, in the example, **$HOME** was processed before the **echo** command. Actually, this is the case of **command with shell expansion** while the command **ls -l** considered as a simple command.

Here is a sample code (PyGoogle/FFMpeg/iframe_extract.py (https://github.com/PyGoogle/FFMpeg.git)). It downloads YouTube video and then extracts I-frames to sub folder:

```
'''
iframe_extract.py - download video and ffmpeg i-frame ext
Usage:
(ex) python iframe_extract.py -u https://www.youtube.com/
This code does two things:
1. Download using youtube-dl
cmd = ['youtube-dl', '-f', videoSize, '-k', '-o', video_o
2. Extract i-frames via ffmpeg
cmd = [ffmpeg,'-i', inFile,'-f', 'image2','-vf',
        "select='eq(pict_type,PICT_TYPE_I)'",'-vsync','vf
'''

from __future__ import unicode_literals
import youtube_dl

import sys
import os
import subprocess
import argparse
import glob

if sys.platform == "Windows":
    FFMPEG_BIN = "ffmpeg.exe"
    MOVE = "move"
    MKDIR = "mkdir"
else:
    FFMPEG_BIN = "ffmpeg"
    MOVE = "mv"
    MKDIR = "md"


def iframe_extract(inFile):
# ffmpeg -i inFile -f image2 -vf \
#   "select='eq(pict_type,PICT_TYPE_I)'" -vsync vfr oStri

    # infile : video file name
    #          (ex) 'FoxSnowDive-Yellowstone-BBCTwo.mp4'
    imgPrefix = inFile.split('.')[0]
    # imgPrefix : image file

    # start extracting i-frames
    home = os.path.expanduser("~")
    ffmpeg = home + '/bin/ffmpeg'

    imgFilenames = imgPrefix + '%03d.png'

    cmd = [ffmpeg,'-i', inFile,'-f', 'image2','-vf',
        "select='eq(pict_type,PICT_TYPE_I)'",'-vsync','vf

    # create iframes
    print "creating iframes ...."
    subprocess.call(cmd)

    # Move the extracted iframes to a subfolder
    # imgPrefix is used as a subfolder name that stores i
    cmd = 'mkdir -p ' + imgPrefix
    os.system(cmd)
    print "make subdirectoy", cmd
    mvcmd = 'mv ' + imgPrefix + '*.png ' + imgPrefix
    print "moving images to subdirectoy", mvcmd
    os.system(mvcmd)



def get_info_and_download(download_url):

    # Get video meta info and then download using youtube

    ydl_opts = {}

    # get meta info from the video
    with youtube_dl.YoutubeDL(ydl_opts) as ydl:
        meta = ydl.extract_info(download_url, download=Fa

    # renaming the file
    # remove special characters from the file name
    print('meta[title]=%s' %meta['title'])
    out = ''.join(c for c in meta['title'] if c.isalnum()
    print('out=%s' %out)
    extension = meta['ext']
    video_out = out + '.' + extension
    print('video_out=%s' %video_out)
    videoSize = 'bestvideo[height<=540]+bestaudio/best[he
    cmd = ['youtube-dl', '-f', videoSize, '-k', '-o', vid
    print('cmd=%s' %cmd)
    # download the video
    subprocess.call(cmd)

    # Sometimes output file has format code in name such
    # so, in this case, we want to rename it 'out.webm'
    found = False
    extension_list = ['mkv', 'mp4', 'webm']
    for e in extension_list:
       glob_str = '*.' + e
       for f in glob.glob(glob_str):
          if out in f:
             if os.path.isfile(f):
                video_out = f
                found = True
                break
       if found:
          break

    # call iframe-extraction : ffmpeg
    print('before iframe_extract() video_out=%s' %video_o
    iframe_extract(video_out)
    return meta



def check_arg(args=None):

# Command line options
# Currently, only the url option is used

    parser = argparse.ArgumentParser(description='downloa
    parser.add_argument('-u', '--url',
                        help='download url',
                        required='True')
    parser.add_argument('-i', '--infile',
                        help='input to iframe extract')
    parser.add_argument('-o', '--outfile',
                        help='output name for iframe imag

    results = parser.parse_args(args)
    return (results.url,
            results.infile,
            results.outfile)
```

```
# Usage sample:
#    syntax: python iframe_extract.py -u url
#    (ex) python iframe_extract.py -u https://www.youtube

if __name__ == '__main__':
    u,i,o = check_arg(sys.argv[1:])
    meta = get_info_and_download(u)
```



## subprocess.check_call()

```
subprocess.check_call(args, *, stdin=None, stdout=None, s
```

The **check_call()** function works like **call()** except that the exit code is checked, and if it indicates an error happened then a **CalledProcessError** exception is raised.

```
>>> import subprocess
>>> subprocess.check_call(['false'])
Traceback (most recent call last):
...
subprocess.CalledProcessError: Command '['false']' return
```

## subprocess.check_output()

```
subprocess.check_output(args, *, stdin=None, stderr=None,
                                 shell=False, universal_n
```

The standard input and output channels for the process started by **call()** are bound to the parent's input and output. That means the calling program cannot capture the output of the command. **To capture the output**, we can use **check_output()** for later processing.

```
>>> import subprocess
>>> output = subprocess.check_output(['ls','-l'])
>>> print output
total 181
drwxr-xr-x    2 root root  4096 Mar  3  2012 bin
drwxr-xr-x    4 root root  1024 Oct 26  2012 boot
...
>>> output = subprocess.check_output(['echo','$HOME'], sh
>>> print output
/user/khong
```

This function was added in Python 2.7.

## subprocess.Popen()

The underlying process creation and management in this module is handled by the **Popen** class. It offers a lot of flexibility so that developers are able to handle the less common cases not covered by the convenience functions.

**subprocess.Popen()** executes a child program in a new process. On Unix, the class uses **os.execvp()**-like behavior to execute the child program. On Windows, the class uses the Windows **CreateProcess()** function.

## args of subprocess.Popen()

```
class subprocess.Popen(args, bufsize=0, executable=None,
                       stdin=None, stdout=None, stderr=No
                       preexec_fn=None, close_fds=False,
                       shell=False, cwd=None, env=None, u
                       startupinfo=None, creationflags=0)
```

1. **args**:

should be a sequence of program arguments or else a single string. By default, the program to execute is the first item in args if args is a sequence. If args is a string, the interpretation is platform-dependent. It is recommended to pass args as a sequence.

2. **shell**:
   **shell** argument (which defaults to **False**) specifies whether to use the shell as the program to execute. If shell is **True**, it is recommended to pass args as a string rather than as a sequence.
   On Unix with **shell=True**, the shell defaults to **/bin/sh**.
    1. If **args** is a **string**, the string specifies the command to execute through the shell. This means that the string must be formatted exactly as it would be when typed at the shell prompt. This includes, for example, quoting or backslash escaping filenames with spaces in them.
    2. If **args** is a **sequence**, the first item specifies the command string, and any additional items will be treated as additional arguments to the shell itself. That is to say, **Popen** does the equivalent of:

       ```
       Popen(['/bin/sh', '-c', args[0], args[1], ...])
       ```

3. **bufsize**:
   if given, has the same meaning as the corresponding argument to the built-in **open()** function:
    1. 0 means unbuffered
    2. 1 means line buffered
    3. any other positive value means use a buffer of (approximately) that size
    4. A negative bufsize means to use the system default,

   which usually means fully buffered
   5. The default value for bufsize is 0 (unbuffered)

4. **executable**:
   specifies a replacement program to execute. It is very seldom needed.

5. **stdin, stdout and stderr**:
   1. specify the executed program's standard input, standard output and standard error file handles, respectively.
   2. Valid values are **PIPE**, an existing file descriptor (a positive integer), an existing file object, and None.
   3. **PIPE** indicates that a new pipe to the child should be created.
   4. With the default settings of **None**, no redirection will occur; the child's file handles will be inherited from the parent.
   5. Additionally, stderr can be STDOUT, which indicates that the stderr data from the child process should be captured into the same file handle as for stdout.

6. **preexec_fn**:
   is set to a callable object, this object will be called in the child process just before the child is executed. (Unix only)

7. **close_fds**:
   is true, all file descriptors except 0, 1 and 2 will be closed before the child process is executed. (Unix only). Or, on Windows, if **close_fds** is true then no handles will be inherited by the child process. Note that on Windows, we cannot set **close_fds** to true and also redirect the standard handles by setting stdin, stdout or stderr.

8. **cwd**:
   is not None the child's current directory will be changed to **cwd** before it is executed. Note that this directory is not considered when searching the executable, so we can't specify the program's path relative to **cwd**.

9. **env**:
   is not None, it must be a mapping that defines the environment variables for the new process; these are used instead of inheriting the current process' environment, which is the default behavior.

10. **universal_newlines**:
    is True, the file objects stdout and stderr are opened as text files in universal newlines mode. Lines may be terminated by any of **'\n'**, the Unix end-of-line convention, **'\r'**, the old Macintosh convention or **'\r\n'**, the Windows convention. All of these external representations are seen as **'\n'** by the Python program.

11. **startupinfo**:
    will be a **STARTUPINFO** object, which is passed to the underlying **CreateProcess** function.

12. **creationflags**:
    can be **CREATE_NEW_CONSOLE** or **CREATE_NEW_PROCESS_GROUP**. (Windows only)

## os.popen vs. subprocess.Popen()

This is intended as a replacement for **os.popen**, but it is more complicated. For example, we use

```
subprocess.Popen("echo Hello World", stdout=subprocess.PI
```

instead of

```
os.popen("echo Hello World").read()
```

But it is comprehensive and it has all of the options in one unified class instead of different **os.popen** functions.

## subprocess.Popen() - stdout and stderr

```
>>> import subprocess
>>> proc = subprocess.Popen(['echo', '"Hello world!"'],
...                          stdout=subprocess.PIPE)
>>> stddata = proc.communicate()
>>> stddata
('"Hello world!"\n', None)
```

Note that the **communicate()** method returns a **tuple** (stdoutdata, stderrdata) : ('"Hello world!"\n' ,None). If we don't include **stdout=subprocess.PIPE** or **stderr=subprocess.PIPE** in the **Popen** call, we'll just get **None** back.

```
Popen.communicate(input=None)
```

**Popen.communicate()** interacts with process: Send data to **stdin**. Read data from **stdout** and **stderr**, until end-of-file is reached. Wait for process to terminate. The optional input argument should be a string to be sent to the child process, or **None**, if no data should be sent to the child.

So, actually, we could have done as below:

```
>>> import subprocess
>>> proc = subprocess.Popen(['echo', '"Hello world!"'],
...                              stdout=subprocess.PIPE)
>>> (stdoutdata, stderrdata) = proc.communicate()
>>> stdoutdata
'"Hello world!"\n'
```

or we can explicitly specify which one we want from **proc.communicate()**:

```
>>> import subprocess
>>> proc = subprocess.Popen(['echo', '"Hello world!"'],
...                              stdout=subprocess.PIPE)
>>> stdoutdata = proc.communicate()[0]
>>> stdoutdata
'"Hello world!"\n'
```

The simplest code for the example above might be sending the stream directly to console:

```
>>> import subprocess
>>> proc = subprocess.Popen(['echo', '"Hello world!"'],
...                              stdout=subprocess.PIPE)
>>> proc.communicate()[0]
'"Hello world!"\n'
```

The code below is to test the stdout and stderr behaviour:

```
# std_test.py
import sys
sys.stdout.write('Testing message to stdout\n')
sys.stderr.write('Testing message to stderr\n')
```

If we run it:

```
>>> proc = subprocess.Popen(['python', 'std_test.py'],
...                              stdout=subprocess.PIPE)
>>> Testing message to stderr
>>> proc.communicate()
(Testing message to stdout\n', None)
>>>
```

Note that the message to **stderr** gets displayed as it is generated but the message to **stdout** is read via the pipe. This is because we only set up a pipe to **stdout**.

Then, let's make both stdout and stderr to be accessed from Python:

```
>>> proc = subprocess.Popen(['python', 'std_test.py'],
...                              stdout=subprocess.PIPE,
...                              stderr=subprocess.PIPE)
>>> proc.communicate()
(Testing message to stdout\n', Testing message to stderr\
```

The **communicate()** method only reads data from **stdout** and **stderr**, until end-of-file is reached. So, after all the messages have been printed, if we call **communicate()** again we get an error:

```
>>> proc.communicate()
Traceback (most recent call last):
...
ValueError: I/O operation on closed file
```

If we want messages to **stderr** to be piped to **stderr**, we do: **stderr=subprocess.STDOUT**.

```
>>> proc = subprocess.Popen(['python', 'std_test.py'],
...                          stdout=subprocess.PIPE,
...                          stderr=subprocess.STDOUT)
>>> proc.communicate()
('Testing message to stdout\r\nTesting message to stderr\
```

As we see from the output, we do not have **stderr** because it's been redirected to **stderr**.

## subprocess.Popen() - stdin

Writing to a process can be done in a very similar way. If we want to send data to the process's **stdin**, we need to create the **Popen** object with **stdin=subprocess.PIPE**.

To test it let's write another program (**write_to_stdin.py**) which simply prints Received: and then repeats the message we send it:

```
# write_to_stdin.py
import sys
input = sys.stdin.read()
sys.stdout.write('Received: %s'%input)
```

To send a message to **stdin**, we pass the string we want to send as the input argument to **communicate()**:

```
>>> proc = subprocess.Popen(['python', 'write_to_stdin.py
>>> proc.communicate('Hello?')
Received: Hello?(None, None)
```

Notice that the message created in the **write_to_stdin.py** process was printed to **stdout** and then the return value **(None, None)** was printed. That's because no pipes were set up to **stdout** or **stderr**.

Here is another output after we specified **stdout=subprocess.PIPE** and **stderr=subprocess.PIPE** just as before to set up the pipe.

```
>>> proc = subprocess.Popen(['python', 'write_to_stdin.py
...                          stdin=subprocess.PIPE,
...                          stdout=subprocess.PIPE,
...                          stderr=subprocess.PIPE)
>>> proc.communicate('Hello?')
('Received: Hello?', '')
```

## subprocess.Popen() - shell pipeline

```
>>> p1 = subprocess.Popen(['df','-h'], stdout=subprocess.
>>> p2 = subprocess.Popen(['grep', 'sda1'], stdin=p1.stdo
>>> p1.stdout.close()  # Allow p1 to receive a SIGPIPE if
>>> output = p2.communicate()[0]
>>> output
'/dev/sda1              19G  9.2G  8.3G  53% /\n'
```

The **p1.stdout.close()** call after starting the **p2** is important in order for **p1** to receive a **SIGPIPE** if **p2** exits before **p1**.

Here is another example for a piped command. The code gets window ID for the currently active window. The command looks like this:

```
xprop -root | awk '/_NET_ACTIVE_WINDOW\(WINDOW\)/{print $
```

Python code:

```
# This code runs the following awk to get a window id for
# xprop -root | awk '/_NET_ACTIVE_WINDOW\(WINDOW\)/{print

import subprocess

def py_xwininfo():
        winId = getCurrentWinId()
        print 'winId = %s' %winId

def getCurrentWinId():
        cmd_1 = ['xprop', '-root']
        cmd_2 = ['awk', '/_NET_ACTIVE_WINDOW\(WINDOW\)/{p
        p1 = subprocess.Popen(cmd_1, stdout = subprocess.
        p2 = subprocess.Popen(cmd_2, stdin = p1.stdout, s
        id = p2.communicate()[0]

        return id

if __name__ == '__main__':
        py_xwininfo()
```

Output:

```
winId = 0x3c02035
```

## subprocess.Popen() - shell=True

Avoid **shell=True** by all means.

**shell=True** means executing the code through the shell. In other words, executing programs through the shell means, that all user input passed to the program is interpreted according to the syntax and semantic rules of the invoked shell. At best, this only causes inconvenience to the user, because the user has to obey these rules. For instance, paths containing special shell characters like quotation marks or blanks must be escaped. At worst, it causes security leaks, because the user can execute arbitrary programs.

**shell=True** is sometimes convenient to make use of specific shell features like word splitting or parameter expansion. However, if such a feature is required, make use of other modules are given to you (e.g. **os.path.expandvars()** for parameter expansion or shlex for word splitting). This means more work, but avoids other problems. - from Actual meaning of 'shell=True' in subprocess (http://stackoverflow.com/questions/3172470/actual-meaning-of-shell-true-in-subprocess).

If **args** is a **string**, the string specifies the command to execute through the shell. This means that the string must be formatted exactly as it would be when typed at the shell prompt. This includes, for example, quoting or backslash escaping filenames with spaces in them:

```
>>> proc = subprocess.Popen('echo $HOME', shell=True)
>>> /user/khong
```

The string is an exactly the formatted as it would be typed at the shell prompt:

```
$ echo $Home
```

So, the following would not work:

```
>>> proc = subprocess.Popen('echo $HOME', shell=False)
Traceback (most recent call last):
...
OSError: [Errno 2] No such file or directory
```

Following would not work, either:

```
>>> subprocess.Popen('echo "Hello world!"', shell=False)
Traceback (most recent call last):
...
OSError: [Errno 2] No such file or directory
```

That's because we are still passing it as a string, Python assumes the entire string is the name of the program to execute and there isn't a program called **echo "Hello world!"** so it fails. Instead we have to pass each argument separately.

## psutil and subprocess - psutil.popen()

**psutil.Popen(*args, **kwargs)** is a more convenient interface to stdlib **subprocess.Popen()**.

"It starts a sub process and deals with it exactly as when using **subprocess.Popen** class but in addition also provides all the properties and methods of **psutil.Process** class in a single interface".
- see http://code.google.com/p/psutil/wiki/Documentation

(http://code.google.com/p/psutil/wiki/Documentation)

The following code runs **python -c "print 'hi, psutil'"** on a subprocess:

```
>>> import psutil
>>> import subprocess
>>> proc = psutil.Popen(["/usr/bin/python", "-c", "print
>>> proc
<psutil.Popen(pid=4304, name='python') at 140431306151888
>>> proc.uids
user(real=1000, effective=1000, saved=1000)
>>> proc.username
'khong'
>>> proc.communicate()
('hi, psuti\n', None)
```

## References

1. docs.python.org (http://docs.python.org/2/library/subprocess.html)
2. subprocess - Work with additional processes (http://www.doughellmann.com/PyMOTW/subprocess/)
3. subprocess - working with Python subprocess - Shells, Processes, Streams, Pipes, Redirects and More (http://jimmyg.org/blog/2009/working-with-python-subprocess.html)