POWERING POTENTIAL

Python 3 Programming

Multitasking

MULTITASKING

Contents

Family life

- **A process is an instance of a program loaded and ready to run**
- **Every process has a parent, so every process is a child**
- **Child usually inherits attributes of parent**
  - Environment, current directory, security
  - Open files – depends on the Python release (see notes)
- **Relationship depends on the operating system**
  - UNIX has strong family ties
  - → If a parent dies, the child is an 'orphan'
  - Microsoft Windows has few ties between parent and child
  - → Parent has to explicitly maintain a HANDLE to the child
  - → Can disown children, but still supports inheritance

Most modern operating systems allow a user to run several applications simultaneously, so that non-interactive tasks can be run in the background while the user continues with other work in the foreground. The user can also run multiple copies of the same program at the same time.

The two key operating-system object types that have a major role to play in multitasking are the 'process' and the 'thread'. A process is an instance of a running program. Each process owns its own resources (code, data and the like) which are located in its own private address space. Any such resources created by a process are destroyed when the process terminates.

Before creating a process, we should consider the implications. Aside from the obvious performance overhead, creating another process forms a relationship between creator (the parent) and created process (the child).

Depending on the operating system, some items are inherited (copied) by the child process. On Microsoft Windows and UNIX, the child inherits the Environment block, the current directory, and security ID. Other things may be inherited, and both operating systems allow a degree of control over what exactly is inherited.

Open files are handled differently depending on the release of Python. Prior to Python 3.4, open file handles were inherited – and this is still the case in Python 2. In Python 3.4, open handles are no longer inherited by default, see PEP 446.

Creating a process from Python

- **Process interfaces can be platform specific**
- os.fork
→ UNIX specific
→ Creates another process - does not run another program
exec*type* is required to run another program after fork
→ Requires os.wait or os.waitpid to avoid a zombie
- os.system
→ Passes the command to the shell (cmd.exe or the Bourne shell)
→ Runs an additional shell process
- os.spawn*type*
→ Some types are not available on Windows
→ Can run in a similar mode to **exec***type*
- os.popen
→ Run a process connected through pipes

All these interfaces are deprecated

The **`os`** methods shown are system specific, and their behaviour varies wildly between operating systems. They reflect the differing architectures of the operating systems they run on.
On UNIX (and Linux), when we wish to run another program, we first have to create a copy of the *current* process - current program and all. That action is known as a **fork**. In the copy (known as a *child process*), we can overlay the current program with a different one, and this is known as an **exec**. There are various forms of **exec**, depending on whether you wish to supply a different environment block, use the PATH environment variable to find the program, and the way that arguments are passed.
The old DEC operating system VMS did not use that two stage approach, but created a new process running a different program in one go - an action called **spawn**. Windows inherited some of the architectural features of VMS, including **spawn** which, like **exec** on UNIX, also had different forms depending on how we wished to run the program.
Meanwhile, at an attempt to be portable, the C language standard came up with **`system()`**, which called a shell program to launch another process. This was not particularly efficient and could not handle asynchronous requests.
These architectural differences are all reflected in these older

interfaces, which are now considered to be deprecated, so we will not discuss them further, although, you may see them still used by die-hards.
Older versions of Python on UNIX also had the **`commands`** module, which was withdrawn at Python 3.
`os.startfile()` runs on Windows and runs the associated program on the specified file (like double-clicking on the file in Windows Explorer).

Old interface examples

- **Run a process and wait for it to complete**
- Invokes a surrogate shell

```
import os
status = os.system('hello.py')
print("Child exited with", status)
```

- **Run a process at the other end of a pipe**
- Returns a file object

```
for line in os.popen('tasklist').readlines():
    print(":", line, end="")
```

All these interfaces are deprecated

The simplest way to run another program from Python is to use **`os.system`**. Unfortunately, this launches a shell whether you need one or not.
`os.popen` has two parameters, mode, which defaults to 'r', and Buffering, which defaults to None. There are some portability issues with `popen`, and a `win32pipe` module also exists specifically for Windows.
To read stderr from popen, you will need the assistance of the shell, typically with the decoration `2>&1` (*not* UNIX csh). If you need to read stdin from another program then open the pipe with write access. If you need to open both, stdin and stdout, then use **`os.pipe`**.

Using the subprocess module

- **Unifying process creation**
  - Introduced at Python 2.4
  - Intended to replace os.system and os.spawn
- **From Python 3.5, the preferred interface is subprocess.run**
  - Meant for the majority of simple tasks
- **For more complex tasks use Popen**
  - Returns a subprocess object
  - Parameters are discussed later
- **Other shortcuts are available**
  - call and check_call
  - getoutput and getstatusoutput (UNIX specific)
- **We discuss the `multiprocessing` package later**
  - Runs processes in a similar way to threads

At an attempt to resolve the different interfaces used, the **`subprocess`** module was introduced into the Python standard library at Python 2.4. It was supposed to be unifying with no operating system specific quirks - an aim not entirely achieved. The older interfaces are still supported, but should not be used for new applications. Check specifically, the *Replacing Older Functions with the subprocess Module* section in that documentation page. Since using Python 3 is an opportunity to move to new practices, this is a good time to ditch the old methods.
The `multiprocess` module in the Python Standard Library runs processes using a different approach, which we shall discuss later...

subprocess.run

Run a program and wait for it to complete

- This API was added in Python 3.5 for the majority of simple jobs
- It is a wrapper around Popen
- Returns a subprocess.CompletedProcess object

```
run(*args, input=None, timeout=None, check=False, **kwargs)
```

`args`	Command-line to execute (a sequence)
`input`	Data to be passed to stdin of the program
`timeout`	A timeout value in seconds. Raise a subprocess.TimeoutExpired exception if exceeded
`check`	If True, raise a subprocess.CalledProcessError if exit code != 0
`kwargs`	Optional Popen arguments (see later)

The CompletedProcess object includes:

- `returncode`
- `stdout` only if routed to a PIPE (see later)
- `stderr` only if routed to a PIPE (see later)

py3

The subprocess.run method was introduced in 3.5 to simplify the use of subprocess. It is a wrapper about Popen, which is described over, and can appear intimidating. Popen is more than most people need, and there is a requirement to call a method to wait for the process to complete. The simpler subprocess.run will wait – which is often what you need.
The timeout parameter is something which is often requested, and is an add-on which is not availalbe with Popen.
Should you need the power of Popen, then it is still there, all its parameters can be appended.
Note that the template shown is from the help text, not from the documentation.

The subprocess.Popen class parameters

`args`	Command-line to execute (a sequence)
`bufsize=-1`	Buffersize 0: unbuffered < 0: default buffsize
`executable=None`	Program to be executed, rarely needed
`stdin=None`	Handle used for stdin (can be PIPE)
`stdout=None`	Handle used for stdout (can be PIPE)
`stderr=None`	Handle used for stderr (can be `STDOUT`)
`preexec_fn=None`	Code to call before the program (UNIX)
`close_fds=True`	Do not inherit open file handles
`shell=False`	Use a shell to execute the command
`cwd=None`	Working directory of the child process
`env=None`	Environment block of the child process
`universal_newlines=False`	See any of '\r' or '\n' as newlines
`startupinfo=None`	Windows only STARTUPINFO struct
`creationflags=0`	Windows only creation flags
`restore_signals=True`	Restore signals to default action (UNIX) 3.2
`start_new_session=False`	Used for creating daemons (UNIX) 3.2
`pass_fds=()`	Sequence of file descriptors (UNIX) 3.2

Which buffer size should I use? If in doubt, default the parameter, as usual. The default size of -1 was changed in Python 3.3.1 to use the system's default buffer size.

Open file handles (called *file descriptors* on UNIX) are inherited by a child process in many languages, and by the older interfaces (file locks are not inherited). To prevent that, we often had to resort to low-level interfaces, like **`fcntl`** on UNIX. Fortunately, the `subprocess` module switches this off by default (**`close_fds`**), except for `stdin`, `stdout`, and `stderr`, which is usually what we want.

The Windows `startupinfo` mostly determines features of the main window for the new process, like x/y starting position, title, etc. The `creationflags` cover, for example, the priority of the child process. For details of these Windows specifics, see the MSDN and the low-level C function `CreateProcess`.

Running a basic process

- **Run a process and wait for it to complete**
- **A shell is sometimes required**
  - When using shell meta-characters
  - → Wildcards, pipes, redirections, etc.
  - On Windows, no file association is done unless shell=True

```
import subprocess
proc = subprocess.run('hello.py', shell=True)
print('Child exited with', proc.returncode)
```

- Don't use a shell if you don't need to
- → It can add an unnecessary overhead

Typically:
C:\Python36\python.exe

```
import subprocess
proc = subprocess.run([sys.executable, 'hello.py'])
```

Here we look at the simplest technique to run another program, which is similar in some ways to **`os.system()`**. It uses **`subprocess.run`**, often with just the command-line as the single argument.

The first parameter to run can be any sequence - including either a string or a list. If using a string, make sure that there is at least one space between each component. For example, the final example of the slide could be written:

```
cmd = sys.executable + ' hello.py '
proc = subprocess.run(cmd)
```

Unlike `system()`, **`run`** does not use a surrogate shell to run the program unless you ask it to – and this is a **`Popen`** parameter. You will need a shell if you require shell features, such as globbing (also known as wildcards) - why not use the Python **`glob`** module instead?

On Windows, users are so used to clicking on a file and expecting "it" to run the right program that they often forget who or what "it" is. File association, associating a file extension with a particular program, is not done by the operating system, it is done by the application which launches it - Windows Explorer, or `cmd.exe` for

example. The **`subprocess`** module does not do file association, so you will need a shell to do the association for you, or add the program name yourself (which is more efficient). For Python programs, the full path name is conveniently in **`sys.executable`**. If you need file association on Windows then use **`os.startfile()`**.

Capturing the output

- **Use the returned CompletedProcess object**
- Includes stdout, and stderr, but only when using PIPE
- Use *bytes*`.decode()` to convert bytes to string
- Use *string*`.encode()` to convert string to bytes (for stdin)

```
import subprocess
proc = subprocess.run("tasklist",
                      stdout=subprocess.PIPE,
                      stderr=subprocess.PIPE)

if proc.stderr != None:
    print ("error:", proc.stderr.decode())

print("output:", proc.stdout.decode())
```

- Remember that data has to be stored in memory - too much may crash your program!

This is roughly equivalent to using `back-ticks` or `$( )` *command substitution* in UNIX shells - capturing the stdout from the child process. It is only useful with relatively small amounts of data, since the whole output is captured in memory before we can proceed. The CompletedProcess object includes the output from stdout and stderr, but only if the process is run using PIPE for these streams. Both are byte-streams rather than strings, so you might have to **decode** them to manipulate the data.

Passing data into stdin

- **Suitable for simple text**
  - Bytes objects only

```
import subprocess
proc = subprocess.run("stuff.py", input=b"some text")
```

  - stuff.py

```
ans = input("Enter stuff: ")
print(f"<{ans}>" )
```

  - Output

```
Enter stuff: <some text>
```

Passing data through stdin is less common, but can be useful. The `subprocess.run` interface makes this very simple for single strings by using the `input` parameter.
Notice that native strings cannot be used, a bytes object is required.

Very basic threads in Python

Python threading usually uses the threading module

- Call `threading.Thread` (*function*)

```
from threading import Thread
import time

def my_func(*args):
    print("From thread", args)
    time.sleep(5)

th1 = Thread(target=my_func, args="1")
th2 = Thread(target=my_func, args="2")
th1.start()
th2.start()
print("From main")
th1.join()
th2.join()
```

?

```
From thread ('1')
From thread From main
('2')
```

- Or create our own class derived from `threading.Thread`

The Python **`threading`** module is a high-level interface based on the `thread` module. If you have used threads procedurally, for example Win32 threads or pthreads from C/C++, then you will be familiar with the procedural interface. Alternatively, we can derive our own class from the threading base class:

```
import threading
import time
class MyThread (threading.Thread):
    def run (self):
        print ("From thread", self.name)
        time.sleep(5)

th1 = MyThread()
th2 = MyThread()
th1.start()
th2.start()
print ("From main")
th1.join()
th2.join()
```

Python threads support various locking mechanisms: Condition, Event, Semaphore, Locks (and RLock), and Thread local data. Take note of the output from our simple program, can you see how the output from the threads and main are interleaved? Oops!

Synchronisation objects in threading

- **Several objects are available for thread synchronisation**
- Condition variables
→ Similar to those used by pthreads
- Events
→ Similar to those used by Win32
- Thread local storage
→ Enables global variables to be local to a thread
- Locks
→ Similar to a mutex, has a concept of ownership
- Semaphores
→ A counting lock, e.g. allow up to 3 threads to access a resource
- Timers
→ Similar to waitable timers on Win32 and interval timers on UNIX

The **`threading`** module contains a ranges of objects that can be used for thread synchronisation.
Condition variables come from the POSIX pthreads runtime library and include a Lock to protect a predicate. They are generally more complex to use than Events.
Events are very easy to use. Threads wait on an event and another releases them. They include a timeout parameter.
Thread local storage is to enable a thread to share a global variable between functions, but for that variable to be different for each thread. Generally, this is a hack to allow a single threaded program to be converted to multi-threaded! Avoid global variables and you won't need to use this.
Locks are often required, and represent the basic locking mechanism. A thread either has ownership of a lock or waits for it.
Semaphore on the other hand are not "owned" by anyone. We put a limit on the number of threads that can lock a semaphore, if more come along then they wait until a thread releases the semaphore.
Timers can be useful for triggering functions at specific times, or in specific intervals.

Simple use of lock

To fix the print issue, and to protect a global list

```
from threading import Lock
csScreen      = Lock()
csSharePrices = Lock()

dSharePrices = []

def GetStockPrice():
    global dSharePrices

    csSharePrices.acquire()
    dPrices = dSharePrices[:]
    csSharePrices.release()
    return dPrices

def Sessions:
    csScreen.acquire()
    print("\nWaiting for requests\n")
    csScreen.release()
```

This shows the use of `Threading.Lock` to create a "Critical Section" of code (the term *Critical Section* comes from Windows and indicates code which can only run in one thread at a time). While the lock has been acquired, no other thread can access code protected by the lock.
There are two lock objects: `csScreen` protects the STDOUT buffer, and is acquired and released around each `print()`, and `csSharePrices` protects the global list `dSharePrices`. Notice we are taking a copy of the list by using the slice, otherwise we would be returning a reference to the list which would not be protected.

The trouble with threads

- **They are very difficult to code**
  - Sharing variables requires locking mechanisms
  - Subtle timing differences can make debugging difficult
- **The Python Global Interpreter Lock (GIL)**
  - The GIL locks the interpreter
  - → Threads are locked for about 100 byte-code instructions
  - → Simplifies and protects the interpreter
  - The GIL *does not* mean that:
  - → Python is not multi-threaded - C modules can multi-thread
  - → You don't need to worry about locking - you certainly do!

> "Multi-threading is a way of shooting yourself in both feet"

Guido van Rossum : "Unfortunately, for most mortals, thread programming is just Too Hard to get right.... Even in Python...".
The CPython interpreter, when working with pure Python code, will force the GIL to be released every hundred byte code instructions. This means that if you have a complex line of code, like a complex math function that in reality acts as a single byte code, the GIL will not be released for the period that statement takes to run.
There is an exception though: C modules! C extension modules (and built in C modules) can be built in such a way that they release the GIL voluntarily and do their own magic.
By the way, don't think it is just Python which is affected by this kind of issue, in Ruby the GIL is called the Global VM Lock.
Because of these issues, we are not taking threading any further here.
Parallel Python is available here: http://www.parallelpython.com, but not currently on Python 3.0. Another alternative is to use Stackless Python, from http://www.stackless.com. Originally, the Google "Unladen Swallow" project was to remove the GIL, but that extension has now been dropped.

Using the multiprocessing module

- **Uses processes rather than threads**
- Default number of processes is one for each core
- Also supports process pools, and processes across systems
- Pipes and queues for synchronised communication

```
from multiprocessing import Process

def my_func(*args):
    print("From proc", args)
    time.sleep(5)

if __name__ == "__main__":
    p1 = Process(target=my_func, args="1")
    p2 = Process(target=my_func, args="2")
    p1.start()
    p2.start()
    print("From main")
    p1.join()
    p2.join()
```

```
From main
From proc ('2',)
From proc ('1',)
```

The **`multiprocessing`** module is suitable for sharing data or tasks between processor cores. It does not use threading, but processes instead. Processes are inherently more "expensive" than threads, so they are not worth using for trivial data sets or tasks.
Since they run in different processes, then any data items sent must use a kernel object, which again uses more system resources than using shared variables within the same address space. However, shared variables require synchronisation whether they are within the same process or not (technically, in-process synchronisation is cheaper than synchronisation between processes). We shall address that on the next slide.
You will note that the code on the slide is very similar to the threading example. The main noticeable difference (apart from the names) is the inclusion of the **`if __name__ ==“__main__”`** test. This is there because the whole script is repeated for the child processes, much like `fork()` does things (it uses `fork` on UNIX). If you wondered, in the child processes the value of **`__name__`** is “`__parents_main__`”. This `if` statement is important on Windows, since it does not have a `fork()` but imports the entire script. So you can get away with not having it on UNIX/Linux, but it is

probably a good idea to always include it for portability. Strictly speaking, if `process` is called from elsewhere, such as a class, then it might not be required.
Just as with threading, it is common to derive a child class from `multiprocessing` in a similar way.

Queue objects

- **Used by threads and multiprocessing**
- Provides a serialised method of communication
- multiprocessing also supports JoinableQueue

```
from multiprocessing import Process, Queue
import os

def my_func(*args):
    queue = args[0]

    word = ""
    while word != "END":
        word = queue.get()
        if len(word) == 7:
            print(os.getpid(), ":", word)
```

Get an item from the queue

Continued on next slide...

The **`Queue.queue`** module supplies a serialised communication mechanism between threads, and the **`multiprocessing`** module has them built-in. All the locking is done for us - we do not need to worry about atomicity and other nasty details - every operation is atomic.

Queues are ideal for the producer-consumer module, where one set of processes adds data items into the queue and another set removes and processes them. One of the good things about queues is that it does not matter if the data items take different lengths of time to process, each "thread" gets the next item from the queue when it has nothing else to do.

In the (simple) example code, we just have one producer and two consumers. The code shown above is for the consumer child processes. All it is doing is printing out each 7 character word in the queue.

The multiprocessing module also supports other synchronisation primitives like events and semaphores.

Queue objects example (2)

```
if __name__ == "__main__":
    queue = Queue()
    p1 = Process(target=my_func, args=(queue, "1"))
    p2 = Process(target=my_func, args=(queue, "2"))

    p1.start()
    p2.start()

    for line in open("words"):
        queue.put(line[:-1])

    queue.put("END")
    queue.put("END")

    p1.join()
    p2.join()
    print("All done")
```

Put an item onto the queue

Make sure there is an 'END' marker for each child process

Here is the main part of the example code, the producer.
Something which is easy to miss is that we have to send the terminator ('END') to each consumer, otherwise one would end and the producer would hang waiting for the other.
In this case, with a little more work, we could have used a JoinableQueue instead.
Can you think how we could distribute the workload without using a queue? We would have to divide the 'words' file into two, probably by record number, and give each half to each child. There might or might not have been an even distribution of 7 character words in each half and, depending on the processing to be done on each hit, one child could have finished a lot earlier than the other - possibly resulting in an idle processor.

SUMMARY

- **Running a program using the older interfaces was platform specific**
- These functions are now considered deprecated
- **The subprocess module, and the Popen method, provides a more unified approach**
- Although, there are still platform specific methods
- **The communicate method can be used to pass data through pipes**
- **Threads can create more problems than they solve**
- **For true multiprocessing, consider another way**

Running a basic process using subprocess.Popen

- **Run a process and wait for it to complete**
- **A shell is sometimes required**
  - When using shell meta-characters
  - → Wildcards, pipes, redirections, etc.
  - On Windows, no file association is done unless shell=True

```
from subprocess import *
proc = Popen("hello.py", shell=True)
proc.wait()
print("Child exited with", proc.returncode)
```

examples that follow assume this

- Don't use a shell if you don't need to
- → It can add an unnecessary overhead

Typically: C:\Python34\python.exe

```
proc = Popen([sys.executable, "hello.py"])
proc.wait()
```

Here we look at the simplest technique to run another program, which is similar in some ways to **`os.system()`**. It uses **`Popen`**, often with just the command-line as the single argument. It also introduces the **`wait`**`()` method, which sets the `returncode` attribute on completion.

The first parmeter to Popen can be any sequence - including either a string or a list. If using a string, make sure that there is at least one space between each component. For example, the final example of the slide could be written:

```
cmd = sys.executable + ' hello.py '
proc = Popen(cmd)
```

Unlike `system()`, **`Popen`** does not use a surrogate shell to run the program unless you ask it to. You will need a shell if you require shell features, such as globbing (also known as wildcards) - why not use the Python **`glob`** module instead?

On Windows, users are so used to clicking on a file and expecting "it" to run the right program that they often forget who or what "it" is. File association, associating a file extension with a particular program, is not done by the operating system, it is done by the application which launches it - Windows Explorer, or `cmd.exe` for

example. The **`subprocess`** module does not do file association, so you will need a shell to do the association for you, or add the program name yourself (which is more efficient). For Python programs, the full path name is conveniently in **`sys.executable`**. If you need file association on Windows then use **`os.startfile()`**.

Capturing the output using subprocess.Popen

- **Use the `communicate` method**
  - Returns a tuple of byte objects containing stdout, and stderr
  - Use *bytes*`.decode()` to convert bytes to string
  - Use *string*`.encode()` to convert string to bytes (for stdin)

```
proc = Popen("tasklist", stdout=PIPE, stderr=PIPE)
(output, error) = proc.communicate()

if error is not None:
    print("error:", error.decode())

print("output:", output.decode())
```

- Remember that data has to be stored in memory - too much may crash your program!

This is roughly equivalent to using `back-ticks` in UNIX shells - capturing the stdout from the child process. It is only useful with relatively small amounts of data, since the whole output is captured in memory before we can proceed.
The magic is done by the **`communicate`** method, which returns a two-element tuple consisting of the output from stdout and stderr. Both are byte-streams rather than strings, so you might have to **`decode`** them to manipulate the data.
If we only wish to capture stdout, we can slice the result:

```
output = Popen('tasklist',
stdout=PIPE).communicate()[0]
```

Passing data through a pipe using Popen

- **stdout and stdin are file objects**
  - Read program output one record at a time
  - Means there is less data held in memory

```
cmd = "gzip -dc compressed_file.gz"
proc = Popen(cmd, stdout=PIPE)
pipe = proc.stdout

for line in pipe:
    print(":", line.decode(), end="")
```

- **Write to program's input stream one record at a time**

```
cmd = "lp -"
proc = Popen(cmd, stdin=PIPE)
proc.communicate(some_data.encode())
```

Driving a program through a pipe has a number of advantages. We can capture output one record at a time, rather than waiting for the program to complete and storing the whole lot in memory. This solves the problem of running out of memory with large volumes of output data.
Passing data through stdin is less common, but can be useful. Remember that the client program will block waiting for input, and the parent will also block until the client does a read.
Notice that in both cases it is often necessary to decode or encode the data between single-byte strings and Unicode.
This is not suitable for controlling data through both pipe streams, use **`os.pipe`** instead (*not* the `pipes` module, which uses a shell).

Waiting for a child

- **`os.wait()` - UNIX only**
  - No arguments
  - Waits for the child after a `fork` or `spawn`
  - Returns a tuple of the child's PID and its exit status
  - Avoids creating a zombie
- **`os.waitpid (`*PID, options)***
  - Waits for the child after a `fork` or `spawn`

→ *PID* Process ID to wait on, -1 to wait on any child

→ *options* 0, or system specific flag, ignored on Windows

Return value as `wait()`

  - Avoids creating a zombie

If we wish to wait for our child process using the older functions, we can call os.wait, or os.waitpid.
The os.wait function waits on the last child process, so if you call os.system, close a pipe, or after the os.fork, make sure you use os.waitpid instead of os.wait.
The flags which may be supplied with waitpid are system dependant, but defined in os. Typical are WEXITSTATUS, WIFEXITED, WIFSIGNALED, and WIFSTOPPED, which are Boolean macros to be applied to the child status. WSTOPSIG and WTERMSIG give the signal which caused the child to terminate or stop. Others affect the running of waitpid:

os.WNOHANG Return at once if no child has exited

os.WUNTRACED Return for children which are stopped

A zombie is a child process which has completed but cannot end because a connection to the parent still exists. On UNIX, this is caused by an unhandled CHILD (or CHLD) signal. On Microsoft Windows, a zombie is caused by failure to close a thread or process handle. Both os.wait and os.waitpid avoid zombies, as do popen and os.system.