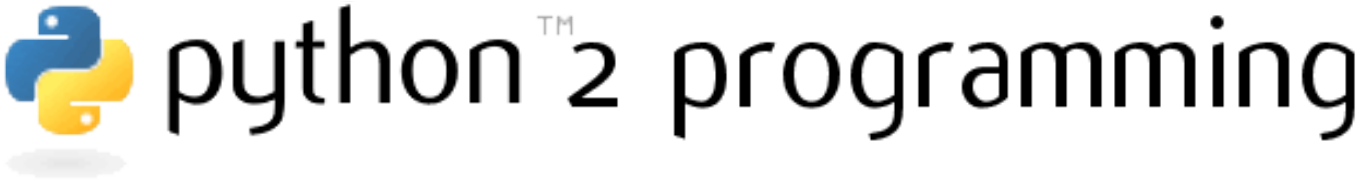
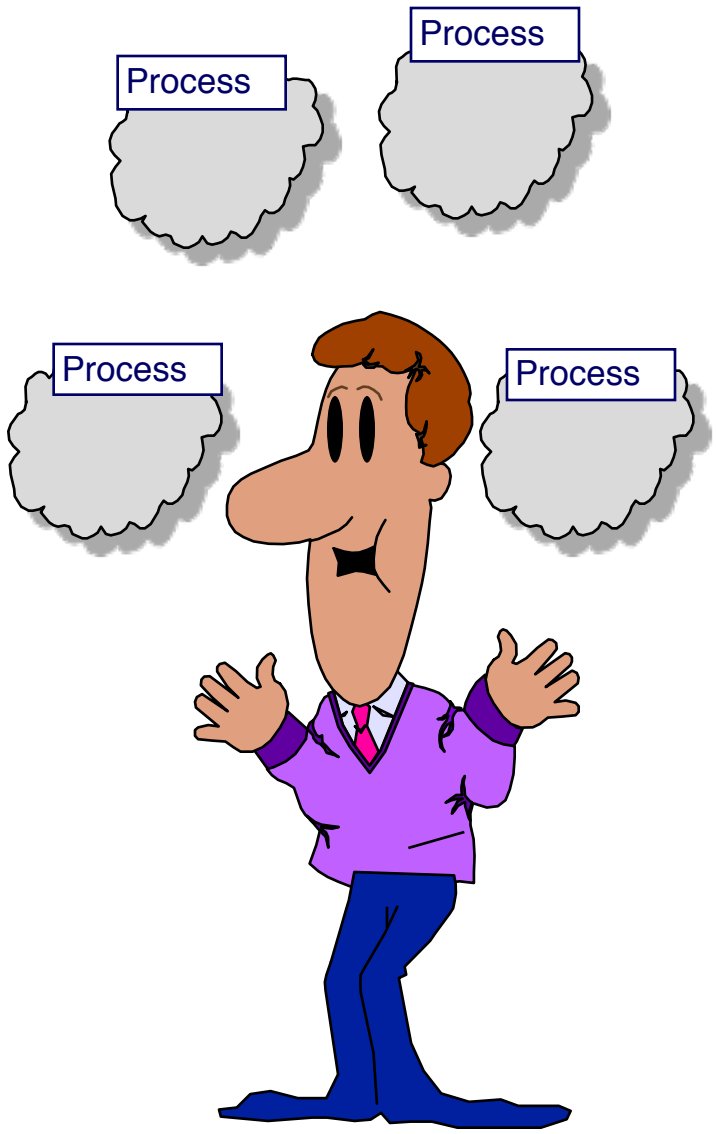


Multitasking

Multitasking

- Contents
  - Family life
  - Creating a process from Python
  - Old interface examples
  - Using the subprocess module
  - The subprocess.Popen class
  - Running a basic process
  - Capturing the output
  - Very basic threads in Python
  - Using the multiprocessing module
  - Queue objects
- Summary



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
  - Open files
- 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

Process groups are single parent families

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

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,
```

All these interfaces are deprecated

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

Using the subprocess module

- Unifying process creation
  - Introduced at Python 2.4
  - Intended to replace os.system and os.spawn
- Main method is Popen
  - Returns a subprocess object
  - Parameters are discussed over…
- 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

The subprocess.Popen class

- **Constructor parameters:**

`args`	Command-line to execute
`bufsize=0`	Buffersize, 0: unbuffered
`executable=None`	Program to be executed, rarely needed
`stdin=None`	Handle to be used for stdin (can be PIPE)
`stdout=None`	Handle to be used for stdout (can be PIPE)
`stderr=None`	Handle to be used for stderr (can be STDOUT)
`preexec_fn=None`	Code to call before the program (UNIX)
`close_fds=False`	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

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**

```
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:\Python26\python.exe

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

Capturing the output

- Use the `communicate` method
  - Returns a tuple of byte objects containing stdout, and stderr

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

if error != None:
    print "error:", error

print "output:", output
```

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

Passing data through a pipe

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

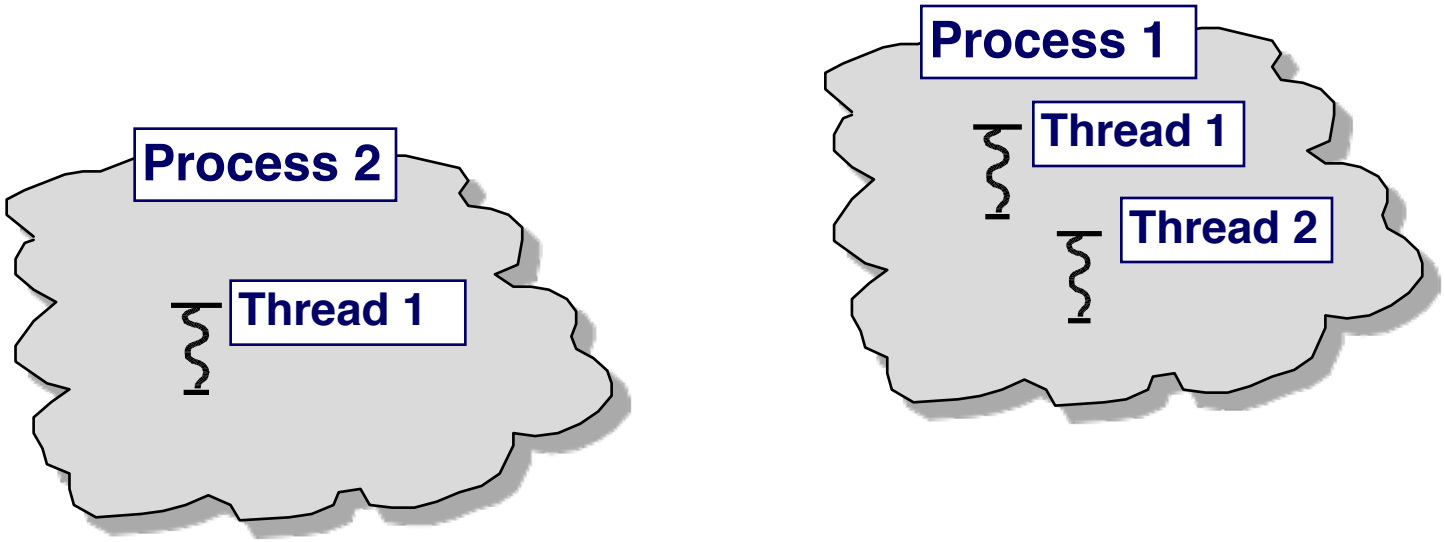
```
cmd = 'gzip -dc compressed_file.gz'
pipe = Popen(cmd, stdout=PIPE).stdout

for line in pipe:
    print ":", line,
```

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

```
cmd = 'lp -'
proc = Popen(cmd, stdin=PIPE)
proc.communicate(some_date)
```

Processes and threads

- A process is an instance of a running program
  - It owns a collection of system resources
- A thread is an asynchronous unit of execution within a process
  - A process may have several threads
- Great for spreading processing among processor cores



Very basic threads in Python

- **Python threading usually uses the threading module**
  - **Call `threading.Thread` (*function*)**

```
from threading import Thread
import time

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

th1 = Thread(target=myfunc, args='T1')
th2 = Thread(target=myfunc, args='T2')
th1.start()
th2.start()
print "From main"
th1.join()
th2.join()
```

?

```
From thread ('T'From threadFrom main,
'1'()'
T', '2')
```

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

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 3 threads to access a resource, but no more
  - **Timers**
    - Similar to waitable timers on Win32 and interval timers on UNIX

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()
```

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"

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

if __name__ == '__main__':
    p1 = Process(target=myfunc, args='T1')
    p2 = Process(target=myfunc, args='T2')
    p1.start()
    p2.start()
    print "From main"
    p1.join()
    p2.join()
```

```
From main
From proc ('T2',)
From proc ('T1',)
```

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 myfunc(*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...

Queue objects example(2)

```
if __name__ == '__main__':
    queue = Queue()
    p1 = Process(target=myfunc, args=(queue,'1'))
    p2 = Process(target=myfunc, 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

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