Red Hat Python Lab - Lesson 4: Threads, locking, shared resources, signaling

Daniel Mach <dmach@redhat.com>
Martin Sivák <msivak@redhat.com>

Useful links

- Python Quick Reference: http://rgruet.free.fr/#QuickRef
- Python docs: http://www.python.org/doc/
- PEP 8: http://www.python.org/dev/peps/pep-0008/
- pep8.py: http://pypi.python.org/pypi/pep8/
- pylint: http://www.logilab.org/project/pylint
- Epydoc: http://epydoc.sourceforge.net/

Interesting links:

- Letajici cirkus on root.cz: http://www.root.cz/serialy/letajici-cirkus/
- py.cz: http://www.py.cz/
- Unladen swallow: http://code.google.com/p/unladen-swallow/

Don't forget

- **DO NOT PANIC!**
- Batteries included -> read docs, find existing function, use it
- Follow PEP 8
- Pylint is your friend
- Write unit tests
- Objects are not everything, don't make your code over-complicated
- Return from functions as soon as possible -> more readable code, better indentation

Concurrent programming

Sometimes we need to start a task and set it aside to run almost independently while we do other stuff. There are couple of approaches to achieve this.

- Start a new **process** and use the OS' multitasking environment. Separate processes have only **limited** access to their parent's resources (filedescriptors, sockets...).
- Start a new **thread**. All threads have **full** access to process' memory and resources and are usually more lightweight, because they require less expensive context switch calls (but not always!)
- Use specially crafted "coroutines" and cooperative multitasking. This is very advanced technique used by many frameworks and different programming languages, because it easily scales to thousands concurrent "processes". It is possible to use it in Python, but is out of scope of this course.

Even in Python world not everything is perfect. When you're using threading heavily, take time to read about Global Interpreter Lock (http://www.dabeaz.com/python/GIL.pdf)

thread, threading - Python interfaces to threads

Both modules give you access to thread management in python. **thread** provides low level interface on the level of posix threads (pthread in C) while **threading** encapsulates the inteface into a set of Python objects.

```
class MyThread(threading.Thread):
  def __init__(self):
    threading.Thread.__init__(self)

  def run(self):
    my code to be ran in thread

vv -- inherited methods -- vv

  def start(self, args):
    "starts the thread and executes self.run(*args) in it"

  def join(self):
    "waits for the running thread to finish"
```

Shared resources in threads

Because threads have unlimited access to it's process' memory, caution must be excersised when manipulating it to avoid accidental memory corruptions.

To avoid a problem with two (or more) threads rewriting the same place of memory, some kind of lock mechanism has to be used. POSIX threads name the locks as **mutex**ex. Python exports them as a **threading.Lock** object.

```
import threading
lock = threading.Lock()
lock.acquire()
-- critical section --
lock.release()
```

It is important to avoid a few special situations. Most notable of them are:

- Deadlock - two threads are waiting on each other. Imagine a couple during a dinner with only one fork and one knife. Each of them hold one piece of equipment and waits for the second one to lend him the other piece.
- Starvation - the same situation, but one person has both the knife and the fork and refuses to make them available (or just grabs them back immediately after it releases them).
- Race confition - only one thread should be allowed to access the resource at the same time. Consider this code which should ensure that *x* is never less than 0.

```
10:     if (x <= 0) goto 30
20:     x = x-1
30:     goto 10
```

The simpliest way to prevent this is to remember following rules:

- Use locks to make blocks of code working with shared resources atomic.
- Always allocate and deallocate the resources in the same order. [Acq1 Acq2 Rel2 Rel1]
- Do not use active waiting and hold the resources only if you really need them.

Queue and PriorityQueue

To make your life a little simplier, Python library contains a couple of thread-safe (you do not have to lock it yourself) "types" as objects. We will need a queue of some kind.

Queue.Queue object and it's variant **Queue.PriorityQueue** (Python >= 2.6) give us very important and useful tool, because they are thread safe, so we can use them to pass information between threads without any difficult locking logic.

Queue behaves as FIFO whilst PriorityQueue returns the lowest value item first.

```
import Queue
q = Queue.Queue()
q.empty()
q.full()
q.get(block = True, timeout = None) #q.get_nowait()
q.put(item, block = True, timeout = None) #q.put_nowait()
```

Task: Create simple threaded delay loop

```
class Weather(threading.Thread):
    def __init__(self, interface):
        threading.Thread.__init__(self)

        self._queue = Queue.PriorityQueue()
        self._running = True
        self._if = interface

    def run(self):
        # the routine to be run in the thread
        # should end when self._running is False
        pass

    def stop(self):
        # pass a signal by setting the running variable to False
        pass
```

Retrieving data from HTTP

To get some data from HTTP, you can use Python provided abstraction - httplib.

httplib.HTTPConnection allows you to very simply request (or send) data over HTTP protocol. The example below should explain more:

```
import httplib
c = httplib.HTTPConnection("www.fit.vutbr.cz")
c.request("GET", "/")
r = c.getresponse()
print r.status, r.reason
data = r.read()
c.close()
print data
```

If you aren't familiar with HTTP methods, the most used are:

- GET – arguments are part of url in the form of url?arg=val&arg2=val2)
- POST – arguments are passed in request body (body argument for c.request method) one argument per line

There are also number of other methods mostly used in extensions like WebDAV, SVN, .. – PUT, DELETE, ..

Also take a look at the **urlgrabber** library. It is not a standard one, but a very usefull one.

Task: retrieve page titles from urls

Using the newly acquired re, HTTP and threading skills implement the routine which will monitor (filter) text going through the bot and when it encounters url, adds it to a queue in downloading thread. The thread will pick whatever gets to the queue and download it, it will then find the <title></title> tags and put the contents to the interface output.

```
> ahoj
ahoj
> i found a super website http://www.fedora.cz, you should take a look
i found a super website http://www.fedora.cz, you should take a look
> it is really great
it is really great
.... some time
www.fedora.cz: Fedora.cz | Česká komunita linuxové distribuce Fedora
>
```

The regular expression to check a link could be for example **http://(?P<server>[a-zA-Z0-9\-\.]+\.[a-z]{2,6})(?P<url>(/\S*)*)**

Interruptable waiting

Sometimes we have really nothing to do, but a new assignment can arrive at any moment. In that case just sleeping for finite amount of time is not a good idea. But there is solution to this problem in the threading module also.

threading.Event class supports setting the event to "it happened" state or to "nothing has happened" state. Thread can then use Event.wait(timeout) to wait for the event but no more than *timeout* seconds. Compare it with time.sleep(time) which always waits for the specified amount of time.

```
event = threading.Event()
event.wait([timeout=SECONDS])  # wait until the event flag is set or a timeout is reached
event.set()   # set the event flag, terminates waiting
event.clear() # clear the event flag
```