

A Survey of Stackless Python

by

Andrew Francis

af.stackless@gmail.com

Montreal Python User's Group

September 30th, 2009

October 14 2009 Notes

- This is the first version of “Survey of Stackless Python” given at the September 30th Montreal Python talk.
 - Sorry for the roughness
- A second version will be available by early November
- See <http://andrewfr.wordpress.com>

Errata

- Page 22, added extra state 'paused'
- Page 32, removed 'associated'.
- Page 35, added iterable.
- Page 42, removed "and response time" (big conceptual mistake. I should know better)
- Page 52, twisted.web not twisted.internet
- Page 55, client.page() not client.page

What is Stackless Python

- A superset of Python (CPython)
 - Any script executable in CPython is executable in Stackless
- Created by Christian Tismer.

What is Stackless Python

- If known, has a reputation for
 - light-weight threads that facilitate development of massively concurrent programmes.
 - ‘infinite recursion.’
- However there is far more to Stackless Python than this!

Motivation for Talk

- Watching Joe Gregorio's *(The Lack of) Design Patterns in Python*
- Gregorio shows a Python based prime number sieve using PyCSP.
 - Inspired by Robert Pike's Newsqueak example
 - Newsqueak inspired by CSP
- Purpose of exercise: to illustrates use of a high level concurrency construct: *channel*

The Problem

2609

Exception in thread Thread-381:

Traceback (most recent call last):

File "/usr/local/lib/python2.6/threading.py", line 525, in __bootstrap_inner
self.run()

File "build/bdist.linux-i686/egg/pycsp/threads/process.py", line 65, in run
self.fn(*self.args, **self.kwargs)

File "S.py", line 22, in worker

Spawn(worker(IN(child_channel), cout))

File "build/bdist.linux-i686/egg/pycsp/threads/process.py", line 123, in Spawn
_parallel(plist, False)

File "build/bdist.linux-i686/egg/pycsp/threads/process.py", line 135, in _parallel
p.start()

File "/usr/local/lib/python2.6/threading.py", line 471, in start
_start_new_thread(self.__bootstrap, ())

error: can't start new thread

- Reason:
 - Sieve algorithm creates a thread for each prime!
 - PyCSP using OS threads.
 - (Gregorio suggests that greenlets could be used).

Big Picture

- Conclusions of Gregorio talk
 - If one has to implement a design pattern, consider adding features to the language.
 - Concurrency patterns are a rich area
- I can understand this :-)

Great Talk but

- Wouldn't Stackless Python be a better example?
- After all, Stackless Python has those language feature goodies such as:
 - coroutines
 - channels
- And Stackless Python implements much of a concurrency design pattern: *Active Object*.

The Active Object Design Pattern

- A.K.A – *Actor*
- Intent
 - Decouple method invocation from method execution to *enhance concurrency* and *simplify synchronized access* to an object that resides in its own thread of control [Schmitt & Lavender]
- Prominent features
 - A scheduler
 - Queues

The Last Straw

INT. AUDITORIUM QUESTION PERIOD

GREGORIO

Any questions?

AUDIENCE MEMBER STEPS UP TO THE MICROPHONE

AUDIENCE MEMBER

Have you played with Stackless Python and its approach to channels and sub-threads?

GREGORIO

No.

AUDIENCE MEMBER

Okay. If you are interested, it has a very similar approach.

My Reaction

Argggggggghhhh!

(I know I am over-reacting)

Stackless Python Implementation

```
import stackless
```

```
def filter(prime, listen, send):  
    for i in listen:  
        if (i % prime):  
            send.send(i)
```

```
def counter(c):  
    i = 2  
    while (True):  
        c.send(i)  
        i += 1
```

```
def sieve(prime, c):  
    for p in c:  
        prime.send(p)  
        newc = stackless.channel()  
        stackless.tasklet(filter)(p, c, newc)  
        c = newc
```

```
prime = stackless.channel()  
c = stackless.channel()  
stackless.tasklet(counter)(c)  
stackless.tasklet(sieve)(prime, c)
```

```
while(True):  
    print prime.receive()
```

Comments

- Stackless Python example almost as terse as Robert Pike's Newsqueak example.
- Application has no problem finding 9973 (on my machine roughly ten seconds).
- More of a problem finding 1,149,193!

The Case for Stackless

- Stackless Python's strength is not only its light-weight threads.
- Rather it is its high level concurrency model.
- The clutter-free concurrency model allows programmers to focus on algorithms rather than mechanics
- Isn't this what being Pythonic is about?

Part II – Feature Overview

Getting Stackless Python

- Acquiring Stackless Python

- svn checkout <http://svn.python.org/projects/stackless/tags/python-3.01/>
- Tarballs for Stackless Python versions 2.4 to 3.1
- Binary installers for Windows and the Mac
- Thank CCP Games for timely updates of Stackless Python.

- Stackless Python Wiki

- <http://www.stackless.com>

Running Stackless

```
andrew@andrew-laptop:~/MontrealPython$ python
Python 2.6.2 Stackless 3.1b3 060516 (python-2.6.2:74783M, Sep 13 2009,
09:43:23)
[GCC 4.3.3] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> import stackless
>>> dir(stackless)
['__doc__', '__name__', '__reduce__', '__reduce_ex__', '_gc_track',
'_gc_untrack', '_pickle_moduledict', '_wrap', 'bomb', 'cframe', 'channel',
'cstack', 'enable_softswitch', 'get_thread_info', 'getcurrent', 'getmain',
'getruncount', 'run', 'schedule', 'schedule_remove', 'set_channel_callback',
'set_schedule_callback', 'slpmodule', 'stackless', 'tasklet', 'test_cframe',
'test_cframe_nr', 'test_outside']
```

Running Stackless

```
andrew@andrew-laptop:~/MontrealPython$ python
Python 2.6.2 Stackless 3.1b3 060516 (python-2.6.2:74783M, Sep 13 2009,
09:43:23)
[GCC 4.3.3] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> import stackless
>>> help(stackless.channel)
=Help on class channel in module stackless:
```

```
class channel(__builtin__.channel)
| A channel object is used for communication between tasklets.
| By sending on a channel, a tasklet that is waiting to receive
| is resumed. If there is no waiting receiver, the sender is suspended.
| By receiving from a channel, a tasklet that is waiting to send
| is resumed. If there is no waiting sender, the receiver is suspended.
```

The 'Stackless' in Stackless

- Python activation frames stored on the heap rather than the C-Stack
 - stack overflow becomes a memory error *
- This enables
 - Creation of coroutines
 - Pickling of execution state.

Stackless Python's Model

- Small number of classes
 - Tasklets
 - concurrency
 - The Scheduler
 - Channels
 - Communications and synchronisation

The Stackless Scheduler

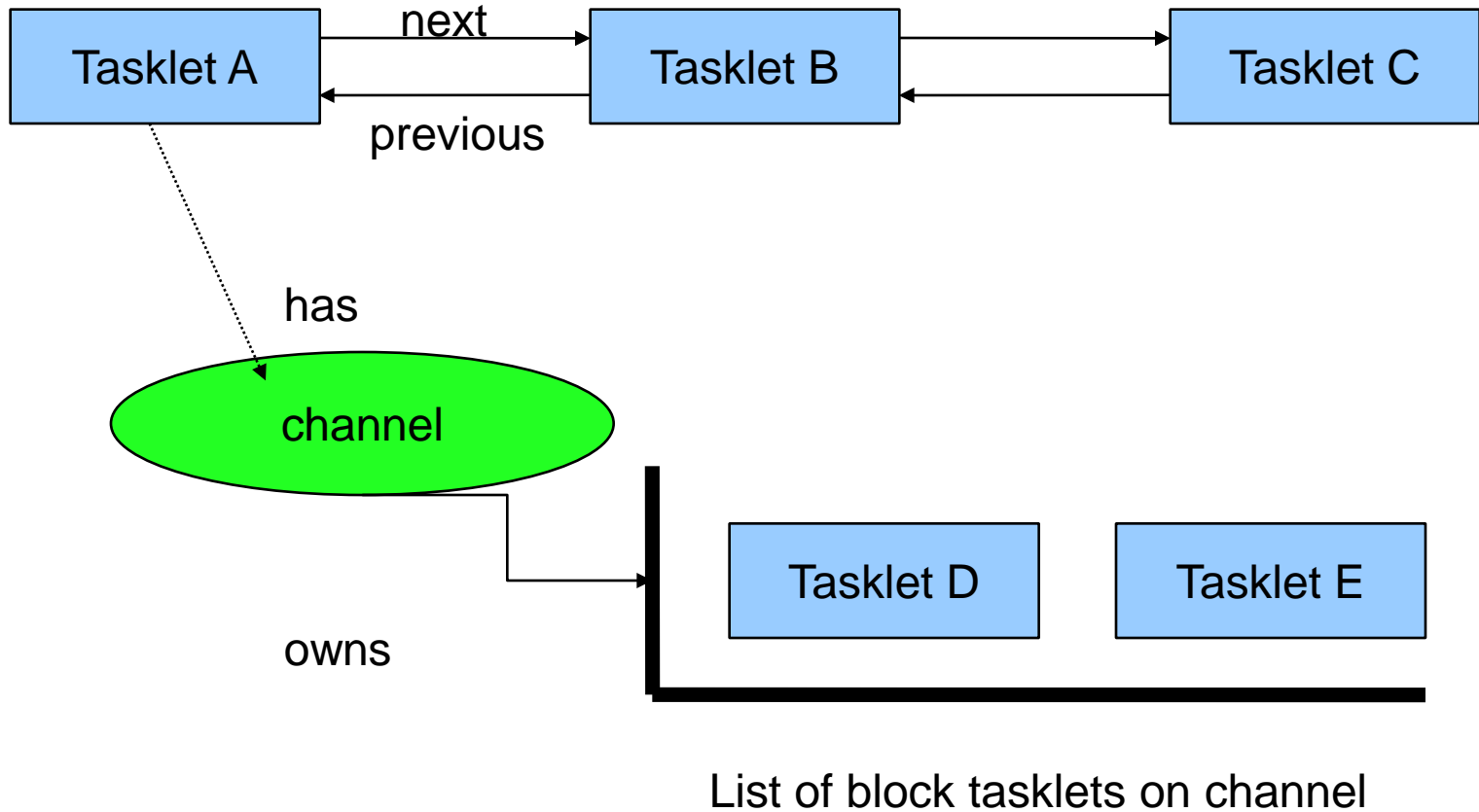
- 'User space' scheduler
- Performs context switches on behalf of the programmer *.
- Tasklets have at least four states
 - Currently executing
 - runnable (scheduled)
 - Blocked
 - paused

Scheduling Algorithm

- Uses round robin algorithm
 - Tasklets that can run are stored on a 'runnable' list
 - Runnable is a doubly linked list
 - Blocked tasklets stored on a list associated with a tasklet's channel.
 - `schedule.getcurrent()` indicates currently running tasklet (the head)

High Level View of Scheduler

Runnable list



Multi-tasking Modes

- Scheduler supports two multi-tasking modes:
 - Preemptive multi-tasking
 - Tasklets give a 'CPU' quota based on instruction ticks.
 - Cooperative multi-tasking
 - Tasklets explicitly yields control to the scheduler

- Talk will focus on cooperative scheduling
 - Far simpler model!
 - Ideal for I/O bound applications
 - Main problem we are dealing with is handling multiple concurrent events.

Tasklets

- Stackless class that represents a thread of execution
 - The class where work gets done.
 - Each tasklet has its own stack frame
- Lives in 'user space'
 - Underlying operating system ignorant of tasklets

Tasklets Continued

- Tasklets have very low overhead
 - Roughly 400 bytes
- Very fast context switching
 - Soft-switching 100 X an OS thread
 - Hard-switching 10 X an OS thread
 - (source “Stackless Python and PyPy: Nuts and Bolts’)

Simple Example

```
import stackless
```

```
def HelloWorld():  
    print "hello world"
```

```
stackless.tasklet(HelloWorld)()  
stackless.run()
```

```
$python helloWorld.py  
hello world
```

More Complex Example

```
import stackless

def numbers():
    for number in [0, 1, 2]:
        print number,
        stackless.schedule()

def letters():
    for letter in ['A','B','C']:
        print letter,
        stackless.schedule()

stackless.tasklet(numbers)()
stackless.tasklet(letters)()
stackless.run()
```

0 A 1 B 2 C

Channels

- The heart of Stackless Python's concurrency model.
- Used for both synchronisation and communications
- Inspired by channels from Newsqueak and Limbo
 - An important source of idioms

Channels continued

- Bi-directional
- Unbuffered
- References including channels and exceptions can be passed in a channel!

Synchronous Communications

- Channels are synchronous
 - If there is no receive, sender blocks
 - If there is no sender, receiver blocks
- Channels are iterable!
 - We saw this in the Prime sieve example
- Straightforward to build more complex solutions from channels.

Simple Example

```
import stackless

def first(channel):
    channel.send("first always starts")
    print channel.receive()

def second(channel):
    message = channel.receive()
    print message
    channel.send("second received message")

channel = stackless.channel()
stackless.tasklet(second)(channel)
stackless.tasklet(first)(channel)
stackless.run()
```

output:
first always starts
second received message

Deadlock

- Channels and message passing is simple .
- However, synchronous nature of channels and co-operative scheduling allow deadlock conditions to occur.
- Often this silent!
 - Source of confusion to the newbie

Example

“””

Tasklet A is waiting for Tasklet B to send a message on ch1 before sending on channel ch2. Tasklet B is waiting for Tasklet A receive a message on ch2 before sending on ch1

“””

```
import stackless
```

```
def task(x, y, message):  
    print "tasklet", stackless.getcurrent()  
    print x.receive()  
    print y.send()
```

```
ch1 = stackless.channel()  
ch2 = stackless.channel()  
stackless.tasklet(task)(ch1, ch2, "Tasklet A sending message")  
stackless.tasklet(task)(ch2, ch1, "Tasklet B sending message")  
stackless.run()
```

A Simplified Dump

```
stackless.run()  
print ch1.__reduce__()  
print ch2.__reduce__()
```

OUTPUT: (-1 tasklet is a receiver)

```
tasklet A  
tasklet B  
(CH1, (), (-1, 1610612736, [A]))  
(CH2, (), (-1, 1610612736, [B]))
```

A Common Idiom - Interface/Protocol

- Common Stackless idiom is to use two channels as a part of request/response protocol
 - Request channel is hidden via a Proxy/Interface.
 - Reply channel is sent as a part of message to request handler
 - Requesting tasklet blocks under message arrives.

```
import stackless

requestChannel = stackless.channel()

def requestInterface(message):
    global requestChannel
    replyChannel = stackless.channel()
    requestChannel.send((message, replyChannel))
    return replyChannel.receive()

def requestHandler(requestChannel):
    for name, replyChannel in requestChannel:
        replyChannel.send("reply for request " + name + "
received")

def client(name):
    print name + " " + requestInterface(name)

for name in ['A','B','C','D','E']:
    stackless.tasklet(client)(name)
stackless.tasklet(requestHandler)(requestChannel)
stackless.run()
```

Channel Preferences

- A limited form of schedule prioritization
- A property of the channel
 - -1 receiver preference
 - 1 sender preference
 - 0 Neutral
- If the preferenced tasklet is blocked, it is made current
- If the preferenced tasklet is running, it continues to run
- Can allow channels to simulate queues

```
import stackless
```

```
PREFERENCE = -1
```

```
def p(channel):  
    while (True):  
        i = channel.receive()  
        print "P" + str(i),
```

```
def c(i, channel):  
    channel.send(i)  
    print "R" + str(i),
```

```
channel = stackless.channel()  
channel.preference = PREFERENCE  
stackless.tasklet(p)(channel)  
for i in range(0,10):  
    stackless.tasklet(c)(i, channel)  
stackless.run()
```

Preferences

- Receiver preference
 - P0 P1 P2 P3 P4 P5 P6 P7 P8 P9 R0 R1 R2 R3
R4 R5 R6 R7 R8 R9
- Sender preference
 - R0 P0 R1 P1 R2 P2 R3 P3 R4 P4 R5 P5 R6 P6
R7 P7 R8 P8 R9 P9
- If numbers sufficiently large, has an impact on execution time.

That's all folks for Part II

Part III

Limitations

Limitations

- Stackless Python uses the Global Interpreter Lock (GIL)
 - Cannot take advantage of multiple CPUs
- Stackless Python tasklets are in user space

GIL

- When CPython and/or PyPy based Python implementation stops using the GIL, Stackless Python will follow.
- Happy this is out of the way.

User Space Threads

- Stackless Python tasklets and scheduler run in the operating system's user space.
- When tasklet makes a system call that blocks, the operating system blocks the OS thread that is associated with the tasklet.
- Often this means, all the tasklets associated with the blocked OS thread, are blocked!

Solution

- This problem is typically associated with network programming.
- The solution is to use tasklets in conjunction with an asynchronous I/O package.
 - Asyncore
 - Libevent
 - Twisted
- Integration technique is similar.

A Happy Consequence

- Integration techniques uses a design pattern called 'Half Sync/Half Async' [Schmitt & Cranor]
- A major consequence is one gets a good compromise between the programming 'ease' of a synchronous model combined with the performance of an asynchronous model.

Chapter Four

Networking Techniques

A Crash Course on Reactors

- Most asynchronous I/O packages based on the Reactor Pattern.
 - Operating support, i.e `select()`, `epoll()`, `ioCompletion` ports.
- Reactors are associated with Event driven programming.
- Reactors provide a limited form of non-preemptive multi-tasking without 'threads'

A Crash Course on Reactors Continued

- General strategy
 - Client registers a callback function associated with an event (i.e recv) with the reactor.
 - When event occurs, reactor calls callback passing it a result
- We wont get into the pros and cons of asynchronous programming.

Simple Asynchronous Programming example

```
from twisted.internet import reactor
from twisted.web import client

def success(page):
    print page
    reactor.stop()

def failure(errorData)
    print errorData
    reactor.stop()

deferred = client.getPage("http://www.google.com")
deferred.addCallback(success).addErrback(failure)
reactor.run()
```

Integration technique

- Associate a callback with a channel

Example

```
# technique used by Christopher Armstrong
def blockOn(deferred):
    channel = stackless.channel()
    def success(data):
        return (True, data)
    def failure(data):
        return (False, data)
    deferred.addCallback(success).addErrback(failure)
    return channel.receive()
.
# result is a tuple (status, data)
result = blockOn(client.getPage())
```

Integration Technique Two

- Execute the reactor in its own tasklet
 - `stackless.tasklet(reactor.run)()`
- Now reactor is just another tasklet!
- From reactor's point of view, tasklets are callbacks!

Make Reactor Tasklet Yield

- Even though underlying I/O call may timeout, reactor does not know about Stackless
- Consequencely reactor will not yield (call `stackless.schedule`).
- Tasklets that can run, will not get a chance.
- Must make reactor implementation yield

has to do this because of a bug between twisted and Stackless 2.6

```
def __tick__():  
    stackless.schedule()  
t = task.loopingCall(__tick__)  
t = l.start(.1)  
stackless.tasklet(reactor.run)()
```

```
def client(url1, url2):
    # note we can now use reactor API in a synchronous fashion!
    print blockOn(client.getPage(url1))
    print blockOn(client.getPage(url2))

t = task.loopingCall(__tick__)
t.start(.1)
stackless.tasklet(reactor.run)()
stackless.tasklet(client)('http://www.google.com', 'http://www.mcgill.ca')
stackless.run()
```

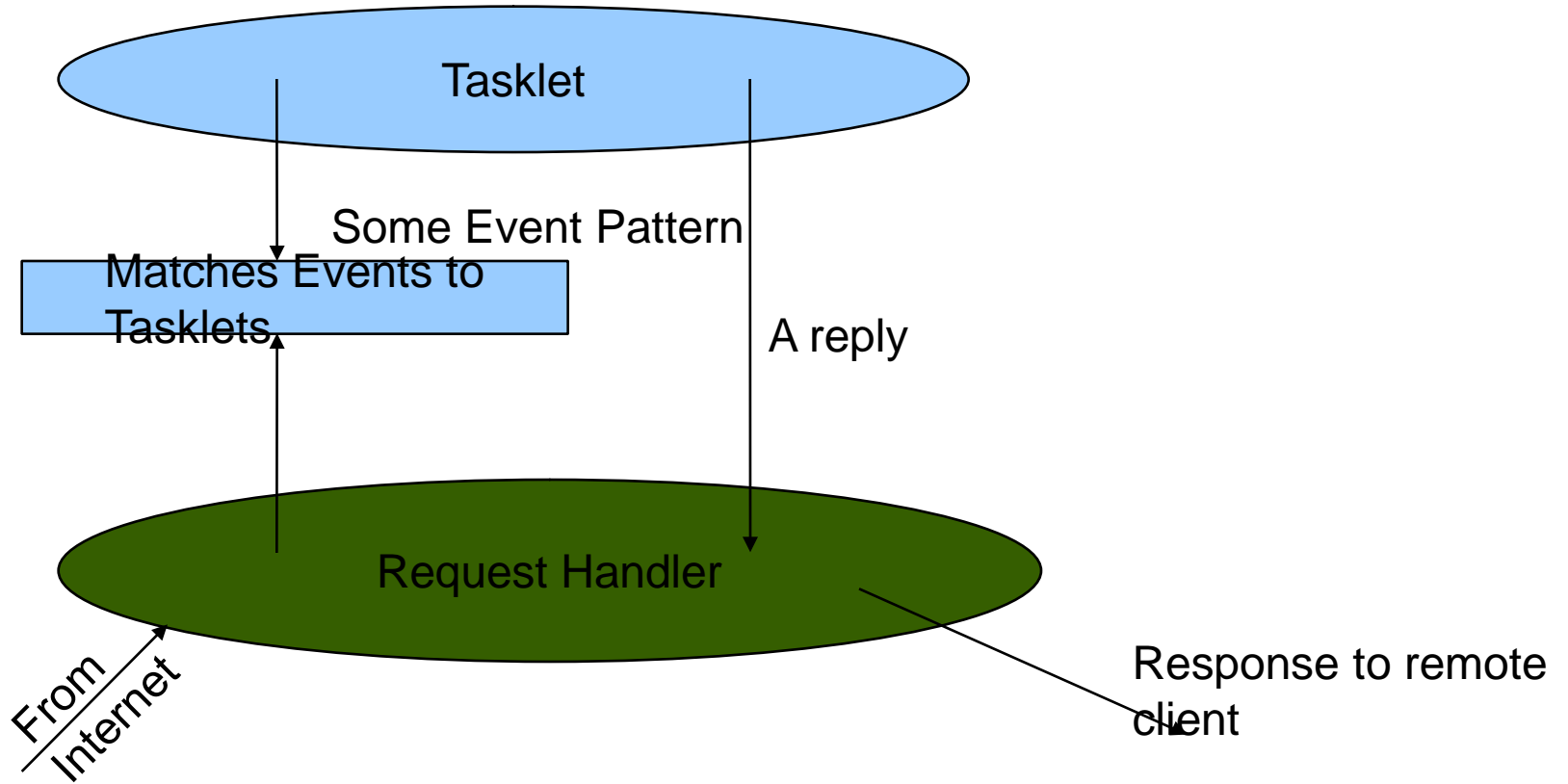
Server Side

- So far, examples have focused on web clients.
- Similar technique for servers
 - Tasklet can rendezvous with a protocol instance (a HTTP connection) via a channel.
 - A tasklet can be created for each new connection.

```
class MyRequestHandler(http.Request):
    def process(self):
        stackless.tasklet(self.doWork)()

    def doWork(self):
        channel = stackless.channel()
        MyRequestHandler.responseChannel.send(self.path,
\
                                                self.content.read(),\
                                                channel)
        reply = channel.receive()
        self.write(reply, message.encode('utf-8'))
        self.finish()
        return
```

Diagram



Final Word - Stacklesssocket

- Plug in replacement for sockets
 - Technique developed by Andrew Dalke and Richard Tew.
- Idea is to create proxy class that implements the socket interface but uses a channel in conjunction with asynchronous module
- For many third party libraries, this just works

The Future

- Two tracks
 - C based Stackless Python develops in step with CPython
 - PyPy
- PyPy is a Python based framework for developing interpreters
 - Original idea was to prototype future Python features in Python (and eliminate need for C extensions)

The Future

- Stackless module has been implemented in PyPy
 - Written in Python!
 - Great way to learn about Stackless's model
- Stackless's model has been generalised
 - Greenlets
 - Coroutines
- Opens the door for experimentation!