

# Using Stackless

Andrew Dalke / [dalke@dalkescientific.com](mailto:dalke@dalkescientific.com)

# Stackless is Python

(with a few additions)

# Stackless is CPython

(with a few modifications)

```
[josiah:~] dalke% spython
Python 2.5 Stackless 3.1b3 060516 (release25-maint:53731,
Feb 10 2007, 23:25:41)
[GCC 4.0.1 (Apple Computer, Inc. build 5367)] on darwin
Type "help", "copyright", "credits" or "license" for more
information.
>>> print "Hello, PyCon 2007!"
Hello, PyCon 2007!
>>>
```

# Stackless adds:

tasklets

channels

cooperative multitasking

# Stackless vs. Threads

tasklet(f)  
channel()  
    .send(x)  
    .receive()  
    .balance

~

Thread(target=f)  
Queue(1)  
    .put(x)  
    .get()

run()  
schedule()    } tasklet scheduling

```
import stackless
```

```
def writeln(s):  
    print s
```

```
stackless.tasklet(writeln)("Hello, PyCon 2007!")  
stackless.run()
```

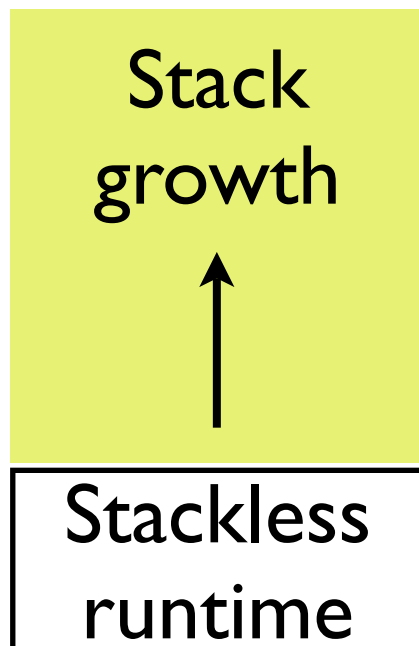
```
% spython simple.py  
Hello, PyCon 2007!
```

# Starting a new tasklet

```
import stackless
```

```
def writeln(s):  
    print s
```

```
stackless.tasklet(writeln)("Hello, PyCon 2007!")  
stackless.run()
```







```
import stackless
```

```
def writeln(s):  
    print s
```

```
stackless.tasklet(writeln)("Hello, PyCon 2007!")  
stackless.run()
```



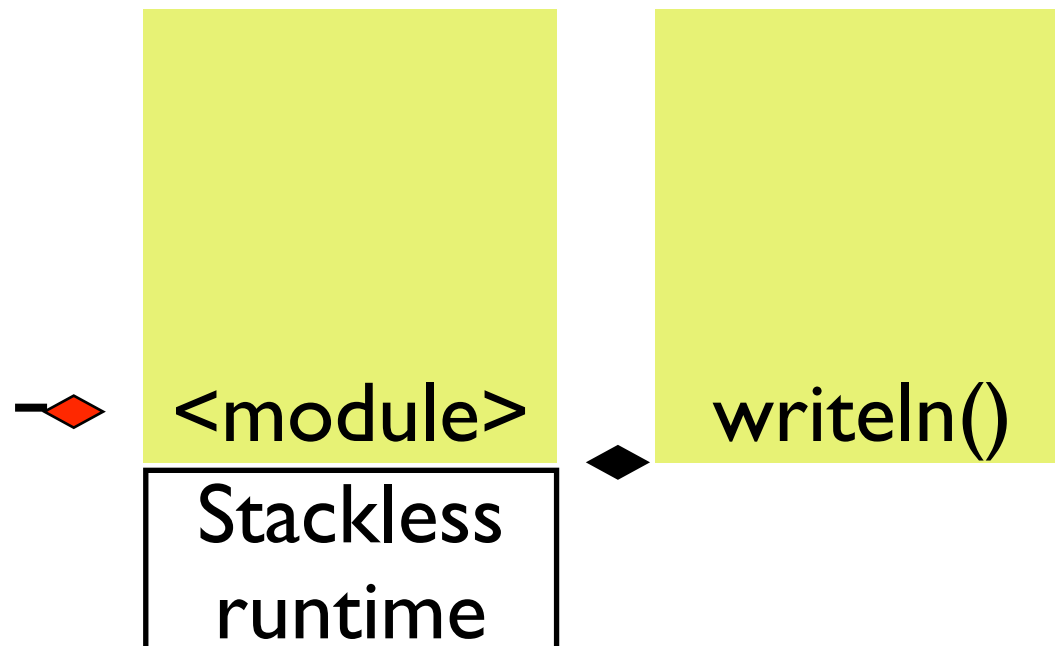
<module>

Stackless  
runtime

```
import stackless
```

```
def writeln(s):  
    print s
```

```
→ stackless.tasklet(writeln)("Hello, PyCon 2007!")  
stackless.run()
```

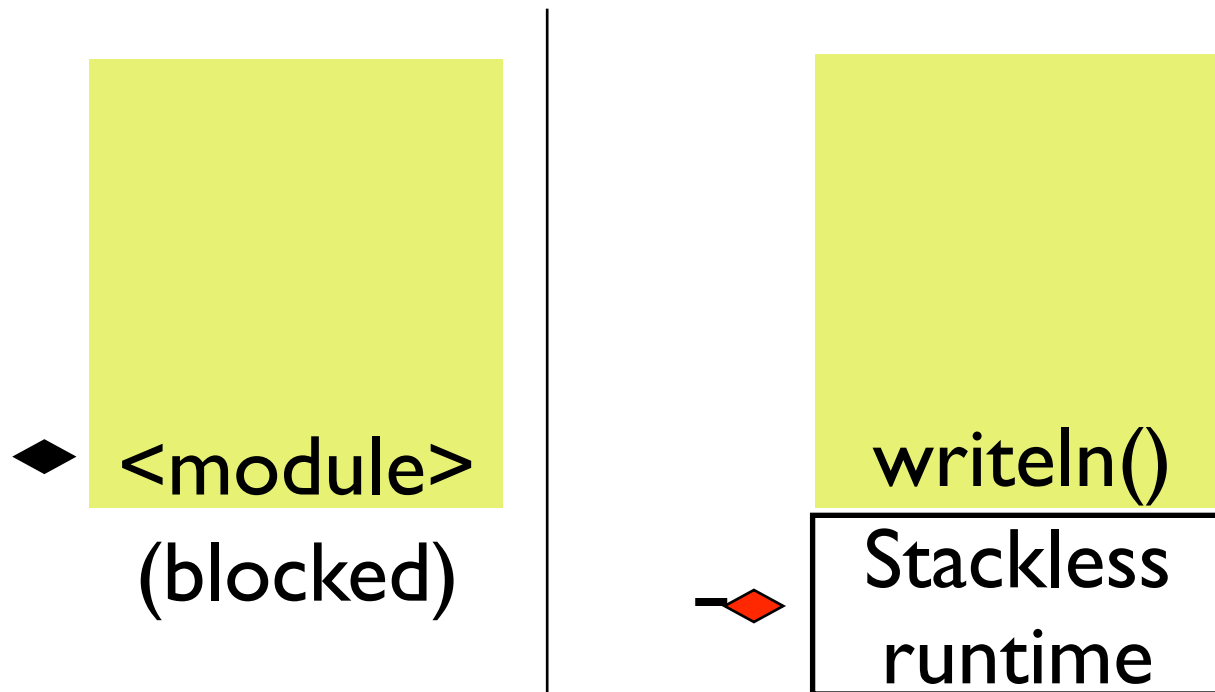


```
import stackless
```

```
def writeln(s):  
    print s
```

```
stackless.tasklet(writeln)("Hello, PyCon 2007!")
```

```
→ stackless.run()
```



```
import stackless
```

```
→ def writeln(s):  
    print s
```

```
stackless.tasklet(writeln)("Hello, PyCon 2007!")  
stackless.run()
```

◆ <module>

→ writeln()  
Stackless  
runtime

```
import stackless
```

```
def writeln(s):
```

```
    print s
```

```
stackless.tasklet(writeln)("Hello, PyCon 2007!")
```

```
stackless.run()
```

Hello, PyCon 2007!



<module>



print  
writeln()

Stackless  
runtime

```
import stackless
```

```
def writeln(s):
```

```
    print s
```

Hello, PyCon 2007!

```
stackless.tasklet(writeln)("Hello, PyCon 2007!")
```

```
stackless.run()
```

◀ <module>

◀ Stackless  
runtime

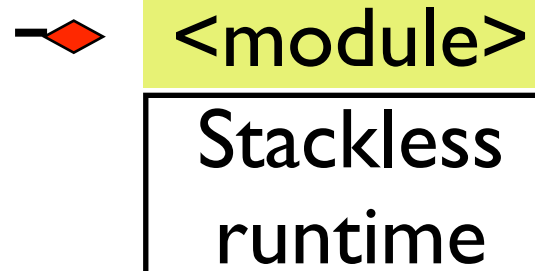
```
import stackless
```

```
def writeln(s):  
    print s
```

```
stackless.tasklet(writeln)("Hello, PyCon 2007!")
```

```
→ stackless.run()
```

Hello, PyCon 2007!



```
import stackless
```

```
def writeln(s):  
    print s
```

```
stackless.tasklet(writeln)("Hello, PyCon 2007!")  
stackless.run()
```

Hello, PyCon 2007!

... and Stackless exits



Stackless  
runtime



```
import stackless

def greet(s):
    print "Hello,", s

stackless.tasklet(greet)("A")
stackless.tasklet(greet)("B")
stackless.run()
```

```
Hello, A
Hello, B
```

# Starting two tasklets



```
import stackless
```

```
def greet(s):  
    print "Hello,", s
```

```
stackless.tasklet(greet)("A")  
stackless.tasklet(greet)("B")  
stackless.run()
```



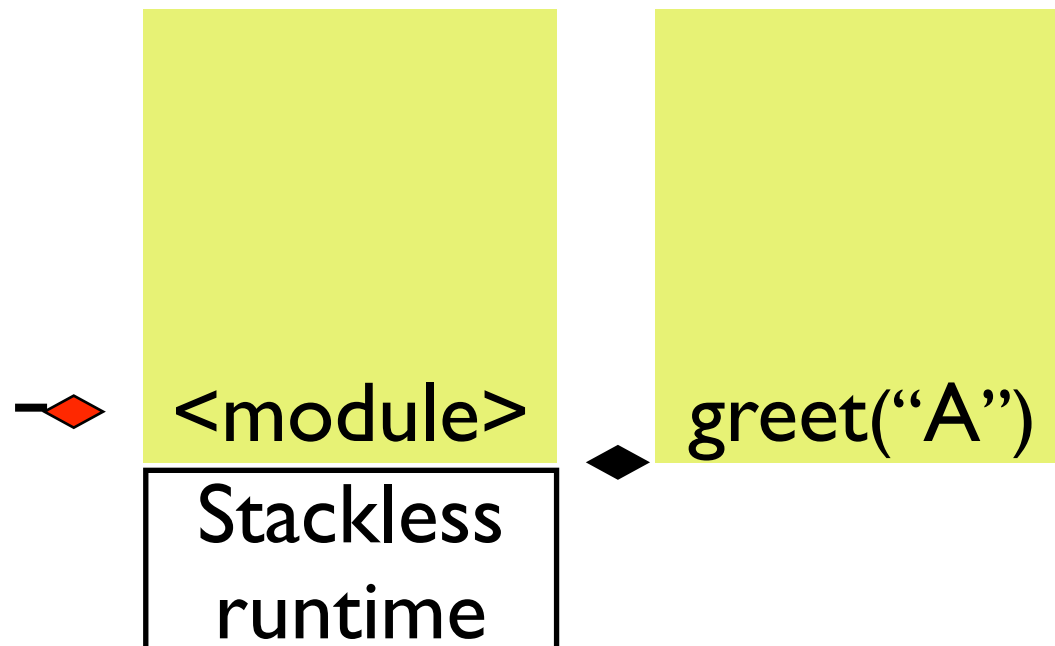
<module>

Stackless  
runtime

```
import stackless
```

```
def greet(s):  
    print "Hello,", s
```

```
➡ stackless.tasklet(greet)("A")  
   stackless.tasklet(greet)("B")  
   stackless.run()
```

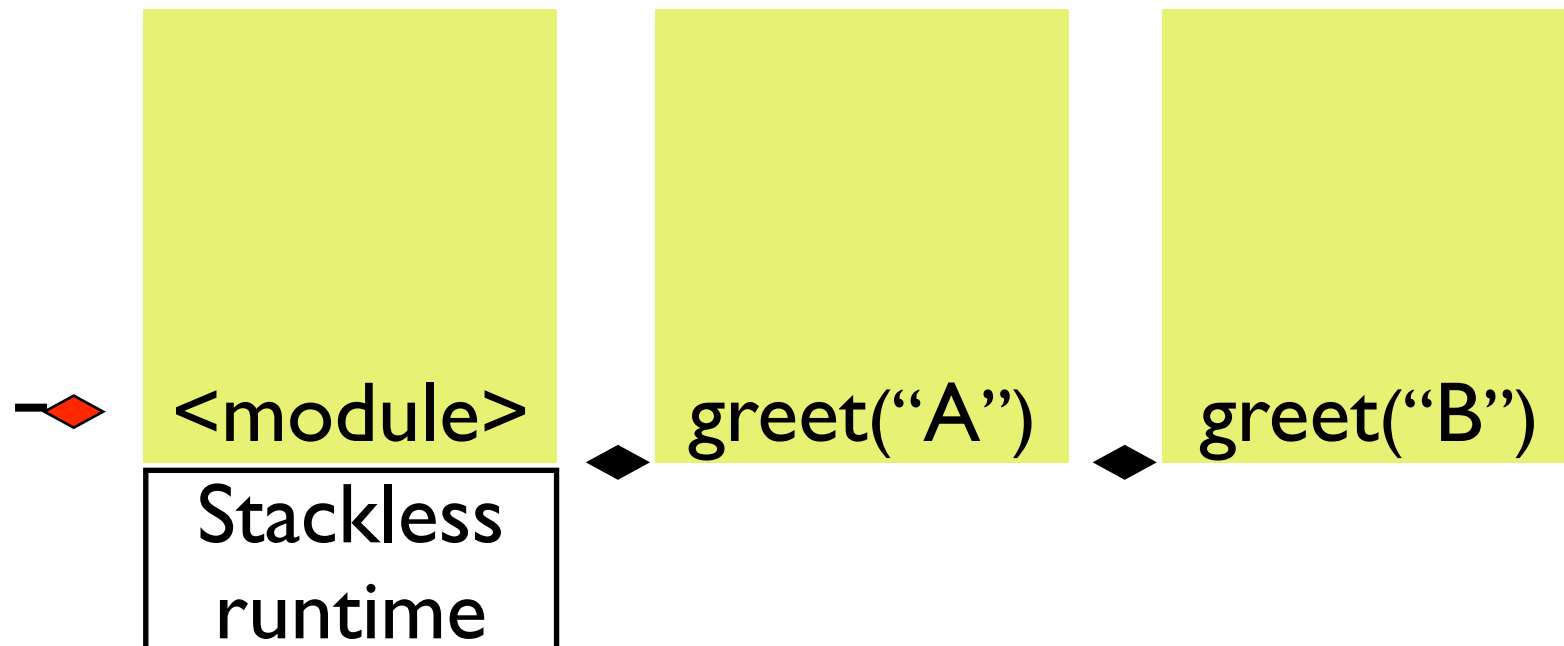


```
import stackless
```

```
def greet(s):  
    print "Hello,", s
```

```
stackless.tasklet(greet)("A")
```

```
→ stackless.tasklet(greet)("B")  
stackless.run()
```



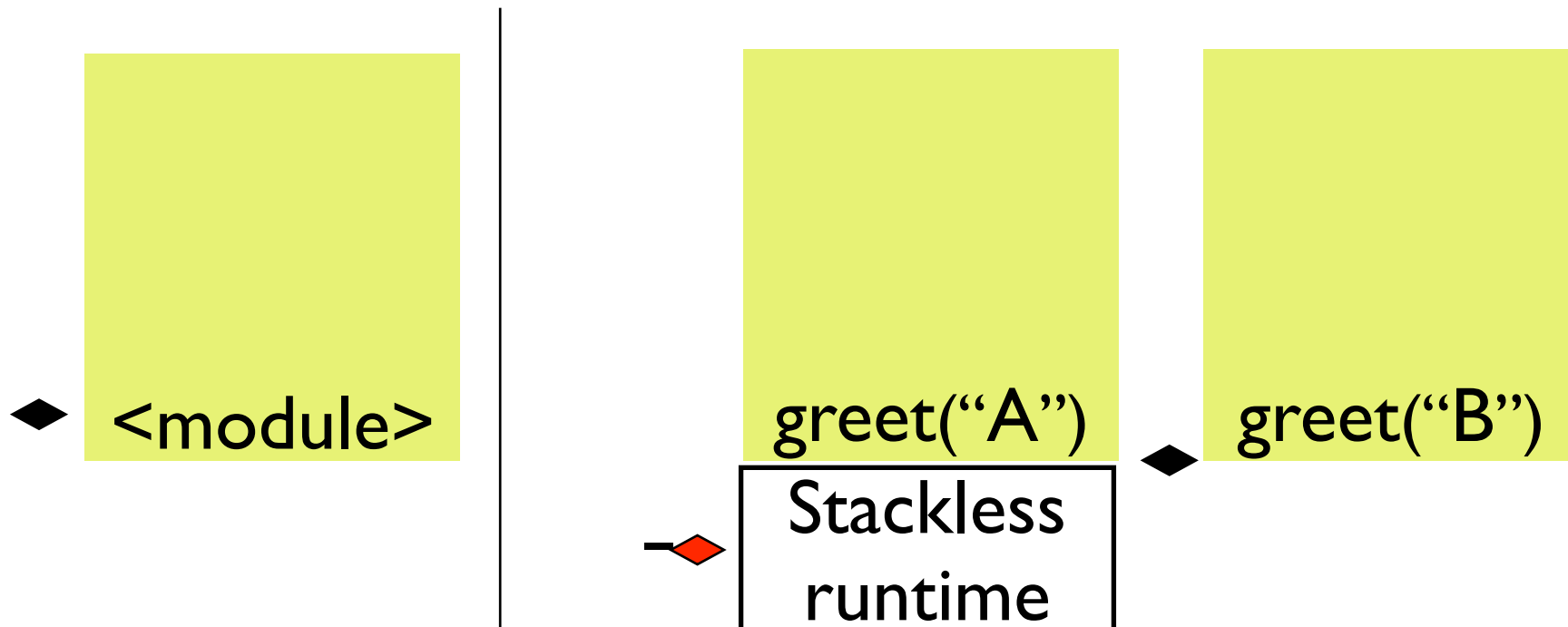
```
import stackless
```

```
def greet(s):  
    print "Hello,", s
```

```
stackless.tasklet(greet)("A")
```

```
stackless.tasklet(greet)("B")
```

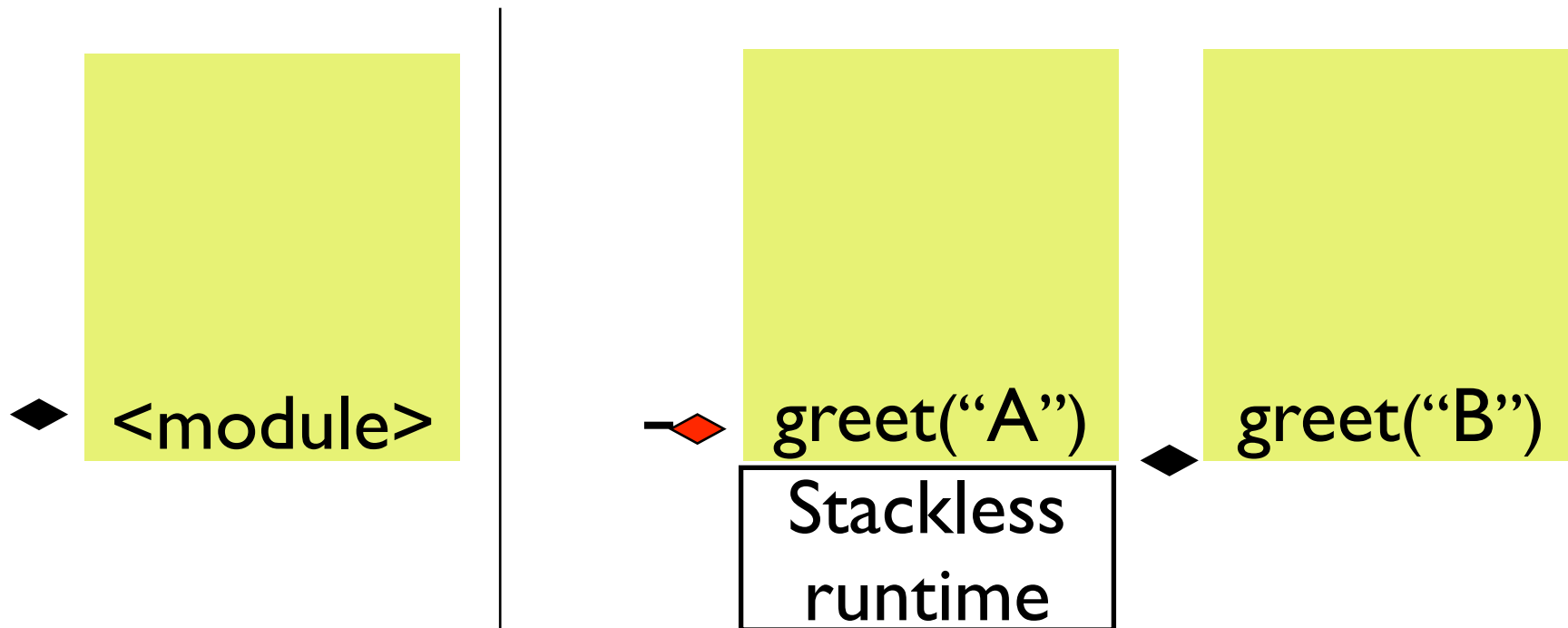
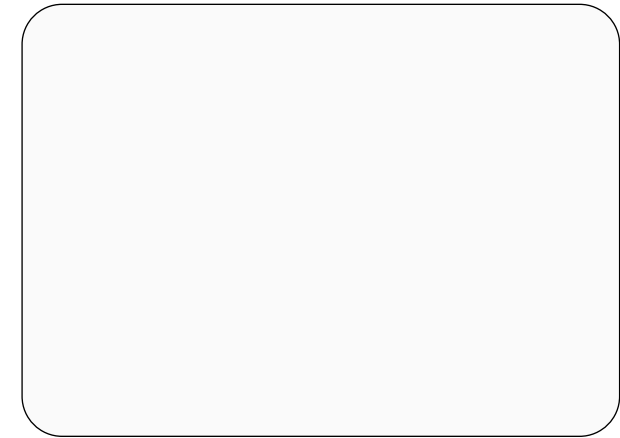
```
➡ stackless.run()
```



```
import stackless
```

```
➤ def greet(s):  
    print "Hello,", s
```

```
stackless.tasklet(greet)("A")  
stackless.tasklet(greet)("B")  
stackless.run()
```



```
import stackless
```

```
def greet(s):
```

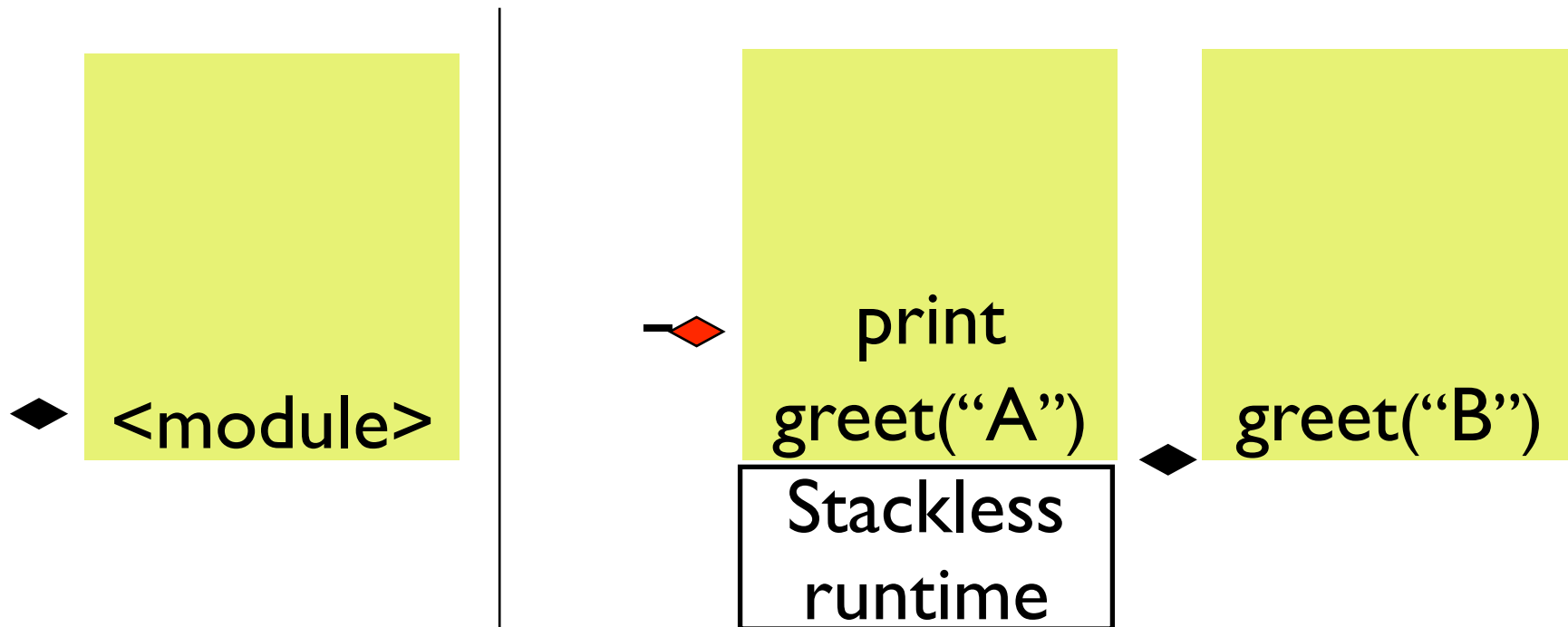
```
    print "Hello,", s
```

```
stackless.tasklet(greet)("A")
```

```
stackless.tasklet(greet)("B")
```

```
stackless.run()
```

Hello, A



```
import stackless
```

```
def greet(s):
```

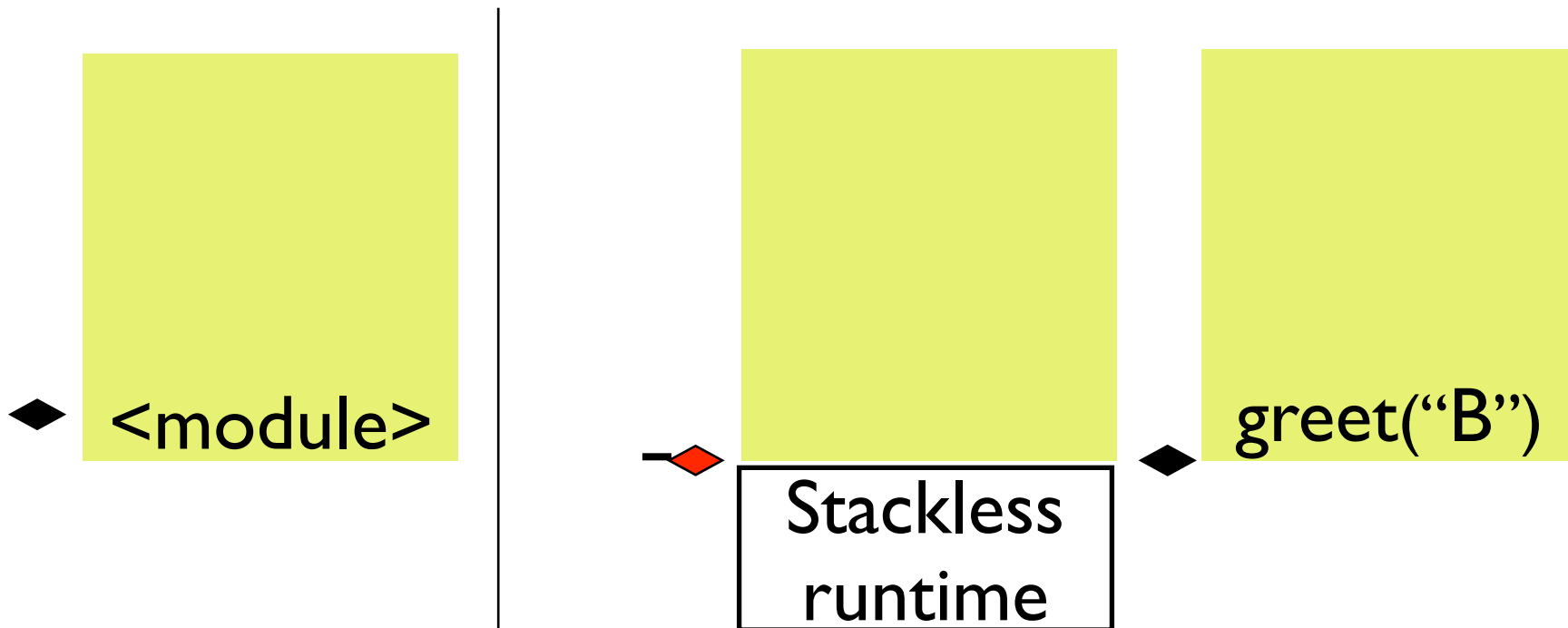
```
    print "Hello,", s
```

```
stackless.tasklet(greet)("A")
```

```
stackless.tasklet(greet)("B")
```

```
stackless.run()
```

Hello, A





```
import stackless
```

```
➡ def greet(s):  
    print "Hello,", s
```

```
stackless.tasklet(greet)("A")  
stackless.tasklet(greet)("B")  
stackless.run()
```

Hello, A

◆ <module>

➡ greet("B")

Stackless  
runtime

```
import stackless
```

```
def greet(s):
```

```
    print "Hello,", s
```

```
stackless.tasklet(greet)("A")
```

```
stackless.tasklet(greet)("B")
```

```
stackless.run()
```

Hello, A  
Hello, B

◀ <module>

◀ print  
greet("B")

Stackless  
runtime

```
import stackless
```

```
def greet(s):
```

```
    print "Hello,", s
```

```
stackless.tasklet(greet)("A")
```

```
stackless.tasklet(greet)("B")
```

```
stackless.run()
```

Hello, A  
Hello, B

<module>

Stackless  
runtime

```
import stackless
```

```
def greet(s):  
    print "Hello,", s
```

```
stackless.tasklet(greet)("A")
```

```
stackless.tasklet(greet)("B")
```

```
stackless.run()
```



Hello, A  
Hello, B



<module>

Stackless  
runtime

... and Stackless exits

```
import stackless
```

```
def greet(s):  
    print "Hello,", s  
    stackless.schedule()  
    print "Goodbye", s
```

```
stackless.tasklet(greet)("A")  
stackless.tasklet(greet)("B")  
stackless.run()
```

```
Hello, A  
Hello, B  
Goodbye A  
Goodbye B
```

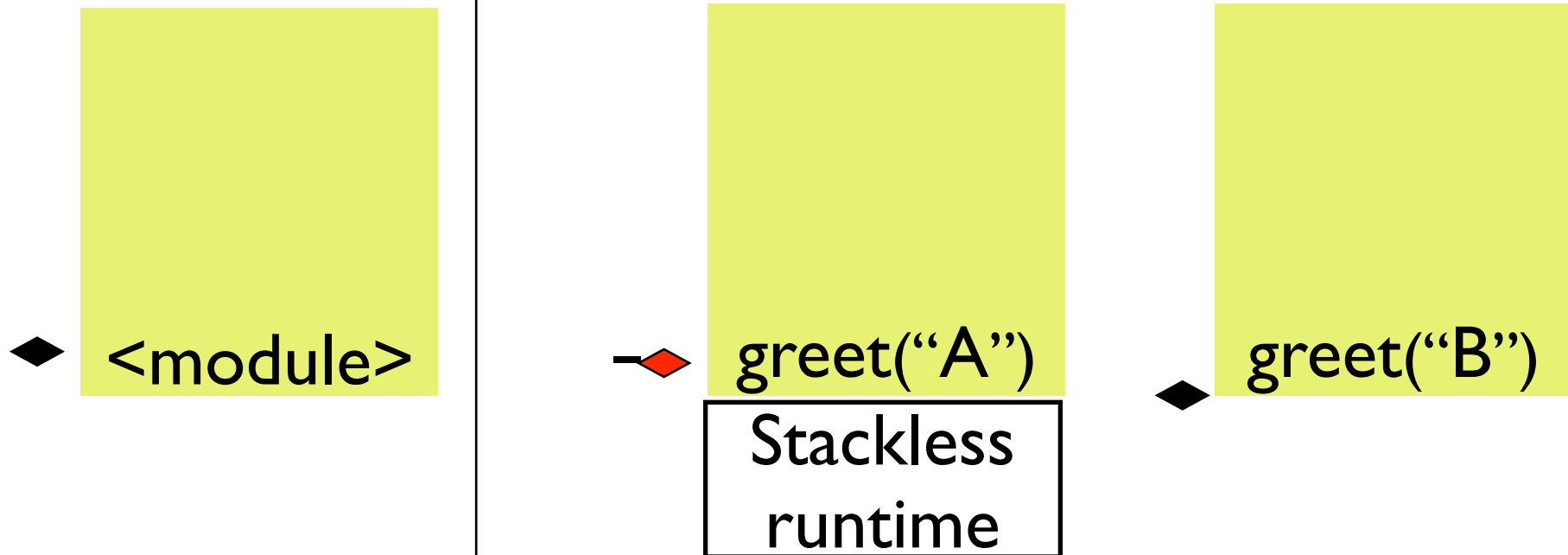
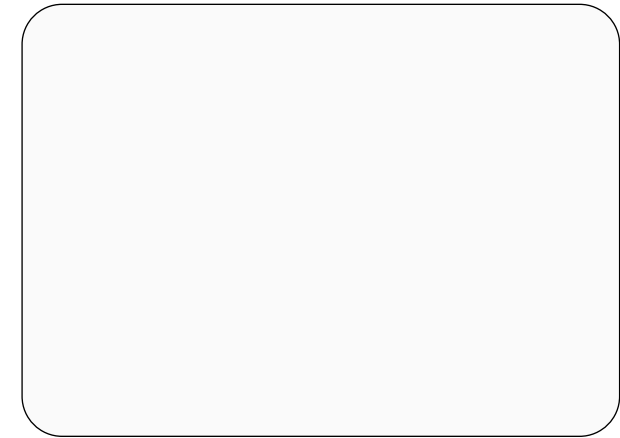
# Round-robin scheduling

Yield control using “schedule()”

```
import stackless
```

```
➤ def greet(s):  
    print "Hello,", s  
    stackless.schedule()  
    print "Goodbye", s
```

```
stackless.tasklet(greet)("A")  
stackless.tasklet(greet)("B")  
stackless.run()
```



```
import stackless
```

```
def greet(s):
```

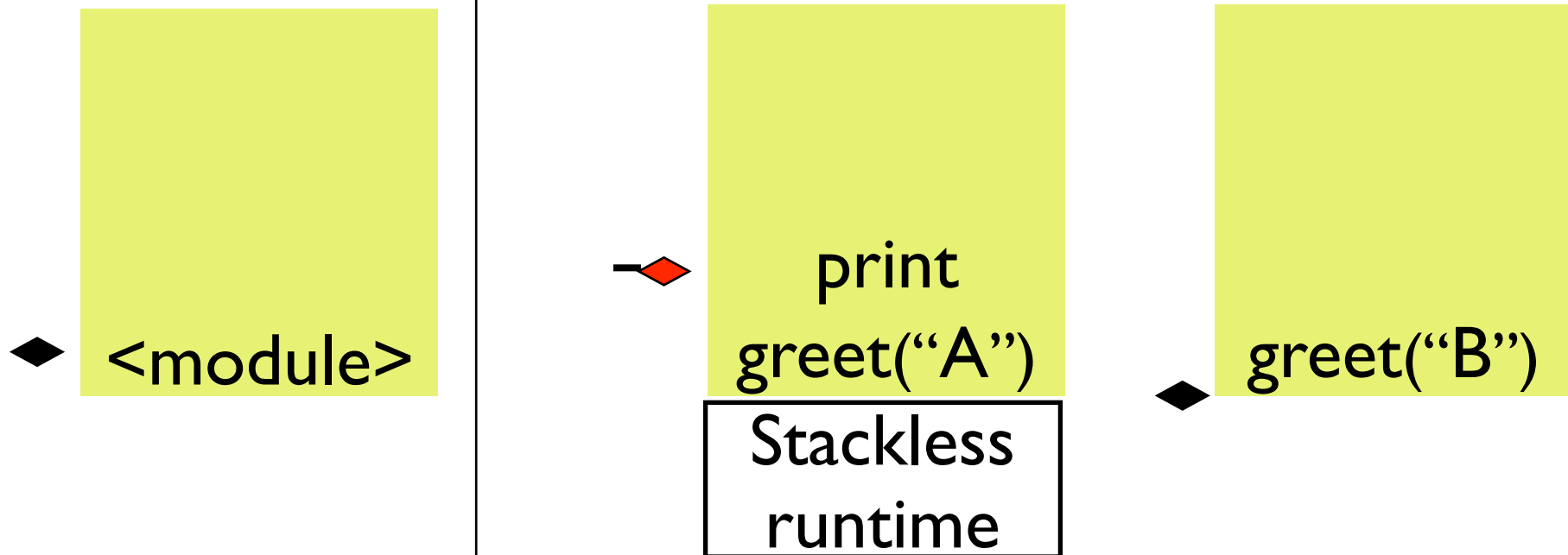
```
    print "Hello,", s  
    stackless.schedule()  
    print "Goodbye", s
```

```
stackless.tasklet(greet)("A")
```

```
stackless.tasklet(greet)("B")
```

```
stackless.run()
```

Hello, A



```
import stackless
```

```
def greet(s):
```

```
    print "Hello,", s
```

```
    ◀ stackless.schedule()
```

```
    print "Goodbye", s
```

```
stackless.tasklet(greet)("A")
```

```
stackless.tasklet(greet)("B")
```

```
stackless.run()
```

Hello, A

◀ <module>

◀ schedule()  
greet("A")

Stackless  
runtime

◀ greet("B")

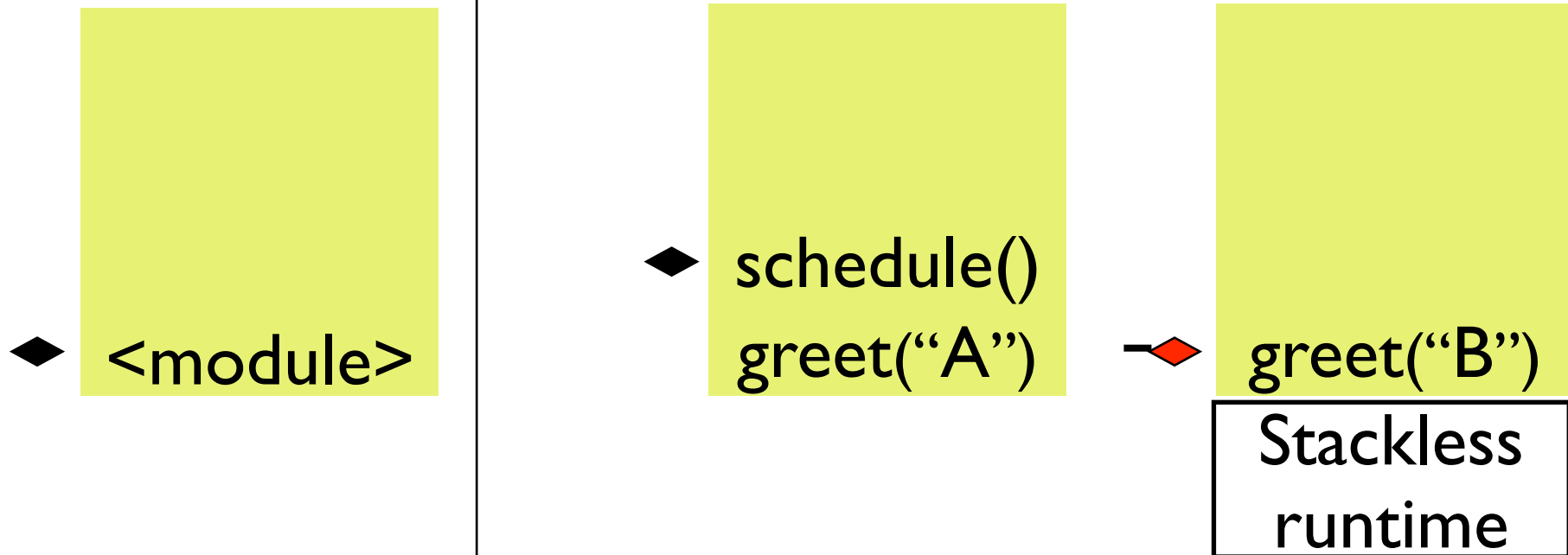


```
import stackless
```

```
➤ def greet(s):  
    print "Hello,", s  
    stackless.schedule()  
    print "Goodbye", s
```

```
stackless.tasklet(greet)("A")  
stackless.tasklet(greet)("B")  
stackless.run()
```

Hello, A



```
import stackless
```

```
def greet(s):
```

```
    print "Hello,", s  
    stackless.schedule()  
    print "Goodbye", s
```

```
stackless.tasklet(greet)("A")
```

```
stackless.tasklet(greet)("B")
```

```
stackless.run()
```

Hello, A  
Hello, B

◆ <module>

◆ schedule()  
greet("A")

◆ print  
greet("B")

Stackless  
runtime

```
import stackless
```

```
def greet(s):
```

```
    print "Hello,", s
```

```
    ◀ stackless.schedule()
```

```
    print "Goodbye", s
```

```
stackless.tasklet(greet)("A")
```

```
stackless.tasklet(greet)("B")
```

```
stackless.run()
```

Hello, A  
Hello, B

◀ <module>

◀ schedule()  
greet("A")

◀ schedule()  
greet("B")

Stackless  
runtime

```
import stackless
```

```
def greet(s):
```

```
    print "Hello,", s
```

```
    ◀ stackless.schedule()
```

```
    print "Goodbye", s
```

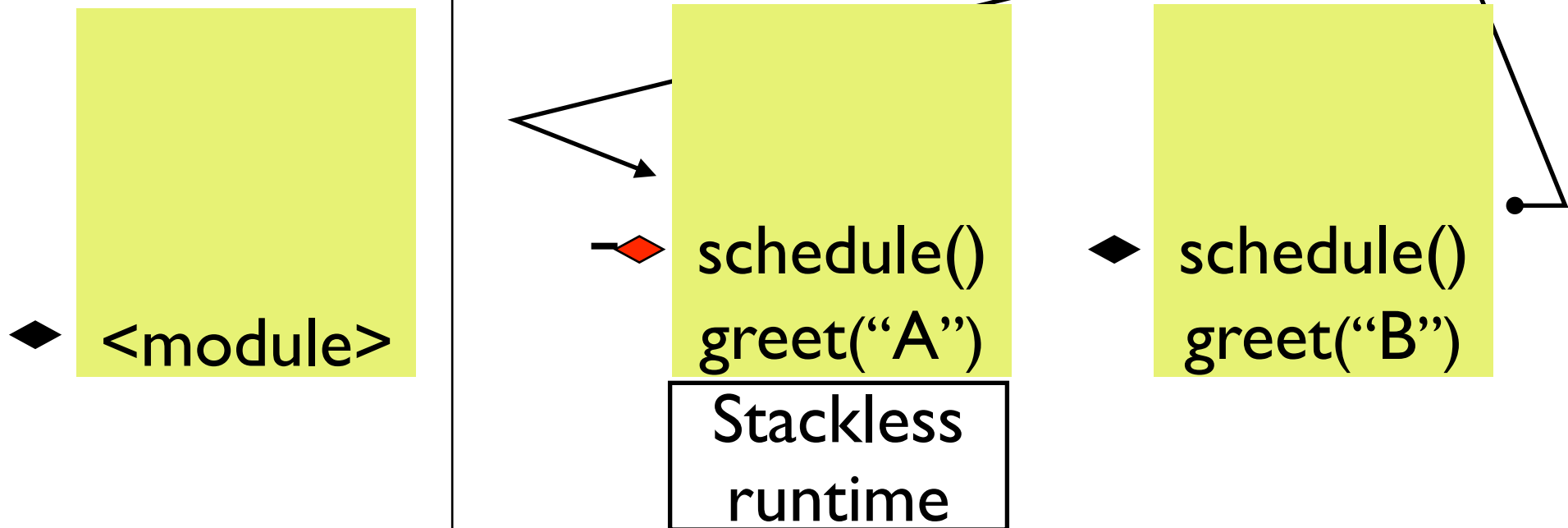
```
stackless.tasklet(greet)("A")
```

```
stackless.tasklet(greet)("B")
```

```
stackless.run()
```

Hello, A  
Hello, B

round-robin  
to 1<sup>st</sup> tasklet

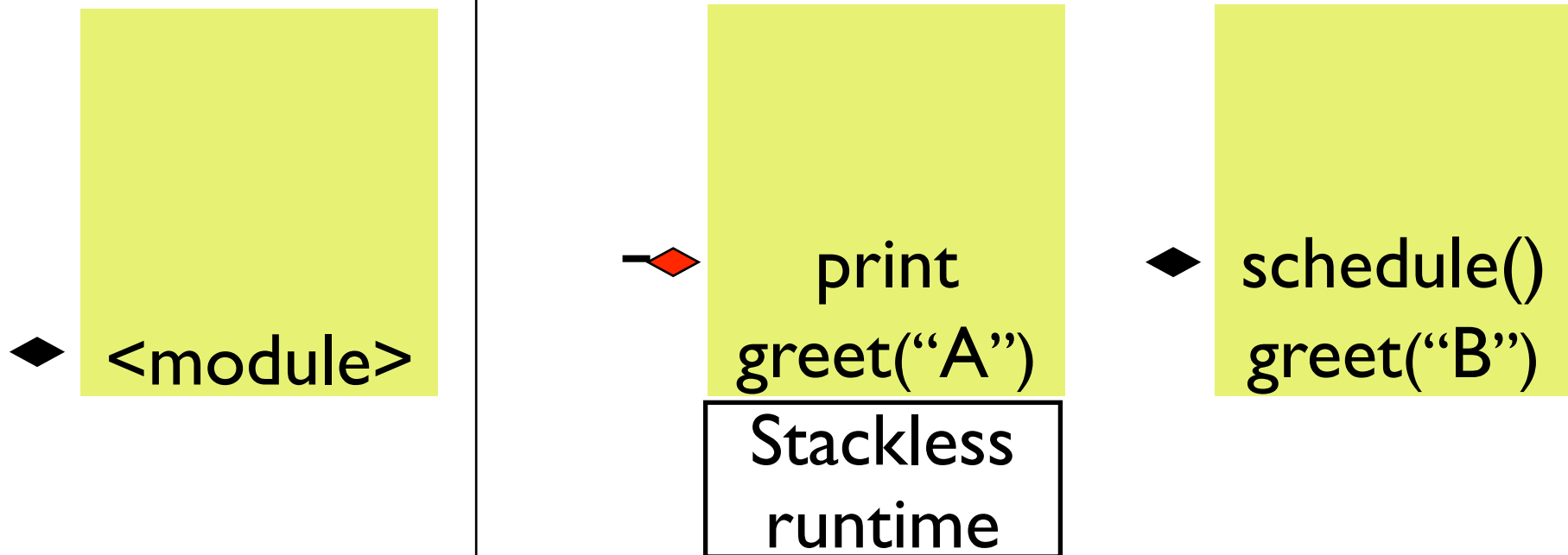


```
import stackless
```

```
def greet(s):  
    print "Hello,", s  
    stackless.schedule()  
    print "Goodbye", s
```

Hello, A  
Hello, B  
Goodbye, A

```
stackless.tasklet(greet)("A")  
stackless.tasklet(greet)("B")  
stackless.run()
```

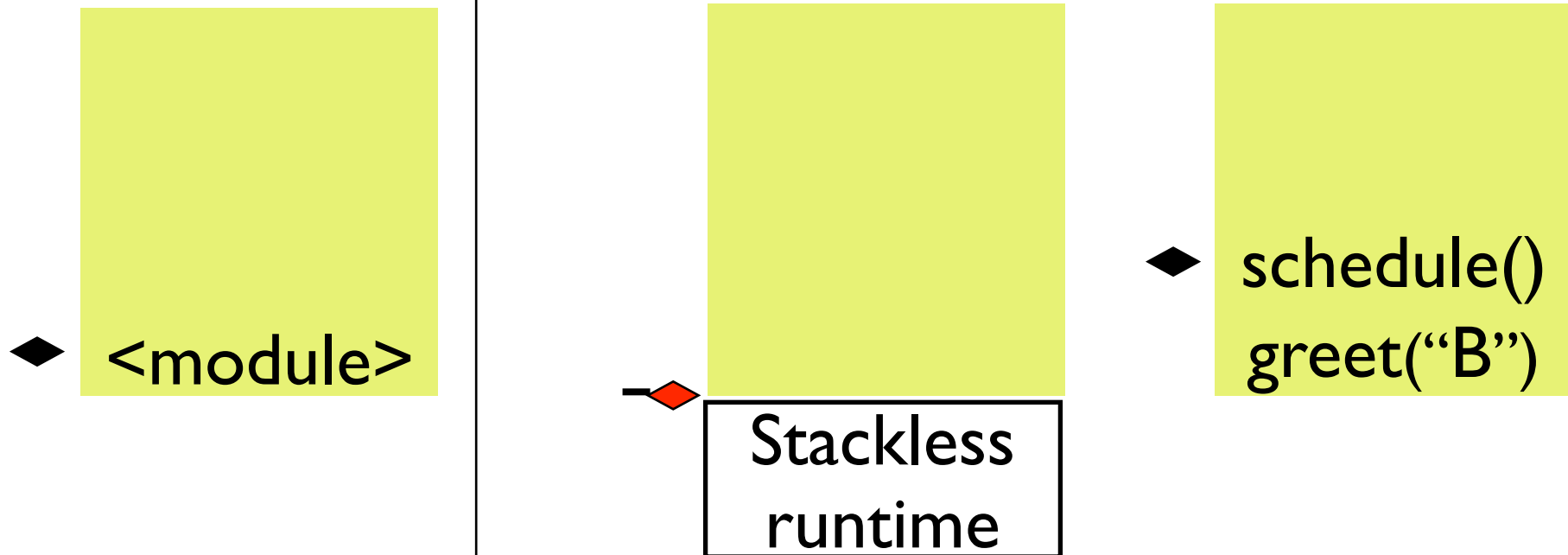


```
import stackless
```

```
def greet(s):  
    print "Hello,", s  
    stackless.schedule()  
    print "Goodbye", s
```

Hello, A  
Hello, B  
Goodbye, A

```
stackless.tasklet(greet)("A")  
stackless.tasklet(greet)("B")  
stackless.run()
```



```
import stackless
```

```
def greet(s):
```

```
    print "Hello,", s
```

```
    ↖ stackless.schedule()
```

```
    print "Goodbye", s
```

```
stackless.tasklet(greet)("A")
```

```
stackless.tasklet(greet)("B")
```

```
stackless.run()
```

Hello, A

Hello, B

Goodbye, A

◆ <module>

↖ schedule()  
greet("B")

Stackless  
runtime

```
import stackless
```

```
def greet(s):
```

```
    print "Hello,", s
```

```
    stackless.schedule()
```

```
    print "Goodbye", s
```

```
stackless.tasklet(greet)("A")
```

```
stackless.tasklet(greet)("B")
```

```
stackless.run()
```

◆ <module>

Hello, A  
Hello, B  
Goodbye, A  
Goodbye, B

◆ print  
greet("B")

Stackless  
runtime



```
import stackless
```

```
def greet(s):
```

```
    print "Hello,", s
```

```
    stackless.schedule()
```

```
    print "Goodbye", s
```



```
stackless.tasklet(greet)("A")
```

```
stackless.tasklet(greet)("B")
```

```
stackless.run()
```



<module>

Hello, A

Hello, B

Goodbye, A

Goodbye, B



Stackless  
runtime

```
import stackless
```

```
def greet(s):  
    print "Hello,", s  
    stackless.schedule()  
    print "Goodbye", s
```

```
stackless.tasklet(greet)("A")  
stackless.tasklet(greet)("B")  
stackless.run()
```

```
Hello, A  
Hello, B  
Goodbye, A  
Goodbye, B
```



<module>

Stackless  
runtime

... and Stackless exits

# Channels

```
ch = stackless.channel()
```

Sending tasklet

```
x = 5  
ch.send(5)  
print "Sent."
```

ch

Receiving tasklet

```
x = ch.receive()  
assert x == 5
```

“send” and “receive” block if the channel has no waiting receiving/sending tasklets.

Blocked tasklets are removed from the run queue and moved to the channel’s queue.

```
import stackless
```

```
def sending():  
    x = 5  
    ch.send(x)  
    print "Sent."
```

```
def receiving():  
    x = ch.receive()  
    assert x == 5
```

```
ch = stackless.channel()  
stackless.tasklet(sending)()  
stackless.tasklet(receiving)()  
stackless.run()
```

Sent.

## Sending tasklet

➡ `x = 5`  
`ch.send(x)`  
`print "Sent."`

## Receiving tasklet

◆ `x = ch.receive()`  
`assert x == 5`

'ch' channel

`balance = 0`  
`queue = None`

sending



`x = 5`

receiving



## Sending tasklet

➡ `x = 5`  
`ch.send(x)`  
`print "Sent."`

## Receiving tasklet

◆ `x = ch.receive()`  
`assert x == 5`

'ch' channel

`balance = 0`  
`queue = None`

sending

➡ `ch.send(5)`

receiving



## Sending tasklet

➡  
`x = 5`  
`ch.send(x)`  
`print "Sent."`

## Receiving tasklet

◆  
`x = ch.receive()`  
`assert x == 5`

## 'ch' channel

`balance = 1`  
`queue`

→  
◆ `print "Sent."`  
`.tempval = 5`

sending

➡  
`ch.send(5)`

receiving



## Sending tasklet

```
x = 5  
ch.send(x)  
print "Sent."
```



## Receiving tasklet

```
x = ch.receive()  
assert x == 5
```

## 'ch' channel

```
balance = 1  
queue
```



```
print "Sent."  
.tempval = 5
```

## receiving

```
x = ch.receive()
```



## Sending tasklet

◆  
x = 5  
ch.send(x)  
print "Sent."

## Receiving tasklet

◆  
x = ch.receive()  
assert x == 5

## 'ch' channel

balance = 1  
queue

◆  
print "Sent."  
.tempval = 5

## receiving

◆  
x = 5

## Sending tasklet

◆  
`x = 5`  
`ch.send(x)`  
`print "Sent."`

## Receiving tasklet

◆  
`x = ch.receive()`  
`assert x == 5`

## 'ch' channel

`balance = 0`  
`queue = None`

◆  
`print "Sent."`

sending

◆  
`print "Sent."`

receiving

◆  
`x = 5`



## Sending tasklet

◆  
`x = 5`  
`ch.send(x)`  
`print "Sent."`

## Receiving tasklet

◆  
`x = ch.receive()`  
`assert x == 5`

'ch' channel

`balance = 0`  
`queue = None`

sending

◆  
`print "Sent."`

receiving

◆  
`assert x == 5`

## Sending tasklet

```
x = 5  
ch.send(x)  
print "Sent."
```

## 'ch' channel

```
balance = 0  
queue = None
```

## Receiving tasklet

```
x = ch.receive()  
assert x == 5
```

## sending

```
print "Sent."
```

## receiving

## Sending tasklet

```
x = 5  
ch.send(x)  
print "Sent."
```

## 'ch' channel

```
balance = 0  
queue = None
```

## Receiving tasklet

```
x = ch.receive()  
assert x == 5
```

## sending

```
print "Sent."
```

# Producer / Consumer

```
import stackless
```

```
def producer1():  
    print "Starting producer1 "  
    ch.send("A")  
    ch.send("B")  
  
def producer2():  
    print "Starting producer2"  
    ch.send_sequence( (1, 2, 3) )
```

```
def consumer():  
    print "Starting consumer"  
    for x in ch:  
        print repr(x)
```

```
ch = stackless.channel()  
#ch.preference = 1  
for f in (consumer, producer1, producer2):  
    stackless.tasklet(f())  
stackless.run()
```

```
Starting consumer  
Starting producer1  
'A'  
Starting producer2  
1  
'B'  
2  
3
```

# Bidirectional

```
import stackless

request_ch = stackless.channel()
response_ch = stackless.channel()

def divider():
    while 1:
        x, y = request_ch.receive()
        response_ch.send( x/y )

def divide(top, bottom):
    request_ch.send( (top, bottom) )
    return response_ch.receive()

def main(top, bottom):
    result = divide(top, bottom)
    print "%s/%s = %s" % (top, bottom, result)

stackless.tasklet(divider)()
stackless.tasklet(main)(8, 4)
stackless.run()
```

$$8 / 4 = 2$$

... import and channel setup ...

```
def divider():
    while 1:
        x, y = request_ch.receive()
        response_ch.send( x/y )

def divide(top, bottom):
    request_ch.send( (top, bottom) )
    return response_ch.receive()

def main(top, bottom):
    try:
        result = divide(top, bottom)
    except ZeroDivisionError:
        result = None
    print "%s/%s = %s" % (top, bottom, result)

stackless.tasklet(divider)()
stackless.tasklet(main)(8, 0)
stackless.run()
```

# Exceptions

```
Traceback (most recent call last):
  File "divider.py", line 24, in
    stackless.run()
  File "divider.py", line 9, in
    response_ch.send( x/y )
ZeroDivisionError: integer divi
```

**Uncaught  
exceptions forwarded  
to main tasklet**



```
def divider():
    while 1:
        x, y = request_ch.receive()
        try:
            response_ch.send( x/y )
        except ZeroDivisionError, err:
            response_ch.send_exception(ZeroDivisionError, *err.args)
```

```
def divide(top, bottom):
    request_ch.send( (top, bottom) )
    return response_ch.receive()
```

```
def main(top, bottom):
    try:
        result = divide(top, bottom)
    except ZeroDivisionError:
        result = None
    print "%s/%s = %s" % (top, bottom, result)
```


```
stackless.tasklet(divider)()
stackless.tasklet(main)(8, 0)
stackless.run()
```

8 / 0 = None

# Forwarding exceptions

```
while 1:
    x, y = request_ch.receive()
    try:
        response_ch.send( x/y )
    except ZeroDivisionError, err:
        response_ch.send_exception(ZeroDivisionError, *err.args)
```

**Ugly**



---

```
while 1:
    x, y = request_ch.receive()
    try:
        response_ch.send( x/y )
    except ZeroDivisionError, err:
        response_ch.send_exception(err)
```

What I would prefer

---

...

```
except Exception:
    response_ch.send_exception(WrapperException, sys.exc_info())
```

Can fake it with

plus an unwrapper and 3-arg raise  
around the channel receive

# Recursion and the stack

```
def factorial(n):  
    if n <= 1:  
        return 1  
    return n * factorial(n-1)
```

```
print "5! =", factorial(5)
```

```
try:
```

```
    factorial(1000)
```

```
except Exception, err:
```

```
    print "1000! :", err
```

```
>>> sys.getrecursionlimit()
```

```
1000
```

```
>>>
```

```
5! = 120
```

```
1000! : maximum recursion depth exceeded
```

# Stackless Recursion

```
import stackless
```

```
def call_wrapper(f, args, kwargs, result_ch):  
    result_ch.send(f(*args, **kwargs))  
    # ... should also catch and forward exceptions ...
```

Create a new tasklet and  
channel for each call

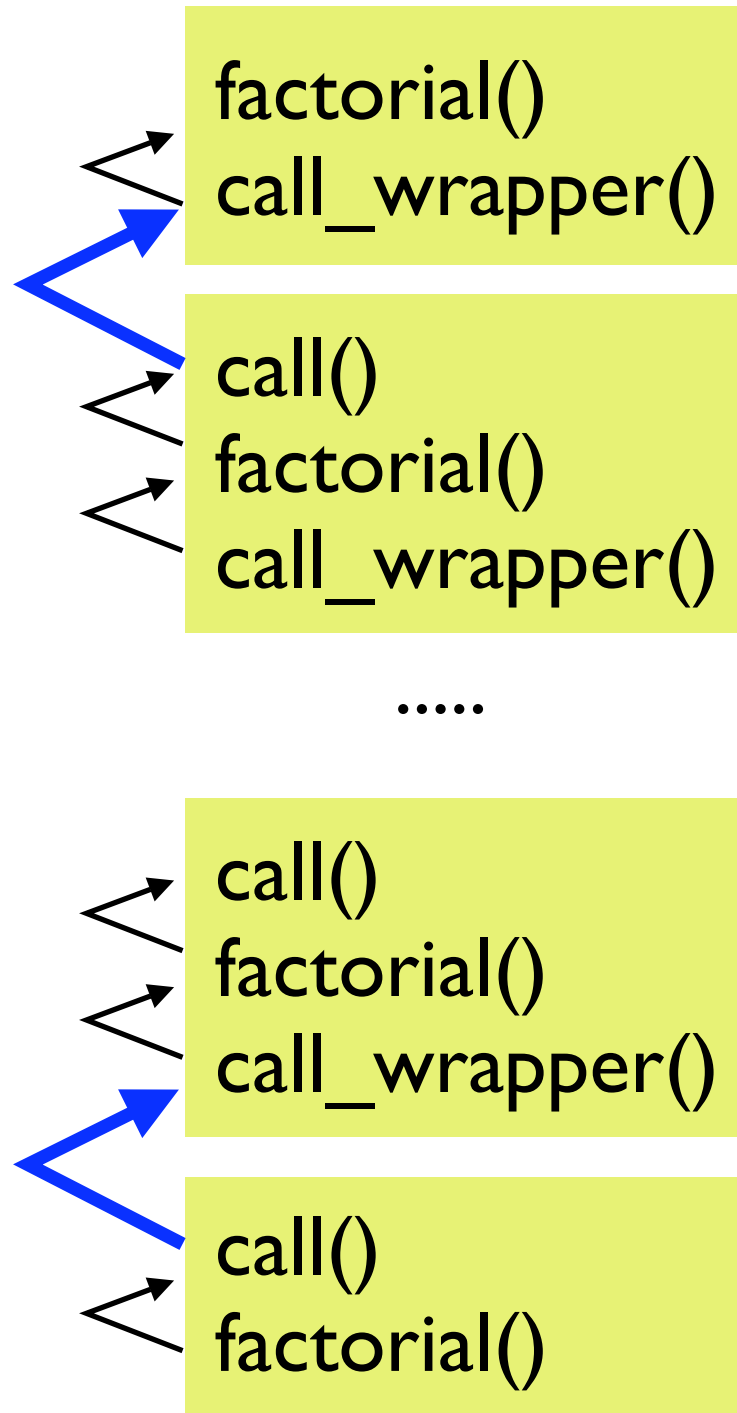
```
def call(f, *args, **kwargs):  
    result_ch = stackless.channel()  
    stackless.tasklet(call_wrapper)(f, args, kwargs, result_ch)  
    return result_ch.receive()
```

```
def factorial(n):  
    if n <= 1:  
        return 1  
    return n * call(factorial, n-1)
```

```
print "5! =", factorial(5)  
print "1000! / 998! =", factorial(1000)/factorial(998)
```

$$5! = 120$$

$$1000! / 998! = 999000$$



# Tasklets

Running



```
x = 1  
send(x) on channel 2!
```



2! channel

....

Blocked.  
Waiting in a channel queue



```
create channel 997!  
x = 998 * receive()  
send(x) on channel 998!
```



998! channel



```
create channel 998!  
x = 999 * receive()  
send(x) on channel 999!
```



999! channel



```
create channel 999!  
x = 1000 * receive()  
return x
```



Tasklet  
creation  
order

# Scalability

1000 channels, 1000 tasklets

Try that with threads!

What's the overhead in computing 998! ?

Normal Python, recursive algorithm: 2.93 ms

Stackless Python, recursive algorithm: 3.62 ms

Stackless Python, Stackless algorithm: 10.3 ms

Stackless algorithm has 6x more function calls.

The send/receive calls add only 0.8 ms.

1000 calls of object() takes 0.27 ms

# Why use Stackless?

Scalable, light-weight threads

Fast inter-tasklet communication

Can use Stackless solutions where you might be cautious of threads

- Convert callback processing into an iterator
- Implement blocking APIs with asynchronous I/O
- Games and simulation programming



# XML parsing with SAX

```
import xml.sax
```

```
class ElementCountHandler(xml.sax.ContentHandler):  
    def startDocument(self):  
        self.count = 0  
    def startElement(self, tag, attrs):  
        self.count += 1
```

```
handler = ElementCountHandler()
```

```
parser = xml.sax.make_parser()  
parser.setContentHandler(handler)  
# manually removed the DTD link  
parser.parse("iTunes Music Library.xml")
```

```
print handler.count, "elements"
```

220523 elements

# What's the problem?

Most people don't like callbacks.

```
count = 0
for event, args in iterparse("iTunes Music Library.xml"):
    if event == "startElement":
        count += 1

print count, "elements"
```

(BTW, Python 2.5 includes ElementTree in the stdlib.  
xml.etree.ElementTree.iterparse has a different API.)

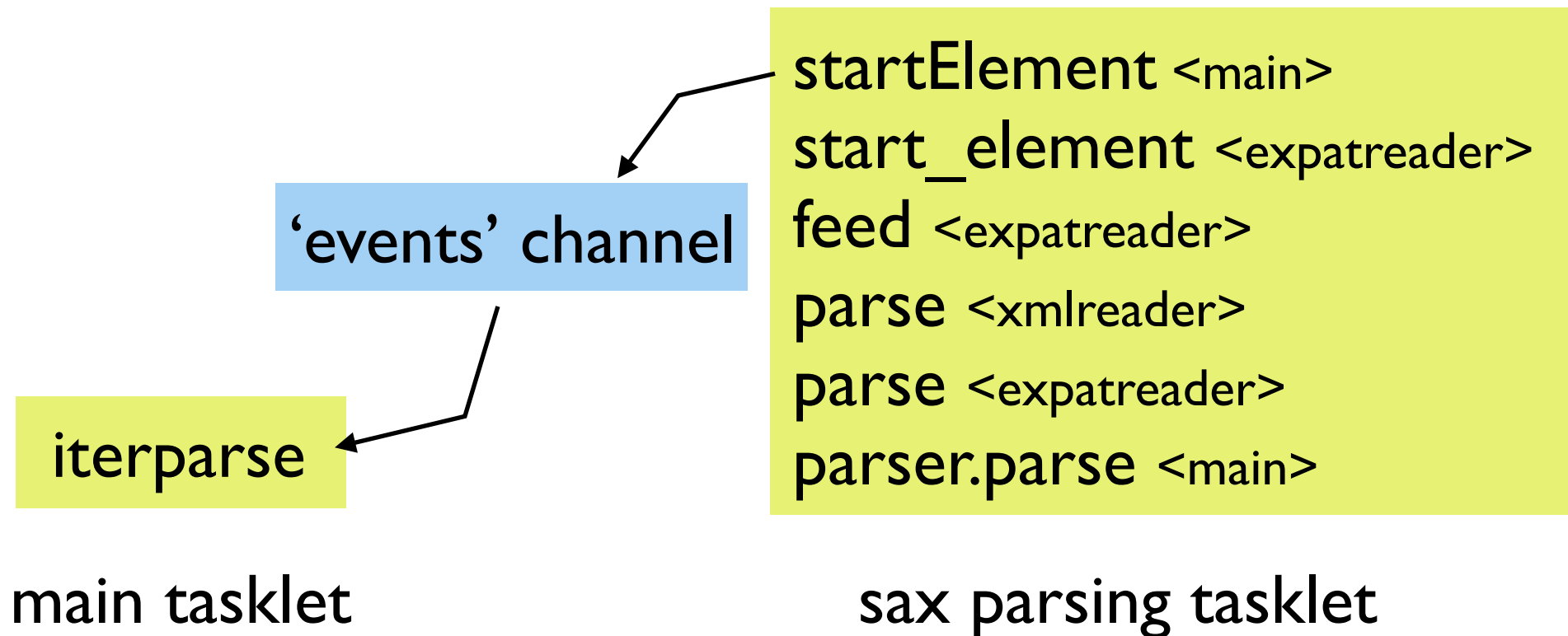
# Can't (easily) switch to a generator

Calls expat,  
a C library

```
File "pycon_sax.py", line 13, in <module>
    parser.parse("iTunes Music Library.xml")
File "....../python2.5/xml/sax/expatreader.py", line 107, in parse
    xmlreader.IncrementalParser.parse(self, source)
File "....../python2.5/xml/sax/xmlreader.py", line 123, in parse
    self.feed(buffer)
File "....../python2.5/xml/sax/expatreader.py", line 207, in feed
    self._parser.Parse(data, isFinal)
File "....../python2.5/xml/sax/expatreader.py", line 301, in start_element
    self._cont_handler.startElement(name, AttributesImpl(attrs))
File "pycon_sax.py", line 7, in startElement
    1/0
```

Added "1 / 0" to get the stack trace

# With Stackless you control the stack



```
import xml.sax
import stackless
```

```
class StacklessIterHandler(xml.sax.ContentHandler):
    def __init__(self, events):
        self.events = events

    def startElement(self, tag, attrs):
        self.events.send( ("startElement", (tag, attrs)) )
```

```
def iterparse(filename):
    events = stackless.channel()

    handler = StacklessIterHandler(events)
    parser = xml.sax.make_parser()
    parser.setContentHandler(handler)
    stackless.tasklet(parser.parse)(filename)

    return iter(events)
```

220523 elements

# “robust” implementation must handle XML errors

set an error handler and send any errors to  
the ‘events’ channel as an exception

- and/or -

put a wrapper around `parser.parse()` which  
catches all exceptions and forwards them  
to the ‘events’ channel

# Scalability

Parsing an 6 MB XML file with 220,523 elements

Normal Python, counter in handler: 1.46 s

Stackless Python, counter in handler: 1.49 s

Stackless iterparser: 2.25 s

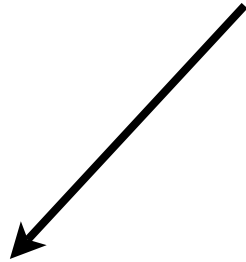
threaded, lock around a deque: 2.34 s

threaded, using Queue.Queue(1000): 13.1 s

lock+deque based on approach by Santiago Gala

Queue version almost identical to Stackless version

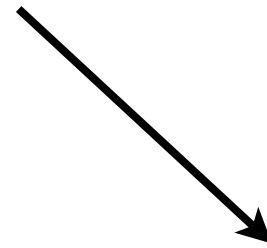
# Networking APIs



## Blocking

Use threads for  
parallel I/O

```
f = urllib.urlopen(url)
print f.read()
```



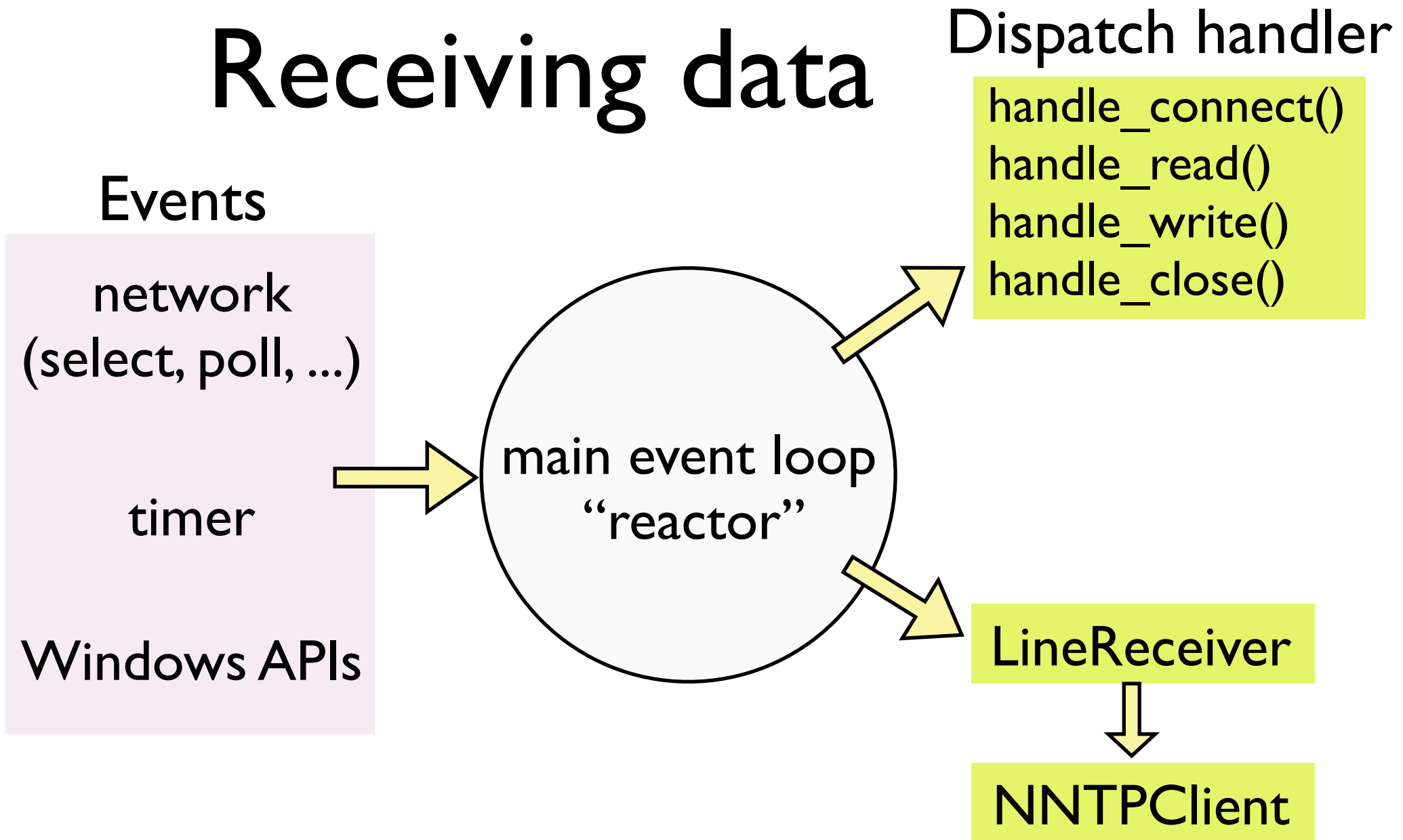
## Asynchronous

Event and callback based

- asyncore (in stdlib)
- Twisted



# Receiving data



Handlers are callback functions

```
import asyncore, socket

class http_client(asyncore.dispatcher):
    def __init__(self, host, path):
        asyncore.dispatcher.__init__(self)
        self.create_socket(socket.AF_INET, socket.SOCK_STREAM)
        self.connect( (host, 80) )
        self.buffer = ('GET %s HTTP/1.0\r\nHost: %s\r\n\r\n' %
                        (path, host))

    def handle_connect(self):
        pass

    def writable(self):
        return (len(self.buffer) > 0)

    def handle_write(self):
        sent = self.send(self.buffer)
        self.buffer = self.buffer[sent:]

    def handle_read(self):
        print self.recv(8192)

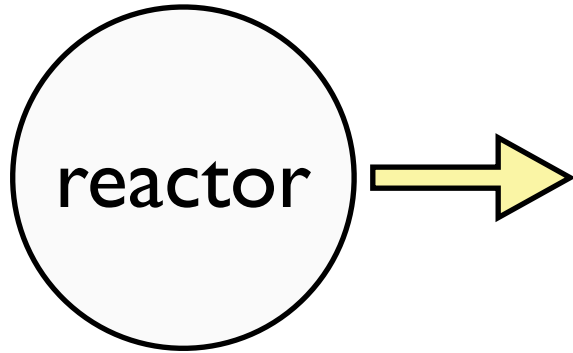
    def handle_close(self):
        self.close()

c = http_client('www.python.org', '/')
asyncore.loop()
```

**Based on the example  
in the `asyncore`  
documentation**

## stackless\_http\_client

```
handle_read():  
    read_channel.send(self.recv(8192))  
handle_close()  
    read_channel.close()
```



## stackless\_urlopen

```
parse url  
make 'read_channel'  
make stackless_http_client tasklet  
return StacklessFileAdapter
```

'read\_channel'

## StacklessFileAdapter

```
read():  
    s = "".join(read_channel)  
    i = s.index("\r\n\r\n")  
    return s[i+4:]
```

```
f = stackless_urlopen(url)  
print f.read()
```

# Details

```
import asyncore, socket, urlparse, stackless
```

```
class stackless_http_client(asyncore.dispatcher):
```

```
    def __init__(self, host, path, read_channel):
```

```
        asyncore.dispatcher.__init__(self)
```

```
        self.create_socket(socket.AF_INET, socket.SOCK_STREAM)
```

```
        self.connect( (host, 80) )
```

```
        self.buffer = 'GET %s HTTP/1.0\r\nHost: %s\r\n\r\n' % (path, host)
```

```
        self.read_channel = read_channel
```

```
    def handle_connect(self):
```

```
        pass
```

```
    def handle_read(self):
```

```
        self.read_channel.send(self.recv(8192))
```

```
    def handle_close(self):
```

```
        self.read_channel.close()
```

```
        self.close()
```

```
    def writable(self):
```

```
        return (len(self.buffer) > 0)
```

```
    def handle_write(self):
```

```
        sent = self.send(self.buffer)
```

```
        self.buffer = self.buffer[sent:]
```

```
class StacklessFileAdapter(object):
```

```
    def __init__(self, read_channel):
```

```
        self.read_channel = read_channel
```

```
    def read(self):
```

```
        s = "".join(self.read_channel)
```

```
        # Skip the http response headers
```

```
        i = s.index("\r\n\r\n")
```

```
        return s[i+4:]
```

```
def stackless_urlopen(url):
```

```
    scheme, netloc, path, query, fragment = \
        urlparse.urlsplit(url)
```

```
    if ":" in netloc:
```

```
        raise TypeError("bad netloc")
```

```
    read_channel = stackless.channel()
```

```
    stackless.tasklet(stackless_http_client)(
```

```
        netloc, path, read_channel)
```

```
    return StacklessFileAdapter(read_channel)
```

# This almost works

```
def call_asyncore_loop():  
    while 1:  
        stackless.schedule()  
        asyncore.loop(count=1)  
  
def fetch(url):  
    f = stackless_urlopen(url)  
    print f.read()  
  
stackless.tasklet(call_asyncore_loop)()  
stackless.tasklet(fetch)("http://www.python.org/")  
stackless.tasklet(fetch)("http://docs.python.org/")  
stackless.run()
```

Daemon does not know when to exit

Blocks for 30 seconds while waiting for I/O:  
Non-I/O tasklets don't run

Never returns (call\_asyncore\_loop never finishes)

# App-specific questions

- When does your program exit?
- Are there background/low-priority tasks?
- Any non-I/O tasks yielding time with `schedule()`?
- Can tasks sleep?
- Does it use gtk, qt, wx, Windows, ... event loop?
- Or work with multiple event loops?
- Do tasklets communicate with system threads?
- Is busy-waiting okay?

# Don't block in asyncore if other tasklets are running

```
def call_asyncore_loop():
    while 1:
        stackless.schedule()
        # Are there any non-blocked tasklets?
        # This tasklet and the main tasklet are
        # always running.
        if stackless.gettruncount() == 2:
            timeout = 30
        else:
            timeout = 0
        asyncore.loop(timeout=timeout, count=1)
```

# Distinguish between daemononic and non-daemononic tasklets

```
# Assume normal tasklets are daemononic
stackless.tasklet(call_asyncore_loop)()

# Track "run tasklets" which must go to completion
run_tasklet(fetch)("http://www.python.org/")
run_tasklet(fetch)("http://docs.python.org/")

# Run until all "run tasklets" are finished
run_all_tasklets()
```

(You might prefer using an "is\_finished" flag.)



# Implementation

```
_running_tasklet_count = 0
```

```
def run_tasklet(f):  
    global _running_tasklet_count  
    _running_tasklet_count += 1  
  
    def tasklet_wrapper(*args, **kwargs):  
        global _running_tasklet_count  
        try:  
            return f(*args, **kwargs)  
        finally:  
            _running_tasklet_count -= 1  
  
    return stackless.tasklet(tasklet_wrapper)
```

(kinda ugly)

```
def run_all_tasklets():  
    while 1:  
        stackless.schedule()  
        if _running_tasklet_count == 0:  
            break
```

# Turn it into a library?

Maybe.

Experimental

Does 80/20 apply?

Use Twisted? It understands the hard parts.  
("Stackless and Twisted" talk for next year?)

Does anyone really want this approach?

# Monkeypatching

Richard Tew's “stacklesssocket.py” module implements the socket module API

```
import sys
import stacklesssocket
sys.modules["socket"] = stacklesssocket
```

```
>>> import socket
>>> socket is stacklesssocket
True
>>> import httpplib
>>> httpplib.socket is stacklesssocket
True
>>>
```

# Some stdlib modules

## “just work”

```
import urllib2
```

```
def fetch(url):  
    f = urllib2.urlopen(url)  
    print f.read()
```

```
# Provide my own manager  
stacklesssocket.managerRunning = True  
stackless.tasklet(call_asyncore_loop)()
```

```
run_tasklet(fetch)("http://www.python.org/")  
run_tasklet(fetch)("http://docs.python.org/")  
run_all_tasklets()
```

It's a hack! But very cool.

## Blocking calls still block

- “subprocess” module uses `waitpid` and `select`
- can those be monkeypatch?

## Some stdlib modules aren't thread-safe

- but Stackless has weaker requirements than system threads

# Actor-based programming

Semi-independent interacting objects with complex actions

Games: each character, spaceship, gun, bullet, door, ... gets its own tasklet

```
def squarebot(bot):
    while 1:
        for _ in range(3):
            bot.shoot()
            bot.forward(10)
            bot.turn(90)
        bot.shoot()
        bot.forward(10)
        bot.turn(randrange(0, 360))

def spinbot(bot):
    while 1:
        bot.turn(5)
        bot.shoot()

def bullet(bullet, dx, dy):
    x, y = bullet.pos
    while in_bounds(bullet):
        x += dx; y += dy
        bullet.move_to(x, y)
        bot = find_bot(bullet)
        if bot is not None:
            bot.add_damage(-10)
            break
```

# Actor-based programming

EVE Online from CCP Games uses 100,000s of tasklets

For more on games and simulation programming,  
see Grant Olson's excellent tutorial

*“Introduction to Concurrent Programming with  
Stackless Python”*

[http://members.verizon.net/olsongt/stackless/why\\_stackless.html](http://members.verizon.net/olsongt/stackless/why_stackless.html)

or follow link from the Stackless home page

# State of Stackless

Stable. Working. Supported.

Will not be part of core CPython.

Is implemented in PyPy

See Christian Tismer's talk on Sunday, 1:30pm

Not widely used but is in successful commercial use

Needs documentation, tools, and advocacy

Missing experience and concrete idea of best practices



<http://stackless.com/>