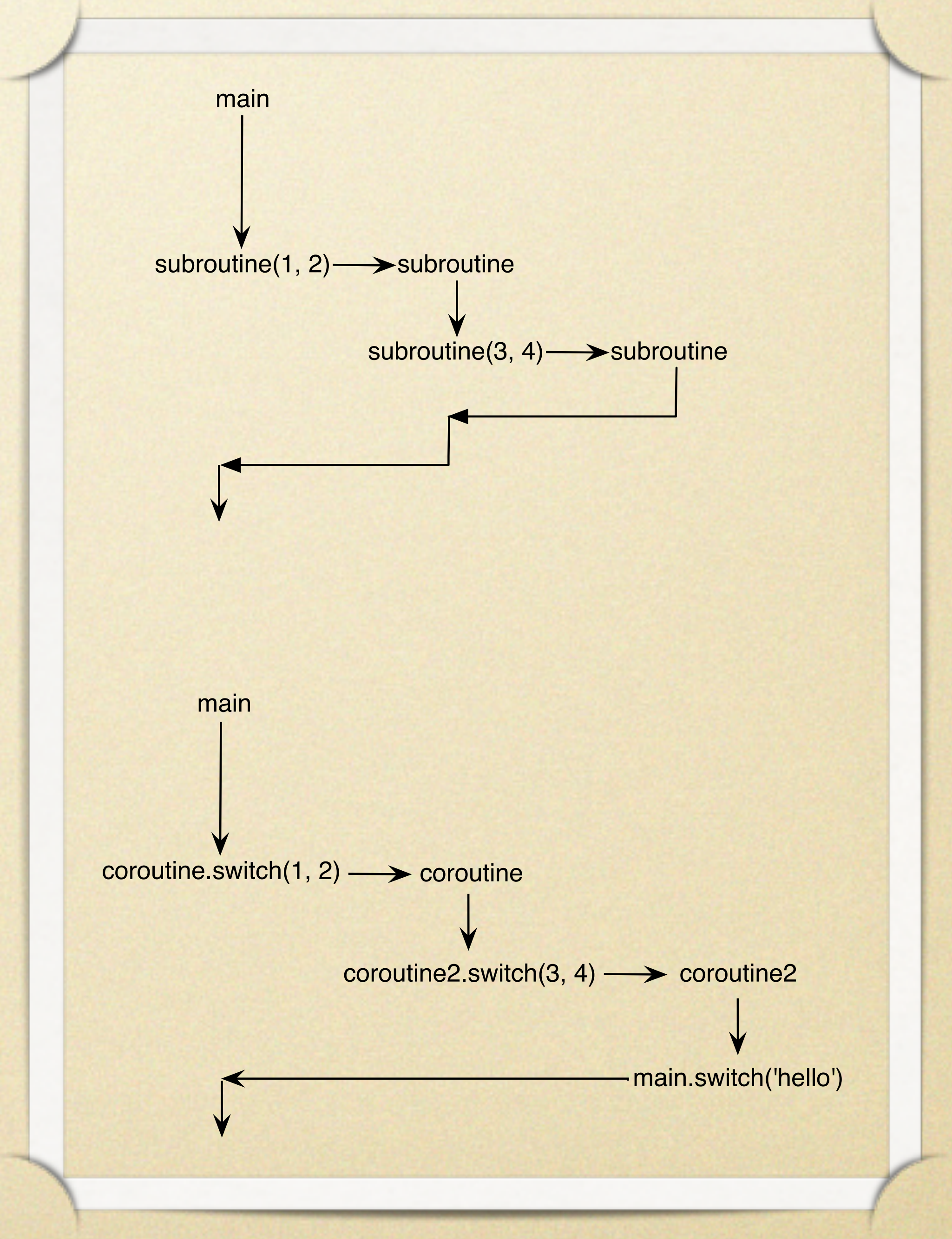
eventlet

eventlet

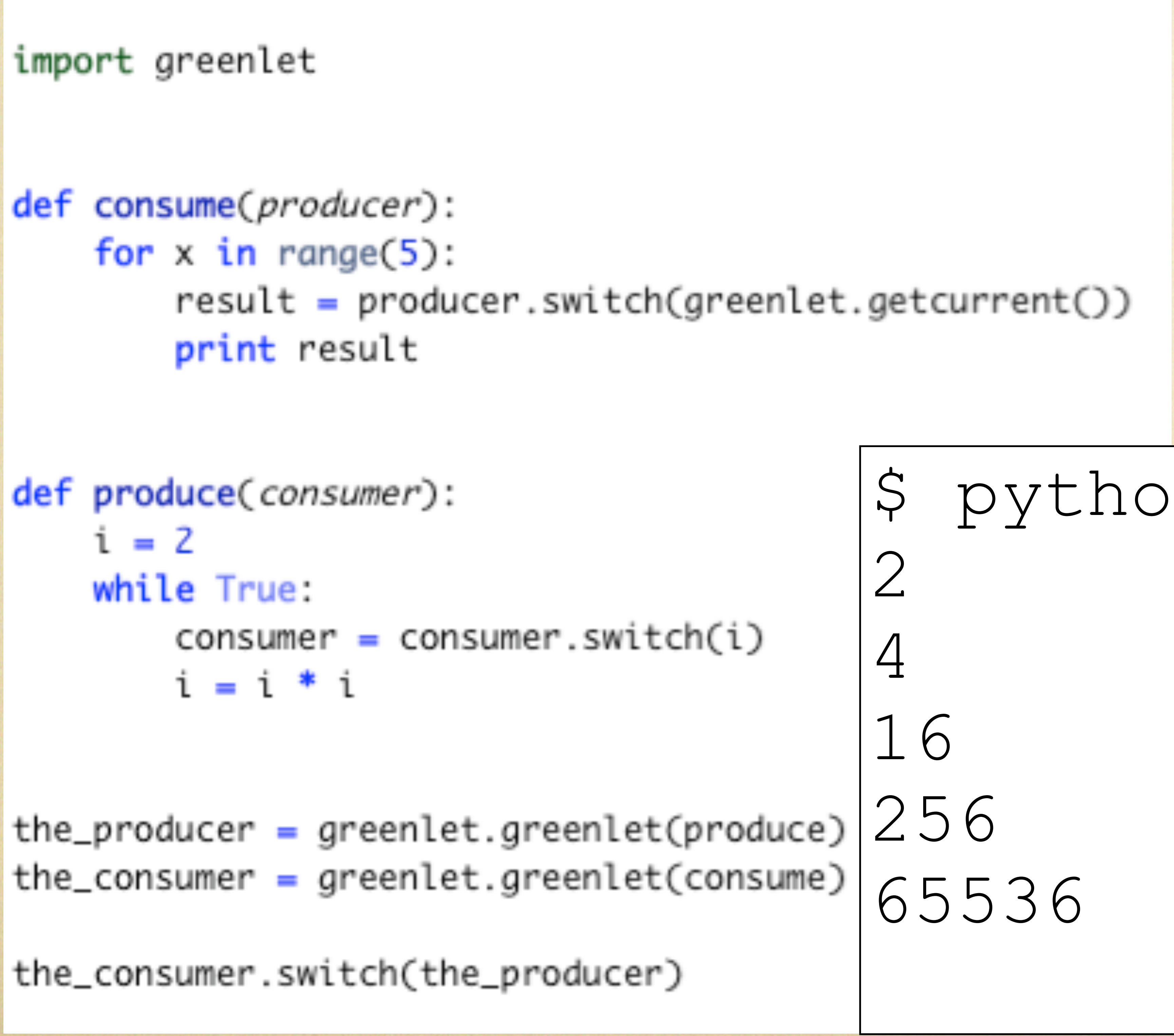
- coroutines — flexible efficient control flow
  - greenlet
- non-blocking i/o — efficient network i/o
  - select/poll/epoll
- threads — switch between async and sync
  - queues/pipes

coroutines

- subroutine:
  - continue by returning to caller
- coroutine:
  - continue by calling another coroutine



greenlet



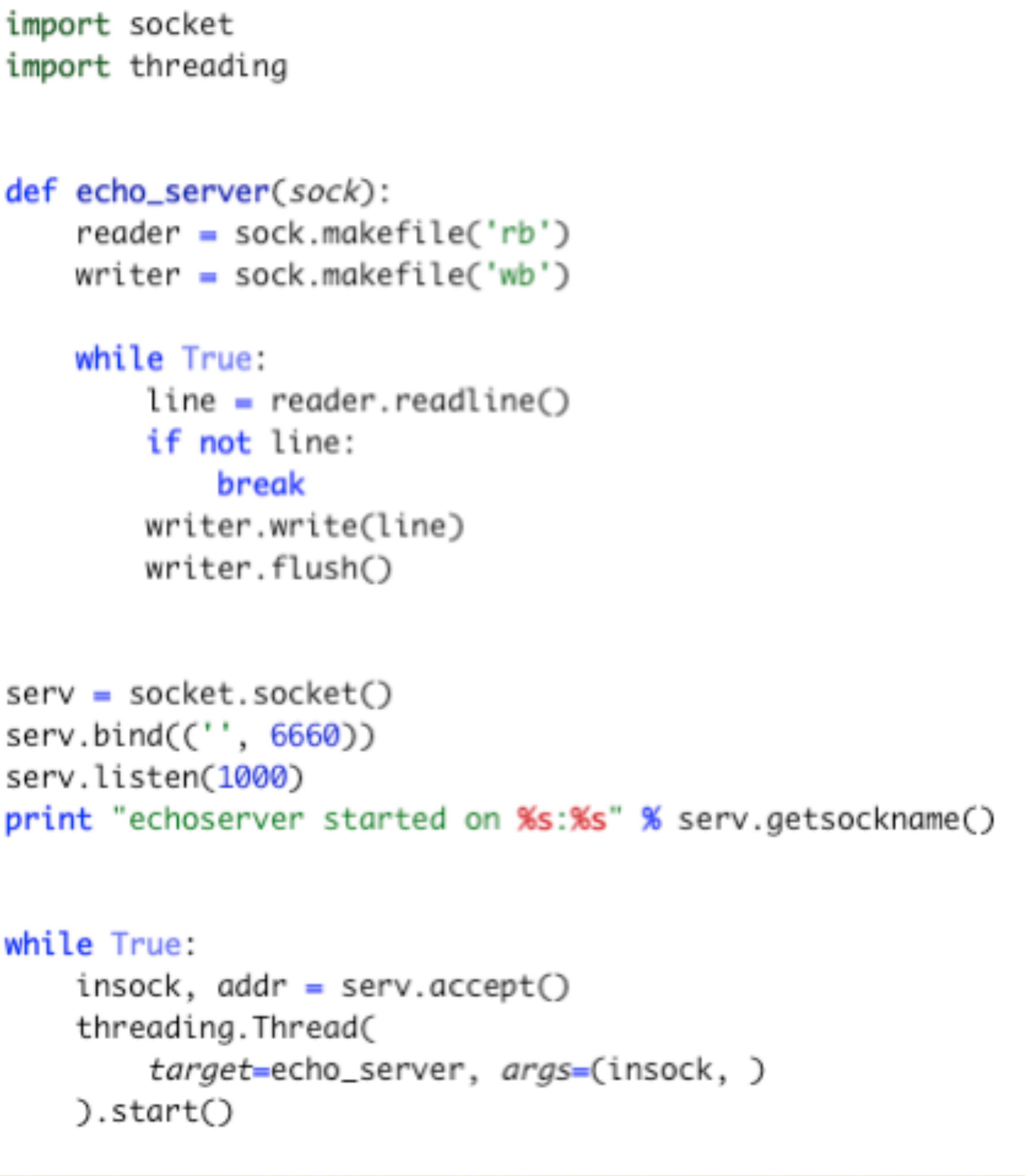
```
import greenlet


def consume(producer):
    for x in range(5):
        result = producer.switch(greenlet.getcurrent())
        print result


def produce(consumer):
    i = 2
    while True:
        consumer = consumer.switch(i)
        i = i * i


the_producer = greenlet.greenlet(produce)
the_consumer = greenlet.greenlet(consume)

the_consumer.switch(the_producer)
```

```
$ python coros.py
2
4
16
256
65536
```

non-blocking i/o

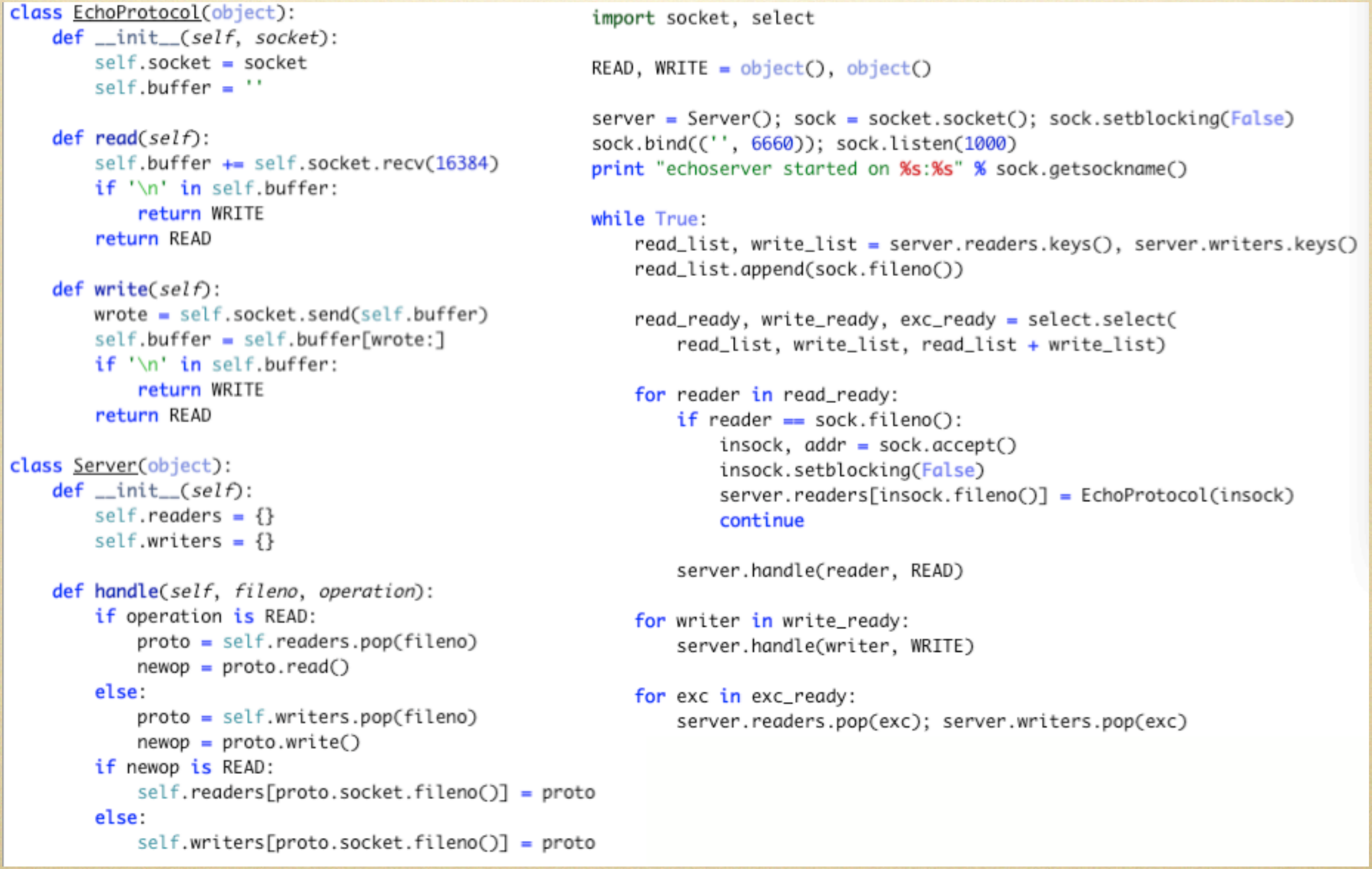
- blocking i/o:
  - each “thread of control” can read or write on one file descriptor at a time
    - process, thread
- non-blocking i/o:
  - reads and writes are multiplexed using select, poll, epoll, kqueue, etc.

blocking i/o



```
import socket
import threading


def echo_server(sock):
    reader = sock.makefile('rb')
    writer = sock.makefile('wb')

    while True:
        line = reader.readline()
        if not line:
            break
        writer.write(line)
        writer.flush()


serv = socket.socket()
serv.bind(('', 6660))
serv.listen(1000)
print "echoserver started on %s:%s" % serv.getsockname()


while True:
    insock, addr = serv.accept()
    threading.Thread(
        target=echo_server, args=(insock, )
    ).start()
```

non-blocking i/o



```python
class EchoProtocol(object):
    def __init__(self, socket):
        self.socket = socket
        self.buffer = ''

    def read(self):
        self.buffer += self.socket.recv(16384)
        if '\n' in self.buffer:
            return WRITE
        return READ

    def write(self):
        wrote = self.socket.send(self.buffer)
        self.buffer = self.buffer[wrote:]
        if '\n' in self.buffer:
            return WRITE
        return READ

class Server(object):
    def __init__(self):
        self.readers = {}
        self.writers = {}

    def handle(self, fileno, operation):
        if operation is READ:
            proto = self.readers.pop(fileno)
            newop = proto.read()
        else:
            proto = self.writers.pop(fileno)
            newop = proto.write()
        if newop is READ:
            self.readers[proto.socket.fileno()] = proto
        else:
            self.writers[proto.socket.fileno()] = proto
```

```python
import socket, select

READ, WRITE = object(), object()

server = Server(); sock = socket.socket(); sock.setblocking(False)
sock.bind(('', 6660)); sock.listen(1000)
print "echoserver started on %s:%s" % sock.getsockname()

while True:
    read_list, write_list = server.readers.keys(), server.writers.keys()
    read_list.append(sock.fileno())

    read_ready, write_ready, exc_ready = select.select(
        read_list, write_list, read_list + write_list)

    for reader in read_ready:
        if reader == sock.fileno():
            insock, addr = sock.accept()
            insock.setblocking(False)
            server.readers[insock.fileno()] = EchoProtocol(insock)
            continue

        server.handle(reader, READ)

    for writer in write_ready:
        server.handle(writer, WRITE)

    for exc in exc_ready:
        server.readers.pop(exc); server.writers.pop(exc)
```

eventlet: coroutines + non-blocking i/o

- main loop (Hub) is responsible for calling i/o multiplexer function and scheduling timers
- eventlet.greenio provides a socket object which registers with the Hub and cooperatively switches instead of blocking
- code looks blocking, but all network i/o is non-blocking

eventlet.greenio

- socket.read(...)
  - while not enough data:
    - trampoline(socket, read=True)
      - api.get_hub().add_descriptor(
        socket, read=api.get_current().switch)
        - self.readers[socket] = callback
      - api.get_hub().switch()

greenio part 2

- ready_to_read, ready_to_write, exc = select(...)

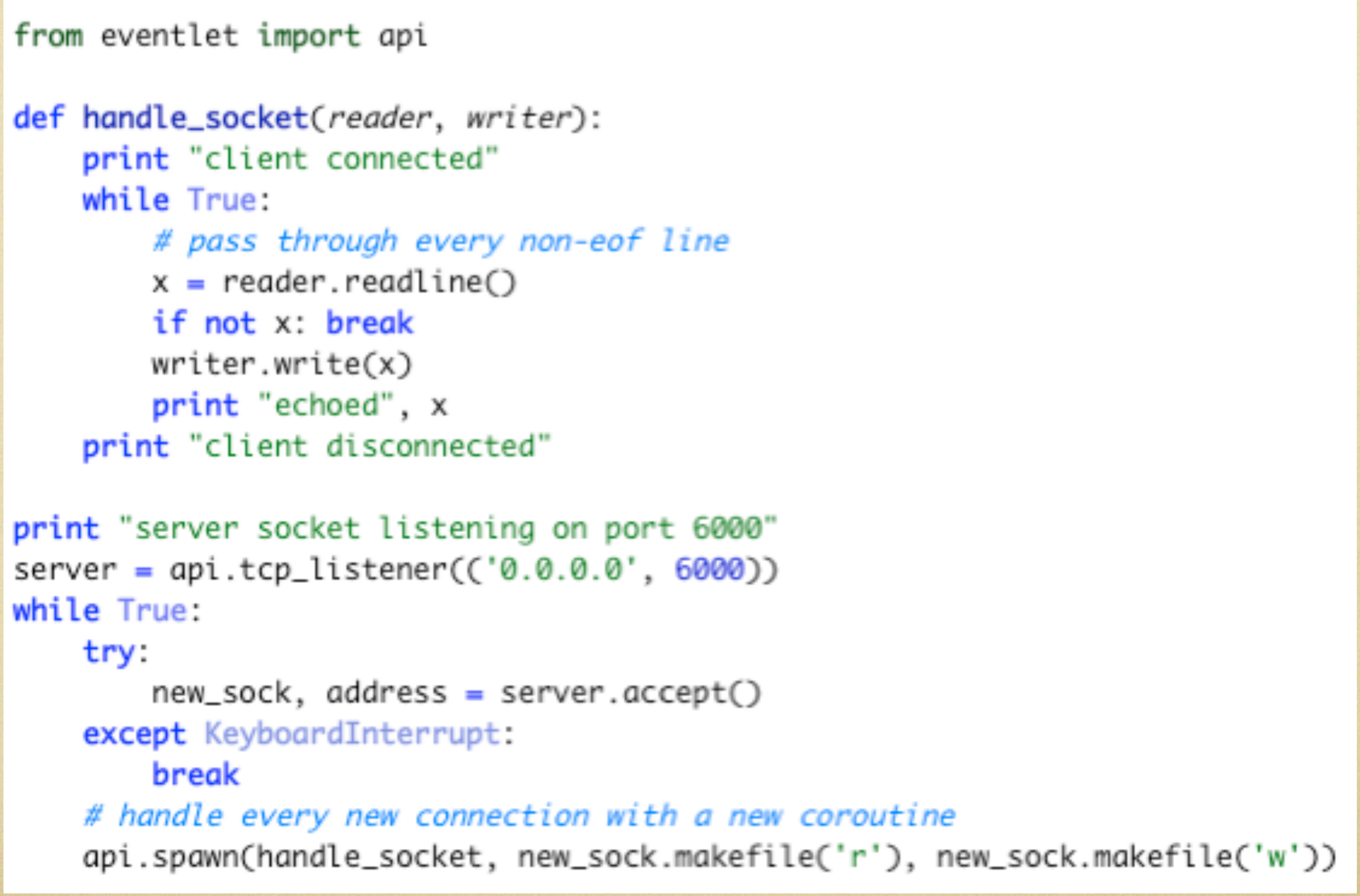
- for read in ready_to_read:

  self.readers[read].switch()

  - socket.recv(4096)

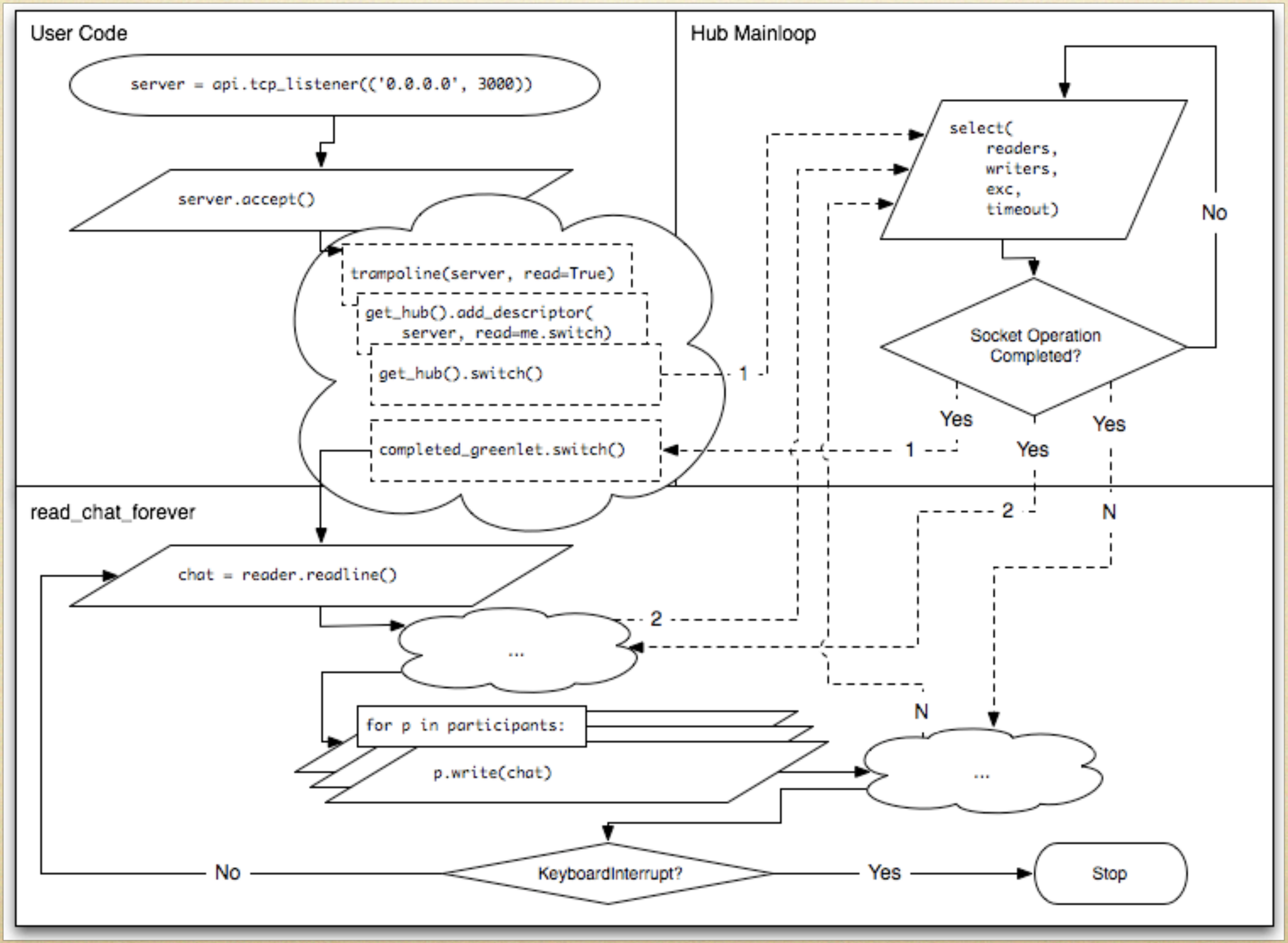
- once all requested data has been read, the socket.read(...) returns data

eventlet echo server



```
from eventlet import api

def handle_socket(reader, writer):
    print "client connected"
    while True:
        # pass through every non-eof line
        x = reader.readline()
        if not x: break
        writer.write(x)
        print "echoed", x
    print "client disconnected"

print "server socket listening on port 6000"
server = api.tcp_listener(('0.0.0.0', 6000))
while True:
    try:
        new_sock, address = server.accept()
    except KeyboardInterrupt:
        break
    # handle every new connection with a new coroutine
    api.spawn(handle_socket, new_sock.makefile('r'), new_sock.makefile('w'))
```

eventlet flowchart



```
server = api.tcp_listener(('0.0.0.0', 3000))
server.accept()
trampoline(server, read=True)
get_hub().add_descriptor(
    server, read=me.switch)
get_hub().switch()
completed_greenlet.switch()
select(
    readers,
    writers,
    exc,
    timeout)
chat = reader.readline()
for p in participants:
p.write(chat)
```

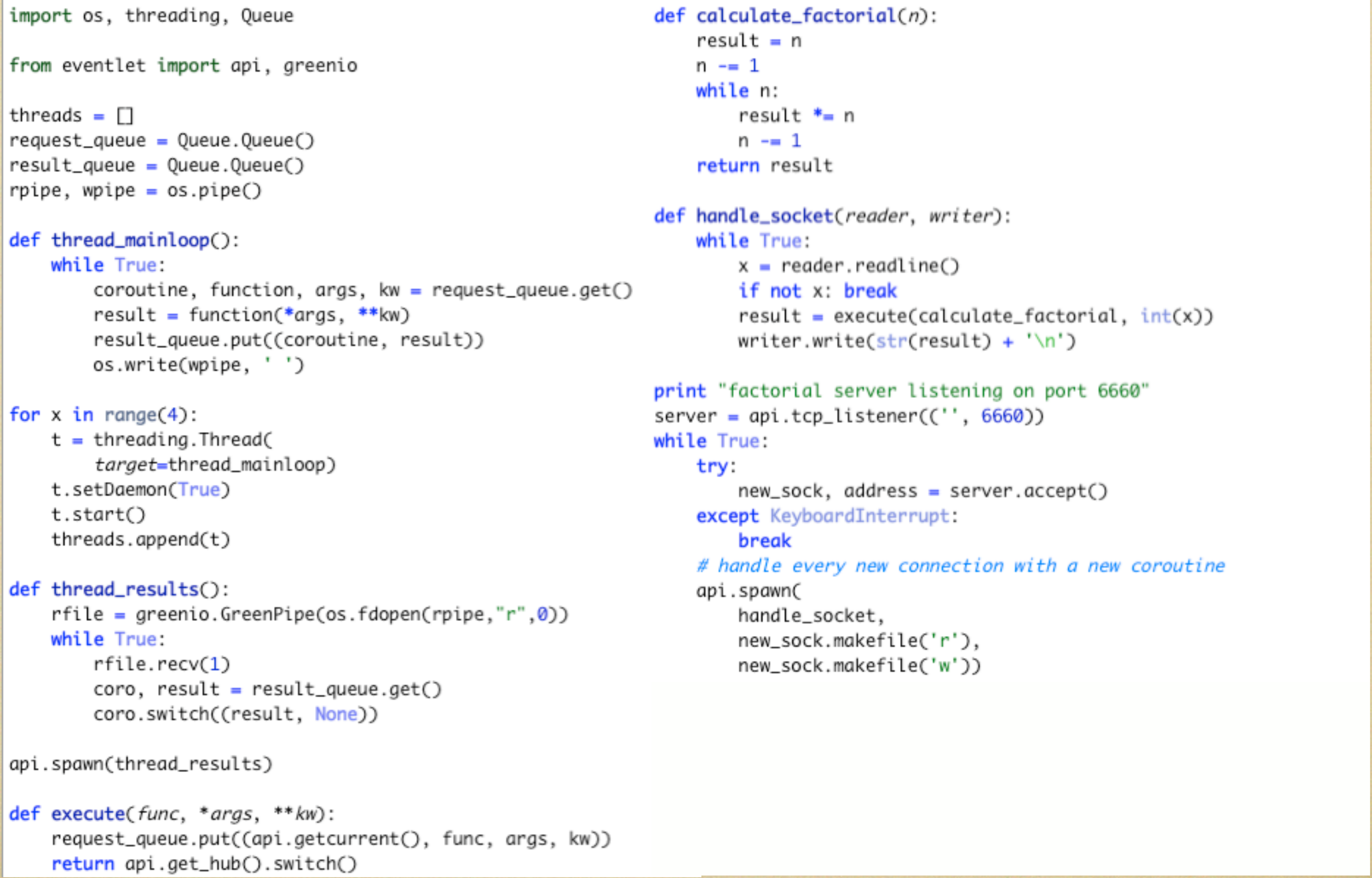
integration with blocking code

- eventlet uses a cooperative single thread
- blocking code must cooperate
- eventlet provides cooperative:
  - sockets
  - pipes
  - processes
- eventlet.tpool can mix blocking code with cooperative coroutines using a threadpool

threadpool details

- to call a function in a threadpool, eventlet puts the function, arguments, and current coroutine in a request queue
- threads in the pool block on the request queue
- the function is executed in the thread
- the result is put in the response queue
- a byte is written into a pipe which is being read by the main thread
- the result is sent to the original coroutine

naive threadpool



```
import os, threading, Queue

from eventlet import api, greenio

threads = []
request_queue = Queue.Queue()
result_queue = Queue.Queue()
rpipe, wpipe = os.pipe()

def thread_mainloop():
    while True:
        coroutine, function, args, kw = request_queue.get()
        result = function(*args, **kw)
        result_queue.put((coroutine, result))
        os.write(wpipe, ' ')

for x in range(4):
    t = threading.Thread(
        target=thread_mainloop)
    t.setDaemon(True)
    t.start()
    threads.append(t)

def thread_results():
    rfile = greenio.GreenPipe(os.fdopen(rpipe,"r",0))
    while True:
        rfile.recv(1)
        coro, result = result_queue.get()
        coro.switch((result, None))

api.spawn(thread_results)

def execute(func, *args, **kw):
    request_queue.put((api.getcurrent(), func, args, kw))
    return api.get_hub().switch()
```

```
def calculate_factorial(n):
    result = n
    n -= 1
    while n:
        result *= n
        n -= 1
    return result

def handle_socket(reader, writer):
    while True:
        x = reader.readline()
        if not x: break
        result = execute(calculate_factorial, int(x))
        writer.write(str(result) + '\n')

print "factorial server listening on port 6660"
server = api.tcp_listener(('', 6660))
while True:
    try:
        new_sock, address = server.accept()
    except KeyboardInterrupt:
        break
    # handle every new connection with a new coroutine
    api.spawn(
        handle_socket,
        new_sock.makefile('r'),
        new_sock.makefile('w'))
```

spawning

spawning

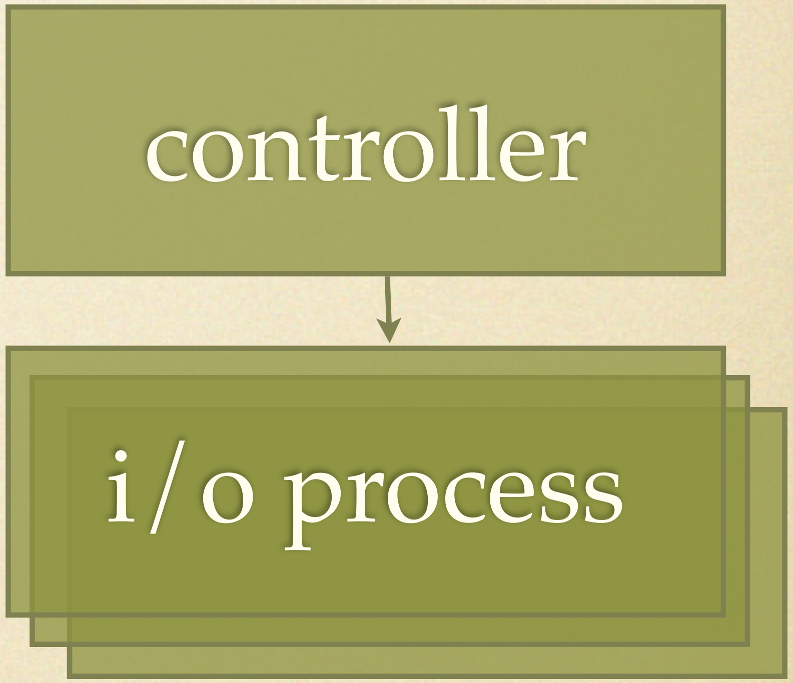
- http server
- wsgi server
- multiple network i/o processes
- multiple wsgi worker threads
- graceful code reloading

process model options

- single i/o process, multiple threads
  - good for stateful applications
- multiple i/o process, single thread
  - good for comet applications
- multiple i/o process, multiple thread
  - good for the majority of applications

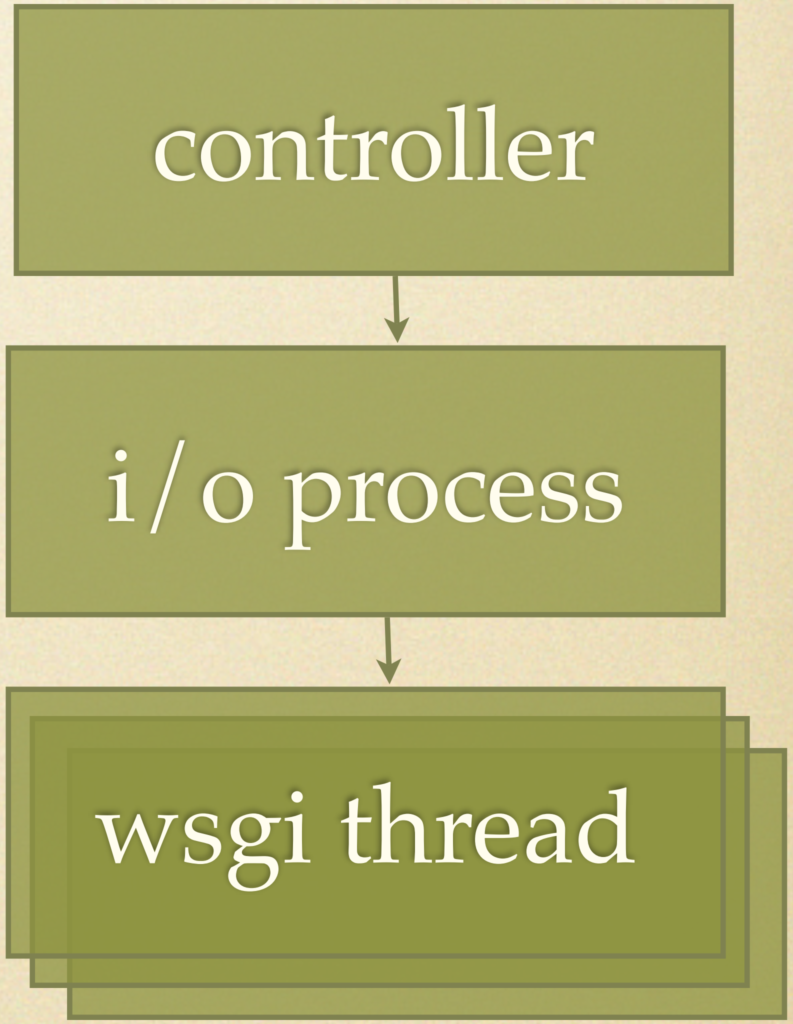
spawning controller

- main spawning process
- binds network socket
- forks network i/o processes
- multiple i/o processes can take advantage of multiple cpus



spawning child

- i/o processes use eventlet to scale to many keepalive sockets
- http protocol implementation in eventlet.wsgi
- dispatches to wsgi applications in threadpool



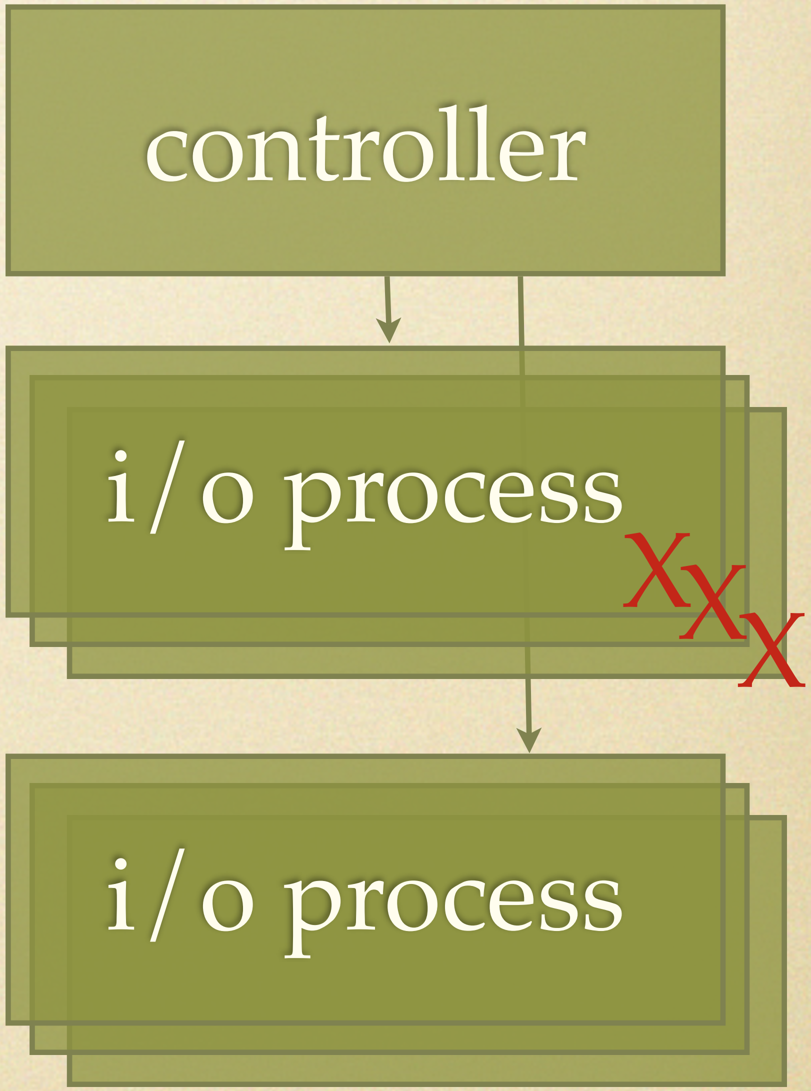
controller

i/o process

wsgi thread

graceful reloading

- send controller sighup
- controller forks new processes with new code
- existing processes stop accepting and complete outstanding requests, then exit



using spawning

- with paster serve:
  - [server:main]

    use = egg:Spawning
- command line:
  - spawn my_package.my_module.wsgi_app

spawn options

- spawn wsgi_app [wsgi_middleware, ...]
- --port=8080
- --host=127.0.0.1
- --processes=4
- --threads=8
    - --threads=0 will use eventlet cooperation monkeypatching