Async I/O in Python
We're here to talk about asynchronous I/O in Python
Widely accepted as a weakness of Python
People are working on fixing this in 3.4
Extremely broad, nuanced topic
I expect there's a huge range in people's level of understanding here
People often talk past each other, debating obscure points
The goal here is to learn more about the topic, not debate I/O strategies
Waiting
You're doing some I/O
It can't complete immediately (e.g. waiting for server to respond)
What do you do now?
If there's anything else useful to be done, surely do that
Async Strategies
Multiple processes
Multiple threads
Single thread, non-blocking I/O
A combination of all three might make sense
We're here to talk about frameworks for the latter
Non-blocking I/O
Usually, if an I/O operation (read/write/etc.) can't complete, you block. Blocking means that the kernel puts you to sleep. If you're sleeping, you don't have the opportunity to do other useful work. Instead, use non-blocking I/O operations. Kernel returns immediately if it can't complete the I/O. You end up with a bunch of in-progress I/O operations. You use select() to have the kernel tell you when one of them can progress.
import socket

sock = socket.socket()
sock.connect(('localhost', 1234))

sock.send('foo
' * 10 * 1024 * 1024)
Simple example, connect to a server on port 1234, write a bunch of stuff
This send() function won't finish until all the data is written
While you're waiting for the I/O to complete, you can't do anything else
If you're not familiar with berkley sockets ...
socket() creates a socket ... it's just a data structure
connect() initiates the TCP connection with the server
Some handshake packets will go back and forth before this completes
send() takes the data buffer, splits up and sends as TCP packets
import socket

sock = socket.socket()
sock.connect(('localhost', 1234))
sock.setblocking(0)
sock.send('foo
' * 10 * 1024 * 1024)
import socket

sock = socket.socket()
sock.connect(('localhost', 1234))
sock.setblocking(0)

buf = b'foo\n' * 10 * 1024 * 1024
while len(buf):
    buf = buf[:sock.send(buf):]
import errno
import select
import socket

sock = socket.socket()
sock.connect(('localhost', 1234))
sock.setblocking(0)

buf = buffer('foo
' * 10 * 1024 * 1024)
while len(buf):
    try:
        buf = buf[:sock.send(buf):]
    except socket.error, e:
        if e.errno != errno.EAGAIN:
            raise e
import errno
import select
import socket

sock = socket.socket()
sock.connect(('localhost', 1234))
sock.setblocking(0)

buf = buffer('foo\n' * 10 * 1024 * 1024)
while len(buf):
    try:
        buf = buf[:sock.send(buf)]
    except socket.error, e:
        if e.errno != errno.EAGAIN:
            raise e

select.select([], [sock], [])
import errno
import select
import socket

sock = socket.socket()
sock.connect(('localhost', 1234))
sock.setblocking(0)

buf = buffer('foo\n' * 10 * 1024 * 1024)
while len(buf):
    try:
        buf = buf[sock.send(buf):
    except socket.error, e:
        if e.errno != errno.EAGAIN:
            raise e

i = 0
while i < 5000000:
    i += 1

select.select([], [sock], [])
Eventlet
Eventlet is one of several async I/O frameworks for Python
It’s magic, to its detriment IMHO
Co-operative co-routine scheduling
Eventlet coroutines are called green threads (from Java)
green threads != threads
Many greenthreads in one OS thread
They are very cheap
We call them coroutines because they cooperatively yield to one another
greenlet allows you to save a stack and switch to another stack
from eventlet.green import socket

sock = socket.socket()
sock.connect(('localhost', 1234))
sock.send('foo
' * 10 * 1024 * 1024)
import eventlet
from eventlet.green import socket

def busy_loop():
    pass

eventlet.spawn(busy_loop)

sock = socket.socket()
sock.connect(('localhost', 1234))
sock.send('foo\n' * 10 * 1024 * 1024)
import eventlet
from eventlet.green import socket

def busy_loop():
    while True:
        i = 0
        while i < 5000000:
            i += 1

eventlet.spawn(busy_loop)

sock = socket.socket()
sock.connect(('localhost', 1234))
sock.send('foo\n' * 10 * 1024 * 1024)
import eventlet
from eventlet.green import socket

def busy_loop():
    while True:
        i = 0
        while i < 5000000:
            i += 1
            eventlet.sleep()

    eventlet.spawn(busy_loop)

    sock = socket.socket()
    sock.connect(('localhost', 1234))
    sock.send('foo\n' * 10 * 1024 * 1024)
Twisted
Eventlet has a scheduler, twisted has an event loop
Event loop is a loop (i.e. while loop) which checks for I/O events
Twisted invokes callbacks for events, rather than scheduling coroutines
Eventlet is confusing because magic
Twisted is confusing because abstractions
from twisted.internet import reactor
reactor.run()
from twisted.internet import protocol
from twisted.internet import reactor

reactor.connectTCP('localhost', 1234, protocol.ClientFactory())
reactor.run()
from twisted.internet import protocol
from twisted.internet import reactor

class Test(protocol.Protocol):
    pass

class TestFactory(protocol.ClientFactory):
    def buildProtocol(self, addr):
        return Test()

reactor.connectTCP('localhost', 1234, TestFactory())

reactor.run()
from twisted.internet import protocol
from twisted.internet import reactor

class Test(protocol.Protocol):
    def connectionMade(self):
        self.transport.write('foo
' * 2 * 1024 * 1024)

class TestFactory(protocol.ClientFactory):
    def buildProtocol(self, addr):
        return Test()

reactor.connectTCP('localhost', 1234, TestFactory())
reactor.run()
from twisted.internet import protocol
from twisted.internet import reactor

class Test(protocol.Protocol):
    def connectionMade(self):
        self.transport.write('foo\n' * 2 * 1024 * 1024)

class TestFactory(protocol.ClientFactory):
    def buildProtocol(self, addr):
        return Test()

reactor.connectTCP('localhost', 1234, TestFactory())

def busy_loop():
    i = 0
    while i < 5000000:
        i += 1
    reactor.callLater(0, busy_loop)

reactor.run()
from twisted.internet import protocol
from twisted.internet import reactor

class Test(protocol.Protocol):
    def connectionMade(self):
        self.transport.write('foo\n' * 2 * 1024 * 1024)

class TestFactory(protocol.ClientFactory):
    def buildProtocol(self, addr):
        return Test()

reactor.connectTCP('localhost', 1234, TestFactory())

def busy_loop():
    i = 0
    while i < 5000000:
        i += 1
        reactor.callLater(0, busy_loop)
    reactor.callLater(0, busy_loop)

reactor.run()
Tulip
Guido's "Async I/O Support Rebooted"
Support for both coroutine and callback models
It's event loop is called an event loop
import tulip

event_loop = tulip.get_event_loop()

event_loop.run_forever()
import socket
import tulip

sock = socket.socket()
sock.connect(('localhost', 1234))
sock.setblocking(0)

event_loop = tulip.get_event_loop()

event_loop.run_forever()
import errno
import socket
import tulip

sock = socket.socket()
sock.connect(('localhost', 1234))
sock.setblocking(0)

buf = memoryview(str.encode('foo\n' * 2 * 1024 * 1024))
def do_write():
    global buf
    while True:
        try:
            buf = buf[sock.send(buf):]
        except socket.error as e:
            if e.errno != errno.EAGAIN:
                raise e
            return

event_loop = tulip.get_event_loop()
event_loop.add_writer(sock, do_write)
event_loop.run_forever()
import errno
import socket

import tulip

sock = socket.socket()
sock.connect(('localhost', 1234))
sock.setblocking(0)

buf = memoryview(str.encode('foo\n' * 2 * 1024 * 1024))

def do_write():
    global buf
    while True:
        try:
            buf = buf[sock.send(buf):]
        except socket.error as e:
            if e.errno != errno.EAGAIN:
                raise e
            return

def busy_loop():
    i = 0
    while i < 5000000:
        i += 1
        event_loop.call_soon(busy_loop)

event_loop = tulip.get_event_loop()
event_loop.add_writer(sock, do_write)
event_loop.call_soon(busy_loop)
event_loop.run_forever()
import tulip

event_loop = tulip.get_event_loop()

event_loop.run_forever()
import tulip

class Protocol(tulip.Protocol):
    buf = b'foo\n' * 10 * 1024 * 1024
    def connection_made(self, transport):
        transport.write(self.buf)
        transport.close()

event_loop = tulip.get_event_loop()
event_loop.run_forever()
import tulip

class Protocol(tulip.Protocol):
    buf = b'foo\n' * 10 * 1024 * 1024
    def connection_made(self, transport):
        transport.write(self.buf)
        transport.close()

event_loop = tulip.get_event_loop()
tulip.Task(event_loop.create_connection(Protocol, 'localhost', 1234))
event_loop.run_forever()
```python
import tulip

class Protocol(tulip.Protocol):
    buf = b'foo\n' * 10 * 1024 * 1024
    def connection_made(self, transport):
        transport.write(self.buf)
        transport.close()
    
def connection_lost(self, exc):
        event_loop.stop()

event_loop = tulip.get_event_loop()
tulip.Task(event_loop.create_connection(Protocol, 'localhost', 1234))
event_loop.run_forever()
```
import tulip

class Protocol(tulip.Protocol):
    buf = b'foo\n' * 10 * 1024 * 1024

def connection_made(self, transport):
    event_loop.call_soon(busy_loop)
    transport.write(self.buf)
    transport.close()

def connection_lost(self, exc):
    event_loop.stop()

def busy_loop():
    i = 0
    while i < 5000000:
        i += 1
        event_loop.call_soon(busy_loop)

event_loop = tulip.get_event_loop()
tulip.Task(event_loop.create_connection(Protocol, 'localhost', 1234))
event_loop.run_forever()
Generators as Coroutines
import os

def contents_of_files(dir):
    ret = []
    for f in os.listdir(dir):
        path = os.path.join(dir, f)
        ret.append((path, file(path).read()))
    return ret

for path, contents in contents_of_files(dir):
    if search_for in contents:
        print("found", search_for, "in", path)
        break
import os

def contents_of_files(dir):
    for f in os.listdir(dir):
        path = os.path.join(dir, f)
        yield (path, file(path).read())

for path, contents in contents_of_files(dir):
    if search_for in contents:
        print("found", search_for, "in", path)
        break
def gen():
    i = 0
    while i < 2:
        print(i)
        yield
        i += 1

i = gen()
print("yo!")
next(i)
print("hello!")
next(i)
print("bye!")
try:
    next(i)
except StopIteration:
    print("stopped")
Task = collections.namedtuple('Task', ['generator', 'wfd', 'idle'])

running = True

def quit():
    global running
    running = False

while running:
    finished = []
    for n, t in enumerate(tasks):
        try:
            next(t.generator)
        except StopIteration:
            finished.append(n)
        map(tasks.pop, finished)

    wfds = [t.wfd for t in tasks if t.wfd]
    timeout = 0 if [t for t in tasks if t.idle] else None

    select.select([], wfds, [], timeout)
def busy_loop():
    while True:
        i = 0
        while i < 5000000:
            i += 1
            yield

tasks.append(Task(busy_loop(), wfd=None, idle=True))
sock = socket.socket()
sock.connect(('localhost', 1234))
sock.setblocking(0)

def write(data):
    buf = memoryview(data)
    while len(buf):
        try:
            buf = buf[sock.send(buf):]
        except socket.error as e:
            if e.errno != errno.EAGAIN:
                raise e
        yield

def write_stuff():
    yield from write(b'foon' * 2 * 1024 * 1024)
    yield from write(b'barn' * 2 * 1024 * 1024)
    quit()

tasks.append(Task(write_stuff(), wfd=sock, idle=False))
References

- Eventlet - http://eventlet.net
- Twisted - http://twistedmatrix.com
- Tulip - http://www.python.org/dev/peps/pep-3156
- yield from - http://www.python.org/dev/peps/pep-0380