Tulip: Async I/O for Python 3

Guido van Rossum
guido@python.org
SF Python Meetup at Yelp
10/16/2013


Dropbox

Overview

- About asynchronous I/O
- The road to Tulip
- Architecture
- Examples
- Q & A

ABOUT ASYNCHRONOUS I/O

What is asynchronous I/O?

- Opinions differ...
- Non-blocking file descriptors?
- Callbacks?
  - on ready or on completion?
- Threads?
- Micro-threads?

Back to fundamentals

- select, poll etc. (UNIX)
- IOCP (Windows, Solaris, maybe others)
- Whoa... these are really different!
- Different DNA goes back many years

Select and friends

- Use select, poll etc. to wait until I/O possible
- Then attempt the I/O operation
  - may still block, so use non-blocking mode

Select and friends

- Use select, poll etc. to wait until I/O possible
- Then attempt the I/O operation
  - may still block, so use non-blocking mode
- This is how most UNIX systems do it
  - most UNIX: select(), poll() — don't scale well
  - Linux: epoll()
  - BSD/OSX: kqueue()

Windows

- Windows has select(), but it's a toy (mostly)

Windows: IOCP

- Start the I/O operation in "overlapped" mode
- Receive a callback when it's done
- Uses "Completion Ports"
- High performance, but really different
  - have to use this if you want to scale on Windows

How to unify these APIs?

- Build good abstractions
- This is why we have Transports and Protocols
- More in the Architecture section

THE ROAD TO TULIP

Tulip's goals

- State-of-the art asynchronous I/O for Python
- Future-proof, but practical
- Make good use of PEP 380 **yield from**
  - new in Python 3.3
- No new language features
  - beyond Python 3.3
- Interoperability with legacy frameworks

What's interoperability?

- Being able to use different frameworks IN THE SAME PROGRAM
- Share an event loop
- Smoothly transfer control between paradigms

Non-goals

- Perfection
- The kitchen sink
- Replace third-party frameworks
- Specific protocol implementations
- Replace httplib, smptlib, etc.
- Support older Python versions (3.2, 2.7)

Building consensus

- Open development
  - python-tulip@googlegroups.com
  - code.google.com/p/tulip
- In-person meetings with experts
- Encourage collaborators and contributors

Personal development

- Kept an open mind
- Learned a lot about other async frameworks
  - e.g. in-depth study of Deferred in Twisted
- Changed my mind publicly
- Didn't lose my way

Some more goals

- UNIX, Windows, OS X
- IPv4 and IPv6
- TCP and UDP
- Basic SSL (using stdlib/openssl only)
  - default to pragmatically secure
- Pipes and subprocess
  - named pipes on Windows

Even more goals

- No dependencies on 3rd party software
  - standard library only
  - no new extension modules
    - exception: IOCP ("overlapped I/O") on Windows
- Don't force people to use **yield from**
  - even though personally I like **yield from** best
  - callbacks are the lowest common denominator
    - but don't force their use either (in most cases)

ARCHITECTURE

Main components

- Coroutines, Futures, Tasks
  - because I don't like callbacks much
- Event loop and event loop policy
  - interoperability happens here
- Transports and Protocols
  - the actual I/O happens here

ARCHITECTURE:
COROUTINES, FUTURES, TASKS

Coroutines, Futures, Tasks

- Coroutine
  - really just a generator function
  - decorated with @coroutine
- Future
  - object representing an eventual result (or error)
- Task
  - a Future wrapping a coroutine

Coroutines: using **yield from**

```
@coroutine
def fetch(host, port):
    r, w = yield from open_connection(host, port)
    w.write(b'GET / HTTP/1.0\r\n\r\n')
    while (yield from r.readline()).decode('latin-1').strip():
        pass
    body = yield from r.read()
    return body

@coroutine
def start():
    data = yield from fetch('python.org', 80)
    print(data.decode('utf-8'))
```

I can't actually explain this

- You just have to squint :-)

- Imagine **yield from** isn't there
- Interpret each function as sequential code
- Now you know what it does

- Forget **yield from**'s formal (PEP 380) definition

(Have another look)

```
@coroutine
def fetch(host, port):
    r, w = yield from open_connection(host, port)
    w.write(b'GET / HTTP/1.0\r\n\r\n')
    while (yield from r.readline()).decode('latin-1').strip():
        pass
    body = yield from r.read()
    return body

@coroutine
def start():
    data = yield from fetch('python.org', 80)
    print(data.decode('utf-8'))
```

(Did you blink?)

```
@coroutine
def fetch(host, port):
    r, w = open_connection(host, port)
    w.write(b'GET / HTTP/1.0\r\n\r\n')
    while (r.readline()).decode('latin-1').strip():
        pass
    body = r.read()
    return body

@coroutine
def start():
    data = fetch('python.org', 80)
    print(data.decode('utf-8'))
```

Futures

- Almost, but not quite, like PEP 3148
  - concurrent.futures.Future
- Tulip API is (almost) the same:
  - f.set_result(res); res = f.result()
  - f.set_exception(exc); exc = f.exception()
  - f.done()
  - f.add_done_callback(cb) # will call cb(f)
  - f.cancel(); f.cancelled()

Futures and coroutines

- **yield from** works with Futures too!
  - f = Future()
    - then arrange that f.set_result() is eventually called
  - res = **yield from** f
    - blocks until f.done(), then sets res = f.result()
- Typically f is returned by a function
  - res = **yield from** some_function(...)
  - set_result() typically called by some I/O handler

Tasks

A riddle wrapped in a mystery inside an enigma

Tasks

A riddle wrapped in a mystery inside an enigma

—Churchil, about Russia

Tasks

- Actually, it's a coroutine wrapped in a Future
- class Task is a subclass of class Future
- So it works with **yield from** too!

- Example:
  - r = **yield from** Task(some_coroutine(...))

Tasks vs. coroutines

Tasks vs. coroutines

- Compare:
  - res = **yield from** some_coroutine(...)
  - res **= yield from** Task(some_coroutine(...))

- Task can make progress without waiting for it
  - as long as you wait for something else
    - i.e. **yield from** <something else>

Exception handling

- Coroutines can raise exceptions
  - catch these in the caller with try/except
- Futures have exceptions too
  - f.result() raises if set_exception() used
  - use f.exception() to check without raising
- **raise** *exc* in a Task calls t.set_exception(*exc*)
  - then **yield from** Task(...) will raise *exc*

Asynchronous exceptions

- If a Task raises in the forest, and no one is around to catch it, does it print a traceback?

Asynchronous exceptions

- If a Task raises in the forest, and no one is around to catch it, does it print a traceback?
- Yes!
  - we log it
    - easier said than done, but we managed
      - use an auxiliary object with a __del__() method
  - we don't log:
    - ignored successes; incomplete Tasks; plain Futures
    - coroutines (sadly)

ARCHITECTURE:
THE EVENT LOOP

The event loop

- Multiplexes different activities
  - Immediate callbacks, timed callbacks
  - I/O callbacks (registered by FD)
  - UNIX signals
  - interface with threads
- APIs for creating connections and servers
  - TCP, SSL, UDP, pipes, subprocesses

Event loop callbacks

- loop.call_soon(callback, *args)
- loop.call_later(delay, callback, *args)
- loop.call_at(when, callback, *args)
- loop.time()

- No API for repeated callbacks; DIY is easy :-)

I/O callbacks

- loop.add_reader(fd, callback, *args)
- loop.add_writer(fd, callback, *args)
- loop.remove_reader(fd)
- loop.remove_writer(fd)

- Low-level — do not use
- Built on "selectors" module (new in 3.4)

UNIX signal callbacks

- loop.add_signal_handler(sig, callback, *args)
- loop.remove_signal_handler(sig)

Thread interface

- loop.call_soon_threadsafe(callback, *args)
- loop.run_in_executor(executor, callback, *args)
- loop.set_default_executor(executor)

- Executor is a thread pool (PEP 3148)

Starting and stopping

- loop.run_forever()
- loop.stop()
- loop.run_until_complete(f)

- No recursive loops
- Do not use except in main()
  - or in unit tests

Event loop access

- Get the current event loop:
  - get_event_loop()
- Set the current event loop:
  - set_event_loop(loop)
- Create a new default event loop:
  - new_event_loop()
- Custom event loops can be instantiate

Default event loop policy

- Only one event loop per thread
- Only main thread has a default event loop
- In other threads, use
  - set_event_loop(new_event_loop())
- Otherwise get_event_loop() returns None

Event loop policy

- get/set/new_event_loop() are configurable
- To change the policy:
  - set_event_loop_policy(policy)
- To read the policy:
  - get_event_loop_policy()
- The policy is the only really global thing
- Primarily used to change new_event_loop()

ARCHITECTURE: TRANSPORTS AND PROTOCOLS

Transports and Protocols

- These come in pairs, call each other
  - symbiotic relationship
- A Transport represents a connection
  - e.g. a socket, pipe, or SSL connection
    - typically implemented by the framework
- A Protocol represents an application
  - e.g. an HTTP server or client
    - typically implemented by you!

Client or server?

Client or server?

- Both!
- loop.create_connection(...):
  - creates one Transport and one Protocol
- loop.create_server(...):
  - creates a T+P pair for each accepted connection

Transport/Protocol interface

- Transport calls Protocol methods:
  - connection_made(transport)
  - data_received(data)
  - eof_received()
  - connection_lost(exc)

- UDP is a bit different
- So are pipes and subprocesses

Protocol/Transport interface

- Protocol may call Transport methods:
  - write(data)
  - writelines(list_of_data)
  - write_eof()
  - close()

- UDP and pipes/subprocesses different again

Auxiliary Transport methods

- Some more stuff:
  - can_write_eof()
  - abort()
  - get_extra_info(key)
    - 'socket'
    - 'sockname', 'peername'
    - 'sslcontext', 'peercert', ...

Flow control

- Protocol can call t.pause(), t.resume()
- Transport can call p.pause_writing(), p.resume_writing()
  - not yet implemented
  - maybe the others should be called pause_reading(), resume_reading()

Are these coroutines?

Are these coroutines?

- No!
- Protocol methods are plain callbacks
- Transport methods are just, um, methods

- This is done for interoperability's sake
- There's a higher-level API offering coroutines
  - open_connection() -> reader, writer

EXAMPLES

(Click for references)

Q & A

References

- code.google.com/p/tulip/
- groups.google.com/forum/#!forum/python-tulip
- www.python.org/dev/peps/pep-3156/
- bugs.python.org/issue19262