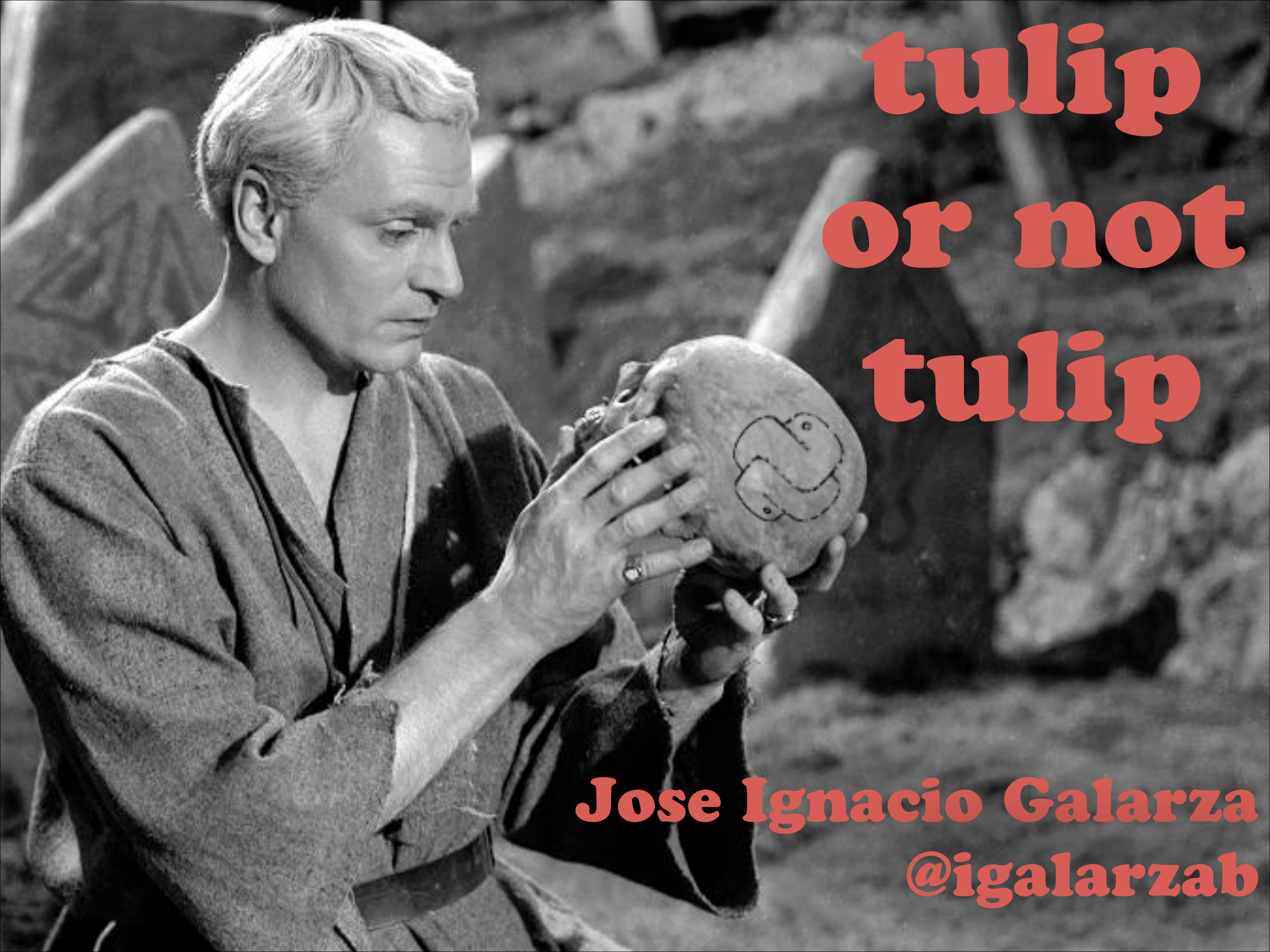
tulip
or not
tulip
Jose Ignacio Galarza
@igalarzab

Index

what is tulip?

generators

coroutines

tulip components

asyncio

~~tulip~~

“asynchronous IO support rebooted”

Python >= 3.3


event loop
coroutines
futures
tasks
transports
protocols

some

necessary

history

python includes generators

PEP 0255

yield

```
def work_hard_normal():
    results = []

    for i in range(1, 10):
        print('Working very hard %d times...' % i)
        results.append(i)

    return results

def working_hard_generator():
    for i in range(1, 10):
        print('Working very hard %d times...' % i)
        yield i

if __name__ == '__main__':
   for result in work_hard_normal():
        if result % 5 == 0:
            print('Eureka!')
            break

   for result in working_hard_generator():
        if result % 5 == 0:
            print ('Eureka!')
            break
```

result

```
$ python3 001-generator.py

  Working normal 1…
  Working normal 2…
  […]
  Working normal 10…
  Eureka!
  Generatoorrr…1
  Generatoorrr…2
  […]
  Generatoorrr…5
  Eureka!
```

python includes coroutines

PEP 0342

send values to a generator

generators

vs

coroutines

```
def list_dir(path, target):
    for dirpath, dirnames, filenames in os.walk(path):
        for filename in filenames:
            target.send(filename)

@coroutine
def filter_str(pattern, target):
    while True:
        filename = (yield)
        if pattern in filename:
            target.send(filename)

@coroutine
def print_match():
    while True:
        result = (yield)
        print(result)

if __name__ == '__main__':
    list_dir('.', filter_str('py', print_match()))
```

```
def list_dir(path, target):
    for dirpath, dirnames, filenames in os.walk(path):
        for filename in filenames:
            target.send(filename)

@coroutine
def filter_str(pattern, target):
    while True:
        filename = (yield)
        if pattern in filename:
            target.send(filename)

@coroutine
def print_match():
    while True:
        result = (yield)
        print(result)

if __name__ == '__main__':
    list_dir('.', filter_str('py', print_match()))
```

```
def list_dir(path, target):
    for dirpath, dirnames, filenames in os.walk(path):
        for filename in filenames:
            target.send(filename)

@coroutine
def filter_str(pattern, target):
    while True:
        filename = (yield)
        if pattern in filename:
            target.send(filename)

@coroutine
def print_match():
    while True:
        result = (yield)
        print(result)

if __name__ == '__main__':
    list_dir('.', filter_str('py', print_match()))
```

```python
def list_dir(path, target):
    for dirpath, dirnames, filenames in os.walk(path):
        for filename in filenames:
            target.send(filename)

@coroutine
def filter_str(pattern, target):
    while True:
        filename = (yield)
        if pattern in filename:
            target.send(filename)

@coroutine
def print_match():
    while True:
        result = (yield)
        print(result)

if __name__ == '__main__':
    list_dir('.', filter_str('mp3', print_match()))
```

what the hell is that decorator?

```
def coroutine(func):
    """
    Decorator to auto-start coroutines.
    Got it from: PEP-0342
    """

    def wrapper(*args, **kwargs):
        gen = func(*args, **kwargs)
        next(gen)
        return gen

    wrapper.__name__ = func.__name__
    wrapper.__dict__ = func.__dict__
    wrapper.__doc__  = func.__doc__
    return wrapper
```

result

```
$ python3 003-coroutines.py

  El Fary - La Mandanga.mp3
  Julito Iglesias - Grandes exitos.mp3
  […]
```

python enhances generators

PEP 0380

“A syntax is proposed for a generator to delegate part of its operations to another generator”

yield from

without yield from

```
class TreeBasic:

    def __init__(self, data, left=None, right=None):
        self.left = left
        self.data = data
        self.right = right

    def __iter__(self):
        if self.left:
            for node in self.left:
                yield node

        yield self.data

        if self.right:
            for node in self.right:
                yield node
```

with yield from

```
class TreeYieldFrom:

    def __init__(self, data, left=None, right=None):
        self.left = left
        self.data = data
        self.right = right

    def __iter__(self):
        if self.left:
            yield from self.left

        yield self.data

        if self.right:
            yield from self.right
```

let's do an scheduler

declaring the scheduler

```
class Scheduler:

    def __init__(self):
        self.tasks = deque()

    def schedule(self, task):
        self.tasks.append(task)

    def run(self):
        while self.tasks:
            task = self.tasks.popleft()

            try:
                task.run()
            except StopIteration:
                print('Task %s has finished' % task)
            else:
                self.tasks.append(task)
```

declaring what's a task

```
class Task:

    ID = 0

    def __init__(self, runner):
        Task.ID += 1
        self.id = Task.ID
        self.runner = runner

    def __str__(self):
        return str(self.id)

    def run(self):
        result = next(self.runner)
        print('[%d] %s' % (self.id, result))
```

some tasks examples

```
def list_dir(directory):
    for item in os.listdir(directory):
        yield item


def echo_text(number_times):
    for i in range(number_times):
        yield 'Hi dude!'
```

creating the tasks...

```
if __name__ == '__main__':
    s = Scheduler()

    s.schedule(Task(list_dir('.')))
    s.schedule(Task(echo_text(5)))
    s.schedule(Task(echo_text(3)))

    s.run()
```

result...

```
$ python3 004-scheduler.py

  [1] 001-generator.py
  [2] Hi dude!
  [3] Hi dude!
  [1] 002-pipeline.py
  [2] Hi dude!
  [3] Hi dude!
  [1] 003-coroutine.py
  [2] Hi dude!
  [3] Hi dude!
  [1] 004-tree.py
  [2] Hi dude!
Task 3 has finished
  [1] 005-scheduler.py
  [2] Hi dude!
  [1] 00X-scheduler.pyc
Task 2 has finished
  […]
```

python introduces tulip

PEP 3156

event

loop

the event loop *multiplexes* a variety of events

IO events use the best possible * *selector* for the platform

* new module in Python 3.4
epoll, kqueue, IOCP

interoperability **with other frameworks is one of the main focuses**

how to run the event loop?

```
# Get the main event loop
loop = asyncio.get_event_loop()

# Execute it until the future returns
loop.run_until_complete(future)

# Run forever (until stop() is called)
loop.run_forever()
```

how to run callbacks?

```
# Run the callback as soon as possible
loop.call_soon(callback, *args)

# Run the callback in `delay` seconds or more
loop.call_later(delay, callback, *args)

# Run the callback at the provided `when` or more
loop.call_at(when, callback, *args)
```

much more about
this in @saghul's talk

check his slides!

coroutines

it's not mandatory to use them, but tulip does it really well

we already know what's a coroutine

```
@coroutine
def get_url():
    r, w = yield from open_connection('google.es', 80)

    w.write(b'GET / HTTP/1.0\r\n\r\n')
    result = yield from r.read()
    print(result)


if __name__ == '__main__':
    loop = asyncio.get_event_loop()
    loop.run_until_complete(get_url())
```

futures

promises to return
a *result* or an
exception sometime
in the future

they are really

* *similar* to

concurrent.futures

* *almost the same API*

generators!

use *yield from* with *futures!*

an easy example!

```
@asyncio.coroutine
def wait_and_resolve_future(future):
    for i in range(3):
        print('Sleeping 1 second')
        yield from asyncio.sleep(1)

    future.set_result('Future is done!')


if __name__ == '__main__':
    loop = asyncio.get_event_loop()

    future = asyncio.Future()
    asyncio.Task(wait_and_resolve_future(future))

    loop.run_until_complete(future)
    print(future.result())
```

tasks

it's a coroutine

* *wrapped* in a future

* *in fact, it's a subclass*

tasks can make *progress* alone, unlike coroutines

why?

the __*init*__ schedules a *callback* with the next step of the coroutine

```
# asyncio/task.py:110

class Task(futures.Future):

    def __init__(self, coro, *, loop=None):
        # . . .
        self._loop.call_soon(self._step)
        # . . .
```

_step runs the coroutine

```
# asyncio/task.py:246

def _step(self, value=None, exc=None):
    # . . .
    try:
        if exc is not None:
            result = coro.throw(exc)
        elif value is not None:
            result = coro.send(value)
        else:
            result = next(coro)
    except StopIteration as exc:
        self.set_result(exc.value)
    except futures.CancelledError as exc:
        super().cancel()
    except Exception as exc:
        self.set_exception(exc)
    # . . .
```

awesome

:D

transports and protocols

transports and protocols are used in pairs

*“the transport is concerned about **how** bytes are transmitted”*

“the protocol determines which bytes to transmit”

protocols call transport methods (TCP)

```
# Write data to the transport
write(data)

# Write data using an iterator
writelines(list_of_data)

# Checks if the protocol allows to write EOF
can_write_eof()

# Close the writing end
write_eof()

# Close the connection
close()
```

protocol's callbacks (TCP)

```
# A new connection has been made
connection_made(transport)

# New data has been received
data_received(transport)

# EOF received (not all protocols support it)
eof_received(transport)

# Broken connection
connection_lost(exc)
```

simple ECHO protocol using TCP as the transport

the protocol of the server

```
class EchoServer(asyncio.Protocol):

    def connection_made(self, transport):
        print('Connected')
        self.transport = transport

    def data_received(self, data):
        print('[R] ', data.decode())
        print('[S] ', data.decode())
        self.transport.write(data)

    def eof_received(self):
        pass

    def connection_lost(self, exc):
        print('Connection lost')
```

the protocol of the client

```
class EchoClient(asyncio.Protocol):

    def connection_made(self, transport):
        self.transport = transport
        self.transport.write(b'Hola caracola')
        print('[S] ', 'Hola caracola')

    def data_received(self, data):
        print('[R] ', data)

    def eof_received(self):
        pass

    def connection_lost(self, exc):
        print('Connection lost')
        asyncio.get_event_loop().stop()
```

executing both endpoints

```
def start_client(event_loop):
    task = asyncio.Task(
        event_loop.create_connection(
            EchoClient,
            '127.0.0.1',
            8080
        )
    )
    event_loop.run_until_complete(task)

def start_server(event_loop):
    server = event_loop.create_server(
        EchoServer,
        '127.0.0.1',
        8080
    )
```

main program

```
if __name__ == '__main__':

    if len(sys.argv) != 2:
        print('Call with --server or --client flag')
        sys.exit()

    loop = asyncio.get_event_loop()
    loop.add_signal_handler(signal.SIGINT, loop.stop)

    if sys.argv[1] == '--server':
        start_server(loop)
    else:
        start_client(loop)

    loop.run_forever()
```

and with UDP?

*almost the same!

* check the examples!

questions?

http://twitter.com/igalarzab

http://github.com/igalarzab

thank you!