

Exploration de la boucle d'événements asyncio

Pycon 2014, Lyon



Victor STINNER
victor.stinner@gmail.com

Victor STINNER



- Core developer Python depuis 2010
- Contributeur à asyncio (code, doc)
- Auteur de Trollius, portage d'asyncio sur Python 2.6
- Libriste convaincu : publie sur github et bitbucket
- Travaille pour **eNovance** sur OpenStack

Mise en garde



- Code simplifié, API proche d'asyncio, mais différent
- Pas de gestion d'erreur ni d'optimisation
- Code écrit pour Python 3

Fonctions de rappel

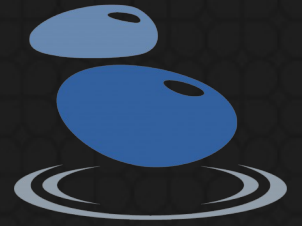


```
class CallbackEventLoop:
    def __init__(self):
        self.callbacks = []

    def call_soon(self, func):
        self.callbacks.append(func)

    def execute_callbacks(self):
        callbacks = self.callbacks
        self.callbacks = []
        for cb in callbacks:
            cb()
```

Fonctions de rappel



Code

```
loop.call_soon(hello_world)
```

Callbacks

```
hello_world()
```

Output

Fonctions de rappel



Code

```
loop.call_soon(hello_world)  
loop.execute_callbacks()
```

Callbacks

```
hello_world()
```

Output

Hello World!

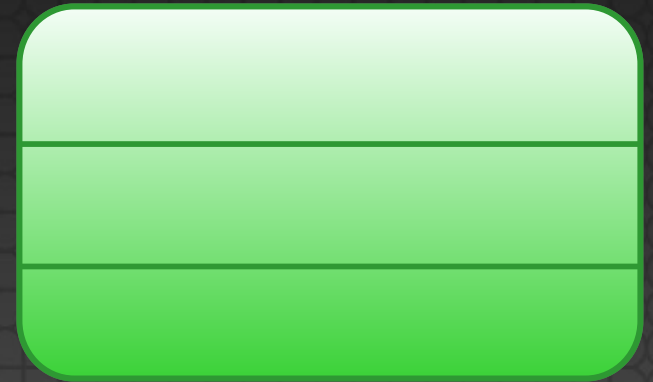
Fonctions de rappel



Code

```
loop.call_soon(hello_world)  
loop.execute_callbacks()
```

Callbacks



Output

Hello World!

Minuteurs



```
class TimerEventLoop(CallbackEventLoop):
```

```
    def __init__(self):  
        super().__init__()  
        self.timers = []
```

```
    def call_at(self, when, func):  
        timer = (when, func)  
        self.timers.append(timer)
```

```
    ...
```


Minuteurs



```
class TimerEventLoop(CallbackEventLoop):  
    ...  
    def execute_timers(self):  
        now = time.time()  
        new_timers = []  
        for when, func in self.timers:  
            if when <= now:  
                self.call_soon(func)  
            else:  
                new_timers.append((when, func))  
        self.timers = new_timers  
  
        self.execute_callbacks()
```

Minuteurs



Code

```
loop.call_at(1, hello_world)
```

Timers

```
(1, hello_world)
```

Output

Callbacks

Minuteurs



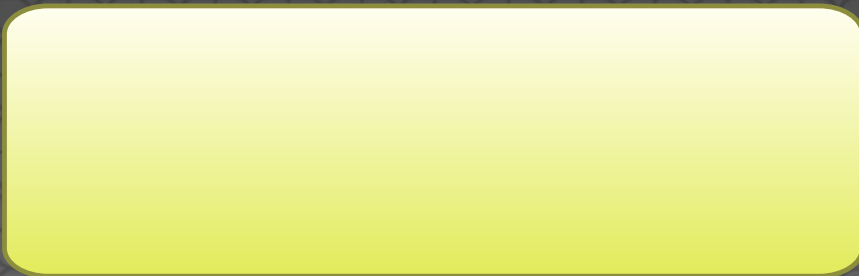
Code

```
loop.call_at(1, hello_world)  
loop.call_at(5, exit)
```

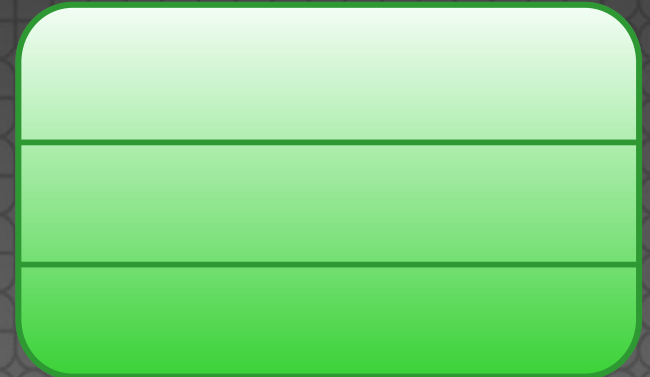
Timers

(1, hello_world)
(5, exit)

Output



Callbacks



Minuteurs



Code

```
loop.call_at(1, hello_world)
loop.call_at(5, exit)
loop.call_at(2, good_bye)
```

Timers

```
(1, hello_world)
(5, exit)
(2, good_bye)
```

Output

Callbacks

Minuteurs



Code

```
loop.call_at(1, hello_world)
loop.call_at(5, exit)
loop.call_at(2, good_bye)
loop.execute_timers()
```

Output



Timers

(1, hello_world)

(5, exit)

(2, good_bye)

Callbacks

hello_world()

good_bye()

Minuteurs



Code

```
loop.call_at(1, hello_world)
loop.call_at(5, exit)
loop.call_at(2, good_bye)
loop.execute_timers()
```

Timers

(5, exit)

Output

Hello World!
Good bye.

Callbacks

hello_world()
good_bye()

Minuteurs



Code

```
loop.call_at(1, hello_world)
loop.call_at(5, exit)
loop.call_at(2, good_bye)
loop.execute_timers()
```

Timers

(5, exit)

Callbacks

Output

Hello World!
Good bye.

Multiplexeur E/S



- Sockets : réseau, TCP, UDP et UNIX
- Pipes : processus, signaux
- Module **select** : **select()** Windows, **poll()**, **epoll()** Linux, **kqueue()** BSD et Mac OS X, **devpoll()** Solaris
- Module **selectors** de Python 3.4

Multiplexeur E/S



```
from selectors import DefaultSelector

class SelectorEventLoop(TimerEventLoop):

    def __init__(self):
        super().__init__()
        self.selector = DefaultSelector()

    def add_reader(self, sock, func):
        self.selector.register(sock,
                               selectors.EVENT_READ,
                               data=func)

    ...
```


Multiplexeur E/S



```
class SelectorEventLoop(TimerEventLoop):  
    ...  
    def select(self):  
        timeout = self.compute_timeout()  
  
        events = self.selector.select(timeout)  
        for key, mask in events:  
            func = key.data  
            self.call_soon(func)  
  
        self.execute_timers()
```

Multiplexeur E/S



```
class SelectorEventLoop(TimerEventLoop):  
    ...  
    def compute_timeout(self):  
        if self.callbacks:  
            # already something to do  
            return 0  
        elif self.timers:  
            next_timer = min(self.timers)[0]  
            timeout = next_timer - time.time()  
            return max(timeout, 0.0)  
        else:  
            # blocking call  
            return None
```

Multiplexeur E/S



Code

```
s, c = socket.socketpair()  
loop.add_reader(s, reader)
```

Selector

s: idle

Callbacks

Output

Multiplexeur E/S



Code

```
s, c = socket.socketpair()  
loop.add_reader(s, reader)  
c.send(b'abc')
```

Selector

s: read event

Callbacks

Output

Multiplexeur E/S



Code

```
s, c = socket.socketpair()
loop.add_reader(s, reader)
c.send(b'abc')
loop.select()
```

Output



Selector

s: read event

Callbacks

reader()

Multiplexeur E/S



Code

```
s, c = socket.socketpair()
loop.add_reader(s, reader)
c.send(b'abc')
loop.select()
```

Output

Received: b'abc'

Selector

s: idle

Callbacks

reader()

Multiplexeur E/S



Code

```
s, c = socket.socketpair()
loop.add_reader(s, reader)
c.send(b'abc')
loop.select()
```

Selector

s: idle

Callbacks

Output

Received: b'abc'

Générateur



- Fonction qui peut être mise en pause
- Mot clé **yield**
- Python 3.3 apporte **yield from** et **return** aux générateurs
- **yield from** : chaîne l'exécution d'un autre générateur

Générateur



Code

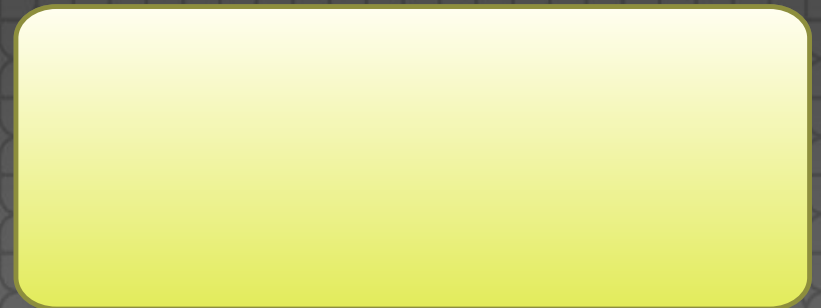
```
gen = producer()
```

producer() generator

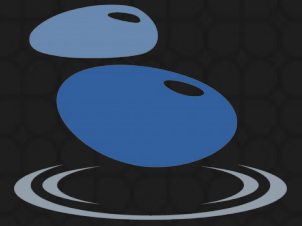


```
yield "start"  
return "stop"
```

Output



Générateur



Code

```
gen = producer()  
print(next(gen))
```

producer() generator



```
yield "start"  
return "stop"
```

Output

start

Générateur



Code

```
gen = producer()
print(next(gen))
try:
    next(gen)
except StopIteration as e:
    print(e.value)
```

producer() generator



```
yield "start"
return "stop"
```

Output

```
start
stop
```

Coroutine



- Générateur utilisant uniquement **yield from**

```
def my_coroutine(future):  
    result = yield from future  
    return result
```


BaseTask



```
class BaseTask:
    def __init__(self, coro):
        self.coro = coro

    def step(self):
        try:
            next(self.coro)
        except StopIteration:
            pass
```

BaseTask



Code

```
coro = test_coro()  
task = BaseTask(coro)
```

test_coro() coroutine



```
print("begin")  
yield from ["hack"]  
print("end")
```

Output




BaseTask



Code

```
coro = test_coro()  
task = BaseTask(coro)  
task.step()
```

test_coro() coroutine



```
print("begin")  
yield from ["hack"]  
print("end")
```

Output

```
begin
```


BaseTask



Code

```
coro = test_coro()
task = BaseTask(coro)
task.step()
task.step()
```

test_coro() coroutine

```
print("begin")
yield from ["hack"]
print("end")
```



Output

```
begin
end
```

Future et tâche



- Future sert à stocker un résultat futur
- Future permet de déclarer le flot d'exécution
- Task exécute une coroutine dans la boucle d'événements

Future



```
class Future:
```

```
    def __init__(self, loop):  
        self.loop = loop  
        self._result = None  
        self.callbacks = []
```

```
    def result(self):  
        return self._result
```

```
    def add_done_callback(self, func):  
        self.callbacks.append(func)
```

```
    ...
```


Future



```
class Future:
    ...
    def set_result(self, result):
        self._result = result

        for func in self.callbacks:
            self.loop.call_soon(func)

    def __iter__(self):
        # used by "yield from future"
        yield self
```


Tâche



```
class Task:
```

```
    def __init__(self, coro, loop):  
        self.coro = coro  
        loop.call_soon(self.step)  
    ...
```

Tâche



```
class Task:
    ...
    def step(self):
        try:
            result = next(self.coro)
        except StopIteration:
            return

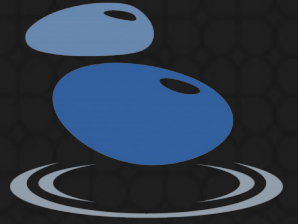
        if isinstance(result, Future):
            result.add_done_callback(self.step)
```

Pause



```
def sleep(delay, loop):  
    fut = Future(loop)  
    cb = functools.partial(fut.set_result,  
                           None)  
    loop.call_later(delay, cb)  
    yield from fut  
  
def slow_print(loop):  
    print("Hello")  
    yield from sleep(1.0)  
    print("World")
```


Pause



slow_print() coroutine

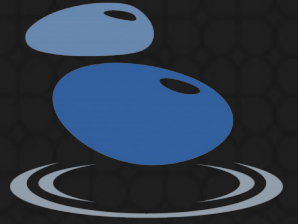
→ print("Hello")
yield from sleep(1.0, loop)
print("World")

Callbacks

slow_print.step

Timers

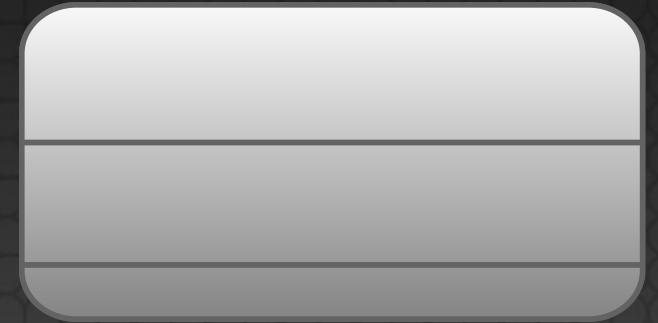
Pause



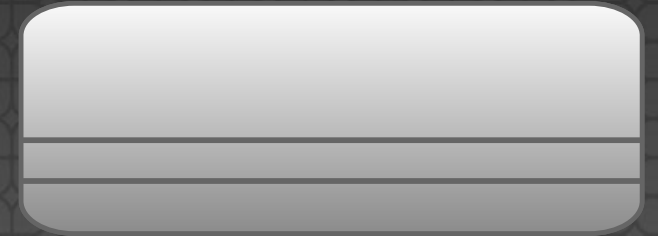
slow_print() coroutine

→ print("Hello")
yield from sleep(1.0, loop)
print("World")

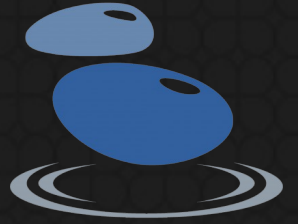
Callbacks



Timers



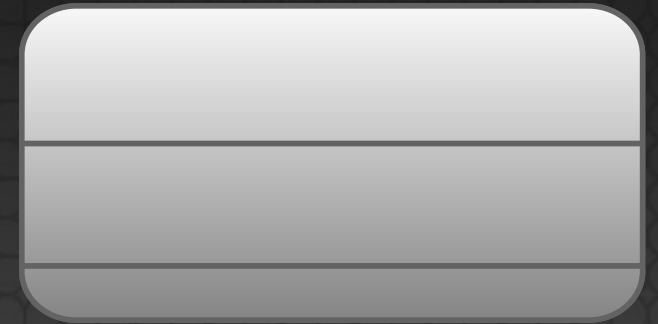
Pause



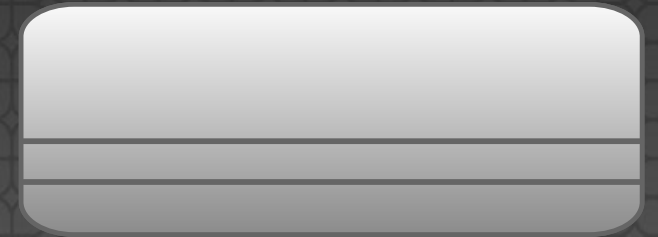
slow_print() coroutine

```
print("Hello")  
→ yield from sleep(1.0, loop)  
print("World")
```

Callbacks



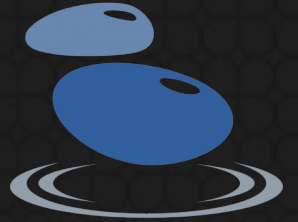
Timers



sleep() coroutine

```
→ f = Future()  
loop.call_later(1.0, f.set_result)  
yield from f
```

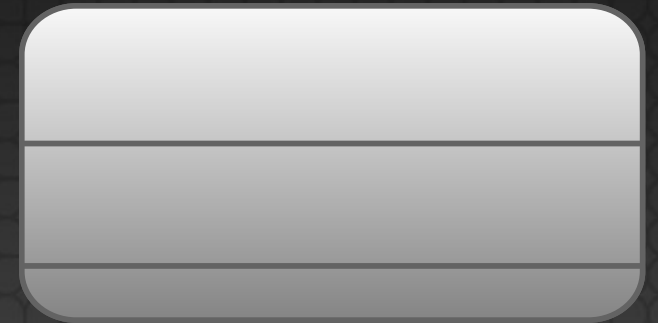
Pause



slow_print() coroutine

```
print("Hello")  
→ yield from sleep(1.0, loop)  
print("World")
```

Callbacks

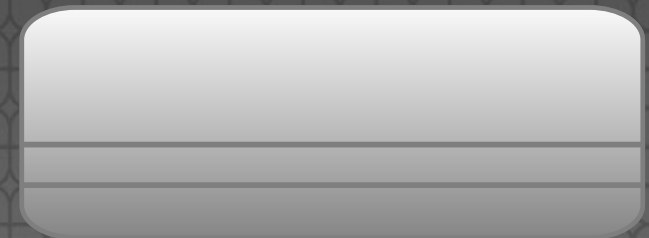


Timers

(1, f.set_result)



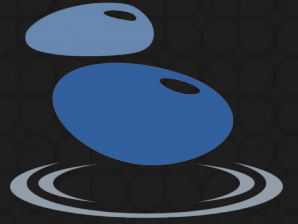
f callbacks



sleep() coroutine

```
f = Future()  
loop.call_later(1.0, f.set_result)  
→ yield from f
```

Pause



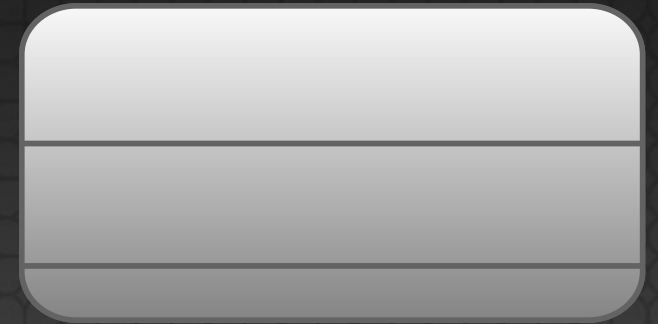
slow_print() coroutine

```
print("Hello")  
→ yield from sleep(1.0, loop)  
print("World")
```

sleep() coroutine

```
f = Future()  
loop.call_later(1.0, f.set_result)  
⇒ yield from f
```

Callbacks

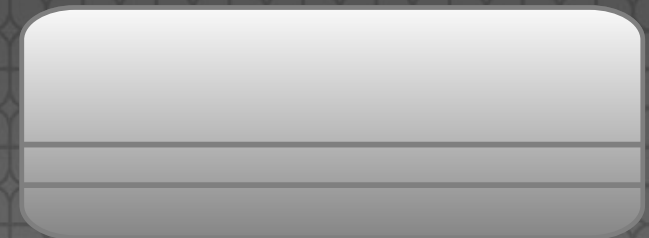


Timers

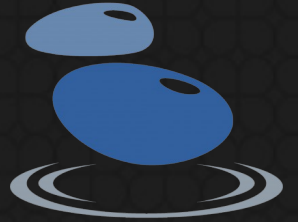
(1, f.set_result)



f callbacks



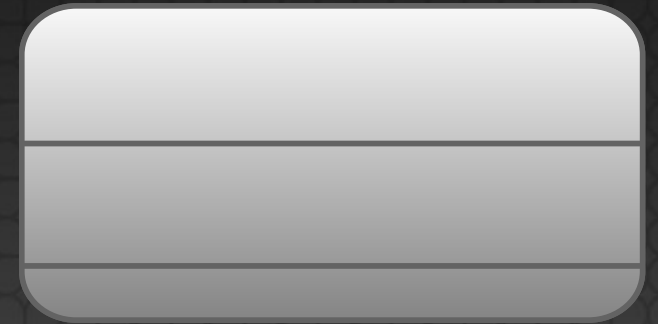
Pause



slow_print() coroutine

```
print("Hello")  
=> yield from sleep(1.0, loop)  
print("World")
```

Callbacks



Timers

(1, f.set_result)



f callbacks

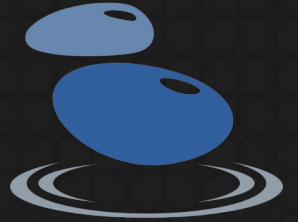
slow_print.step



sleep() coroutine

```
f = Future()  
loop.call_later(1.0, f.set_result)  
=> yield from f
```

Pause



slow_print() coroutine

```
print("Hello")  
=> yield from sleep(1.0, loop)  
print("World")
```

sleep() coroutine

```
f = Future()  
loop.call_later(1.0, f.set_result)  
=> yield from f
```

Callbacks

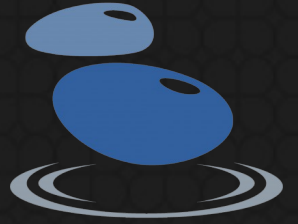
f.set_result

Timers

f callbacks

slow_print.step

Pause



slow_print() coroutine

```
print("Hello")  
=> yield from sleep(1.0, loop)  
print("World")
```

Callbacks

slow_print.step

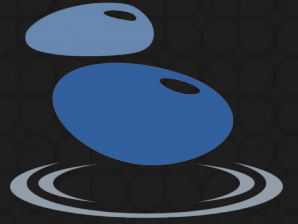
Timers

sleep() coroutine

```
f = Future()  
loop.call_later(1.0, f.set_result)  
=> yield from f
```

f callbacks

Pause



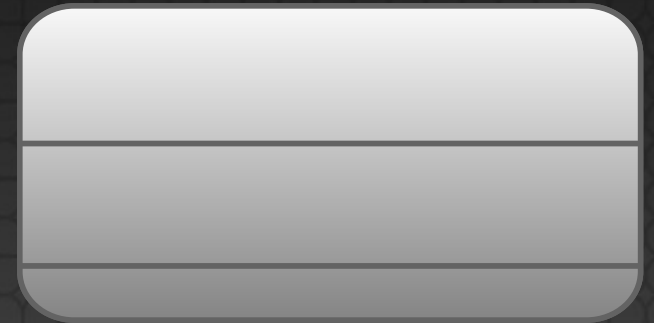
slow_print() coroutine

```
print("Hello")  
→ yield from sleep(1.0, loop)  
print("World")
```

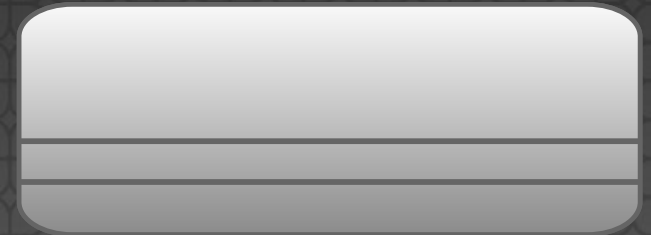
sleep() coroutine

```
f = Future()  
loop.call_later(1.0, f.set_result)  
⇒ yield from f
```

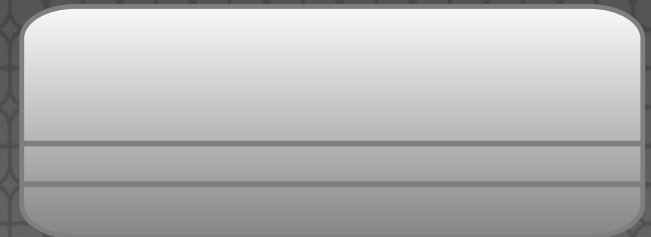
Callbacks



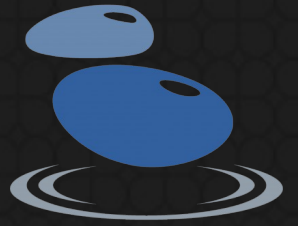
Timers



f callbacks



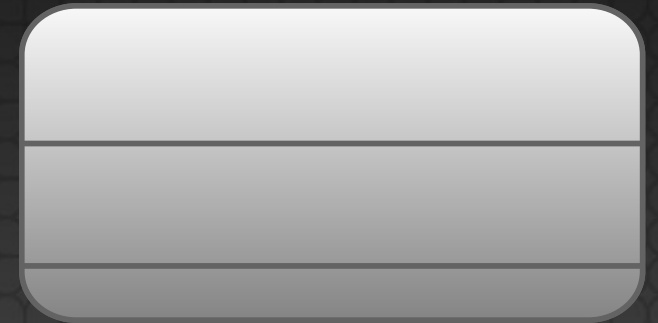
Pause



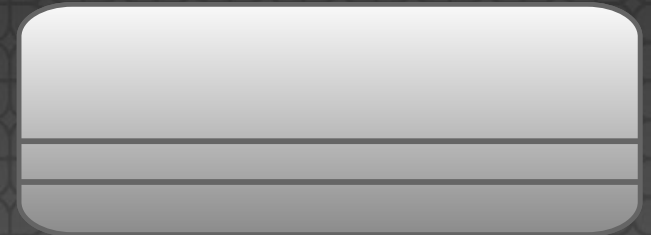
slow_print() coroutine

```
print("Hello")  
→ yield from sleep(1.0, loop)  
print("World")
```

Callbacks



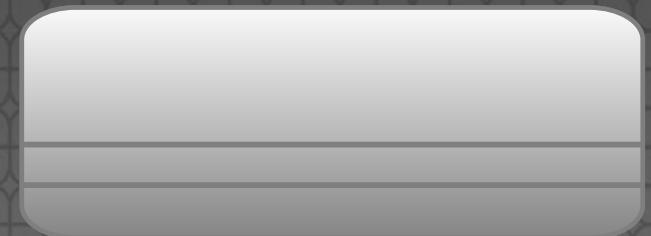
Timers



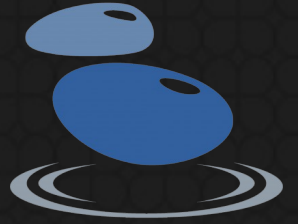
sleep() coroutine

```
f = Future()  
loop.call_later(1.0, f.set_result)  
→ yield from f
```

f callbacks



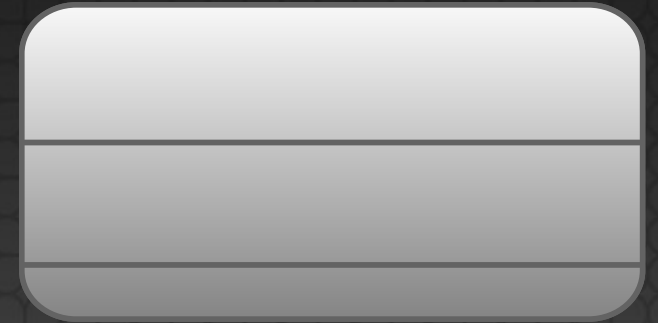
Pause



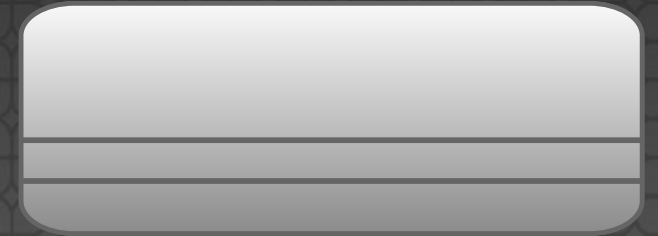
slow_print() coroutine

```
print("Hello")  
→ yield from sleep(1.0, loop)  
print("World")
```

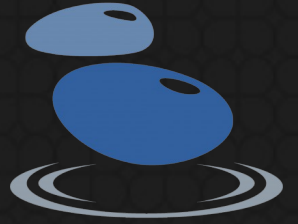
Callbacks



Timers



Pause

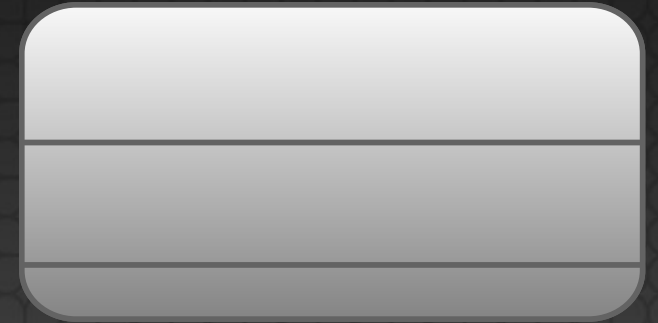


slow_print() coroutine

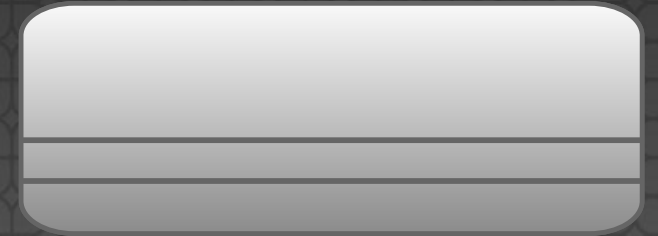
```
print("Hello")  
yield from sleep(1.0, loop)  
print("World")
```



Callbacks



Timers



Résumé



- Fonctions de rappel, Future
- Minuterries
- Multiplexeur E/S
- Tâche, coroutine

Questions ?

<http://docs.python.org/dev/library/asyncio.html>

<http://github.com/haypo/>
<http://bitbucket.org/haypo/>

Victor Stinner
victor.stinner@gmail.com

Merci à David Malcom
pour le modèle LibreOffice

<http://dmalcolm.livejournal.com/>