Python classes and cocotb concurrency

This page answers: *a testbench has to drive, watch and check all at once, and hand packets from one piece to the next. What do I need to know about Python objects and cocotb tasks before I write a monitor and a scoreboard?*

In Lab 02 you'll write the router's monitor and scoreboard. They're plain Python classes, and the testbench they go into has no pyuvm in it. The rest of that testbench comes with the lab: a generator that makes packets, a driver that offers them to the router's input, a sink for each output, and an environment that makes all of them and runs the test. When the test runs, seven pieces run at the same time and pass packets to each other through three queues. They're the generator, the driver, the two sinks, the scoreboard, and two monitors made from your class, one on the router's input and one on output 0. The first half of this page is about the objects those pieces pass around, and the second half is about running the pieces side by side as cocotb tasks.

We ran every example on this page in the course image. The source is in `guide-examples/python-classes-and-cocotb-concurrency/`, and the section for this page in `guide-examples/README.md` gives the command behind each output.

Names and objects

The examples use a cut-down version of Lab 02's `Packet`, with a serial number added:

```
class Packet:
    n_made = 0                    # a class attribute: one for the class

    def __init__(self, data=None):
        self.id = Packet.n_made   # instance attributes: one set per object
        Packet.n_made += 1
        self.data = list(data or [])

    def __str__(self):
        return f"pkt {self.id}: " + " ".join(f"{b:02x}" for b in self.data)
```

Each call of `Packet(...)` makes a new object and runs `__init__` on it. Lab 02's generator makes a new `Packet` for every packet it sends, and your monitor will make one for every packet it sees on its stream. A Python variable is a name that refers to an object, and assigning one name to another gives the same object a second name. `handles_test` does that, and then makes two copies of the packet, one with `copy.copy()` and one with `copy.deepcopy()`:

```
a = None
print(f"a is {a}")
a = Packet([0x80, 0x11])
b = a
b.data.append(0x22)
c = copy.copy(a)
c.data.append(0x33)
d = copy.deepcopy(a)
d.data.append(0x44)
```

It then prints the four packets, four `is` comparisons and the count in `Packet.n_made`:

```
a is None
a: pkt 0: 80 11 22 33
```

```
b: pkt 0: 80 11 22 33
c: pkt 0: 80 11 22 33
d: pkt 0: 80 11 22 33 44
a is b: True, a is c: False, a.data is c.data: True, a.data is d.data: False, Packet.n_made: 1
```

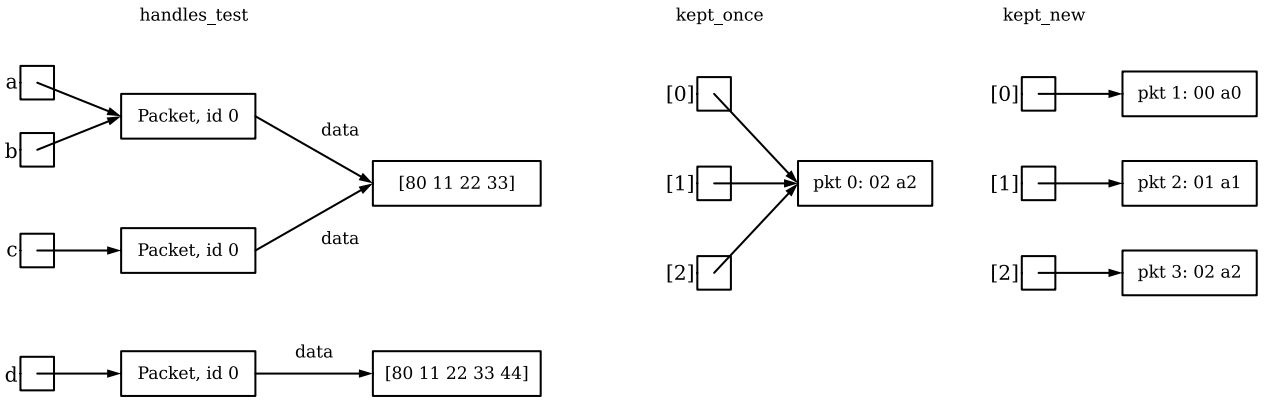


Figure 1: The objects that handles_test and reuse_test leave behind

Left, `handles_test`: `a` and `b` name one `Packet`; `c` is a second `Packet` that shares the first one's list; `d` is a third `Packet`, with a list of its own. Right, `reuse_test`, from the next section: `kept_once` holds three references to one reused `Packet`, and `kept_new` holds three `Packet`s. The values are the ones the tests printed.

`b = a` didn't make a packet, so the `22` appended through `b` shows up when you print `a`. `a is b` is `True`, because `is` asks whether two names refer to the same object. Two packets made separately are two objects even when their beats match, so `is` can't tell you whether two packets carry the same data. That's why Lab 02's `Packet` has a `same()` method, which compares the beats.

`copy.copy(a)` made a *shallow* copy. It made a new `Packet` and copied each of `a`'s attributes into it, but where an attribute refers to a list, it copied the reference and not the list. So `c.data` is `a.data`, and the `33` appended through `c` changed `a` too. `copy.deepcopy(a)` made a *deep* copy, which copies the list as well, so the `44` went into `d` alone. Neither copy ran `__init__`, so all three packets have `id` 0 and `Packet.n_made` is still 1.

You won't often copy packets in the labs, but you'll pass references to them all the time. In Lab 02, the `Packet` that your input monitor builds is the same object that the scoreboard takes out of the queue and, if it's predicted for output 0, keeps in its list of expected packets. A change made through any of those references shows through all of them.

The same object kept three times

`reuse_test` has two loops, and each one stands in for a monitor. Each loop builds the same three packets, two bytes each, and keeps them in a list, as Lab 02's scoreboard keeps the packets it expects. The first loop makes one `Packet` before it starts and reuses it for every packet, and the second makes a new `Packet` for each one:

```
p = Packet()
for k in range(3):
    p.data.clear()
    p.data.append(k)
    p.data.append(0xA0 + k)
    kept_once.append(p)
```

```
for k in range(3):
    p = Packet()
    p.data.append(k)
    p.data.append(0xA0 + k)
    kept_new.append(p)
```

```
one Packet, reused: [0] pkt 0: 02 a2
one Packet, reused: [1] pkt 0: 02 a2
one Packet, reused: [2] pkt 0: 02 a2
a new Packet each:  [0] pkt 1: 00 a0
a new Packet each:  [1] pkt 2: 01 a1
a new Packet each:  [2] pkt 3: 02 a2
```

`kept_once` holds three references to one object. That object holds the beats of the last packet, `02 a2`, so the list shows them three times, and the first two packets' beats are gone, cleared by `p.data.clear()` when the loop reused the object. A queue behaves the same way, because what you put into it is a reference. That's why the `TODO` in Lab 02's `monitor.py` has you start a new packet when `p` is `None`, and set `p = None` once you've put a finished packet into the queue. The next packet then gets an object of its own, and the object that the scoreboard was given keeps the beats it had.

None

If you use an attribute of a name that still refers to `None`, Python raises an `AttributeError`, and the test stops there. `none_test` sets `p = None`, prints a line, and then appends a beat to `p.data`:

```
appending a beat to p
    0.00ns WARNING  cocotb.Test none_test.none_test   'NoneType' object has no attribute 'data'
                                                         Traceback (most recent call last):
                                                           File "/w/guide-examples/python-
classes-and-cocotb-concurrency/classes_demo.py", line 78, in none_test
                                                             p.data.append(0x80)
                                                             ^^^^^^
                                                       AttributeError: 'NoneType' object has
no attribute 'data'
```

and the course's verdict on the run is

```
 BROKEN: AttributeError   0.23 s   log: build/runs/none_test-s1/run.log
```

The traceback names the file and the line where the `None` was used. The graders mark a run like this `BROKEN`, because the exception is a mistake in the testbench, and a run counts as catching a bug only when a check fires, such as a failed `assert`. In Lab 02 you'll see this error if your monitor appends a beat to `p.data` before it has made `p`.

Inheritance and super()

A class can derive from another class, which is then its base class. It inherits the base class's methods, `__init__` among them, and it can override any of them with a version of its own. In Python every method call is looked up on the object's own class when the call happens, so when a method written in the base class calls another method,

the call goes to the derived class's version if there is one. In `dispatch_test`, `Component` has a `report()` method that calls `self.kind()`, and `Scoreboard` derives from `Component` and overrides `kind()`:

```
class Component:
    def __init__(self, name):
        self.name = name
        self.n_errors = 0

    def kind(self):
        return "a Component"

    def report(self):
       print(f"report() of Component, on {self.name}: {self.kind()}, {self.n_errors} errors")


class Scoreboard(Component):
    def __init__(self, name):
        super().__init__(name)
        self.expected = []

    def kind(self):
        return "a Scoreboard"
```

`Component("c").report()` and `Scoreboard("sb").report()` both ran `Component`'s `report()`, and the call to `self.kind()` inside it went to each object's own class:

```
report() of Component, on c: a Component, 0 errors
report() of Component, on sb: a Scoreboard, 0 errors
```

pyuvm relies on the same lookup. It calls `build_phase()` and `run_phase()` on every component, and when the component is Lab 01's `AluDriver`, those calls run the versions written in `AluDriver`.

`Scoreboard.__init__` starts with `super().__init__(name)`, which runs `Component`'s `__init__` and sets `name` and `n_errors`. If you leave that line out, Python raises nothing when the object is made. The error comes later, when something first uses an attribute that `Component.__init__` would have set. `super_test` makes a `ForgetfulScoreboard`. It derives from `Component`, as `Scoreboard` does, and its `__init__` is `Scoreboard`'s without the `super()` line. The test prints the object's attributes and then calls `report()`:

```
made a ForgetfulScoreboard; its attributes: ['expected']
     0.00ns WARNING  cocotb.Test super_test.super_test  'ForgetfulScoreboard' object has no
attribute 'name'
```

The traceback in the log points at the `print` in `report()`, not at the constructor that left the attribute out. The same rule applies in pyuvm to any method you override when the base class's version does work of its own. `RouterBaseTest.build_phase()` puts the router in `ConfigDB`, and `BaseTest.final_phase()` prints the closing line that the graders look for and fails the test if any component logged an error, so a test that overrides either one has to call the base version through `super()`.

Class attributes

An attribute assigned in the class body belongs to the class, so every object shares it. `Packet.n_made` is a class attribute on purpose, because it counts the packets made from the class. A list made in the class body is shared the same way, which is almost never what you meant. `class_attr_test` uses two classes. `Sink` has `beats = []` in its class body, and `OwnSink` sets `self.beats = []` in `__init__`. The test makes two sinks, `out0` and `out1`, from each class, appends `0x11` to `out0`'s list and `0x22` to `out1`'s, and prints both lists:

```
Sink: out0 took ['0x11', '0x22'], out1 took ['0x11', '0x22']
OwnSink: out0 took ['0x11'], out1 took ['0x22']
```

Lab 02's classes make every list and counter in `__init__`, and yours should too.

Tasks

A coroutine is an `async def` function, and `await coro()` runs it inside your own task until it returns. To run a coroutine alongside your own code instead, you pass `coro()` to `cocotb.start_soon()`, which wraps it in a **task** and returns the `Task` object. `start_soon()` returns straight away, and the task first runs when the code that started it reaches its next `await`. You can wait for a task later, or never.

`start_test` starts three tasks, A, B and C, which finish 10, 20 and 30 ns after they start, and it does this three times. The first time it waits for all three with `await Combine(a, b, c)`, the second time it waits for the first one to finish with `await First(a, b, c)`, and the third time it waits for none of them:

```
 0 ns: start A, B and C, and await Combine(a, b, c)
10 ns:   A finished
20 ns:   B finished
30 ns:   C finished
30 ns: the test goes on after Combine
30 ns: start A, B and C, and await First(a, b, c)
40 ns:   A finished
40 ns: the test goes on after First
50 ns:   B finished
60 ns:   C finished
60 ns: Combine returns
60 ns: start A, B and C, and await none of them
60 ns: the test goes on at once
70 ns:   A finished
80 ns:   B finished
90 ns:   C finished
```

- `Combine` waits for every task, so the test went on at 30 ns, when the slowest one finished.
- `First` waits for the first task to finish, so the test went on at 40 ns, when A finished. `First` leaves the other two running, and B and C finished at 50 and 60 ns. The test then waited for them with `Combine` before it went on.
- When the test waited for none of them, it went on at once, at 60 ns, and the three tasks ran beside it.

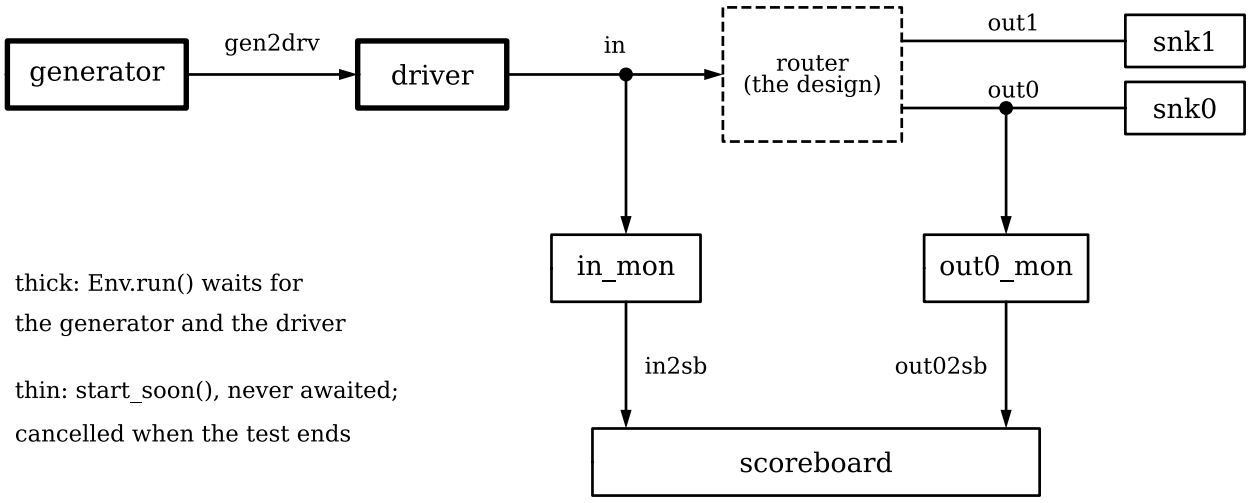


Figure 2: Lab 02's environment as tasks and queues

From Lab 02's `env.py`. Each solid box is an object whose `run()` method runs as a task, except the driver, whose `run()` runs inside the test's own task. The dashed box is the router. The arrows labelled `gen2drv`, `in2sb` and `out02sb` are the three queues, and `in`, `out0` and `out1` are the router's streams.

Lab 02's `Env.run()` starts the two sinks, the two monitors and the scoreboard with `cocotb.start_soon()` and never waits for them, because none of their `run()` methods ever returns. Each sink, and each monitor once you've written it, runs a `while True` loop, and the scoreboard's `run()` waits for two such loops, which it starts as two more tasks. `Env.run()` starts the generator with `cocotb.start_soon()` too, and then runs the driver itself, with `await self.drv.run(self.cfg.n)`, so the driver and the generator work at the same time. When the driver has sent every packet, `await gen` waits for the generator's task, which by then has finished. Then the environment waits 50 cycles for the outputs to drain and calls the scoreboard's `report()`, while the sinks, the monitors and the scoreboard are still running.

Tasks started in a loop

`capture_test` starts three tasks in a loop, and then three more in a second loop. In the first loop each task is a coroutine defined inside the loop, which reads the loop variable `i`. In the second, each task is a coroutine that is given `i` as an argument:

```
for i in range(3):
    async def show():
        say(f"  i = {i}")
    cocotb.start_soon(show())
...
for i in range(3):
    cocotb.start_soon(show_arg(i))
```

```
  0 ns: three tasks, each a closure that reads i
  0 ns: the loop has finished
  0 ns:   i = 2
  0 ns:   i = 2
  0 ns:   i = 2
  5 ns: three tasks, each given i as an argument
```

```
 5 ns: the loop has finished
 5 ns:   j = 0
 5 ns:   j = 1
 5 ns:   j = 2
```

None of the six tasks ran before its loop had finished. The first three read `i` when they ran, by which time the loop had left it at 2. A function defined inside another one reads the outer function's variables when it runs, not when it's defined. The second three were each given the value of `i` at the moment `show_arg(i)` was called, so they printed 0, 1 and 2. Give a task what it needs as arguments. Lab 02′s environment starts its tasks in a loop too, with `cocotb.start_soon(piece.run())`, and that's safe, because `piece.run()` is called inside the loop, while `piece` still names that pass's object, and the coroutine it returns keeps that object as its `self`.

Queues

A `cocotb.queue.Queue` carries values from tasks that `put()` them to tasks that `get()` them, in the order they were put. `await q.get()` waits while the queue is empty, and if you gave the queue a `maxsize`, `await q.put()` waits while it's full. `queue_test` has a generator that makes four items as fast as it can, and a driver that spends 10 ns on each item. It runs them twice, first with an unbounded queue between them and then with a queue that has room for one item:

```
 0 ns: Queue(maxsize=0)
 0 ns: gen put 0
 0 ns: gen put 1
 0 ns: gen put 2
 0 ns: gen put 3
 0 ns: gen has made all four
 0 ns:   drv got 0
10 ns:   drv got 1
20 ns:   drv got 2
30 ns:   drv got 3
40 ns: Queue(maxsize=1)
40 ns: gen put 0
40 ns:   drv got 0
40 ns: gen put 1
50 ns:   drv got 1
50 ns: gen put 2
60 ns:   drv got 2
60 ns: gen put 3
60 ns: gen has made all four
70 ns:   drv got 3
```

`maxsize=0`, the default, means no limit. With it, the generator made all four items at once, and the driver worked through them, one every 10 ns. With room for one, each `put()` waited until the driver had taken the item before it, so the generator kept pace with the driver and finished 20 ns after it started. The driver got the same items, 10 ns apart, both ways.

Lab 02′s queues are all unbounded (`Queue()`), so its generator makes every packet at the start and the driver works through them. The monitors' queues are the other way round, with the reader waiting for the writer. A monitor puts a packet into its queue each time one finishes on its stream, and the scoreboard's `get()` waits until the next one arrives. The scoreboard reads two of these queues, `from_in` and `from_out0`, and each fills at its own pace. A

task waiting in `from_out0.get()` can't take a packet from `from_in` at the same time, so the scoreboard's `run()`, which you write in Part 3, starts two tasks of its own. One predicts each input packet as it arrives, and the other checks each packet from output 0 against the predictions made so far.

Events

A `cocotb.triggers.Event` lets one task tell others that something has happened. `e.set()` sets it, and every task waiting in `await e.wait()` goes on. The event stays set until something calls `e.clear()`, so a task that calls `e.wait()` after the `set()` goes straight on. `event_test` starts a waiter, sets the event at 10 ns and starts a second waiter at the same time, then clears the event at 20 ns and starts a third:

```
 0 ns:   first: waiting
10 ns: done.set()
10 ns:   first: done.wait() returned
10 ns:   second: waiting
10 ns:   second: done.wait() returned
20 ns: done.clear()
20 ns:   third: waiting
30 ns: end of the test
```

The third waiter never returned, because nothing set the event again. Lab 02 has no events. You'll meet one in Lab 04, whose scoreboard sets an `Event` called `stop` at its 50th error. Lab 04's tests wait on that event, so the run ends there. pyuvm also uses an event to end the run phase, and sets it when the last objection is dropped.

Stopping tasks

`t.cancel()` stops a task. cocotb raises `CancelledError` inside the task, at the `await` where it's waiting, and unless the task catches the exception, the task ends there. `cancel_test` starts a monitor task that counts a tick every 10 ns, and a parent task that counts its own ticks and has started a child task that counts too. At 25 ns the test prints the three counts and cancels the monitor and the parent, and 50 ns later it prints the counts again:

```
25 ns: cancel the monitor and the parent: {'mon': 2, 'parent': 2, 'child': 2}
75 ns: 50 ns later: {'mon': 2, 'parent': 2, 'child': 7}
75 ns: mon.cancelled() is True, par.cancelled() is True
```

The monitor and the parent stopped at 2, and the child went on counting. cocotb doesn't keep track of which task started which, so cancelling a task leaves the tasks it started running. To stop those too, keep their `Task` objects and cancel each one.

You don't have to stop the tasks that run for the whole test, because when a test ends, cocotb cancels every task that is still running. `end_test` starts a monitor that catches `CancelledError`, prints a line and raises the exception again, and the test returns 35 ns later:

```
35 ns: the test returns
35 ns:   the monitor was cancelled
```

Lab 02's sinks, monitors and scoreboard are stopped this way. They're still running when `Env.run()` returns and the test makes its `assert`, and cocotb cancels them when the test function returns.

An exception in a task

A mistake in a task, such as a misspelt name, raises an exception in that task, and what happens next depends on whether anything is waiting for the task. The example's `broken_monitor()` waits 20 ns and then appends to a list called `pkts`, which doesn't exist. Four tests start it as a task:

Test	What it does after starting the task	Verdict
`unwatched_test`	waits 100 ns and never looks at the task	`BROKEN: NameError`, at 20 ns
`joined_test`	`await t`	`BROKEN: NameError`, at 20 ns
`first_test`	`await First(t.complete, Timer(100, unit="ns"))`	`ok: passed`, at 20 ns
`first_result_test`	the same, then `t.result()`	`BROKEN: NameError`, at 20 ns

When nothing waits for a task, cocotb fails the test with the task's exception as soon as it happens:

```
    20.00ns WARNING  cocotb.Task 2.broken_monitor      name 'pkts' is not defined
```

`await t` raises the task's exception in the task that waits for it, so `joined_test` failed the same way. `first_test` passed. While `First` waits on the task's `complete` trigger, cocotb counts the task as watched and leaves its exception to the code that is waiting. `First` returns once the task has finished, and nothing raises the exception again:

```
20 ns: First returned; t.done() is True
20 ns: the test reached its end
```

The test passed even though its monitor had crashed at 20 ns. A monitor that has crashed sends nothing more to the scoreboard, so a test like this can pass on a broken design, unless the scoreboard checks at the end that every packet it expected arrived, as Lab 02's `report()` does.

If you ever wait on `t.complete`, call `t.result()` once the task is done. `t.result()` raises the task's exception if it had one, as `first_result_test` shows. Lab 04's tests all derive from `RouterLabTest`, which does this. Its `run_phase()` waits with `First` for its stimulus task, a watchdog task or the scoreboard's `stop` event. Then, for each of the two tasks, it calls `result()` if the task has finished and `cancel()` if it hasn't. Lab 02's environment waits for only one of its tasks, with `await gen`, and starts the rest without waiting for them, so an exception in your monitor or your scoreboard fails the test as soon as it happens. The verdict is `BROKEN`, and the traceback is in the log.

Check yourself

1. Your scoreboard reports that every packet it expected has the same beats as the last packet that the input monitor saw. The monitor does `p = Packet()` once, in its `__init__`. What's wrong, and how do you fix it?

   Answer: A1, in Answers at the end.

2. A test starts eight tasks in a loop, `for k in range(8)`. Each task is a coroutine, defined inside the loop, that logs `k`. Every task logs 7. Why, and what's the fix?

   Answer: A2, in Answers at the end.

3. What would happen if `Env.run()` started the scoreboard with `await self.sb.run()` instead of `cocotb.start_soon(self.sb.run())`?

Answer: A3, in Answers at the end.

4. Your scoreboard's output loop has a typo that raises `NameError` at the first packet from output 0. What does the verdict say in Lab 02, and what would it say if `Env.run()` had waited for the scoreboard's task with `First(task.complete, ...)`?

   Answer: A4, in Answers at the end.

Further reading

- The Python tutorial's chapter on classes. Its section on class and instance variables shows the shared-list mistake that `class_attr_test` makes.
- The `copy` module: shallow and deep copies, and how a class can define its own.
- The Python FAQ's answer to why lambdas defined in a loop all return the same result, which is `capture_test`'s first loop with a `lambda` in place of a coroutine.
- cocotb 2.0.1's Coroutines and Tasks: `start_soon`, `First` and `Combine`, and waiting for several things at once. It says that `First` doesn't cancel the tasks that hadn't finished, as `start_test` shows. It doesn't say what happens to tasks still running when a test ends, or to an exception in a task that nothing waits for; `end_test` and the four exception tests on this page show what cocotb 2.0.1 does.

Next: Driving and sampling on the clock.

Answers

A1. Check yourself

The monitor puts the same object into the queue for every packet and refills it each time, so every reference that the scoreboard kept refers to one object, and that object holds whatever was written into it last. Make a new `Packet` for each packet the monitor sees, and stop using the old one once it's in the queue, which is what setting `p = None` after the `put()` does.

A2. Check yourself

Each coroutine reads `k` when it runs, and none of the tasks runs until the test reaches its next `await`. By then the loop has finished and left `k` at 7. Pass `k` to the coroutine as an argument, as `capture_test`'s second loop passes `i` to `show_arg()`, so that each task gets the value that `k` had when the task was started.

A3. Check yourself

The scoreboard's `run()`, once you've written it in Part 3, waits for its two loops, and they never return, so `Env.run()` would wait there for ever. It would never start the generator or the driver, so no packet would be sent, and the run would go on until `make` stopped it after `RUN_TIMEOUT`, 300 s by default, and graded it `BROKEN`.

A4. Check yourself

In Lab 02 the exception fails the test as soon as it happens, and the verdict is `BROKEN: NameError`, with the traceback in the log. With `First(task.complete, ...)` on the scoreboard's task, the verdict is the same if your `run()` awaits the input loop's task before the output loop's. The typo raises in the output loop, which is a separate task, and nothing is waiting on that task when it raises.

If the exception does reach the scoreboard's own task, for example because `run()` waits for both loops with `Combine`, `First` counts that task as watched and the exception is dropped. The test then runs to the end, where `report()` finds every packet predicted for output 0 still expected and no packet checked. Those errors fail the test with an `AssertionError`, which the verdict counts as a check that fired, so the golden router's run says `FAIL` instead of `BROKEN`. Calling `task.result()` after `First` returns would raise the `NameError` and make the verdict `BROKEN` again.