Discrete-event simulation in Python

Rémi Varloot (Nokia Bell Labs)

Tools, Tips and Tricks @ LINCS

7 march, 2025

Discrete-event simulation

- Simulation is a sequence of atomic events
- Each event is scheduled to occur at a given time
- Events can schedule future events

Basic implementation

In [ ]:
```python
from queue import PriorityQueue

priority_queue = PriorityQueue()
counter = 0
time = 0

def reset():
    global priority_queue, counter, time
    priority_queue = PriorityQueue()
    counter = 0
    time = 0

def schedule(when, what):
    global counter
    counter += 1
    priority_queue.put((when, counter, what))

def run(stop=None):
    global priority_queue, time
    while not priority_queue.empty():
        t, _, func = priority_queue.get()

        if stop is not None and t >= stop:
            reset()
            return
        else:
            time = t

        func()
```

In action

In [ ]:
```python
schedule(1, lambda: print(time, 'a'))
schedule(3, lambda: print(time, 'b'))
schedule(2, lambda: print(time, 'c'))
```

```
schedule(1, lambda: print(time, 'd'))

run()
```

Recurring events

In [ ]:
```
def print_time_loop():
    print(time)
    schedule(time+1, print_time_loop)

schedule(0, print_time_loop)

run(stop=4)
```

We can achieve the same effect more conveniently using decorators.

Decorator basics

In [ ]:
```
from functools import wraps

def log(func):

    @wraps(func)    # makes `wrapper` look like `func`
    def wrapper(*args, **kwds):
        print(f'running {func.__name__} with args {args} and kwds {kwds}')
        res = func(*args, **kwds)
        print(f'{func.__name__} returned: {res}')
        return res

    return wrapper

@log                # Equivalent to `double = log(double)`
def double(x):
    return 2 * x

double(3)
```

With parameters

In [ ]:
```
def run_x_times(x):

    def decorator(func):

        @wraps(func)
        def wrapper(*args, **kwds):
            for _ in range(x):
                func(*args, **kwds)

        return wrapper

    return decorator

@run_x_times(3)
def print_hello(name='World'):
    print(f'Hello, {name}!')

print_hello('Alice')
```

Decorators and simulation

In [ ]:
```
from functools import wraps

def loop(*, start=0, every): # parameters after the `*` are keyword-only paramet

    def decorator(func):

        @wraps(func)
        def wrapper():
            func()
            next_call = time + every
            schedule(next_call, wrapper)

        schedule(start, wrapper)

        return wrapper

    return decorator

# Could have been `loop(every=1)(print)`
@loop(every=1)
def print_time_loop():
    print(time)

run(stop=4)
```

Example: simple random walk on {0, 1, 2, ..., 10}

In [ ]:
```
from random import randint

position = 0
bins = [0 for _ in range(11)]

@loop(every=1)
def step():
    global position
    if randint(0, 1) == 0:  # left
```

```
        position = max(0, position - 1)
    else:                    # right
        position = min(10, position + 1)
    bins[position] += 1

run(stop=1000)

print(bins)
```

Separating the random walk and the sampling

In [ ]:
```
position = 0
bins = [0 for _ in range(11)]

@loop(every=1)
def step():
    global position
    if randint(0, 1) == 0: # left
        position = max(0, position - 1)
    else: # right
        position = min(10, position + 1)

@loop(every=10, start=100)
def measure():
    bins[position] += 1

run(stop=1000)

print(bins)
```

Example: ping-pong

In [ ]:
```
def alice():
    print(time, 'ping')
    schedule(time+1, bob)

def bob():
    print(time, 'pong')
    schedule(time+1, alice)

schedule(0, alice)

run(stop=10)
```

Asynchronous operations

Example: downloading chunks from peers

In [ ]:
```
from random import random, choice, shuffle

N = 10

class Client:
    def __init__(self, index, initial_chunk):
        self.index = index
        self.missing_chunks = [i for i in range(N) if i != initial_chunk]
```

```
        self.current_query = None
        self.not_asked = None

    def download_missing_chunk(self):
        if self.current_query is None:
            self.current_query = choice(self.missing_chunks)
            self.not_asked = [i for i in range(N) if i != self.index]

        peer = choice(self.not_asked)
        schedule(time + random(), lambda: clients[peer].on_query(self.index, sel

    def on_query(self, peer, chunk):
        if chunk in self.missing_chunks:
            schedule(time + random(), lambda: clients[peer].on_reply(self.index,
        else:
            schedule(time + random() + random(), lambda: clients[peer].on_reply(

    def on_reply(self, peer, chunk):
        if chunk != None:
            self.missing_chunks.remove(chunk)
            self.current_query = None

            print(f'{self.index} got {chunk} from {peer}')
            if len(self.missing_chunks) == 0:
                print(f'{self.index} finished')
                return

        else:
            self.not_asked.remove(peer)

        schedule(time + random(), self.download_missing_chunk)

initial_chunks = [i for i in range(N)]
shuffle(initial_chunks)

clients = [Client(i, initial_chunks[i]) for i in range(N)]

for c in clients:
    schedule(0, c.download_missing_chunk)

run()

print(time)
```

What we want

In [ ]:

```
class Client:
    def __init__(self, index, initial_chunk):
        self.index = index
        self.missing = [i for i in range(N) if i != initial_chunk]

    def download_missing_chunk(self):
        missing = self.missing[:]
        shuffle(missing)

        for chunk in missing:
            peers = [clients[i] for i in range(N) if i != self.index]
            shuffle(peers)
```

```
            for peer in peers:
#               ??????????????????????????????????????????????????????????????
                success = schedule(time+random(), lambda: peer.on_query(chunk))
#               ??????????????????????????????????????????????????????????????

                if success is not None:
                    self.missing.remove(chunk)
                    break

    def on_query(self, chunk):
        if chunk in self.missing:
#           ????????????????????????????????????
            return after_some_time(time + random(), None)
#           ????????????????????????????????????
        else:
#           ??????????????????????????????????????????????????
            return after_some_time(time + random() + random(), chunk)
#           ??????????????????????????????????????????????????
```

A likely solution

(we could be tempted to use `async` / `await`*, but this is impracticle due to how asyncio is conceived)*

In [ ]:
```
class Client:
    def __init__(self, index, initial_chunk):
        self.index = index
        self.missing = [i for i in range(N) if i != initial_chunk]

    def download_missing_chunk(self):
        missing = self.missing[:]
        shuffle(missing)

        for chunk in missing:
            peers = [i for i in range(N) if i != self.index]
            shuffle(peers)

            for peer in peers:
                success = yield schedule(time+random(), lambda: peer.on_query(ch

                if success is not None:
                    self.missing.remove(chunk)
                    break

    def on_query(self, chunk):
        yield wait(random())

        if chunk in self.missing:
            return None
        else:
            yield wait(random())
            return chunk
```

Generator functions

Creating a generator

In [ ]:
```
def generator(n):
    i = 0
    while i < n:
        yield i
        i += 1
```

Basic usage

In [ ]:
```
for i in generator(4):
    print(i)
```

Under the hood

In [ ]:
```
g = generator(4)

while True:
    try:
        i = next(g)
        print(i)
    except StopIteration:
        break
```

With input values

In [ ]:
```
def game(secret):
    guess = yield
    while guess != secret:
        guess = yield f'{'less' if guess > secret else 'more'} than {guess}'
    return secret

g = game(6)

try:
    g.send(None) # equivalent to `next(g)`
    print(g.send(5))
    print(g.send(10))
    print(g.send(7))
    print(g.send(6))
except StopIteration as return_value:
    print('success:', return_value.value)
```

Back to our simulation

- A pending value is a `PendingResponse`, i.e. in `yield schedule(...)`, `schedule` is saying "I'll get back to you with the result"

In [ ]:
```
class PendingResponse:
    def __init__(self):
        self.callback = None

    def on_resolved(self, func):    # Caller sets a callback
        self.callback = func

    def resolve(self, res):         # Called later with the result by the callee
        if self.callback:
```

```
            self.callback(res)
```

- Generators need to be handled properly:
  - We need a handler to "resume" execution when `PendingResponse`s are resolved
  - They must themselves return a `PendingResponse`

In [ ]:
```
class GeneratorHandler(PendingResponse):
    def __init__(self, generator):
        super().__init__()
        self.generator = generator

    def wake(self, arg=None):
        try:
            response = self.generator.send(arg) # If the generator yields a `Pen
            response.on_resolved(self.wake)     # ... tell it to wake up again w
        except StopIteration as return_value:
            self.resolve(return_value.value)    # The generator itself (finally)
```

- `schedule` must now return a `GeneratorHandler`, and enqueue its `wake` function

In [ ]:
```
def schedule(when, what):
    global counter
    counter += 1
    generator = what()
    handler = GeneratorHandler(generator)
    priority_queue.put((when, counter, handler.wake))
    return handler
```

- Let's implement the `wait` function

In [ ]:
```
def no_op_generator():
    return
    yield   # Idiomatic hack to create a generator with no yield statement

def wait(delay):
    return schedule(time+delay, no_op_generator)
```

Let's test

In [ ]:
```
class Client:
    def __init__(self, index, initial_chunk):
        self.index = index
        self.missing = [i for i in range(N) if i != initial_chunk]

    def download_missing_chunk(self):
        missing = self.missing[:]
        shuffle(missing)

        for chunk in missing:
            peers = [clients[i] for i in range(N) if i != self.index]
            shuffle(peers)
```

```
            for peer in peers:
                success = yield schedule(time+random(), lambda: peer.on_query(ch


                if success is not None:
                    print(f'{self.index} got {chunk} from {peer.index}')

                    self.missing.remove(chunk)
                    break

        print(f'{self.index} finished')

    def on_query(self, chunk):
        yield wait(random())
        if chunk in self.missing:
            return None
        else:
            yield wait(random())
            return chunk

reset()

initial_chunks = [i for i in range(N)]
shuffle(initial_chunks)

clients = [Client(i, initial_chunks[i]) for i in range(N)]

for c in clients:
    schedule(0, c.download_missing_chunk)

run()

print(time)
```

Other possible improvements

- Encapsulate `priority_queue` and `counter`, possible using a context manager

In [ ]:
```
with simulator(stop=100) as simu:

    @simu.schedule(delay=10)
    def some_call():
        ...

    @simu.loop(every=1)
    def some_other_call():
        ...

# Simulation runs on close

analyze_results(simu)
```

- Define `Join` and `Race`

In [ ]:
```
response = Join(pending_1, pending_2, pending_3)
@response.on_resolved
def do_stuff(res_1, res_2, res_3):  # When all `PendingResponse`s are resolved
    ...
# response.on_resolved(lambda res_1, res_2, res_3: ...)

response = Race(pending_1, pending_2, pending_3)
@response.on_resolved
def do_stuff(res_1, res_2, res_3):  # When the first `PendingResponse`s is resol
    ...

def timeout(delay, generator):
    return Race(GeneratorHandler(generator), wait(delay))
```

- Automatically transform scheduled functions into generators (use `isinstance`)
- Handle `PendingResponse`s raising exceptions.
- Rather than call `wake` when a `PendingResponse` is resolved, it may be better to "schedule it for now"...
- Etc.