Discrete event simulation for Python

PyPI | GitLab | Issues | Mailing list

Overview

SimPy is a process-based discrete-event simulation framework based on standard Python.

Processes in SimPy are defined by Python generator functions and may, for example, be used to model active components like customers, vehicles or agents. SimPy also provides various types of shared resources to model limited capacity congestion points (like servers, checkout counters and tunnels).

Simulations can be performed "as fast as possible", in real time (wall clock time) or by manually stepping through the events.

Though it is theoretically possible to do continuous simulations with SimPy, it has no features that help you with that. On the other hand, SimPy is overkill for simulations with a fixed step size where your processes don't interact with each other or with shared resources.

A short example simulating two clocks ticking in different time intervals looks like this:

Documentation

```
>>> import simpy
>>>
>>> def clock(env, name, tick):
...     while True:
...         print(name, env.now)
...         yield env.timeout(tick)
...
>>> env = simpy.Environment()
>>> env.process(clock(env, 'fast', 0.5))
<Process(clock) object at 0x...>
>>> env.process(clock(env, 'slow', 1))
<Process(clock) object at 0x...>
>>> env.run(until=2)
fast 0
slow 0
fast 0.5
slow 1
fast 1.0
fast 1.5
```

The documentation contains a tutorial, several guides explaining key concepts, a number of examples and the API reference.

SimPy is released under the MIT License. Simulation model developers are encouraged to share their SimPy modeling techniques with the SimPy community. Please post a message to the SimPy mailing list.

There is an introductory talk that explains SimPy's concepts and provides some examples: watch the video or get the slides.

SimPy has also been reimplemented in other programming languages. See the list of ports for more details.

Documentation for SimPy

Contents:

Indices and tables

SimPy in 10 Minutes

In this section, you'll learn the basics of SimPy in just a few minutes. Afterwards, you will be able to implement a simple simulation using SimPy and you'll be able to make an educated decision if SimPy is what you need. We'll also give you some hints on how to proceed to implement more complex simulations.

Topical Guides

This sections covers various aspects of SimPy more in-depth. It assumes that you have a basic understanding of SimPy's capabilities and that you know what you are looking for.

Examples

All theory is grey. In this section, we present various practical examples that demonstrate how to uses SimPy's features.

Here is a list of examples grouped by the features they demonstrate.

Condition events

- Bank Renege
- Movie Renege

Interrupts

- Machine Shop

Monitoring

Resources: Container

- Gas Station Refueling

Resources: Preemptive Resource

- Machine Shop

Resources: Resource

- Bank Renege
- Carwash
- Gas Station Refueling
- Movie Renege

Resources: Store

- Event Latency
- Process Communication

Shared events

- Movie Renege

Waiting for other processes

- Carwash
- Gas Station Refueling

All examples

- Bank Renege
- Carwash
- Machine Shop
- Movie Renege
- Gas Station Refueling
- Process Communication
- Event Latency

You have ideas for better examples? Please send them to our mailing list or make a pull request on GitLab.

API Reference

The API reference provides detailed descriptions of SimPy's classes and functions. It should be helpful if you plan to extend SimPy with custom components.

About SimPy

This sections is all about the non-technical stuff. How did SimPy evolve? Who was responsible for it? And what the heck were they thinking when they made it?

Discrete event simulation for Python

PyPI | GitLab | Issues | Mailing list

Overview

SimPy is a process-based discrete-event simulation framework based on standard Python.

Processes in SimPy are defined by Python generator functions and may, for example, be used to model active components like customers, vehicles or agents. SimPy also provides various types of shared resources to model limited capacity congestion points (like servers, checkout counters and tunnels).

Simulations can be performed "as fast as possible", in real time (wall clock time) or by manually stepping through the events.

Though it is theoretically possible to do continuous simulations with SimPy, it has no features that help you with that. On the other hand, SimPy is overkill for simulations with a fixed step size where your processes don't interact with each other or with shared resources.

A short example simulating two clocks ticking in different time intervals looks like this:

Documentation

```
>>> import simpy
>>>
>>> def clock(env, name, tick):
...     while True:
...         print(name, env.now)
...         yield env.timeout(tick)
...
>>> env = simpy.Environment()
>>> env.process(clock(env, 'fast', 0.5))
<Process(clock) object at 0x...>
>>> env.process(clock(env, 'slow', 1))
<Process(clock) object at 0x...>
>>> env.run(until=2)
fast 0
slow 0
fast 0.5
slow 1
fast 1.0
fast 1.5
```

The documentation contains a tutorial, several guides explaining key concepts, a number of examples and the API reference.

SimPy is released under the MIT License. Simulation model developers are encouraged to share their SimPy modeling techniques with the SimPy community. Please post a message to the SimPy mailing list.

There is an introductory talk that explains SimPy's concepts and provides some examples: watch the video or get the slides.

SimPy has also been reimplemented in other programming languages. See the list of ports for more details.

Installation

SimPy is implemented in pure Python and has no dependencies. SimPy runs on Python 3 (>= 3.8). PyPy3 is also supported. If you have pip installed, just type

```
$ pip install simpy
```

and you are done.

Installing from source

Alternatively, you can download SimPy and install it manually. Extract the archive, open a terminal window where you extracted SimPy and type:

```
$ python setup.py install
```

You can now optionally run SimPy's tests to see if everything works fine. You need pytest for this. Run the following command within the source directory of SimPy:

```
$ py.test --pyargs simpy
```

Upgrading from SimPy 2

If you are already familiar with SimPy 2, please read the Guide Porting from SimPy 2 to 3.

What's Next

Now that you've installed SimPy, you probably want to simulate something. The next section will introduce you to SimPy's basic concepts.

Basic Concepts

SimPy is a discrete-event simulation library. The behavior of active components (like vehicles, customers or messages) is modeled with *processes*. All processes live in an *environment*. They interact with the environment and with each other via *events*.

Processes are described by simple Python generators. You can call them *process function* or *process method*, depending on whether it is a normal function or method of a class. During their lifetime, they create events and `yield` them in order to wait for them to occur.

When a process yields an event, the process gets *suspended*. SimPy *resumes* the process, when the event occurs (we say that the event is *processed*). Multiple processes can wait for the same event. SimPy resumes them in the same order in which they yielded that event.

An important event type is the `Timeout`. Events of this type occur (are processed) after a certain amount of (simulated) time has passed. They allow a process to sleep (or hold its state) for the given time. A `Timeout` and all other events can be created by calling the appropriate method of the `Environment` that the process lives in (`Environment.timeout()` for example).

Our First Process

Our first example will be a *car* process. The car will alternately drive and park for a while. When it starts driving (or parking), it will print the current simulation time.

So let's start:

```
>>> def car(env):
...     while True:
...         print('Start parking at %d' % env.now)
...         parking_duration = 5
...         yield env.timeout(parking_duration)
...
...         print('Start driving at %d' % env.now)
...         trip_duration = 2
...         yield env.timeout(trip_duration)
```

Our *car* process requires a reference to an `Environment` (`env`) in order to create new events. The *car*'s behavior is described in an infinite loop. Remember, this function is a generator. Though it will never terminate, it will pass the control flow back to the simulation once a

`yield` statement is reached. Once the yielded event is processed ("it occurs"), the simulation will resume the function at this statement.

As I said before, our car switches between the states *parking* and *driving*. It announces its new state by printing a message and the current simulation time (as returned by the `Environment.now` property). It then calls the `Environment.timeout()` factory function to create a `Timeout` event. This event describes the point in time the car is done *parking* (or *driving*, respectively). By yielding the event, it signals the simulation that it wants to wait for the event to occur.

Now that the behavior of our car has been modeled, lets create an instance of it and see how it behaves:

```
>>> import simpy
>>> env = simpy.Environment()
>>> env.process(car(env))
<Process(car) object at 0x...>
>>> env.run(until=15)
Start parking at 0
Start driving at 5
Start parking at 7
Start driving at 12
Start parking at 14
```

The first thing we need to do is to create an instance of `Environment`. This instance is passed into our *car* process function. Calling it creates a *process generator* that needs to be started and added to the environment via `Environment.process()`.

Note, that at this time, none of the code of our process function is being executed. Its execution is merely scheduled at the current simulation time.

The `Process` returned by `process()` can be used for process interactions (we will cover that in the next section, so we will ignore it for now).

Finally, we start the simulation by calling `run()` and passing an end time to it.

What's Next?

You should now be familiar with SimPy's terminology and basic concepts. In the next section, we will cover process interaction.

Process Interaction

The `Process` instance that is returned by `Environment.process()` can be utilized for process interactions. The two most common examples for this are to wait for another process to finish and to interrupt another process while it is waiting for an event.

Waiting for a Process

As it happens, a SimPy `Process` can be used like an event (technically, a process actually *is* an event). If you yield it, you are resumed once the process has finished. Imagine a car-wash simulation where cars enter the car-wash and wait for the washing process to finish. Or an airport simulation where passengers have to wait until a security check finishes.

Lets assume that the car from our last example magically became an electric vehicle. Electric vehicles usually take a lot of time charging their batteries after a trip. They have to wait until their battery is charged before they can start driving again.

We can model this with an additional `charge()` process for our car. Therefore, we refactor our car to be a class with two process methods: `run()` (which is the original `car()` process function) and `charge()`.

The `run` process is automatically started when `Car` is instantiated. A new `charge` process is started every time the vehicle starts parking. By yielding the `Process` instance that `Environment.process()` returns, the `run` process starts waiting for it to finish:

```
>>> class Car(object):
...     def __init__(self, env):
...         self.env = env
...         # Start the run process everytime an instance is created.
...         self.action = env.process(self.run())
...
...     def run(self):
...         while True:
...             print('Start parking and charging at %d' % self.env.now)
...             charge_duration = 5
...             # We yield the process that process() returns
...             # to wait for it to finish
...             yield self.env.process(self.charge(charge_duration))
...
...             # The charge process has finished and
...             # we can start driving again.
...             print('Start driving at %d' % self.env.now)
...             trip_duration = 2
...             yield self.env.timeout(trip_duration)
...
...     def charge(self, duration):
...         yield self.env.timeout(duration)
```

Starting the simulation is straightforward again: We create an environment, one (or more) cars and finally call `run()`.

```
>>> import simpy
>>> env = simpy.Environment()
>>> car = Car(env)
>>> env.run(until=15)
Start parking and charging at 0
Start driving at 5
Start parking and charging at 7
Start driving at 12
Start parking and charging at 14
```

Interrupting Another Process

Imagine, you don't want to wait until your electric vehicle is fully charged but want to interrupt the charging process and just start driving instead.

SimPy allows you to interrupt a running process by calling its `interrupt()` method:

```
>>> def driver(env, car):
...     yield env.timeout(3)
...     car.action.interrupt()
```

The `driver` process has a reference to the car's `action` process. After waiting for 3 time steps, it interrupts that process.

Interrupts are thrown into process functions as `Interrupt` exceptions that can (should) be handled by the interrupted process. The process can then decide what to do next (e.g., continuing to wait for the original event or yielding a new event):

```
>>> class Car(object):
...     def __init__(self, env):
...         self.env = env
...         self.action = env.process(self.run())
...
...     def run(self):
...         while True:
...             print('Start parking and charging at %d' % self.env.now)
...             charge_duration = 5
...             # We may get interrupted while charging the battery
...             try:
...                 yield self.env.process(self.charge(charge_duration))
...             except simpy.Interrupt:
...                 # When we received an interrupt, we stop charging and
...                 # switch to the "driving" state
...                 print('Was interrupted. Hope, the battery is full enough ...')
...
...             print('Start driving at %d' % self.env.now)
...             trip_duration = 2
...             yield self.env.timeout(trip_duration)
...
...     def charge(self, duration):
...         yield self.env.timeout(duration)
```

When you compare the output of this simulation with the previous example, you'll notice that the car now starts driving at time `3` instead of `5`:

```
>>> env = simpy.Environment()
>>> car = Car(env)
>>> env.process(driver(env, car))
<Process(driver) object at 0x...>
>>> env.run(until=15)
Start parking and charging at 0
Was interrupted. Hope, the battery is full enough ...
Start driving at 3
Start parking and charging at 5
Start driving at 10
Start parking and charging at 12
```

What's Next

We just demonstrated two basic methods for process interactions—waiting for a process and interrupting a process. Take a look at the Topical Guides or the `Process` API reference for more details.

In the next section we will cover the basic usage of shared resources.

Shared Resources

SimPy offers three types of `resources` that help you modeling problems, where multiple processes want to use a resource of limited capacity (e.g., cars at a fuel station with a limited number of fuel pumps) or classical producer-consumer problems.

In this section, we'll briefly introduce SimPy's `Resource` class.

Basic Resource Usage

We'll slightly modify our electric vehicle process `car` that we introduced in the last sections.

The car will now drive to a *battery charging station (BCS)* and request one of its two *charging spots*. If both of these spots are currently in use, it waits until one of them becomes available again. It then starts charging its battery and leaves the station afterwards:

```
>>> def car(env, name, bcs, driving_time, charge_duration):
...     # Simulate driving to the BCS
...     yield env.timeout(driving_time)
...
...     # Request one of its charging spots
...     print('%s arriving at %d' % (name, env.now))
...     with bcs.request() as req:
...         yield req
...
...         # Charge the battery
...         print('%s starting to charge at %s' % (name, env.now))
...         yield env.timeout(charge_duration)
...         print('%s leaving the bcs at %s' % (name, env.now))
```

The resource's `request()` method generates an event that lets you wait until the resource becomes available again. If you are resumed, you "own" the resource until you *release* it.

If you use the resource with the `with` statement as shown above, the resource is automatically being released. If you call `request()` without `with`, you are responsible to call `release()` once you are done using the resource.

When you release a resource, the next waiting process is resumed and now "owns" one of the resource's slots. The basic `Resource` sorts waiting processes in a *FIFO (first in–first out)* way.

A resource needs a reference to an `Environment` and a *capacity* when it is created:

```
>>> import simpy
>>> env = simpy.Environment()
>>> bcs = simpy.Resource(env, capacity=2)
```

We can now create the `car` processes and pass a reference to our resource as well as some additional parameters to them:

```
>>> for i in range(4):
...     env.process(car(env, 'Car %d' % i, bcs, i*2, 5))
<Process(car) object at 0x...>
<Process(car) object at 0x...>
<Process(car) object at 0x...>
<Process(car) object at 0x...>
```

Finally, we can start the simulation. Since the car processes all terminate on their own in this simulation, we don't need to specify an *until* time—the simulation will automatically stop when there are no more events left:

```
>>> env.run()
Car 0 arriving at 0
Car 0 starting to charge at 0
Car 1 arriving at 2
Car 1 starting to charge at 2
Car 2 arriving at 4
Car 0 leaving the bcs at 5
Car 2 starting to charge at 5
Car 3 arriving at 6
Car 1 leaving the bcs at 7
Car 3 starting to charge at 7
Car 2 leaving the bcs at 10
Car 3 leaving the bcs at 12
```

Note that the first two cars can start charging immediately after they arrive at the BCS, while cars 2 and 3 have to wait.

What's Next

You should now be familiar with SimPy's basic concepts. The next section shows you how you can proceed with using SimPy from here on.

How to Proceed

If you are not certain yet if SimPy fulfills your requirements or if you want to see more features in action, you should take a look at the various examples we provide.

If you are looking for a more detailed description of a certain aspect or feature of SimPy, the Topical Guides section might help you.

Finally, there is an API Reference that describes all functions and classes in full detail.

SimPy basics

This guide describes the basic concepts of SimPy: How does it work? What are processes, events and the environment? What can I do with them?

How SimPy works

If you break SimPy down, it is just an asynchronous event dispatcher. You generate events and schedule them at a given simulation time. Events are sorted by priority, simulation time, and an increasing event id. An event also has a list of callbacks, which are executed when the event is triggered and processed by the event loop. Events may also have a return value.

The components involved in this are the `Environment`, `events` and the process functions that you write.

Process functions implement your simulation model, that is, they define the behavior of your simulation. They are plain Python generator functions that yield instances of `Event`.

The environment stores these events in its event list and keeps track of the current simulation time.

If a process function yields an event, SimPy adds the process to the event's callbacks and suspends the process until the event is triggered and processed. When a process waiting for an event is resumed, it will also receive the event's value.

Here is a very simple example that illustrates all this; the code is more verbose than it needs to be to make things extra clear. You find a compact version of it at the end of this section:

```
>>> import simpy
>>>
>>> def example(env):
...     event = simpy.events.Timeout(env, delay=1, value=42)
...     value = yield event
...     print('now=%d, value=%d' % (env.now, value))
>>>
>>> env = simpy.Environment()
>>> example_gen = example(env)
>>> p = simpy.events.Process(env, example_gen)
>>>
>>> env.run()
now=1, value=42
```

The `example()` process function above first creates a `Timeout` event. It passes the environment, a delay, and a value to it. The Timeout schedules itself at `now + delay` (that's why the environment is required); other event types usually schedule themselves at the current simulation time.

The process function then yields the event and thus gets suspended. It is resumed, when SimPy processes the Timeout event. The process function also receives the event's value (42) – this is, however, optional, so `yield event` would have been okay if you were not interested in the value or if the event had no value at all.

Finally, the process function prints the current simulation time (that is accessible via the environment's `now` attribute) and the Timeout's value.

If all required process functions are defined, you can instantiate all objects for your simulation. In most cases, you start by creating an instance of `Environment`, because you'll need to pass it around a lot when creating everything else.

Starting a process function involves two things:

1. You have to call the process function to create a generator object. (This will not execute any code of that function yet. Please read The Python yield keyword explained, to understand why this is the case.)
2. You then create an instance of `Process` and pass the environment and the generator object to it. This will schedule an `Initialize` event at the current simulation time which starts the execution of the process function. The process instance is also an event that is triggered when the process function returns. The guide to events explains why this is handy.

Finally, you can start SimPy's event loop. By default, it will run as long as there are events in the event list, but you can also let it stop earlier by providing an `until` argument (see Simulation control).

The following guides describe the environment and its interactions with events and process functions in more detail.

"Best practice" version of the example above

```
>>> import simpy
>>>
>>> def example(env):
...     value = yield env.timeout(1, value=42)
...     print('now=%d, value=%d' % (env.now, value))
>>>
>>> env = simpy.Environment()
>>> p = env.process(example(env))
>>> env.run()
now=1, value=42
```

Environments

A simulation environment manages the simulation time as well as the scheduling and processing of events. It also provides means to step through or execute the simulation.

Normal simulations use `Environment`. For real-time simulations, SimPy provides a `RealtimeEnvironment` (more on that in Real-time simulations).

Simulation control

SimPy is very flexible in terms of simulation execution. You can run your simulation until there are no more events, until a certain simulation time is reached, or until a certain event is triggered. You can also step through the simulation event by event. Furthermore, you can mix these things as you like.

For example, you could run your simulation until an interesting event occurs. You could then step through the simulation event by event for a while; and finally run the simulation until there are no more events left and your processes have all terminated.

The most important method here is `Environment.run()`:

- If you call it without any argument (`env.run()`), it steps through the simulation until there are no more events left.

  Warning

  If your processes run forever (`while True: yield env.timeout(1)`), this method will never terminate (unless you kill your script by e.g., pressing `Ctrl-C`).

- In most cases it is advisable to stop your simulation when it reaches a certain simulation time. Therefore, you can pass the desired time via the *until* parameter, e.g.: `env.run(until=10)`.

  The simulation will then stop when the internal clock reaches 10 but will not process any events scheduled for time 10. This is similar to a new environment where the clock is 0 but (obviously) no events have yet been processed.

  If you want to integrate your simulation in a GUI and want to draw a process bar, you can repeatedly call this function with increasing *until* values and update your progress bar after each call:

```
for i in range(100):
    env.run(until=i)
    progressbar.update(i)
```

- Instead of passing a number to `run()`, you can also pass any event to it. `run()` will then return when the event has been processed.

  Assuming that the current time is 0, `env.run(until=env.timeout(5))` is equivalent to `env.run(until=5)`.

  You can also pass other types of events (remember, that a **`Process`** is an event, too):

```
>>> import simpy
>>>
>>> def my_proc(env):
...     yield env.timeout(1)
...     return 'Monty Python's Flying Circus'
>>>
>>> env = simpy.Environment()
>>> proc = env.process(my_proc(env))
>>> env.run(until=proc)
'Monty Python's Flying Circus'
```

To step through the simulation event by event, the environment offers **`peek()`** and **`step()`**.

`peek()` returns the time of the next scheduled event or *infinity* (`float('inf')`) if no future events are scheduled.

`step()` processes the next scheduled event. It raises an **`EmptySchedule`** exception if no event is available.

In a typical use case, you use these methods in a loop like:

```
until = 10
while env.peek() < until:
    env.step()
```

State access

The environment allows you to get the current simulation time via the **`Environment.now`** property. The simulation time is a number without unit and is increased via **`Timeout`** events.

By default, `now` starts at 0, but you can pass an `initial_time` to the **`Environment`** to use something else.

Note

Although the simulation time is technically unit-less, you can pretend that it is, for example, in seconds and use it like a timestamp returned by `time.time()` to calculate a date or the day of the week.

The property `Environment.active_process` is comparable to `os.getpid()` and is either `None` or pointing at the currently active `Process`. A process is *active* when its process function is being executed. It becomes *inactive* (or suspended) when it yields an event.

Thus, it only makes sense to access this property from within a process function or a function that is called by your process function:

```
>>> def subfunc(env):
...     print(env.active_process)  # will print "p1"
>>>
>>> def my_proc(env):
...     while True:
...         print(env.active_process)  # will print "p1"
...         subfunc(env)
...         yield env.timeout(1)
>>>
>>> env = simpy.Environment()
>>> p1 = env.process(my_proc(env))
>>> env.active_process  # None
>>> env.step()
<Process(my_proc) object at 0x...>
<Process(my_proc) object at 0x...>
>>> env.active_process  # None
```

An exemplary use case for this is the resource system: If a process function calls `request()` to request a resource, the resource determines the requesting process via `env.active_process`. Take a look at the code to see how we do this :-).

Event creation

To create events, you normally have to import `simpy.events`, instantiate the event class and pass a reference to the environment to it. To reduce the amount of typing, the `Environment` provides some shortcuts for event creation. For example, `Environment.event()` is equivalent to `simpy.events.Event(env)`.

Other shortcuts are:

- `Environment.process()`
- `Environment.timeout()`
- `Environment.all_of()`
- `Environment.any_of()`

More details on what the events do can be found in the guide to events.

Miscellaneous

Since Python 3.3, a generator function can have a return value:

```
def my_proc(env):
    yield env.timeout(1)
    return 42
```

In SimPy, this can be used to provide return values for processes that can be used by other processes:

```
def other_proc(env):
    ret_val = yield env.process(my_proc(env))
    assert ret_val == 42
```

Events

SimPy includes an extensive set of event types for various purposes. All of them inherit `simpy.events.Event`. The listing below shows the hierarchy of events built into SimPy:

```
events.Event
|
+– events.Timeout
|
+– events.Initialize
|
+– events.Process
|
+– events.Condition
|  |
|  +– events.AllOf
|  |
|  +– events.AnyOf
.
.
.
```

This is the set of basic events. Events are extensible and resources, for example, define additional events. In this guide, we'll focus on the events in the `simpy.events` module. The guide to resources describes the various resource events.

Event basics

SimPy events are very similar – if not identical — to deferreds, futures or promises. Instances of the class `Event` are used to describe any kind of events. Events can be in one of the following states. An event

- might happen (not triggered),
- is going to happen (triggered) or
- has happened (processed).

They traverse these states exactly once in that order. Events are also tightly bound to time and time causes events to advance their state.

Initially, events are not triggered and just objects in memory.

If an event gets triggered, it is scheduled at a given time and inserted into SimPy's event queue. The property `Event.triggered` becomes `True`.

As long as the event is not *processed*, you can add *callbacks* to an event. Callbacks are callables that accept an event as parameter and are stored in the `Event.callbacks` list.

An event becomes *processed* when SimPy pops it from the event queue and calls all of its callbacks. It is now no longer possible to add callbacks. The property `Event.processed` becomes `True`.

Events also have a *value*. The value can be set before or when the event is triggered and can be retrieved via `Event.value` or, within a process, by yielding the event (`value = yield event`).

Adding callbacks to an event

"What? Callbacks? I've never seen no callbacks!", you might think if you have worked your way through the tutorial.

That's on purpose. The most common way to add a callback to an event is yielding it from your process function (`yield event`). This will add the process' *_resume()* method as a callback. That's how your process gets resumed when it yielded an event.

However, you can add any callable object (function) to the list of callbacks as long as it accepts an event instance as its single parameter:

```
>>> import simpy
>>>
>>> def my_callback(event):
...     print('Called back from', event)
...
>>> env = simpy.Environment()
>>> event = env.event()
>>> event.callbacks.append(my_callback)
>>> event.callbacks
[<function my_callback at 0x...>]
```

If an event has been *processed*, all of its `Event.callbacks` have been executed and the attribute is set to `None`. This is to prevent you from adding more callbacks – these would of course never get called because the event has already happened.

Processes are smart about this, though. If you yield a processed event, *_resume()* will immediately resume your process with the value of the event (because there is nothing to wait for).

Triggering events

When events are triggered, they can either *succeed* or *fail*. For example, if an event is to be triggered at the end of a computation and everything works out fine, the event will *succeed*. If an exception occurs during that computation, the event will *fail*.

To trigger an event and mark it as successful, you can use `Event.succeed(value=None)`. You can optionally pass a *value* to it (e.g., the results of a computation).

To trigger an event and mark it as failed, call `Event.fail(exception)` and pass an `Exception` instance to it (e.g., the exception you caught during your failed computation).

There is also a generic way to trigger an event: `Event.trigger(event)`. This will take the value and outcome (success or failure) of the event passed to it.

`Event.succeed()` and `Event.fail()` methods return the event instance they are bound to. This allows you to do things like `yield Event(env).succeed()`.

`Event.trigger()` returns None.

Example usages for `Event`

The simple mechanics outlined above provide a great flexibility in the way events (even the basic `Event`) can be used.

One example for this is that events can be shared. They can be created by a process or outside of the context of a process. They can be passed to other processes and chained:

```
>>> class School:
...     def __init__(self, env):
...         self.env = env
...         self.class_ends = env.event()
...         self.pupil_procs = [env.process(self.pupil()) for i in range(3)]
...         self.bell_proc = env.process(self.bell())
...
...     def bell(self):
...         for i in range(2):
...             yield self.env.timeout(45)
...             self.class_ends.succeed()
...             self.class_ends = self.env.event()
...             print()
...
...     def pupil(self):
...         for i in range(2):
...             print(r' \o/', end='')
...             yield self.class_ends
...
>>> school = School(env)
>>> env.run()
 \o/ \o/ \o/
 \o/ \o/ \o/
```

This can also be used like the *passivate* / *reactivate* known from SimPy 2. The pupils *passivate* when class begins and are *reactivated* when the bell rings.

Let time pass by: the `Timeout`

To actually let time pass in a simulation, there is the *timeout* event. A timeout has two parameters: a *delay* and an optional *value*: `Timeout(delay, value=None)`. It triggers itself during its creation and schedules itself at `now + delay`. Thus, the `succeed()` and `fail()` methods cannot be called again and you have to pass the event value to it when you create the timeout.

The delay can be any kind of number, usually an *int* or *float* as long as it supports comparison and addition.

Processes are events, too

SimPy processes (as created by `Process` or `env.process()`) have the nice property of being events, too.

That means, that a process can yield another process. It will then be resumed when the other process ends. The event's value will be the return value of that process:

```
>>> def sub(env):
...     yield env.timeout(1)
...     return 23
...
>>> def parent(env):
...     ret = yield env.process(sub(env))
...     return ret
...
>>> env.run(env.process(parent(env)))
23
```

When a process is created, it schedules an `Initialize` event which will start the execution of the process when triggered. You usually won't have to deal with this type of event.

If you don't want a process to start immediately but after a certain delay, you can use `simpy.util.start_delayed()`. This method returns a helper process that uses a *timeout* before actually starting a process.

The example from above, but with a delayed start of `sub()`:

```
>>> from simpy.util import start_delayed
>>>
>>> def sub(env):
...     yield env.timeout(1)
...     return 23
...
>>> def parent(env):
...     sub_proc = yield start_delayed(env, sub(env), delay=3)
...     ret = yield sub_proc
...     return ret
...
>>> env.run(env.process(parent(env)))
23
```

Pay attention to the additional `yield` needed for the helper process.

Waiting for multiple events at once

Sometimes, you want to wait for more than one event at the same time. For example, you may want to wait for a resource, but not for an unlimited amount of time. Or you may want to wait until a set of events has happened.

SimPy therefore offers the **`AnyOf`** and **`AllOf`** events which both are a **`Condition`** event.

Both take a list of events as an argument and are triggered when any (at least one) or all of them are triggered.

```
>>> from simpy.events import AnyOf, AllOf, Event
>>> events = [Event(env) for i in range(3)]
>>> a = AnyOf(env, events)  # Triggers if at least one of "events" is triggered.
>>> b = AllOf(env, events)  # Triggers if all each of "events" is triggered.
```

The value of a condition event is an ordered dictionary with an entry for every triggered event. In the case of `AllOf`, the size of that dictionary will always be the same as the length of the event list. The value dict of `AnyOf` will have at least one entry. In both cases, the event instances are used as keys and the event values will be the values.

As a shorthand for `AllOf` and `AnyOf`, you can also use the logical operators `&` (and) and `|` (or):

```
>>> def test_condition(env):
...     t1, t2 = env.timeout(1, value='spam'), env.timeout(2, value='eggs')
...     ret = yield t1 | t2
...     assert ret == {t1: 'spam'}
...
...     t1, t2 = env.timeout(1, value='spam'), env.timeout(2, value='eggs')
...     ret = yield t1 & t2
...     assert ret == {t1: 'spam', t2: 'eggs'}
...
...     # You can also concatenate & and |
...     e1, e2, e3 = [env.timeout(i) for i in range(3)]
...     yield (e1 | e2) & e3
...     assert all(e.processed for e in [e1, e2, e3])
...
>>> proc = env.process(test_condition(env))
>>> env.run()
```

The order of condition results is identical to the order in which the condition events were specified. This allows the following idiom for conveniently fetching the values of multiple events specified in an *and* condition (including `AllOf`):

```
>>> def fetch_values_of_multiple_events(env):
...     t1, t2 = env.timeout(1, value='spam'), env.timeout(2, value='eggs')
...     r1, r2 = (yield t1 & t2).values()
...     assert r1 == 'spam' and r2 == 'eggs'
...
>>> proc = env.process(fetch_values_of_multiple_events(env))
>>> env.run()
```

Process Interaction

Discrete event simulation is only made interesting by interactions between processes.

So this guide is about:

- Sleep until woken up (passivate/reactivate)
- Waiting for another process to terminate
- Interrupting another process

The first two items were already covered in the Events guide, but we'll also include them here for the sake of completeness.

Another possibility for processes to interact are resources. They are discussed in a separate guide.

Sleep until woken up

Imagine you want to model an electric vehicle with an intelligent battery-charging controller. While the vehicle is driving, the controller can be passive but needs to be reactivate once the vehicle is connected to the power grid in order to charge the battery.

In SimPy 2, this pattern was known as *passivate / reactivate*. In SimPy 3, you can accomplish that with a simple, shared `Event`:

```
>>> from random import seed, randint
>>> seed(23)
>>>
>>> import simpy
>>>
>>> class EV:
...     def __init__(self, env):
...         self.env = env
...         self.drive_proc = env.process(self.drive(env))
...         self.bat_ctrl_proc = env.process(self.bat_ctrl(env))
...         self.bat_ctrl_reactivate = env.event()
...
...     def drive(self, env):
...         while True:
...             # Drive for 20-40 min
...             yield env.timeout(randint(20, 40))
...
...             # Park for 1–6 hours
...             print('Start parking at', env.now)
...             self.bat_ctrl_reactivate.succeed()  # "reactivate"
...             self.bat_ctrl_reactivate = env.event()
...             yield env.timeout(randint(60, 360))
...             print('Stop parking at', env.now)
...
...     def bat_ctrl(self, env):
...         while True:
...             print('Bat. ctrl. passivating at', env.now)
...             yield self.bat_ctrl_reactivate  # "passivate"
...             print('Bat. ctrl. reactivated at', env.now)
...
...             # Intelligent charging behavior here …
...             yield env.timeout(randint(30, 90))
...
>>> env = simpy.Environment()
>>> ev = EV(env)
>>> env.run(until=150)
Bat. ctrl. passivating at 0
Start parking at 29
Bat. ctrl. reactivated at 29
Bat. ctrl. passivating at 60
Stop parking at 131
```

Since `bat_ctrl()` just waits for a normal event, we no longer call this pattern *passivate / reactivate* in SimPy 3.

Waiting for another process to terminate

The example above has a problem: it may happen that the vehicle wants to park for a shorter duration than it takes to charge the battery, since the minimum park time of 60 minutes is less than the maximum charge time of 90 minutes.

To fix this problem we have to slightly change our model. A new `bat_ctrl()` will be started every time the EV starts parking. The EV then waits until the parking duration is over *and* until the charging has stopped:

```
>>> class EV:
...     def __init__(self, env):
...         self.env = env
...         self.drive_proc = env.process(self.drive(env))
...
...     def drive(self, env):
...         while True:
...             # Drive for 20-40 min
...             yield env.timeout(randint(20, 40))
...
...             # Park for 1–6 hours
...             print('Start parking at', env.now)
...             charging = env.process(self.bat_ctrl(env))
...             parking = env.timeout(randint(60, 360))
...             yield charging & parking
...             print('Stop parking at', env.now)
...
...     def bat_ctrl(self, env):
...         print('Bat. ctrl. started at', env.now)
...         # Intelligent charging behavior here …
...         yield env.timeout(randint(30, 90))
...         print('Bat. ctrl. done at', env.now)
...
>>> env = simpy.Environment()
>>> ev = EV(env)
>>> env.run(until=310)
Start parking at 29
Bat. ctrl. started at 29
Bat. ctrl. done at 83
Stop parking at 305
```

Again, nothing new (if you've read the Events guide) and special is happening. SimPy processes are events, too, so you can yield them and will thus wait for them to get triggered. You can also wait for two events at the same time by concatenating them with `&` (see Waiting for multiple events at once).

Interrupting another process

As usual, we now have another problem: Imagine, a trip is very urgent, but with the current implementation, we always need to wait until the battery is fully charged. If we could somehow interrupt that …

Fortunate coincidence, there is indeed a way to do exactly this. You can call `interrupt()` on a `Process`. This will throw an `Interrupt` exception into that process, resuming it immediately:

```
>>> class EV:
...     def __init__(self, env):
...         self.env = env
...         self.drive_proc = env.process(self.drive(env))
...
...     def drive(self, env):
...         while True:
...             # Drive for 20-40 min
...             yield env.timeout(randint(20, 40))
...
...             # Park for 1 hour
...             print('Start parking at', env.now)
...             charging = env.process(self.bat_ctrl(env))
...             parking = env.timeout(60)
...             yield charging | parking
...             if not charging.triggered:
...                 # Interrupt charging if not already done.
...                 charging.interrupt('Need to go!')
...             print('Stop parking at', env.now)
...
...     def bat_ctrl(self, env):
...         print('Bat. ctrl. started at', env.now)
...         try:
...             yield env.timeout(randint(60, 90))
...             print('Bat. ctrl. done at', env.now)
...         except simpy.Interrupt as i:
...             # Onoes! Got interrupted before the charging was done.
...             print('Bat. ctrl. interrupted at', env.now, 'msg:',
...                   i.cause)
...
>>> env = simpy.Environment()
>>> ev = EV(env)
>>> env.run(until=100)
Start parking at 31
Bat. ctrl. started at 31
Stop parking at 91
Bat. ctrl. interrupted at 91 msg: Need to go!
```

What `process.interrupt()` actually does is scheduling an **`Interruption`** event for immediate execution. If this event is executed it will remove the victim process' `_resume()` method from the callbacks of the event that it is currently waiting for (see **`target`**). Following that it will throw the `Interrupt` exception into the process.

Since we don't do anything special to the original target event of the process, the interrupted process can yield the same event again after catching the `Interrupt` – Imagine someone waiting for a shop to open. The person may get interrupted by a phone call. After finishing the call, he or she checks if the shop already opened and either enters or continues to wait.

Shared Resources

Shared resources are another way to model Process Interaction. They form a congestion point where processes queue up in order to use them.

SimPy defines three categories of resources:

- Resources – Resources that can be used by a limited number of processes at a time (e.g., a gas station with a limited number of fuel pumps).
- Containers – Resources that model the production and consumption of a homogeneous, undifferentiated bulk. It may either be continuous (like water) or discrete (like apples).
- Stores – Resources that allow the production and consumption of Python objects.

The basic concept of resources

All resources share the same basic concept: The resource itself is some kind of a container with a, usually limited, *capacity*. Processes can either try to *put* something into the resource or try to *get* something out. If the resource is full or empty, they have to *queue* up and wait.

This is roughly how every resource looks:

```
BaseResource(capacity):
   put_queue
   get_queue

   put(): event
   get(): event
```

Every resource has a maximum capacity and two queues: one for processes that want to put something into it and one for processes that want to get something out. The `put()` and `get()` methods both return an event that is triggered when the corresponding action was successful.

Resources and interrupts

While a process is waiting for a put or get event to succeed, it may be interrupted by another process. After catching the interrupt, the process has two possibilities:

1. It may continue to wait for the request (by yielding the event again).

2. It may stop waiting for the request. In this case, it has to call the event's `cancel()` method.

   Since you can easily forget this, all resources events are *context managers* (see the Python docs for details).

The resource system is modular and extensible. Resources can, for example, use specialized queues and event types. This allows them to use sorted queues, to add priorities to events, or to offer preemption.

Resources

Resources can be used by a limited number of processes at a time (e.g., a gas station with a limited number of fuel pumps). Processes *request* these resources to become a user (or to "own" them) and have to *release* them once they are done (e.g., vehicles arrive at the gas station, use a fuel-pump, if one is available, and leave when they are done).

Requesting a resource is modeled as "putting a process' token into the resource" and releasing a resource correspondingly as "getting a process' token out of the resource". Thus, calling `request()` / `release()` is equivalent to calling `put()` / `get()`. Releasing a resource will always succeed immediately.

SimPy implements three *resource* types:

1. **`Resource`**
2. **`PriorityResource`**, where queueing processes are sorted by priority
3. **`PreemptiveResource`**, where processes additionally may preempt other processes with a lower priority

Resource

The `Resource` is conceptually a *semaphore*. Its only parameter – apart from the obligatory reference to an **`Environment`** – is its *capacity*. It must be a positive number and defaults to 1: `Resource(env, capacity=1)`.

Instead of just counting its current users, it stores the request event as an "access token" for each user. This is, for example, useful for adding preemption (see below).

Here is a basic example for using a resource:

```
>>> import simpy
>>>
>>> def resource_user(env, resource):
...     request = resource.request()  # Generate a request event
...     yield request                 # Wait for access
...     yield env.timeout(1)          # Do something
...     resource.release(request)     # Release the resource
...
>>> env = simpy.Environment()
>>> res = simpy.Resource(env, capacity=1)
>>> user = env.process(resource_user(env, res))
>>> env.run()
```

Note, that you have to release the resource under all conditions; for example, if you got interrupted while waiting for or using the resource. In order to help you with that and to avoid too many `try: ... finally: ...` constructs, request events can be used as context manager:

```
>>> def resource_user(env, resource):
...     with resource.request() as req:  # Generate a request event
...         yield req                    # Wait for access
...         yield env.timeout(1)         # Do something
...                                      # Resource released automatically
>>> user = env.process(resource_user(env, res))
>>> env.run()
```

Resources allow you to retrieve lists of the current users or queued users, the number of current users and the resource's capacity:

```
>>> res = simpy.Resource(env, capacity=1)
>>>
>>> def print_stats(res):
...     print(f'{res.count} of {res.capacity} slots are allocated.')
...     print(f'  Users: {res.users}')
...     print(f'  Queued events: {res.queue}')
>>>
>>>
>>> def user(res):
...     print_stats(res)
...     with res.request() as req:
...         yield req
...         print_stats(res)
...     print_stats(res)
>>>
>>> procs = [env.process(user(res)), env.process(user(res))]
>>> env.run()
0 of 1 slots are allocated.
  Users: []
  Queued events: []
1 of 1 slots are allocated.
  Users: [<Request() object at 0x...>]
  Queued events: []
1 of 1 slots are allocated.
  Users: [<Request() object at 0x...>]
  Queued events: [<Request() object at 0x...>]
0 of 1 slots are allocated.
  Users: []
  Queued events: [<Request() object at 0x...>]
1 of 1 slots are allocated.
  Users: [<Request() object at 0x...>]
  Queued events: []
0 of 1 slots are allocated.
  Users: []
  Queued events: []
```

PriorityResource

As you may know from the real world, everything is not equally important. To map that to SimPy, there's the *PriorityResource*. This subclass of *Resource* lets requesting processes provide a priority for each request. More important requests will gain access to the resource earlier than less important ones. Priority is expressed by integer numbers; smaller numbers mean a higher priority.

Apart from that, it works like a normal *Resource*:

```
>>> def resource_user(name, env, resource, wait, prio):
...     yield env.timeout(wait)
...     with resource.request(priority=prio) as req:
...         print(f'{name} requesting at {env.now} with priority={prio}')
...         yield req
...         print(f'{name} got resource at {env.now}')
...         yield env.timeout(3)
...
>>> env = simpy.Environment()
>>> res = simpy.PriorityResource(env, capacity=1)
>>> p1 = env.process(resource_user(1, env, res, wait=0, prio=0))
>>> p2 = env.process(resource_user(2, env, res, wait=1, prio=0))
>>> p3 = env.process(resource_user(3, env, res, wait=2, prio=-1))
>>> env.run()
1 requesting at 0 with priority=0
1 got resource at 0
2 requesting at 1 with priority=0
3 requesting at 2 with priority=-1
3 got resource at 3
2 got resource at 6
```

Although *p3* requested the resource later than *p2*, it could use it earlier because its priority was higher.

PreemptiveResource

Sometimes, new requests are so important that queue-jumping is not enough and they need to kick existing users out of the resource (this is called *preemption*). The *PreemptiveResource* allows you to do exactly this:

```
>>> def resource_user(name, env, resource, wait, prio):
...     yield env.timeout(wait)
...     with resource.request(priority=prio) as req:
...         print(f'{name} requesting at {env.now} with priority={prio}')
...         yield req
...         print(f'{name} got resource at {env.now}')
...         try:
...             yield env.timeout(3)
...         except simpy.Interrupt as interrupt:
...             by = interrupt.cause.by
...             usage = env.now - interrupt.cause.usage_since
...             print(f'{name} got preempted by {by} at {env.now}'
...                   f' after {usage}')
...
>>> env = simpy.Environment()
>>> res = simpy.PreemptiveResource(env, capacity=1)
>>> p1 = env.process(resource_user(1, env, res, wait=0, prio=0))
>>> p2 = env.process(resource_user(2, env, res, wait=1, prio=0))
>>> p3 = env.process(resource_user(3, env, res, wait=2, prio=-1))
>>> env.run()
1 requesting at 0 with priority=0
1 got resource at 0
2 requesting at 1 with priority=0
3 requesting at 2 with priority=-1
1 got preempted by <Process(resource_user) object at 0x...> at 2 after 2
3 got resource at 2
2 got resource at 5
```

PreemptiveResource inherits from *PriorityResource* and adds a *preempt* flag (that defaults to `True`) to `request()`. By setting this to `False` (`resource.request(priority=x, preempt=False)`), a process can decide to not preempt another resource user. It will still be put in the queue according to its priority, though.

The implementation of *PreemptiveResource* values priorities higher than preemption. That means preempt requests are not allowed to cheat and jump over a higher prioritized request. The following example shows that preemptive low priority requests cannot queue-jump over high priority requests:

```
>>> def user(name, env, res, prio, preempt):
...     with res.request(priority=prio, preempt=preempt) as req:
...         try:
...             print(f'{name} requesting at {env.now}')
...             assert isinstance(env.now, int), type(env.now)
...             yield req
...             assert isinstance(env.now, int), type(env.now)
...             print(f'{name} got resource at {env.now}')
...             yield env.timeout(3)
...         except simpy.Interrupt:
...             print(f'{name} got preempted at {env.now}')
>>>
>>> env = simpy.Environment()
>>> res = simpy.PreemptiveResource(env, capacity=1)
>>> A = env.process(user('A', env, res, prio=0, preempt=True))
>>> env.run(until=1)  # Give A a head start
A requesting at 0
A got resource at 0
>>> B = env.process(user('B', env, res, prio=-2, preempt=False))
>>> C = env.process(user('C', env, res, prio=-1, preempt=True))
>>> env.run()
B requesting at 1
C requesting at 1
B got resource at 3
C got resource at 6
```

1. Process *A* requests the resource with priority 0. It immediately becomes a user.
2. Process *B* requests the resource with priority -2 but sets *preempt* to `False`. It will queue up and wait.
3. Process *C* requests the resource with priority -1 but leaves *preempt* `True`. Normally, it would preempt *A* but in this case, *B* is queued up before *C* and prevents *C* from preempting *A*. *C* can also not preempt *B* since its priority is not high enough.

Thus, the behavior in the example is the same as if no preemption was used at all. Be careful when using mixed preemption!

Due to the higher priority of process *B*, no preemption occurs in this example. Note that an additional request with a priority of -3 would be able to preempt *A*.

If your use-case requires a different behaviour, for example queue-jumping or valuing preemption over priorities, you can subclass *PreemptiveResource* and override the default behaviour.

Containers

Containers help you modelling the production and consumption of a homogeneous, undifferentiated bulk. It may either be continuous (like water) or discrete (like apples).

You can use this, for example, to model the gas / petrol tank of a gas station. Tankers increase the amount of gasoline in the tank while cars decrease it.

The following example is a very simple model of a gas station with a limited number of fuel dispensers (modeled as `Resource`) and a tank modeled as `Container`:

```
>>> class GasStation:
...     def __init__(self, env):
...         self.fuel_dispensers = simpy.Resource(env, capacity=2)
...         self.gas_tank = simpy.Container(env, init=100, capacity=1000)
...         self.mon_proc = env.process(self.monitor_tank(env))
...
...     def monitor_tank(self, env):
...         while True:
...             if self.gas_tank.level < 100:
...                 print(f'Calling tanker at {env.now}')
...                 env.process(tanker(env, self))
...             yield env.timeout(15)
>>>
>>>
>>> def tanker(env, gas_station):
...     yield env.timeout(10)  # Need 10 Minutes to arrive
...     print(f'Tanker arriving at {env.now}')
...     amount = gas_station.gas_tank.capacity - gas_station.gas_tank.level
...     yield gas_station.gas_tank.put(amount)
>>>
>>>
>>> def car(name, env, gas_station):
...     print(f'Car {name} arriving at {env.now}')
...     with gas_station.fuel_dispensers.request() as req:
...         yield req
...         print(f'Car {name} starts refueling at {env.now}')
...         yield gas_station.gas_tank.get(40)
...         yield env.timeout(5)
...         print(f'Car {name} done refueling at {env.now}')
>>>
>>>
>>> def car_generator(env, gas_station):
...     for i in range(4):
...         env.process(car(i, env, gas_station))
...         yield env.timeout(5)
>>>
>>>
>>> env = simpy.Environment()
>>> gas_station = GasStation(env)
>>> car_gen = env.process(car_generator(env, gas_station))
>>> env.run(35)
Car 0 arriving at 0
Car 0 starts refueling at 0
Car 1 arriving at 5
Car 0 done refueling at 5
Car 1 starts refueling at 5
Car 2 arriving at 10
Car 1 done refueling at 10
Car 2 starts refueling at 10
Calling tanker at 15
Car 3 arriving at 15
Car 3 starts refueling at 15
Tanker arriving at 25
Car 2 done refueling at 30
Car 3 done refueling at 30
```

Containers allow you to retrieve their current `level` as well as their `capacity` (see `GasStation.monitor_tank()` and `tanker()`). You can also access the list of waiting events via the `put_queue` and `get_queue` attributes (similar to `Resource.queue`).

Stores

Using Stores you can model the production and consumption of concrete objects (in contrast to the rather abstract “amount” stored in containers). A single Store can even contain multiple types of objects.

Beside `Store`, there is a `FilterStore` that lets you use a custom function to filter the objects you get out of the store and `PriorityStore` where items come out of the store in priority order.

Here is a simple example modelling a generic producer/consumer scenario:

```
>>> def producer(env, store):
...     for i in range(100):
...         yield env.timeout(2)
...         yield store.put(f'spam {i}')
...         print(f'Produced spam at', env.now)
>>>
>>>
>>> def consumer(name, env, store):
...     while True:
...         yield env.timeout(1)
...         print(name, 'requesting spam at', env.now)
...         item = yield store.get()
...         print(name, 'got', item, 'at', env.now)
>>>
>>>
>>> env = simpy.Environment()
>>> store = simpy.Store(env, capacity=2)
>>>
>>> prod = env.process(producer(env, store))
>>> consumers = [env.process(consumer(i, env, store)) for i in range(2)]
>>>
>>> env.run(until=5)
0 requesting spam at 1
1 requesting spam at 1
Produced spam at 2
0 got spam 0 at 2
0 requesting spam at 3
Produced spam at 4
1 got spam 1 at 4
```

As with the other resource types, you can get a store’s capacity via the `capacity` attribute. The attribute `items` points to the list of items currently available in the store. The put and get queues can be accessed via the `put_queue` and `get_queue` attributes.

FilterStore can, for example, be used to model machine shops where machines have varying attributes. This can be useful if the homogeneous slots of a *Resource* are not what you need:

```
>>> from collections import namedtuple
>>>
>>> Machine = namedtuple('Machine', 'size, duration')
>>> m1 = Machine(1, 2)  # Small and slow
>>> m2 = Machine(2, 1)  # Big and fast
>>>
>>> env = simpy.Environment()
>>> machine_shop = simpy.FilterStore(env, capacity=2)
>>> machine_shop.items = [m1, m2]  # Pre-populate the machine shop
>>>
>>> def user(name, env, ms, size):
...     machine = yield ms.get(lambda machine: machine.size == size)
...     print(name, 'got', machine, 'at', env.now)
...     yield env.timeout(machine.duration)
...     yield ms.put(machine)
...     print(name, 'released', machine, 'at', env.now)
>>>
>>>
>>> users = [env.process(user(i, env, machine_shop, (i % 2) + 1))
...          for i in range(3)]
>>> env.run()
0 got Machine(size=1, duration=2) at 0
1 got Machine(size=2, duration=1) at 0
1 released Machine(size=2, duration=1) at 1
0 released Machine(size=1, duration=2) at 2
2 got Machine(size=1, duration=2) at 2
2 released Machine(size=1, duration=2) at 4
```

With a `PriorityStore`, we can model items of differing priorities. In the following example, an inspector process finds and logs issues that a separate maintainer process repairs in priority order.

```
>>> env = simpy.Environment()
>>> issues = simpy.PriorityStore(env)
>>>
>>> def inspector(env, issues):
...     for issue in [simpy.PriorityItem('P2', '#0000'),
...                   simpy.PriorityItem('P0', '#0001'),
...                   simpy.PriorityItem('P3', '#0002'),
...                   simpy.PriorityItem('P1', '#0003')]:
...         yield env.timeout(1)
...         print(env.now, 'log', issue)
...         yield issues.put(issue)
>>>
>>> def maintainer(env, issues):
...     while True:
...         yield env.timeout(3)
...         issue = yield issues.get()
...         print(env.now, 'repair', issue)
>>>
>>> _ = env.process(inspector(env, issues))
>>> _ = env.process(maintainer(env, issues))
>>> env.run()
1 log PriorityItem(priority='P2', item='#0000')
2 log PriorityItem(priority='P0', item='#0001')
3 log PriorityItem(priority='P3', item='#0002')
3 repair PriorityItem(priority='P0', item='#0001')
4 log PriorityItem(priority='P1', item='#0003')
6 repair PriorityItem(priority='P1', item='#0003')
9 repair PriorityItem(priority='P2', item='#0000')
12 repair PriorityItem(priority='P3', item='#0002')
```

Real-time simulations

Sometimes, you might not want to perform a simulation as fast as possible but synchronous to the wall-clock time. This kind of simulation is also called *real-time simulation*.

Real-time simulations may be necessary

- if you have hardware-in-the-loop,
- if there is human interaction with your simulation, or
- if you want to analyze the real-time behavior of an algorithm.

To convert a simulation into a real-time simulation, you only need to replace SimPy's default `Environment` with a `simpy.rt.RealtimeEnvironment`. Apart from the *initial_time* argument, there are two additional parameters: *factor* and *strict*: `RealtimeEnvironment(initial_time=0, factor=1.0, strict=True)`.

The *factor* defines how much *real time* passes with each step of simulation time. By default, this is one second. If you set `factor=0.1`, a unit of simulation time will only take a tenth of a second; if you set `factor=60`, it will take a minute.

Here is a simple example for converting a normal simulation to a real-time simulation with a duration of one tenth of a second per simulation time unit:

```
>>> import time
>>> import simpy
>>>
>>> def example(env):
...     start = time.perf_counter()
...     yield env.timeout(1)
...     end = time.perf_counter()
...     print('Duration of one simulation time unit: %.2fs' % (end - start))
>>>
>>> env = simpy.Environment()
>>> proc = env.process(example(env))
>>> env.run(until=proc)
Duration of one simulation time unit: 0.00s
>>>
>>> import simpy.rt
>>> env = simpy.rt.RealtimeEnvironment(factor=0.1)
>>> proc = env.process(example(env))
>>> env.run(until=proc)
Duration of one simulation time unit: 0.10s
```

If the *strict* parameter is set to `True` (the default), the `step()` and `run()` methods will raise a `RuntimeError` if the computation within a simulation time step take more time than the real-time factor allows. In the following example, a process will perform a task that takes 0.02 seconds within a real-time environment with a time factor of 0.01 seconds:

```
>>> import time
>>> import simpy.rt
>>>
>>> def slow_proc(env):
...     time.sleep(0.02)  # Heavy computation :-)
...     yield env.timeout(1)
>>>
>>> env = simpy.rt.RealtimeEnvironment(factor=0.01)
>>> proc = env.process(slow_proc(env))
>>> try:
...     env.run(until=proc)
...     print('Everything alright')
... except RuntimeError:
...     print('Simulation is too slow')
Simulation is too slow
```

To suppress the error, simply set `strict=False`:

```
>>> env = simpy.rt.RealtimeEnvironment(factor=0.01, strict=False)
>>> proc = env.process(slow_proc(env))
>>> try:
...     env.run(until=proc)
...     print('Everything alright')
... except RuntimeError:
...     print('Simulation is too slow')
Everything alright
```

That's it. Real-time simulations are that simple with SimPy!

Monitoring

Monitoring is a relatively complex topic with a lot of different use-cases and lots of variations.

This guide presents some of the more common and more interesting ones. It's purpose is to give you some hints and ideas how you can implement simulation monitoring tailored to your use-cases.

So, before you start, you need to define them:

What do you want to monitor?

- Your processes?
- Resource usage?
- Trace all events of the simulation?

When do you want to monitor?

- Regularly in defined intervals?
- When something happens?

How do you want to store the collected data?

- Store it in a simple list?
- Log it to a file?
- Write it to a database?

The following sections discuss these questions and provide some example code to help you.

Monitoring your processes

Monitoring your own processes is relatively easy, because *you* control the code. From our experience, the most common thing you might want to do is monitor the value of one or more state variables every time they change or at discrete intervals and store it somewhere (in memory, in a database, or in a file, for example).

In the simples case, you just use a list and append the required value(s) every time they change:

```
>>> import simpy
>>>
>>> data = []  # This list will hold all collected data
>>>
>>> def test_process(env, data):
...     val = 0
...     for i in range(5):
...         val += env.now
...         data.append(val)  # Collect data
...         yield env.timeout(1)
>>>
>>> env = simpy.Environment()
>>> p = env.process(test_process(env, data))
>>> env.run(p)
>>> print('Collected', data)  # Lets see what we got
Collected [0, 1, 3, 6, 10]
```

If you want to monitor multiple variables, you can append (named)tuples to your data list.

If you want to store the data in a NumPy array or a database, you can often increase performance if you buffer the data in a plain Python list and only write larger chunks (or the complete dataset) to the database.

Resource usage

The use-cases for resource monitoring are numerous, for example you might want to monitor:

- Utilization of a resource over time and on average, that is,
  - the number of processes that are using the resource at a time
  - the level of a container
  - the amount of items in a store

  This can be monitored either in discrete time steps or every time there is a change.
- Number of processes in the (put|get)queue over time (and the average). Again, this could be monitored at discrete time steps or every time there is a change.
- For *PreemptiveResource*, you may want to measure how often preemption occurs over time.

In contrast to your processes, you don't have direct access to the code of the built-in resource classes. But this doesn't prevent you from monitoring them.

Monkey-patching some of a resource's methods allows you to gather all the data you need.

Here is an example that demonstrate how you can add callbacks to a resource that get called just before or after a *get / request* or a *put / release* event:

```
>>> from functools import partial, wraps
>>> import simpy
>>>
>>> def patch_resource(resource, pre=None, post=None):
...     """Patch *resource* so that it calls the callable *pre* before each
...     put/get/request/release operation and the callable *post* after each
...     operation.  The only argument to these functions is the resource
...     instance.
...
...     """
...     def get_wrapper(func):
...         # Generate a wrapper for put/get/request/release
...         @wraps(func)
...         def wrapper(*args, **kwargs):
...             # This is the actual wrapper
...             # Call "pre" callback
...             if pre:
...                 pre(resource)
...
...             # Perform actual operation
...             ret = func(*args, **kwargs)
...
...             # Call "post" callback
...             if post:
...                 post(resource)
...
...             return ret
...         return wrapper
...
...     # Replace the original operations with our wrapper
...     for name in ['put', 'get', 'request', 'release']:
...         if hasattr(resource, name):
...             setattr(resource, name, get_wrapper(getattr(resource, name)))
>>>
>>> def monitor(data, resource):
...     """This is our monitoring callback."""
...     item = (
...         resource._env.now,  # The current simulation time
...         resource.count,  # The number of users
...         len(resource.queue),  # The number of queued processes
...     )
...     data.append(item)
>>>
>>> def test_process(env, res):
...     with res.request() as req:
...         yield req
...         yield env.timeout(1)
>>>
>>> env = simpy.Environment()
>>>
>>> res = simpy.Resource(env, capacity=1)
>>> data = []
>>> # Bind *data* as first argument to monitor()
>>> # see https://docs.python.org/3/library/functools.html#functools.partial
>>> monitor = partial(monitor, data)
>>> patch_resource(res, post=monitor)  # Patches (only) this resource instance
>>>
>>> p = env.process(test_process(env, res))
>>> env.run(p)
>>>
>>> print(data)
[(0, 1, 0), (1, 0, 0)]
```

The example above is a very generic but also very flexible way to monitor all aspects of all kinds of resources.

The other extreme would be to fit the monitoring to exactly one use case. Imagine, for example, you only want to know how many processes are waiting for a `Resource` at a time:

```
>>> import simpy
>>>
>>> class MonitoredResource(simpy.Resource):
...     def __init__(self, *args, **kwargs):
...         super().__init__(*args, **kwargs)
...         self.data = []
...
...     def request(self, *args, **kwargs):
...         self.data.append((self._env.now, len(self.queue)))
...         return super().request(*args, **kwargs)
...
...     def release(self, *args, **kwargs):
...         self.data.append((self._env.now, len(self.queue)))
...         return super().release(*args, **kwargs)
>>>
>>> def test_process(env, res):
...     with res.request() as req:
...         yield req
...         yield env.timeout(1)
>>>
>>> env = simpy.Environment()
>>>
>>> res = MonitoredResource(env, capacity=1)
>>> p1 = env.process(test_process(env, res))
>>> p2 = env.process(test_process(env, res))
>>> env.run()
>>>
>>> print(res.data)
[(0, 0), (0, 0), (1, 1), (2, 0)]
```

In contrast to the first example, we now haven't patched a single resource instance but the whole class. It also removed all of the first example's flexibility: We only monitor `Resource` typed resources, we only collect data *before* the actual requests are made and we only collect the time and queue length. At the same time, you need less than half of the code.

Event tracing

In order to debug or visualize a simulation, you might want to trace when events are created, triggered and processed. Maybe you also want to trace which process created an event and which processes waited for an event.

The two most interesting functions for these use-cases are `Environment.step()`, where all events get processed, and `Environment.schedule()`, where all events get scheduled and inserted into SimPy's event queue.

Here is an example that shows how `Environment.step()` can be patched in order to trace all processed events:

```
>>> from functools import partial, wraps
>>> import simpy
>>>
>>> def trace(env, callback):
...     """Replace the ``step()`` method of *env* with a tracing function
...     that calls *callbacks* with an events time, priority, ID and its
...     instance just before it is processed.
...
...     """
...     def get_wrapper(env_step, callback):
...         """Generate the wrapper for env.step()."""
...         @wraps(env_step)
...         def tracing_step():
...             """Call *callback* for the next event if one exist before
...             calling ``env.step()``."""
...             if len(env._queue):
...                 t, prio, eid, event = env._queue[0]
...                 callback(t, prio, eid, event)
...             return env_step()
...         return tracing_step
...
...     env.step = get_wrapper(env.step, callback)
>>>
>>> def monitor(data, t, prio, eid, event):
...     data.append((t, eid, type(event)))
>>>
>>> def test_process(env):
...     yield env.timeout(1)
>>>
>>> data = []
>>> # Bind *data* as first argument to monitor()
>>> # see https://docs.python.org/3/library/functools.html#functools.partial
>>> monitor = partial(monitor, data)
>>>
>>> env = simpy.Environment()
>>> trace(env, monitor)
>>>
>>> p = env.process(test_process(env))
>>> env.run(until=p)
>>>
>>> for d in data:
...     print(d)
(0, 0, <class 'simpy.events.Initialize'>)
(1, 1, <class 'simpy.events.Timeout'>)
(1, 2, <class 'simpy.events.Process'>)
```

The example above is inspired by a pull request from Steve Pothier.

Using the same concepts, you can also patch `Environment.schedule()`. This would give you central access to the information when which event is scheduled for what time.

In addition to that, you could also patch some or all of SimPy's event classes, e.g., their *__init__()* method in order to trace when and how an event is initially being created.

Time and Scheduling

The aim of this section is to give you a deeper understanding of how time passes in SimPy and how it schedules and processes events.

What is time?

Time itself is not easy to grasp. The wikipedians describe it this way:

> *«Time is the indefinite continued progress of existence and events that occur in apparently irreversible succession from the past through the present to the future. Time is a component quantity of various measurements used to sequence events, to compare the duration of events or the intervals between them, and to quantify rates of change of quantities in material reality or in the conscious experience. Time is often referred to as the fourth dimension, along with the three spatial dimensions.»*

What's the problem with it?

Often, events (in the real world) appear to happen "at the same time", when they are in fact happening at slightly different times. Here is an obvious example: Alice and Bob have birthday on the same day. If your time scale is in days, both birthday events happen at the same time. If you increase the resolution of you clock, e.g. to minutes, you may realise that Alice was actually born at 0:42 in the morning and Bob at 11:14 and that there's quite a difference between the time of both events.

Doing simulation on computers suffers from similar problems. Integers (and floats, too) are discrete numbers with a lot of void in between them. Thus, events that would occur after each other in the real world (e.g., at $t_1 = 0.1$ and $t_2 = 0.2$) might occur at the "same" time if mapped to an integer scale (e.g., at $t = 0$).

On the other hand, SimPy is (like most simulation frameworks) a single-threaded, deterministic library. It processes events sequentially – one after another. If two events are scheduled at the same time, the one that is scheduled first will also be the processed first (FIFO).

That is very important for you to understand. The processes in your modeled/simulated world may run "in parallel", but when the simulation runs on your CPU, all events are processed sequentially and deterministically. If you run your simulation multiple times (and if you don't use `random` ;-)), you will *always* get the same results.

So keep this in mind:

- In the real world, there's usually no *at the same time*.
- Discretization of the time scale can make events appear to be *at the same time*.
- SimPy processes events *one after another*, even if they have the *same time*.

SimPy Events and time

Before we continue, let's recap the states an event can be in (see Events for details):

- untriggered: not known to the event queue
- triggered: scheduled at a time *t* and inserted into the event queue
- processed: removed from the event queue

SimPy's event queue is implemented as a heap queue: "Heaps are binary trees for which every parent node has a value less than or equal to any of its children." So if we insert events as tuples *(t, event)* (with *t* being the scheduled time) into it, the first element in the queue will by definition always be the one with the smallest *t* and the next one to be processed.

However, storing *(t, event)* tuples will not work if two events are scheduled at the same time because events are not comparable. To fix this, we also store a strictly increasing event ID with them: *(t, eid, event)*. That way, if two events get scheduled for the same time, the one scheduled first will always be processed first.

Porting from SimPy 3 to 4

SimPy 4 drops support for Python 2.7 and requires Python 3.6+. SimPy 4 also breaks compatibility with SimPy 3 in a few minor ways.

Python 3.6+

When porting a SimPy application to SimPy 4, the first step is to ensure that the application works with SimPy 3 on Python >= 3.6. Once the application works with Python 3.6, most of the work in porting to SimPy 4 is complete.

For more information on porting from Python 2 to 3, see this guide from the Python docs.

Environment Subclasses

The `BaseEnvironment` class has been removed in SimPy 4. The **`Environment`** class is now the most base environment class. Any code that inherited from `BaseEnvironment` should be modified to inherit from **`Environment`** instead.

For example, the following SimPy 3 code:

```
class MyEnvironment(simpy.BaseEnvironment):
    ...
```

would be rewritten as follows for SimPy 4:

```
class MyEnvironment(simpy.Environment):
    ...
```

Returning from Process Generators

In Python 2 it is a syntax error for a generator (i.e. a function using the `yield` keyword) to return a value using the `return` keyword. SimPy 3, supports a process generator returning a value via the `Environment.exit(value)` method, which simply raises a `StopProcess` exception. This mechanism was required for Python 2, but also works with Python 3.

In Python 3, it is legal to use `return` to return a value from a generator. Returning from a `Process` generator is also supported in SimPy 3 for applications using Python 3 exclusively.

In SimPy 4, the `Environment.exit()` method and the `StopProcess` exception are eliminated. Applications with generators that need to either return early or return a value *must* use the `return` Python keyword.

> **Note**
>
> No change is required for generators that do not return a value or that do not have control flow that requires returning early. This is the most common case for process generators used with SimPy.

Once a SimPy 3 application is ported to run with Python 3, it may then replace any uses of `Environment.exit(value)` and `raise StopProcess(value)` with `return value`. Once all occurrences are changed, the application will be ready to run with SimPy 4.

Example: Return from Generator

In the following example, `Environment.exit()` is used to return the first the first "needle" from a store of items. When using SimPy 3 with Python 2, this is the only way to return a value from a process generator.

```
def find_first_needle(env, store):
    while True:
        item = yield store.get()
        if is_needle(item):
            env.exit(item)  # Python2 generators cannot use return

def proc(env, store):
    needle = yield env.process(find_first_needle(env, store))
```

In SimPy 4 or with SimPy 3 and Python 3, `find_first_needle()` can be rewritten as:

```
def find_first_needle(env, store):
    while True:
        item = yield store.get()
        if is_needle(item):
            return item  # A Python3 generator can return
```

Sticking with SimPy 3

For applications that are not yet ready to upgrade to SimPy 4 and Python 3, or that may never upgrade, the SimPy dependency must be pinned to version 3.x.

When installing SimPy with `pip`, use the following to force the latest SimPy 3.x to be installed:

```
pip install 'simpy<4'
```

A similar version specification can be used in requirements files:

```
simpy<4
```

Or in the `install_requires` list in a `setup.py` file:

```
setup(
    ...,
    install_requires=['simpy<4'],
    ...,
)
```

Porting from SimPy 2 to 3

Porting from SimPy 2 to SimPy 3 is not overly complicated. A lot of changes merely comprise copy/paste.

This guide describes the conceptual and API changes between both SimPy versions and shows you how to change your code for SimPy 3.

Imports

In SimPy 2, you had to decide at import-time whether you wanted to use a normal simulation (`SimPy.Simulation`), a real-time simulation (`SimPy.SimulationRT`) or something else. You usually had to import `Simulation` (or `SimulationRT`), `Process` and some of the SimPy keywords (`hold` or `passivate`, for example) from that package.

In SimPy 3, you usually need to import much less classes and modules (for example, all keywords are gone). In most use cases you will now only need to import `simpy`.

SimPy 2

```
from Simpy.Simulation import Simulation, Process, hold
```

SimPy 3

```
import simpy
```

The `Simulation*` classes

SimPy 2 encapsulated the simulation state in a `Simulation*` class (e.g., `Simulation`, `SimulationRT` or `SimulationTrace`). This class also had a `simulate()` method that executed a normal simulation, a real-time simulation or something else (depending on the particular class).

There was a global `Simulation` instance that was automatically created when you imported SimPy. You could also instantiate it on your own to uses SimPy's object-orient API. This led to some confusion and problems, because you had to pass the `Simulation` instance around

when you were using the object-oriented API but not if you were using the procedural API.

In SimPy 3, an `Environment` replaces `Simulation` and `RealtimeEnvironment` replaces `SimulationRT`. You always need to instantiate an environment. There's no more global state.

To execute a simulation, you call the environment's `run()` method.

SimPy 2

```
# Procedural API
from SimPy.Simulation import initialize, simulate

initialize()
# Start processes
simulate(until=10)
```

```
# Object-oriented API
from SimPy.Simulation import Simulation

sim = Simulation()
# Start processes
sim.simulate(until=10)
```

SimPy3

```
import simpy

env = simpy.Environment()
# Start processes
env.run(until=10)
```

Defining a Process

Processes had to inherit the `Process` base class in SimPy 2. Subclasses had to implement at least a so called *Process Execution Method (PEM)* (which is basically a generator function) and in most cases `__init__()`. Each process needed to know the `Simulation` instance it belonged to. This reference was passed implicitly in the procedural API and had to be passed explicitly in the object-oriented API. Apart from some internal problems, this made it quite cumbersome to define a simple process.

Processes were started by passing the `Process` and a generator instance created by the generator function to either the global `activate()` function or the corresponding `Simulation` method.

A process in SimPy 3 is a Python generator (no matter if it's defined on module level or as an instance method) wrapped in a `Process` instance. The generator usually requires a reference to a `Environment` to interact with, but this is completely optional.

Processes are can be started by creating a `Process` instance and passing the generator to it. The environment provides a shortcut for this: `process()`.

SimPy 2

```
# Procedural API
from Simpy.Simulation import Process

class MyProcess(Process):
    def __init__(self, another_param):
        super().__init__()
        self.another_param = another_param

    def generator_function(self):
        """Implement the process' behavior."""
        yield something

initialize()
proc = Process('Spam')
activate(proc, proc.generator_function())
```

```
# Object-oriented API
from SimPy.Simulation import Simulation, Process

class MyProcess(Process):
    def __init__(self, sim, another_param):
        super().__init__(sim=sim)
        self.another_param = another_param

    def generator_function(self):
        """Implement the process' behaviour."""
        yield something

sim = Simulation()
proc = Process(sim, 'Spam')
sim.activate(proc, proc.generator_function())
```

SimPy 3

```
import simpy

def generator_function(env, another_param):
    """Implement the process' behavior."""
    yield something

env = simpy.Environment()
proc = env.process(generator_function(env, 'Spam'))
```

SimPy Keywords (`hold` etc.)

In SimPy 2, processes created new events by yielding a *SimPy Keyword* and some additional parameters (at least `self`). These keywords had to be imported from `SimPy.Simulation*` if they were used. Internally, the keywords were mapped to a function that generated the according event.

In SimPy 3, you directly yield `events` if you want to wait for an event to occur. You can instantiate an event directly or use the shortcuts provided by `Environment`.

Generally, whenever a process yields an event, the execution of the process is suspended and resumed once the event has been triggered. To motivate this understanding, some of the events were renamed. For example, the `hold` keyword meant to wait until some time has passed. In terms of events this means that a timeout has happened. Therefore `hold` has been replaced by a `Timeout` event.

Note

`Process` is also an `Event`. If you want to wait for a process to finish, simply yield it.

SimPy 2

```
yield hold, self, duration
yield passivate, self
yield request, self, resource
yield release, self, resource
yield waitevent, self, event
yield waitevent, self, [event_a, event_b, event_c]
yield queueevent, self, event_list
yield get, self, level, amount
yield put, self, level, amount
```

SimPy 3

```
yield env.timeout(duration)        # hold: renamed
yield env.event()                  # passivate: renamed
yield resource.request()           # Request is now bound to class Resource
resource.release()                 # Release no longer needs to be yielded
yield event                        # waitevent: just yield the event
yield env.all_of([event_a, event_b, event_c])  # waitevent
yield env.any_of([event_a, event_b, event_c])  # queuevent
yield container.get(amount)        # Level is now called Container
yield container.put(amount)

yield event_a | event_b            # Wait for either a or b. This is new.
yield event_a & event_b            # Wait for a and b. This is new.
yield env.process(calculation(env))  # Wait for the process calculation to
                                     # to finish.
```

Partially supported features

The following `waituntil` keyword is not completely supported anymore:

```
yield waituntil, self, cond_func
```

SimPy 2 was evaluating `cond_func` after *every* event, which was computationally very expensive. One possible workaround is for example the following process, which evaluates `cond_func` periodically:

```
def waituntil(env, cond_func, delay=1):
    while not cond_func():
        yield env.timeout(delay)

# Usage:
yield waituntil(env, cond_func)
```

Interrupts

In SimPy 2, `interrupt()` was a method of the interrupting process. The victim of the interrupt had to be passed as an argument.

The victim was not directly notified of the interrupt but had to check if the `interrupted` flag was set. Afterwards, it had to reset the interrupt via `interruptReset()`. You could manually set the `interruptCause` attribute of the victim.

Explicitly checking for an interrupt is obviously error prone as it is too easy to be forgotten.

In SimPy 3, you call `interrupt()` on the victim process. You can optionally supply a cause. An `Interrupt` is then thrown into the victim process, which has to handle the interrupt via `try: ... except Interrupt: ...`.

SimPy 2

```
class Interrupter(Process):
    def __init__(self, victim):
        super().__init__()
        self.victim = victim

    def run(self):
        yield hold, self, 1
        self.interrupt(self.victim_proc)
        self.victim_proc.interruptCause = 'Spam'

class Victim(Process):
    def run(self):
        yield hold, self, 10
        if self.interrupted:
            cause = self.interruptCause
            self.interruptReset()
```

SimPy 3

```
def interrupter(env, victim_proc):
    yield env.timeout(1)
    victim_proc.interrupt('Spam')

def victim(env):
    try:
        yield env.timeout(10)
    except Interrupt as interrupt:
        cause = interrupt.cause
```

Conclusion

This guide is by no means complete. If you run into problems, please have a look at the other guides, the examples or the API Reference. You are also very welcome to submit improvements. Just create a merge request at GitLab.

simpy

The `simpy` module aggregates SimPy's most used components into a single namespace. This is purely for convenience. You can of course also access everything (and more!) via their actual submodules.

The following tables list all the available components in this module.

Environments

`Environment` ([initial_time])	Execution environment for an event-based simulation
`RealtimeEnvironment` ([initial_time, factor, ...])	Execution environment for an event-based simulation

Events

`Event` (env)	An event that may happen at some point in time.
`Timeout` (env, delay[, value])	A `Event` that gets processed after a *delay* has passed.
`Process` (env, generator)	Process an event yielding generator.
`AllOf` (env, events)	A `Condition` event that is triggered if all of a list of *events* have been successfully tri
`AnyOf` (env, events)	A `Condition` event that is triggered if any of a list of *events* has been successfully tri
`Interrupt` (cause)	Exception thrown into a process if it is interrupted (see `interrupt()`).

Resources

`Resource` (env[, capacity])	Resource with *capacity* of usage slots that can be requested b
`PriorityResource` (env[, capacity])	A `Resource` supporting prioritized requests.
`PreemptiveResource` (env[, capacity])	A `PriorityResource` with preemption.
`Container` (env[, capacity, init])	Resource containing up to *capacity* of matter which may eithe
`Store` (env[, capacity])	Resource with *capacity* slots for storing arbitrary objects.
`PriorityItem` (priority, item)	Wrap an arbitrary *item* with an order-able *priority*.
`PriorityStore` (env[, capacity])	Resource with *capacity* slots for storing objects in priority orde

`FilterStore`(env[, capacity])	Resource with *capacity* slots for storing arbitrary objects supp

Exceptions

`SimPyException`	Base class for all SimPy specific exceptions.
`Interrupt`(cause)	Exception thrown into a process if it is interrupted (see `interrupt()`).

`simpy.core` — SimPy's core components

Core components for event-discrete simulation environments.

class **simpy.core.Environment**(*initial_time: int | float = 0*)

Execution environment for an event-based simulation. The passing of time is simulated by stepping from event to event.

You can provide an *initial_time* for the environment. By default, it starts at `0`.

This class also provides aliases for common event types, for example `process`, `timeout` and `event`.

property **now**: *int* | *float*

The current simulation time.

property **active_process**: *Process* | *None*

The currently active process of the environment.

process

alias of `Process`

timeout

alias of `Timeout`

event

alias of `Event`

all_of

alias of `AllOf`

any_of

alias of `AnyOf`

schedule(*event: Event*, *priority: EventPriority = 1*, *delay: int | float = 0*) → None

Schedule an *event* with a given *priority* and a *delay*.

peek() → int | float

Get the time of the next scheduled event. Return `Infinity` if there is no further event.

step()→ None

Process the next event.

Raise an `EmptySchedule` if no further events are available.

run(*until: int | float | Event | None = None*)→ Any | None

Executes `step()` until the given criterion *until* is met.

- If it is `None` (which is the default), this method will return when there are no further events to be processed.
- If it is an `Event`, the method will continue stepping until this event has been triggered and will return its value. Raises a `RuntimeError` if there are no further events to be processed and the *until* event was not triggered.
- If it is a number, the method will continue stepping until the environment's time reaches *until*.

class **simpy.core.BoundClass**(*cls: Type[T]*)

Allows classes to behave like methods.

The `__get__()` descriptor is basically identical to `function.__get__()` and binds the first argument of the `cls` to the descriptor instance.

static **bind_early**(*instance: object*)→ None

Bind all `BoundClass` attributes of the *instance's* class to the instance itself to increase performance.

class **simpy.core.EmptySchedule**

Thrown by an `Environment` if there are no further events to be processed.

simpy.core.Infinity: *float* = *inf*

Convenience alias for infinity

`simpy.exceptions` — Exception types used by SimPy

SimPy specific exceptions.

exception **simpy.exceptions.SimPyException**

Base class for all SimPy specific exceptions.

exception **simpy.exceptions.Interrupt**(*cause: Any | None*)

Exception thrown into a process if it is interrupted (see `interrupt()`).

`cause` provides the reason for the interrupt, if any.

If a process is interrupted concurrently, all interrupts will be thrown into the process in the same order as they occurred.

property **cause**: *Any | None*

The cause of the interrupt or `None` if no cause was provided.

`simpy.events` — Core event types

This module contains the basic event types used in SimPy.

The base class for all events is `Event`. Though it can be directly used, there are several specialized subclasses of it.

`Event`(env)	An event that may happen at some point in time.
`Timeout`(env, delay[, value])	A `Event` that gets processed after a *delay* has passed.
`Process`(env, generator)	Process an event yielding generator.
`AnyOf`(env, events)	A `Condition` event that is triggered if any of a list of *events* has been successfully tri
`AllOf`(env, events)	A `Condition` event that is triggered if all of a list of *events* have been successfully tri

`simpy.events.PENDING` *= object()*

Unique object to identify pending values of events.

`simpy.events.URGENT`: *EventPriority = 0*

Priority of interrupts and process initialization events.

`simpy.events.NORMAL`: *EventPriority = 1*

Default priority used by events.

class **`simpy.events.Event`**(*env: Environment*)

An event that may happen at some point in time.

An event

- may happen (`triggered` is `False`),
- is going to happen (`triggered` is `True`) or
- has happened (`processed` is `True`).

Every event is bound to an environment *env* and is initially not triggered. Events are scheduled for processing by the environment after they are triggered by either `succeed()`, `fail()` or `trigger()`. These methods also set the *ok* flag and the *value* of the event.

An event has a list of `callbacks`. A callback can be any callable. Once an event gets processed, all callbacks will be invoked with the event as the single argument. Callbacks can check if the event was successful by examining *ok* and do further processing with the *value* it has produced.

Failed events are never silently ignored and will raise an exception upon being processed. If a callback handles an exception, it must set `defused` to `True` to prevent this.

This class also implements `__and__()` (`&`) and `__or__()` (`|`). If you concatenate two events using one of these operators, a `Condition` event is generated that lets you wait for both or one of them.

env

The `Environment` the event lives in.

callbacks: *List[Callable[[EventType], None]]*

List of functions that are called when the event is processed.

property **triggered**: *bool*

Becomes `True` if the event has been triggered and its callbacks are about to be invoked.

property **processed**: *bool*

Becomes `True` if the event has been processed (e.g., its callbacks have been invoked).

property **ok**: *bool*

Becomes `True` when the event has been triggered successfully.

A “successful” event is one triggered with `succeed()`.

Raises: AttributeError – if accessed before the event is triggered.

property **defused**: *bool*

Becomes `True` when the failed event's exception is "defused".

When an event fails (i.e. with `fail()`), the failed event's *value* is an exception that will be re-raised when the `Environment` processes the event (i.e. in `step()`).

It is also possible for the failed event's exception to be defused by setting `defused` to `True` from an event callback. Doing so prevents the event's exception from being re-raised when the event is processed by the `Environment`.

property **value**: *Any* | *None*

The value of the event if it is available.

The value is available when the event has been triggered.

Raises `AttributeError` if the value is not yet available.

trigger(*event: Event*)→ None

Trigger the event with the state and value of the provided *event*. Return *self* (this event instance).

This method can be used directly as a callback function to trigger chain reactions.

succeed(*value: Any | None = None*)→ Event

Set the event's value, mark it as successful and schedule it for processing by the environment. Returns the event instance.

Raises `RuntimeError` if this event has already been triggerd.

fail(*exception: Exception*)→ Event

Set *exception* as the events value, mark it as failed and schedule it for processing by the environment. Returns the event instance.

Raises `TypeError` if *exception* is not an `Exception`.

Raises `RuntimeError` if this event has already been triggered.

class **simpy.events.Timeout**(*env: Environment, delay: SimTime, value: Any | None = None*)

A `Event` that gets processed after a *delay* has passed.

This event is automatically triggered when it is created.

env

The `Environment` the event lives in.

`callbacks`: *List[Callable[[EventType], None]]*

List of functions that are called when the event is processed.

property **`defused`**: *bool*

Becomes `True` when the failed event's exception is "defused".

When an event fails (i.e. with `fail()`), the failed event's *value* is an exception that will be re-raised when the `Environment` processes the event (i.e. in `step()`).

It is also possible for the failed event's exception to be defused by setting `defused` to `True` from an event callback. Doing so prevents the event's exception from being re-raised when the event is processed by the `Environment`.

`fail`(*exception: Exception*)→ Event

Set *exception* as the events value, mark it as failed and schedule it for processing by the environment. Returns the event instance.

Raises `TypeError` if *exception* is not an `Exception`.

Raises `RuntimeError` if this event has already been triggered.

property **ok**: *bool*

Becomes `True` when the event has been triggered successfully.

A "successful" event is one triggered with `succeed()`.

Raises: AttributeError – if accessed before the event is triggered.

property **`processed`**: *bool*

Becomes `True` if the event has been processed (e.g., its callbacks have been invoked).

`succeed`(*value: Any | None = None*)→ Event

Set the event's value, mark it as successful and schedule it for processing by the environment. Returns the event instance.

Raises `RuntimeError` if this event has already been triggerd.

`trigger`(*event: Event*)→ None

Trigger the event with the state and value of the provided *event*. Return *self* (this event instance).

This method can be used directly as a callback function to trigger chain reactions.

property **`triggered`**: *bool*

Becomes `True` if the event has been triggered and its callbacks are about to be invoked.

property **`value`**: *Any* | *None*

The value of the event if it is available.

The value is available when the event has been triggered.

Raises `AttributeError` if the value is not yet available.

class `simpy.events.Initialize`(*env: Environment, process: Process*)

Initializes a process. Only used internally by `Process`.

This event is automatically triggered when it is created.

`env`

The `Environment` the event lives in.

`callbacks`: *List[Callable[[EventType], None]]*

List of functions that are called when the event is processed.

property **`defused`**: *bool*

Becomes `True` when the failed event's exception is "defused".

When an event fails (i.e. with `fail()`), the failed event's *value* is an exception that will be re-raised when the `Environment` processes the event (i.e. in `step()`).

It is also possible for the failed event's exception to be defused by setting `defused` to `True` from an event callback. Doing so prevents the event's exception from being re-raised when the event is processed by the `Environment`.

`fail`(*exception: Exception*)→ Event

Set *exception* as the events value, mark it as failed and schedule it for processing by the environment. Returns the event instance.

Raises `TypeError` if *exception* is not an `Exception`.

Raises `RuntimeError` if this event has already been triggered.

property **ok**: *bool*

Becomes `True` when the event has been triggered successfully.

A “successful” event is one triggered with `succeed()`.

Raises: AttributeError – if accessed before the event is triggered.

property **processed**: *bool*

Becomes `True` if the event has been processed (e.g., its callbacks have been invoked).

succeed(*value: Any | None = None*)→ Event

Set the event’s value, mark it as successful and schedule it for processing by the environment. Returns the event instance.

Raises `RuntimeError` if this event has already been triggerd.

trigger(*event: Event*)→ None

Trigger the event with the state and value of the provided *event*. Return *self* (this event instance).

This method can be used directly as a callback function to trigger chain reactions.

property **triggered**: *bool*

Becomes `True` if the event has been triggered and its callbacks are about to be invoked.

property **value**: *Any | None*

The value of the event if it is available.

The value is available when the event has been triggered.

Raises `AttributeError` if the value is not yet available.

class **simpy.events.Interruption**(*process: Process, cause: Any | None*)

Immediately schedules an `Interrupt` exception with the given *cause* to be thrown into *process*.

This event is automatically triggered when it is created.

env

The `Environment` the event lives in.

`callbacks`: *List[Callable[[EventType], None]]*

List of functions that are called when the event is processed.

property **`defused`**: *bool*

Becomes `True` when the failed event's exception is "defused".

When an event fails (i.e. with `fail()`), the failed event's *value* is an exception that will be re-raised when the `Environment` processes the event (i.e. in `step()`).

It is also possible for the failed event's exception to be defused by setting `defused` to `True` from an event callback. Doing so prevents the event's exception from being re-raised when the event is processed by the `Environment`.

`fail`(*exception: Exception*)→ Event

Set *exception* as the events value, mark it as failed and schedule it for processing by the environment. Returns the event instance.

Raises `TypeError` if *exception* is not an `Exception`.

Raises `RuntimeError` if this event has already been triggered.

property **`ok`**: *bool*

Becomes `True` when the event has been triggered successfully.

A "successful" event is one triggered with `succeed()`.

Raises: AttributeError – if accessed before the event is triggered.

property **`processed`**: *bool*

Becomes `True` if the event has been processed (e.g., its callbacks have been invoked).

`succeed`(*value: Any | None = None*)→ Event

Set the event's value, mark it as successful and schedule it for processing by the environment. Returns the event instance.

Raises `RuntimeError` if this event has already been triggerd.

`trigger`(*event: Event*)→ None

Trigger the event with the state and value of the provided *event*. Return *self* (this event instance).

This method can be used directly as a callback function to trigger chain reactions.

property **`triggered`**: *bool*

Becomes `True` if the event has been triggered and its callbacks are about to be invoked.

property **`value`**: *Any* | *None*

The value of the event if it is available.

The value is available when the event has been triggered.

Raises `AttributeError` if the value is not yet available.

class **`simpy.events.Process`**(*env: Environment*, *generator: ProcessGenerator*)

Process an event yielding generator.

A generator (also known as a coroutine) can suspend its execution by yielding an event. `Process` will take care of resuming the generator with the value of that event once it has happened. The exception of failed events is thrown into the generator.

`Process` itself is an event, too. It is triggered, once the generator returns or raises an exception. The value of the process is the return value of the generator or the exception, respectively.

Processes can be interrupted during their execution by `interrupt()`.

`env`

The `Environment` the event lives in.

`callbacks`: *List[Callable[[EventType], None]]*

List of functions that are called when the event is processed.

property **target**: *Event*

The event that the process is currently waiting for.

Returns `None` if the process is dead, or it is currently being interrupted.

property **name**: *str*

Name of the function used to start the process.

property **is_alive**: *bool*

`True` until the process generator exits.

interrupt(*cause: Any | None = None*)→ None

Interrupt this process optionally providing a *cause*.

A process cannot be interrupted if it already terminated. A process can also not interrupt itself. Raise a `RuntimeError` in these cases.

property **defused**: *bool*

Becomes `True` when the failed event's exception is "defused".

When an event fails (i.e. with `fail()`), the failed event's *value* is an exception that will be re-raised when the `Environment` processes the event (i.e. in `step()`).

It is also possible for the failed event's exception to be defused by setting `defused` to `True` from an event callback. Doing so prevents the event's exception from being re-raised when the event is processed by the `Environment`.

fail(*exception: Exception*)→ Event

Set *exception* as the events value, mark it as failed and schedule it for processing by the environment. Returns the event instance.

Raises `TypeError` if *exception* is not an `Exception`.

Raises `RuntimeError` if this event has already been triggered.

property **ok**: *bool*

Becomes `True` when the event has been triggered successfully.

A “successful” event is one triggered with `succeed()`.

Raises: AttributeError – if accessed before the event is triggered.

property **processed**: *bool*

Becomes `True` if the event has been processed (e.g., its callbacks have been invoked).

succeed(*value: Any | None = None*)→ Event

Set the event’s value, mark it as successful and schedule it for processing by the environment. Returns the event instance.

Raises `RuntimeError` if this event has already been triggerd.

trigger(*event: Event*)→ None

Trigger the event with the state and value of the provided *event*. Return *self* (this event instance).

This method can be used directly as a callback function to trigger chain reactions.

property **triggered**: *bool*

Becomes `True` if the event has been triggered and its callbacks are about to be invoked.

property **value**: *Any | None*

The value of the event if it is available.

The value is available when the event has been triggered.

Raises `AttributeError` if the value is not yet available.

class **simpy.events.Condition**(*env: Environment, evaluate: Callable[[Tuple[Event, ...], int], bool], events: Iterable[Event]*)

An event that gets triggered once the condition function *evaluate* returns `True` on the given list of *events*.

The value of the condition event is an instance of `ConditionValue` which allows convenient access to the input events and their values. The `ConditionValue` will only contain entries for those events that occurred before the condition is processed.

If one of the events fails, the condition also fails and forwards the exception of the failing event.

The *evaluate* function receives the list of target events and the number of processed events in this list: `evaluate(events, processed_count)`. If it returns `True`, the condition is triggered. The `Condition.all_events()` and `Condition.any_events()` functions are used to implement *and* (`&`) and *or* (`|`) for events.

Condition events can be nested.

static **`all_events`**(*events: Tuple[Event, ...], count: int*)→ bool

An evaluation function that returns `True` if all *events* have been triggered.

static **`any_events`**(*events: Tuple[Event, ...], count: int*)→ bool

An evaluation function that returns `True` if at least one of *events* has been triggered.

property **`defused`**: *bool*

Becomes `True` when the failed event's exception is "defused".

When an event fails (i.e. with `fail()`), the failed event's *value* is an exception that will be re-raised when the `Environment` processes the event (i.e. in `step()`).

It is also possible for the failed event's exception to be defused by setting `defused` to `True` from an event callback. Doing so prevents the event's exception from being re-raised when the event is processed by the `Environment`.

`fail`(*exception: Exception*)→ Event

Set *exception* as the events value, mark it as failed and schedule it for processing by the environment. Returns the event instance.

Raises `TypeError` if *exception* is not an `Exception`.

Raises `RuntimeError` if this event has already been triggered.

property **ok**: *bool*

Becomes `True` when the event has been triggered successfully.

A “successful” event is one triggered with `succeed()`.

Raises: AttributeError – if accessed before the event is triggered.

property **processed**: *bool*

Becomes `True` if the event has been processed (e.g., its callbacks have been invoked).

succeed(*value: Any | None = None*)→ Event

Set the event’s value, mark it as successful and schedule it for processing by the environment. Returns the event instance.

Raises `RuntimeError` if this event has already been triggerd.

trigger(*event: Event*)→ None

Trigger the event with the state and value of the provided *event*. Return *self* (this event instance).

This method can be used directly as a callback function to trigger chain reactions.

property **triggered**: *bool*

Becomes `True` if the event has been triggered and its callbacks are about to be invoked.

property **value**: *Any | None*

The value of the event if it is available.

The value is available when the event has been triggered.

Raises `AttributeError` if the value is not yet available.

callbacks: *List[Callable[[EventType], None]]*

List of functions that are called when the event is processed.

env

The `Environment` the event lives in.

class **simpy.events.AllOf**(*env: Environment*, *events: Iterable[Event]*)

A `Condition` event that is triggered if all of a list of *events* have been successfully triggered. Fails immediately if any of *events* failed.

property **defused**: *bool*

Becomes `True` when the failed event's exception is "defused".

When an event fails (i.e. with `fail()`), the failed event's *value* is an exception that will be re-raised when the `Environment` processes the event (i.e. in `step()`).

It is also possible for the failed event's exception to be defused by setting `defused` to `True` from an event callback. Doing so prevents the event's exception from being re-raised when the event is processed by the `Environment`.

fail(*exception: Exception*) → Event

Set *exception* as the events value, mark it as failed and schedule it for processing by the environment. Returns the event instance.

Raises `TypeError` if *exception* is not an `Exception`.

Raises `RuntimeError` if this event has already been triggered.

property **ok**: *bool*

Becomes `True` when the event has been triggered successfully.

A "successful" event is one triggered with `succeed()`.

Raises: AttributeError – if accessed before the event is triggered.

property **processed**: *bool*

Becomes `True` if the event has been processed (e.g., its callbacks have been invoked).

succeed(*value: Any | None = None*) → Event

Set the event's value, mark it as successful and schedule it for processing by the environment. Returns the event instance.

Raises `RuntimeError` if this event has already been triggerd.

trigger(*event: Event*) → None

Trigger the event with the state and value of the provided *event*. Return *self* (this event instance).

This method can be used directly as a callback function to trigger chain reactions.

property **`triggered`**: *bool*

Becomes `True` if the event has been triggered and its callbacks are about to be invoked.

property **`value`**: *Any* | *None*

The value of the event if it is available.

The value is available when the event has been triggered.

Raises `AttributeError` if the value is not yet available.

`callbacks`: *List[Callable[[EventType], None]]*

List of functions that are called when the event is processed.

`env`

The `Environment` the event lives in.

class **`simpy.events.AnyOf`**(*env: Environment, events: Iterable[Event]*)

A `Condition` event that is triggered if any of a list of *events* has been successfully triggered. Fails immediately if any of *events* failed.

property **`defused`**: *bool*

Becomes `True` when the failed event's exception is "defused".

When an event fails (i.e. with `fail()`), the failed event's *value* is an exception that will be re-raised when the `Environment` processes the event (i.e. in `step()`).

It is also possible for the failed event's exception to be defused by setting `defused` to `True` from an event callback. Doing so prevents the event's exception from being re-raised when the event is processed by the `Environment`.

`fail`(*exception: Exception*)→ Event

Set *exception* as the events value, mark it as failed and schedule it for processing by the environment. Returns the event instance.

Raises `TypeError` if *exception* is not an `Exception`.

Raises `RuntimeError` if this event has already been triggered.

property **ok**: *bool*

Becomes `True` when the event has been triggered successfully.

A “successful” event is one triggered with `succeed()`.

Raises: AttributeError – if accessed before the event is triggered.

property **processed**: *bool*

Becomes `True` if the event has been processed (e.g., its callbacks have been invoked).

succeed(*value: Any | None = None*)→ Event

Set the event’s value, mark it as successful and schedule it for processing by the environment. Returns the event instance.

Raises `RuntimeError` if this event has already been triggerd.

trigger(*event: Event*)→ None

Trigger the event with the state and value of the provided *event*. Return *self* (this event instance).

This method can be used directly as a callback function to trigger chain reactions.

property **triggered**: *bool*

Becomes `True` if the event has been triggered and its callbacks are about to be invoked.

property **value**: *Any | None*

The value of the event if it is available.

The value is available when the event has been triggered.

Raises `AttributeError` if the value is not yet available.

callbacks: *List[Callable[[EventType], None]]*

List of functions that are called when the event is processed.

env

The `Environment` the event lives in.

class **simpy.events.ConditionValue**

Result of a `Condition`. It supports convenient dict-like access to the triggered events and their values. The events are ordered by their occurrences in the condition.

`simpy.resources` — Shared resource primitives

SimPy implements three types of resources that can be used to synchronize processes or to model congestion points:

`resource`	Shared resources supporting priorities and preemption.
`container`	Resource for sharing homogeneous matter between processes, either continuous (like v
`store`	Shared resources for storing a possibly unlimited amount of objects supporting request

They are derived from the base classes defined in the `base` module. These classes are also meant to support the implementation of custom resource types.

Resources — `simpy.resources.resource`

Shared resources supporting priorities and preemption.

These resources can be used to limit the number of processes using them concurrently. A process needs to *request* the usage right to a resource. Once the usage right is not needed any more it has to be *released*. A gas station can be modelled as a resource with a limited amount of fuel-pumps. Vehicles arrive at the gas station and request to use a fuel-pump. If all fuel-pumps are in use, the vehicle needs to wait until one of the users has finished refueling and releases its fuel-pump.

These resources can be used by a limited number of processes at a time. Processes *request* these resources to become a user and have to *release* them once they are done. For example, a gas station with a limited number of fuel pumps can be modeled with a *Resource*. Arriving vehicles request a fuel-pump. Once one is available they refuel. When they are done, the release the fuel-pump and leave the gas station.

Requesting a resource is modelled as “putting a process’ token into the resources” and releasing a resources correspondingly as “getting a process’ token out of the resource”. Thus, calling `request()` / `release()` is equivalent to calling `put()` / `get()`. Note, that releasing a resource will always succeed immediately, no matter if a process is actually using a resource or not.

Besides `Resource`, there is a `PriorityResource`, where processes can define a request priority, and a `PreemptiveResource` whose resource users can be preempted by requests with a higher priority.

class **simpy.resources.resource.Resource**(*env: Environment*, *capacity: int = 1*)

Resource with *capacity* of usage slots that can be requested by processes.

If all slots are taken, requests are enqueued. Once a usage request is released, a pending request will be triggered.

The *env* parameter is the `Environment` instance the resource is bound to.

users: *List*[*Request*]

List of `Request` events for the processes that are currently using the resource.

queue

Queue of pending `Request` events. Alias of `put_queue`.

property **count**: *int*

Number of users currently using the resource.

request

alias of `Request`

release

alias of `Release`

class **simpy.resources.resource.PriorityResource**(*env: Environment*, *capacity: int = 1*)

A `Resource` supporting prioritized requests.

Pending requests in the `queue` are sorted in ascending order by their *priority* (that means lower values are more important).

PutQueue

Type of the put queue. See `put_queue` for details.

alias of `SortedQueue`

GetQueue

Type of the get queue. See `get_queue` for details.

alias of `list`

request

alias of `PriorityRequest`

release

alias of `Release`

class **simpy.resources.resource.PreemptiveResource**(*env: Environment*, *capacity: int = 1*)

A `PriorityResource` with preemption.

If a request is preempted, the process of that request will receive an `Interrupt` with a `Preempted` instance as cause.

users: *List[PriorityRequest]*

List of `Request` events for the processes that are currently using the resource.

class **simpy.resources.resource.Preempted**(*by: Process | None*, *usage_since: SimTime | None*, *resource: Resource*)

Cause of a preemption `Interrupt` containing information about the preemption.

by

The preempting `simpy.events.Process`.

usage_since

The simulation time at which the preempted process started to use the resource.

resource

The resource which was lost, i.e., caused the preemption.

class **simpy.resources.resource.Request**(*resource: ResourceType*)

Request usage of the *resource*. The event is triggered once access is granted. Subclass of `simpy.resources.base.Put`.

If the maximum capacity of users has not yet been reached, the request is triggered immediately. If the maximum capacity has been reached, the request is triggered once an earlier usage request on the resource is released.

The request is automatically released when the request was created within a `with` statement.

usage_since: *SimTime | None* = None

The time at which the request succeeded.

class **simpy.resources.resource.PriorityRequest**(*resource: Resource*, *priority: int = 0*, *preempt: bool = True*)

Request the usage of *resource* with a given *priority*. If the *resource* supports preemption and *preempt* is `True` other usage requests of the *resource* may be preempted (see `PreemptiveResource` for details).

This event type inherits `Request` and adds some additional attributes needed by `PriorityResource` and `PreemptiveResource`

`priority`

The priority of this request. A smaller number means higher priority.

`preempt`

Indicates whether the request should preempt a resource user or not (`PriorityResource` ignores this flag).

`time`

The time at which the request was made.

`key`

Key for sorting events. Consists of the priority (lower value is more important), the time at which the request was made (earlier requests are more important) and finally the preemption flag (preempt requests are more important).

class **`simpy.resources.resource.Release`**(*resource: Resource, request: Request*)

Releases the usage of *resource* granted by *request*. This event is triggered immediately. Subclass of `simpy.resources.base.Get`.

`request`

The request (`Request`) that is to be released.

class **`simpy.resources.resource.SortedQueue`**(*maxlen: int | None = None*)

Queue for sorting events by their `key` attribute.

`maxlen`

Maximum length of the queue.

`append`(*item: Any*)→ None

Sort *item* into the queue.

Raise a `RuntimeError` if the queue is full.

Containers – `simpy.resources.container`

Resource for sharing homogeneous matter between processes, either continuous (like water) or discrete (like apples).

A `Container` can be used to model the fuel tank of a gasoline station. Tankers increase and refuelled cars decrease the amount of gas in the station's fuel tanks.

class **`simpy.resources.container.Container`**(*env: Environment, capacity: int | float = inf, init: int | float = 0*)

Resource containing up to *capacity* of matter which may either be continuous (like water) or discrete (like apples). It supports requests to put or get matter into/from the container.

The *env* parameter is the `Environment` instance the container is bound to.

The *capacity* defines the size of the container. By default, a container is of unlimited size. The initial amount of matter is specified by *init* and defaults to `0`.

Raise a `ValueError` if `capacity <= 0`, `init < 0` or `init > capacity`.

property **`level`**: *int | float*

The current amount of the matter in the container.

`put`

alias of `ContainerPut`

`get`

alias of `ContainerGet`

class **`simpy.resources.container.ContainerPut`**(*container: Container, amount: int | float*)

Request to put *amount* of matter into the *container*. The request will be triggered once there is enough space in the *container* available.

Raise a `ValueError` if `amount <= 0`.

`amount`

The amount of matter to be put into the container.

class **`simpy.resources.container.ContainerGet`**(*container: Container, amount: int | float*)

Request to get *amount* of matter from the *container*. The request will be triggered once there is enough matter available in the *container*.

Raise a `ValueError` if `amount <= 0`.

amount

The amount of matter to be taken out of the container.

Stores — `simpy.resources.store`

Shared resources for storing a possibly unlimited amount of objects supporting requests for specific objects.

The `Store` operates in a FIFO (first-in, first-out) order. Objects are retrieved from the store in the order they were put in. The *get* requests of a `FilterStore` can be customized by a filter to only retrieve objects matching a given criterion.

class **simpy.resources.store.Store**(*env: Environment, capacity: float | int = inf*)

Resource with *capacity* slots for storing arbitrary objects. By default, the *capacity* is unlimited and objects are put and retrieved from the store in a first-in first-out order.

The *env* parameter is the `Environment` instance the container is bound to.

items: *List[Any]*

List of the items available in the store.

put

alias of `StorePut`

get

alias of `StoreGet`

property **capacity**: *float* | *int*

Maximum capacity of the resource.

class **simpy.resources.store.PriorityItem**(*priority, item*)

Wrap an arbitrary *item* with an order-able *priority*.

Pairs a *priority* with an arbitrary *item*. Comparisons of *PriorityItem* instances only consider the *priority* attribute, thus supporting use of unorderable items in a `PriorityStore` instance.

priority: *Any*

Priority of the item.

item: *Any*

The item to be stored.

class **simpy.resources.store.PriorityStore**(*env: Environment, capacity: float | int = inf*)

Resource with *capacity* slots for storing objects in priority order.

Unlike `Store` which provides first-in first-out discipline, `PriorityStore` maintains items in sorted order such that the smallest items value are retrieved first from the store.

All items in a *PriorityStore* instance must be order-able; which is to say that items must implement `__lt__()`. To use unorderable items with *PriorityStore*, use `PriorityItem`.

class **simpy.resources.store.FilterStore**(*env: Environment, capacity: float | int = inf*)

Resource with *capacity* slots for storing arbitrary objects supporting filtered get requests. Like the `Store`, the *capacity* is unlimited by default and objects are put and retrieved from the store in a first-in first-out order.

Get requests can be customized with a filter function to only trigger for items for which said filter function returns `True`.

Note

In contrast to `Store`, get requests of a `FilterStore` won't necessarily be triggered in the same order they were issued.

Example: The store is empty. *Process 1* tries to get an item of type *a*, *Process 2* an item of type *b*. Another process puts one item of type *b* into the store. Though *Process 2* made his request after *Process 1*, it will receive that new item because *Process 1* doesn't want it.

get

alias of `FilterStoreGet`

class **simpy.resources.store.StorePut**(*store: Store, item: Any*)

Request to put *item* into the *store*. The request is triggered once there is space for the item in the store.

item

The item to put into the store.

class **simpy.resources.store.StoreGet**(*resource: ResourceType*)

Request to get an *item* from the *store*. The request is triggered once there is an item available in the store.

class **simpy.resources.store.FilterStoreGet**(*resource: ~simpy.resources.store.FilterStore, filter: ~typing.Callable[[~typing.Any], bool] = <function FilterStoreGet.<lambda>>*)

Request to get an *item* from the *store* matching the *filter*. The request is triggered once there is such an item available in the store.

filter is a function receiving one item. It should return `True` for items matching the filter criterion. The default function returns `True` for all items, which makes the request to behave exactly like `StoreGet`.

filter

The filter function to filter items in the store.

Base classes – `simpy.resources.base`

Base classes of for SimPy's shared resource types.

`BaseResource` defines the abstract base resource. It supports *get* and *put* requests, which return `Put` and `Get` events respectively. These events are triggered once the request has been completed.

class **simpy.resources.base.BaseResource**(*env: Environment, capacity: float | int*)

Abstract base class for a shared resource.

You can `put()` something into the resources or `get()` something out of it. Both methods return an event that is triggered once the operation is completed. If a `put()` request cannot complete immediately (for example if the resource has reached a capacity limit) it is enqueued in the `put_queue` for later processing. Likewise for `get()` requests.

Subclasses can customize the resource by:

- providing custom `PutQueue` and `GetQueue` types,
- providing custom `Put` respectively `Get` events,
- and implementing the request processing behaviour through the methods `_do_get()` and `_do_put()`.

PutQueue

The type to be used for the `put_queue`. It is a plain `list` by default. The type must support index access (e.g. `__getitem__()` and `__len__()`) as well as provide `append()` and `pop()` operations.

alias of `list`

GetQueue

The type to be used for the `get_queue`. It is a plain `list` by default. The type must support index access (e.g. `__getitem__()` and `__len__()`) as well as provide `append()` and `pop()` operations.

alias of `list`

put_queue: *MutableSequence[PutType]*

Queue of pending *put* requests.

get_queue: *MutableSequence[GetType]*

Queue of pending *get* requests.

property **capacity**: *float* | *int*

Maximum capacity of the resource.

put

alias of `Put`

get

alias of `Get`

class **simpy.resources.base.Put**(*resource: ResourceType*)

Generic event for requesting to put something into the *resource*.

This event (and all of its subclasses) can act as context manager and can be used with the `with` statement to automatically cancel the request if an exception (like an `simpy.exceptions.Interrupt` for example) occurs:

```
with res.put(item) as request:
    yield request
```

cancel()→ None

Cancel this put request.

This method has to be called if the put request must be aborted, for example if a process needs to handle an exception like an `Interrupt`.

If the put request was created in a `with` statement, this method is called automatically.

class **simpy.resources.base.Get**(*resource: ResourceType*)

Generic event for requesting to get something from the *resource*.

This event (and all of its subclasses) can act as context manager and can be used with the `with` statement to automatically cancel the request if an exception (like an `simpy.exceptions.Interrupt` for example) occurs:

```
with res.get() as request:
    item = yield request
```

`cancel`()→ None

Cancel this get request.

This method has to be called if the get request must be aborted, for example if a process needs to handle an exception like an `Interrupt`.

If the get request was created in a `with` statement, this method is called automatically.

`simpy.rt` — Real-time simulation

Execution environment for events that synchronizes passing of time with the real-time (aka *wall-clock time*).

class **simpy.rt.RealtimeEnvironment**(*initial_time: int | float = 0, factor: float = 1.0, strict: bool = True*)

Execution environment for an event-based simulation which is synchronized with the real-time (also known as wall-clock time). A time step will take *factor* seconds of real time (one second by default). A step from `0` to `3` with a `factor=0.5` will, for example, take at least 1.5 seconds.

The `step()` method will raise a `RuntimeError` if a time step took too long to compute. This behaviour can be disabled by setting *strict* to `False`.

now

The current simulation time.

active_process

The currently active process of the environment.

factor

Scaling factor of the real-time.

strict

Running mode of the environment. `step()` will raise a `RuntimeError` if this is set to `True` and the processing of events takes too long.

process(*generator*)

Create a new `Process` instance for *generator*.

timeout(*delay*, *value=None*)

Return a new `Timeout` event with a *delay* and, optionally, a *value*.

event()

Return a new `Event` instance. Yielding this event suspends a process until another process triggers the event.

`all_of`(*events*)

Return a new `AllOf` condition for a list of *events*.

`any_of`(*events*)

Return a new `AnyOf` condition for a list of *events*.

`schedule`(*event: Event, priority: EventPriority = 1, delay: int | float = 0*)→ None

Schedule an *event* with a given *priority* and a *delay*.

`peek`()→ int | float

Get the time of the next scheduled event. Return `Infinity` if there is no further event.

`step`()→ None

Process the next event after enough real-time has passed for the event to happen.

The delay is scaled according to the real-time `factor`. With `strict` mode enabled, a `RuntimeError` will be raised, if the event is processed too slowly.

`sync`()→ None

Synchronize the internal time with the current wall-clock time.

This can be useful to prevent `step()` from raising an error if a lot of time passes between creating the RealtimeEnvironment and calling `run()` or `step()`.

`run`(*until: int | float | Event | None = None*)→ Any | None

Executes `step()` until the given criterion *until* is met.

- If it is `None` (which is the default), this method will return when there are no further events to be processed.
- If it is an `Event`, the method will continue stepping until this event has been triggered and will return its value. Raises a `RuntimeError` if there are no further events to be processed and the *until* event was not triggered.
- If it is a number, the method will continue stepping until the environment's time reaches *until*.

`simpy.util` — Utility functions for SimPy

A collection of utility functions:

`start_delayed`(env, generator, delay)	Return a helper process that starts another process for *gener*

simpy.util.start_delayed(*env: Environment, generator: Generator[Event, Any, Any], delay: int | float*) → Process

Return a helper process that starts another process for *generator* after a certain *delay*.

`process()` starts a process at the current simulation time. This helper allows you to start a process after a delay of *delay* simulation time units:

```
>>> from simpy import Environment
>>> from simpy.util import start_delayed
>>> def my_process(env, x):
...     print(f'{env.now}, {x}')
...     yield env.timeout(1)
...
>>> env = Environment()
>>> proc = start_delayed(env, my_process(env, 3), 5)
>>> env.run()
5, 3
```

Raise a `ValueError` if `delay <= 0`.

SimPy History & Change Log

SimPy was originally based on ideas from Simula and Simscript but uses standard Python. It combines two previous packages, SiPy, in Simula-Style (Klaus Müller) and SimPy, in Simscript style (Tony Vignaux and Chang Chui).

SimPy was based on efficient implementation of co-routines using Python's generators capability.

SimPy 3 introduced a completely new and easier-to-use API, but still relied on Python's generators as they proved to work very well.

The package has been hosted on Sourceforge.net since September 15th, 2002. In June 2012, the project moved to Bitbucket.org.

Changelog for SimPy

4.1.1 - 2023-11-12

- [FIX] `EventCallback` typing using `TypeVar`
- [FIX] suppress some pyright typecheck issues involving ClassVars
- [CHANGE] some inner exceptions are now raise from None
- [CHANGE] `Event.fail` raises `TypeError` if it is not passed an `Exception`
- [CHANGE] use ruff for code formatting and linting
- [CHANGE] add pyright typechecking to test suite
- [CHANGE] code refactoring for ruff conformance
- [DOCS] update examples to fix various lint issues

4.1.0 - 2023-11-05

- [BREAKING] Python 3.8 is the minimum supported version
- [BREAKING] Contemporary setuptools based packaging
- [NEW] Add `Process.name` property
- [FIX] Support Python 3.12
- [CHANGE] use PEP-563 postponed evaluation of annotations
- [CHANGE] remove __path__ munging for namespace package
- [DOCS] Fix machine shop example to avoid negative times
- [DOCS] Update to sphinx 7.2.6
- [DOCS] Remove sys.path hack in sphinx config

- [DOCS] Remove sphinx `TYPE_CHECKING` circular import hack
- [DOCS] Remove sphinx extensions circular import hack
- [DOCS] SimJulia renamed to ConcurrentSim.jl

4.0.2 - 2023-07-30

- [CHANGE] Tested with Python 3.9, 3.10, and 3.11
- [CHANGE] Improved docs w.r.t. triggered and processed events
- [CHANGE] Improved gas station example
- [FIX] ClassVar annotations in BaseResource
- [FIX] Documentation typos
- [FIX] Help static analyzers find exported symbols
- [FIX] `license_file` deprecation in setup.cfg
- [FIX] Do not re-annotate type of `__path__`
- [FIX] Annotate `ConditionValue.__init__()` return value
- [FIX] Unbreak docs build by updating to Sphinx 6.2.1
- [FIX] Workaround Sphinx circular import problem

4.0.1 - 2020-04-15

- [FIX] Typing repair for Get and Put as ContextManagers

4.0.0 - 2020-04-06

- [BREAKING] Python 3.6 is the minimum supported version
- [BREAKING] `BaseEnvironment` is eliminated. Inherit `Environment` instead.
- [BREAKING] `Environment.exit()` is eliminated. Use `return` instead.
- [NEW] “Porting from SimPy 3 to 4” topical guide in docs
- [NEW] SimPy is now fully type annotated (PEP-483, PEP-484)
- [NEW] PEP-517/PEP-518 compatible build system

3.0.13 - 2020-04-05

- [FIX] Repair several minor typos in documentation
- [FIX] Possible AttributeError in Process._resume()
- [CHANGE] Use setuptools_scm in distribution build

3.0.12 - 2020-03-12

- [FIX] Fix URLs for GitLab.com and re-release

3.0.11 - 2018-07-13

- [FIX] Repair Environment.exit() to support PEP-479 and Python 3.7.
- [FIX] Fix wrong usage_since calculation in preemptions

- [NEW] Add “Time and Scheduling” section to docs
- [CHANGE] Move Interrupt from events to exceptions
- [FIX] Various minor documentation improvements

3.0.10 – 2016-08-26

- [FIX] Conditions no longer leak callbacks on events (thanks to Peter Grayson).

3.0.9 – 2016-06-12

- [NEW] PriorityStore resource and performance benchmarks were implemented by Peter Grayson.
- [FIX] Support for identifying nested preemptions was added by Cristian Klein.

3.0.8 – 2015-06-23

- [NEW] Added a monitoring guide to the documentation.
- [FIX] Improved packaging (thanks to Larissa Reis).
- [FIX] Fixed and improved various test cases.

3.0.7 – 2015-03-01

- [FIX] State of resources and requests were inconsistent before the request has been processed (issue #62).
- [FIX] Empty conditions were never triggered (regression in 3.0.6, issue #63).
- [FIX] `Environment.run()` will fail if the until event does not get triggered (issue #64).
- [FIX] Callback modification during event processing is now prohibited (thanks to Andreas Beham).

3.0.6 - 2015-01-30

- [NEW] Guide to SimPy resources.
- [CHANGE] Improve performance of condition events.
- [CHANGE] Improve performance of filter store (thanks to Christoph Körner).
- [CHANGE] Exception tracebacks are now more compact.
- [FIX] `AllOf` conditions handle already processed events correctly (issue #52).
- [FIX] Add `sync()` to `RealtimeEnvironment` to reset its internal wall-clock reference time (issue #42).
- [FIX] Only send copies of exceptions into processes to prevent traceback modifications.
- [FIX] Documentation improvements.

3.0.5 – 2014-05-14

- [CHANGE] Move interruption and all of the safety checks into a new event (pull request #30)

- [FIX] `FilterStore.get()` now behaves correctly (issue #49).
- [FIX] Documentation improvements.

3.0.4 – 2014-04-07

- [NEW] Verified, that SimPy works on Python 3.4.
- [NEW] Guide to SimPy events
- [CHANGE] The result dictionary for condition events (`AllOF` / `&` and `AnyOf` / `|`) now is an *OrderedDict* sorted in the same way as the original events list.
- [CHANGE] Condition events now also except processed events.
- [FIX] `Resource.request()` directly after `Resource.release()` no longer successful. The process now has to wait as supposed to.
- [FIX] `Event.fail()` now accept all exceptions derived from `BaseException` instead of only `Exception`.

3.0.3 – 2014-03-06

- [NEW] Guide to SimPy basics.
- [NEW] Guide to SimPy Environments.
- [FIX] Timing problems with real time simulation on Windows (issue #46).
- [FIX] Installation problems on Windows due to Unicode errors (issue #41).
- [FIX] Minor documentation issues.

3.0.2 – 2013-10-24

- [FIX] The default capacity for `Container` and `FilterStore` is now also `inf`.

3.0.1 – 2013-10-24

- [FIX] Documentation and default parameters of `Store` didn't match. Its default capacity is now `inf`.

3.0 – 2013-10-11

SimPy 3 has been completely rewritten from scratch. Our main goals were to simplify the API and code base as well as making SimPy more flexible and extensible. Some of the most important changes are:

- Stronger focus on events. Processes yield event instances and are suspended until the event is triggered. An example for an event is a *timeout* (formerly known as *hold*), but even processes are now events, too (you can wait until a process terminates).
- Events can be combined with `&` (and) and `|` (or) to create *condition events*.
- Process can now be defined by any generator function. You don't have to subclass `Process` anymore.

- No more global simulation state. Every simulation stores its state in an *environment* which is comparable to the old `Simulation` class.
- Improved resource system with newly added resource types.
- Removed plotting and GUI capabilities. Pyside and matplotlib are much better with this.
- Greatly improved test suite. Its cleaner, and the tests are shorter and more numerous.
- Completely overhauled documentation.

There is a guide for porting from SimPy 2 to SimPy 3. If you want to stick to SimPy 2 for a while, change your requirements to `'SimPy>=2.3,<3'`.

All in all, SimPy has become a framework for asynchronous programming based on coroutines. It brings more than ten years of experience and scientific know-how in the field of event-discrete simulation to the world of asynchronous programming and should thus be a solid foundation for everything based on an event loop.

You can find information about older versions on the history page

Historical Releases

2.3.1 – 2012-01-28

- [NEW] More improvements on the documentation.
- [FIX] Syntax error in tkconsole.py when installing on Py3.2.
- [FIX] Added *mock* to the dep. list in SimPy.test().

2.3 – 2011-12-24

- [NEW] Support for Python 3.2. Support for Python <= 2.5 has been dropped.
- [NEW] SimPy.test() method to run the tests on the installed version of SimPy.
- [NEW] Tutorials/examples were integrated into the test suite.
- [CHANGE] Even more code clean-up (e.g., removed prints throughout the code, removed if-main-blocks, ...).
- [CHANGE] Many documentation improvements.

2.2 – 2011-09-27

- [CHANGE] Restructured package layout to be conform to the Hitchhiker's Guide to packaging
- [CHANGE] Tests have been ported to pytest.
- [CHANGE] Documentation improvements and clean-ups.
- [FIX] Fixed incorrect behavior of Store._put, thanks to Johannes Koomer for the fix.

2.1 – 2010-06-03

- [NEW] A function *step* has been added to the API. When called, it executes the next scheduled event. (*step* is actually a method of *Simulation*.)
- [NEW] Another new function is *peek*. It returns the time of the next event. By using peek and step together, one can easily write e.g. an interactive program to step through a simulation event by event.
- [NEW] A simple interactive debugger `stepping.py` has been added. It allows stepping through a simulation, with options to skip to a certain time, skip to the next event of a given process, or viewing the event list.
- [NEW] Versions of the Bank tutorials (documents and programs) using the advanced-
[NEW] object-oriented API have been added.
- [NEW] A new document describes tools for gaining insight into and debugging SimPy models.
- [CHANGE] Major re-structuring of SimPy code, resulting in much less SimPy code – great for the maintainers.
- [CHANGE] Checks have been added which test whether entities belong to the same Simulation instance.
- [CHANGE] The Monitor and Tally methods timeAverage and timeVariance now calculate only with the observed time-series. No value is assumed for the period prior to the first observation.
- [CHANGE] Changed class Lister so that circular references between objects no longer lead to stack overflow and crash.
- [FIX] Functions *allEventNotices* and *allEventTimes* are working again.
- [FIX] Error messages for methods in SimPy.Lib work again.

2.0.1 – 2009-04-06

- [NEW] Tests for real time behavior (testRT_Behavior.py and testRT_Behavior_OO.py in folder SimPy).
- [FIX] Repaired a number of coding errors in several models in the SimPyModels folder.
- [FIX] Repaired SimulationRT.py bug introduced by recoding for the OO API.
- [FIX] Repaired errors in sample programs in documents:
  - Simulation with SimPy - In Depth Manual
  - SimPy's Object Oriented API Manual
  - Simulation With Real Time Synchronization Manual
  - SimPlot Manual
  - Publication-quality Plot Production With Matplotlib Manual

2.0.0 – 2009-01-26

This is a major release with changes to the SimPy application programming interface (API) and the formatting of the documentation.

API changes

In addition to its existing API, SimPy now also has an object oriented API. The additional API

- allows running SimPy in parallel on multiple processors or multi-core CPUs,
- supports better structuring of SimPy programs,
- allows subclassing of class *Simulation* and thus provides users with the capability of creating new simulation modes/libraries like SimulationTrace, and
- reduces the total amount of SimPy code, thereby making it easier to maintain.

Note that the OO API is **in addition** to the old API. SimPy 2.0 is fully backward compatible.

Documentation format changes

SimPy's documentation has been restructured and processed by the Sphinx documentation generation tool. This has generated one coherent, well structured document which can be easily browsed. A search capability is included.

March 2008: Version 1.9.1

This is a bug-fix release which cures the following bugs:

- Excessive production of circular garbage, due to a circular reference between Process instances and event notices. This led to large memory requirements.
- Runtime error for preempts of processes holding multiple Resource objects.

It also adds a Short Manual, describing only the basic facilities of SimPy.

December 2007: Version 1.9

This is a major release with added functionality/new user API calls and bug fixes.

Major changes

- The event list handling has been changed to improve the runtime performance of large SimPy models (models with thousands of processes). The use of dictionaries for timestamps has been stopped. Thanks are due to Prof. Norm Matloff and a team of his students who did a study on improving SimPy performance. This was one of their recommendations. Thanks, Norm and guys! Furthermore, in version 1.9 the 'heapq' sorting package replaces 'bisect'. Finally, cancelling events no longer removes them, but rather marks them. When their event time comes, they are ignored. This was Tony Vignaux' idea!
- The Manual has been edited and given an easier-to-read layout.
- The Bank2 tutorial has been extended by models which use more advanced SimPy commands/constructs.

Bug fixes

- The tracing of 'activate' statements has been enabled.

Additions

- A method returning the time-weighted variance of observations has been added to classes Monitor and Tally.
- A shortcut activation method called “start” has been added to class Process.

January 2007: Version 1.8

Major Changes

- SimPy 1.8 and future releases will not run under the obsolete Python 2.2 version. They require Python 2.3 or later.
- The Manual has been thoroughly edited, restructured and rewritten. It is now also provided in PDF format.
- The Cheatsheet has been totally rewritten in a tabular format. It is provided in both XLS (MS Excel spreadsheet) and PDF format.
- The version of SimPy.Simulation(RT/Trace/Step) is now accessible by the variable ‘version’.
- The *__str__* method of Histogram was changed to return a table format.

Bug fixes

- Repaired a bug in *yield waituntil* runtime code.
- Introduced check for *capacity* parameter of a Level or a Store being a number > 0.
- Added code so that self.eventsFired gets set correctly after an event fires in a compound yield get/put with a waitevent clause (reneging case).
- Repaired a bug in prettyprinting of Store objects.

Additions

- New compound yield statements support time-out or event-based reneging in get and put operations on Store and Level instances.
- *yield get* on a Store instance can now have a filter function.
- All Monitor and Tally instances are automatically registered in list *allMonitors* and *allTallies*, respectively.
- The new function *startCollection* allows activation of Monitors and Tallies at a specified time.
- A *printHistogram* method was added to Tally and Monitor which generates a table-form histogram.
- In SimPy.SimulationRT: A function for allowing changing the ratio wall clock time to simulation time has been added.

June 2006: Version 1.7.1

This is a maintenance release. The API has not been changed/added to.

- Repair of a bug in the _get methods of Store and Level which could lead to synchronization problems (blocking of producer processes, despite space being available in the buffer).
- Repair of Level __init__ method to allow initialBuffered to be of either float or int type.
- Addition of type test for Level get parameter 'nrToGet' to limit it to positive int or float.
- To improve pretty-printed output of 'Level' objects, changed attribute '_nrBuffered' to 'nrBuffered' (synonym for 'amount' property).
- To improve pretty-printed output of 'Store' objects, added attribute 'buffered' (which refers to '_theBuffer' attribute).

February 2006: Version 1.7

This is a major release.

- Addition of an abstract class Buffer, with two sub-classes *Store* and *Level* Buffers are used for modelling inter-process synchronization in producer/ consumer and multi-process cooperation scenarios.
- Addition of two new *yield* statements:
  - *yield put* for putting items into a buffer, and
  - *yield get* for getting items from a buffer.
- The Manual has undergone a major re-write/edit.
- All scripts have been restructured for compatibility with IronPython 1 beta2. This was done by moving all *import* statements to the beginning of the scripts. After the removal of the first (shebang) line, all scripts (with the exception of plotting and GUI scripts) can run successfully under this new Python implementation.

September 2005: Version 1.6.1

This is a minor release.

- Addition of Tally data collection class as alternative to Monitor. It is intended for collecting very large data sets more efficiently in storage space and time than Monitor.
- Change of Resource to work with Tally (new Resource API is backwards-compatible with 1.6).
- Addition of function setHistogram to class Monitor for initializing histograms.
- New function allEventNotices() for debugging/teaching purposes. It returns a prettyprinted string with event times and names of process instances.
- Addition of function allEventTimes (returns event times of all scheduled events).

15 June 2005: Version 1.6

- Addition of two compound yield statement forms to support the modelling of processes reneging from resource queues.
- Addition of two test/demo files showing the use of the new reneging statements.
- Addition of test for prior simulation initialization in method activate().
- Repair of bug in monitoring thw waitQ of a resource when preemption occurs.

- Major restructuring/editing to Manual and Cheatsheet.

1 February 2005: Version 1.5.1

- MAJOR LICENSE CHANGE:

  Starting with this version 1.5.1, SimPy is being release under the GNU Lesser General Public License (LGPL), instead of the GNU GPL. This change has been made to encourage commercial firms to use SimPy in for-profit work.

- Minor re-release
- No additional/changed functionality
- Includes unit test file'MonitorTest.py' which had been accidentally deleted from 1.5
- Provides updated version of 'Bank.html' tutorial.
- Provides an additional tutorial ('Bank2.html') which shows how to use the new synchronization constructs introduced in SimPy 1.5.
- More logical, cleaner version numbering in files.

1 December 2004: Version 1.5

- No new functionality/API changes relative to 1.5 alpha
- Repaired bug related to waiting/queuing for multiple events
- SimulationRT: Improved synchronization with wallclock time on Unix/Linux

25 September 2004: Version 1.5alpha

- New functionality/API additions
  - SimEvents and signalling synchronization constructs, with 'yield waitevent' and 'yield queueevent' commands.
  - A general "wait until" synchronization construct, with the 'yield waituntil' command.
- No changes to 1.4.x API, i.e., existing code will work as before.

19 May 2004: Version 1.4.2

- Sub-release to repair two bugs:
  - The unittest for monitored Resource queues does not fail anymore.
  - SimulationTrace now works correctly with "yield hold,self" form.
- No functional or API changes

29 February 2004: Version 1.4.1

- Sub-release to repair two bugs:
  - The (optional) monitoring of the activeQ in Resource now works correctly.
  - The "cellphone.py" example is now implemented correctly.
- No functional or API changes

1 February 2004: Version 1.4

- Released on SourceForge.net

22 December 2003: Version 1.4 alpha

- New functionality/API changes
  - All classes in the SimPy API are now new style classes, i.e., they inherit from *object* either directly or indirectly.
  - Module *Monitor.py* has been merged into module *Simulation.py* and all *SimulationXXX.py* modules. Import of *Simulation* or any *SimulationXXX* module now also imports *Monitor*.
  - Some *Monitor* methods/attributes have changed. See Manual!
  - *Monitor* now inherits from *list*.
    - A class *Histogram* has been added to *Simulation.py* and all *SimulationXXX.py* modules.
    - A module *SimulationRT* has been added which allows synchronization between simulated and wallclock time.
    - A moduleSimulationStep which allows the execution of a simulation model event-by-event, with the facility to execute application code after each event.
    - A Tk/Tkinter-based module *SimGUI* has been added which provides a SimPy GUI framework.
    - A Tk/Tkinter-based module *SimPlot* has been added which provides for plot output from SimPy programs.

22 June 2003: Version 1.3

- No functional or API changes
- Reduction of sourcecode linelength in Simulation.py to <= 80 characters

June 2003: Version 1.3 alpha

- Significantly improved performance
- Significant increase in number of quasi-parallel processes SimPy can handle
- New functionality/API changes:
  - Addition of SimulationTrace, an event trace utility
  - Addition of Lister, a prettyprinter for instance attributes
  - No API changes
- Internal changes:
  - Implementation of a proposal by Simon Frost: storing the keys of the event set dictionary in a binary search tree using bisect. Thank you, Simon! SimPy 1.3 is dedicated to you!
- Update of Manual to address tracing.
- Update of Interfacing doc to address output visualization using Scientific Python gplt package.

29 April 2003: Version 1.2

- No changes in API.
- Internal changes:
  - Defined “True” and “False” in Simulation.py to support Python 2.2.

22 October 2002

- Re-release of 0.5 Beta on SourceForge.net to replace corrupted file __init__.py.
- No code changes whatever!

18 October 2002

- Version 0.5 Beta-release, intended to get testing by application developers and system integrators in preparation of first full (production) release. Released on SourceForge.net on 20 October 2002.
- More models
- Documentation enhanced by a manual, a tutorial (“The Bank”) and installation instructions.
- Major changes to the API:
  - Introduced ‘simulate(until=0)’ instead of ‘scheduler(till=0)’. Left ‘scheduler()’ in for backward compatibility, but marked as deprecated.
  - Added attribute “name” to class Process. Process constructor is now:

    ```
    def __init__(self,name="a_process")
    ```

    Backward compatible if keyword parameters used.
  - Changed Resource constructor to:

    ```
    def __init__(self,capacity=1,name="a_resource",unitName="units")
    ```

    Backward compatible if keyword parameters used.

27 September 2002

- Version 0.2 Alpha-release, intended to attract feedback from users
- Extended list of models
- Updated documentation

17 September 2002

- Version 0.1.2 published on SourceForge; fully working, pre-alpha code
- Implements simulation, shared resources with queuing (FIFO), and monitors for data gathering/analysis.
- Contains basic documentation (cheatsheet) and simulation models for test and demonstration.

Acknowledgments

SimPy 2 has been primarily developed by Stefan Scherfke and Ontje Lünsdorf, starting from SimPy 1.9. Their work has resulted in a most elegant combination of an object oriented API with the existing API, maintaining full backward compatibility. It has been quite easy to integrate their product into the existing SimPy code and documentation environment.

Thanks, guys, for this great job! **SimPy 2.0 is dedicated to you!**

SimPy was originally created by Klaus Müller and Tony Vignaux. They pushed its development for several years and built the SimPy community. Without them, there would be no SimPy 3.

Thanks, guys, for this great job! **SimPy 3.0 is dedicated to you!**

The many contributions of the SimPy user and developer communities are of course also gratefully acknowledged.

Ports and comparable libraries

Re-implementations of SimPy and libraries similar to SimPy are available in the following languages:

- C#: SimSharp (written by Andreas Beham)
- Julia: ConcurrentSim
- R: Simmer

Defense of Design

This document explains why SimPy is designed the way it is and how its design evolved over time.

Original Design of SimPy 1

SimPy 1 was heavily inspired by *Simula 67* and *Simscript*. The basic entity of the framework was a process. A process described a temporal sequence of actions.

In SimPy 1, you implemented a process by sub-classing `Process`. The instance of such a subclass carried both, process and simulation internal information, whereas the latter wasn't of any use to the process itself. The sequence of actions of the process was specified in a method of the subclass, called the *process execution method* (or PEM in short). A PEM interacted with the simulation by yielding one of several keywords defined in the simulation package.

The simulation itself was executed via module level functions. The simulation state was stored in the global scope. This made it very easy to implement and execute a simulation (despite from having to inherit from *Process* and instantiating the processes before starting their PEMs). However, having all simulation state global makes it hard to parallelize multiple simulations.

SimPy 1 also followed the "batteries included" approach, providing shared resources, monitoring, plotting, GUIs and multiple types of simulations ("normal", real-time, manual stepping, with tracing).

The following code fragment shows how a simple simulation could be implemented in SimPy 1:

```
from SimPy.Simulation import Process, hold, initialize, activate, simulate

class MyProcess(Process):
    def pem(self, repeat):
        for i in range(repeat):
            yield hold, self, 1

initialize()
proc = MyProcess()
activate(proc, proc.pem(3))
simulate(until=10)



sim = Simulation()
proc = MyProcess(sim=sim)
sim.activate(proc, proc.pem(3))
sim.simulate(until=10)
```

Changes in SimPy 2

SimPy 2 mostly stuck with SimPy 1's design, but added an object orient API for the execution of simulations, allowing them to be executed in parallel. Since processes and the simulation state were so closely coupled, you now needed to pass the `Simulation` instance into your process to "bind" them to that instance. Additionally, you still had to activate the process. If you forgot to pass the simulation instance, the process would use a global instance thereby breaking your program. SimPy 2's OO-API looked like this:

```
from SimPy.Simulation import Simulation, Process, hold

class MyProcess(Process):
    def pem(self, repeat):
        for i in range(repeat):
            yield hold, self, 1

sim = Simulation()
proc = MyProcess(sim=sim)
sim.activate(proc, proc.pem(3))
sim.simulate(until=10)
```

Changes and Decisions in SimPy 3

The original goals for SimPy 3 were to simplify and PEP8-ify its API and to clean up and modularize its internals. We knew from the beginning that our goals would not be achievable without breaking backwards compatibility with SimPy 2. However, we didn't expect the API changes to become as extensive as they ended up to be.

We also removed some of the included batteries, namely SimPy's plotting and GUI capabilities, since dedicated libraries like matplotlib or PySide do a much better job here.

However, by far the most changes are—from the end user's view—mostly syntactical. Thus, porting from 2 to 3 usually just means replacing a line of SimPy 2 code with its SimPy3 equivalent (e.g., replacing `yield hold, self, 1` with `yield env.timeout(1)`).

In short, the most notable changes in SimPy 3 are:

- No more sub-classing of `Process` required. PEMs can even be simple module level functions.
- The simulation state is now stored in an `Environment` which can also be used by a PEM to interact with the simulation.
- PEMs now yield event objects. This implicates interesting new features and allows an easy extension with new event types.

These changes are causing the above example to now look like this:

```
from simpy import Environment, simulate

def pem(env, repeat):
    for i in range(repeat):
        yield env.timeout(i)

env = Environment()
env.process(pem(env, 7))
simulate(env, until=10)
```

The following sections describe these changes in detail:

No More Sub-classing of `Process`

In SimPy 3, every Python generator can be used as a PEM, no matter if it is a module level function or a method of an object. This reduces the amount of code required for simple processes. The `Process` class still exists, but you don't need to instantiate it by yourself, though. More on that later.

Processes Live in an Environment

Process and simulation state are decoupled. An `Environment` holds the simulation state and serves as base API for processes to create new events. This allows you to implement advanced use cases by extending the `Process` or `Environment` class without affecting other components.

For the same reason, the `simulate()` method now is a module level function that takes an environment to simulate.

Stronger Focus on Events

In former versions, PEMs needed to yield one of SimPy's built-in keywords (like `hold`) to interact with the simulation. These keywords had to be imported separately and were bound to some internal functions that were tightly integrated with the `Simulation` and `Process` making it very hard to extend SimPy with new functionality.

In SimPy 3, PEMs just need to yield events. There are various built-in event types, but you can also create custom ones by making a subclass of a `BaseEvent`. Most events are generated by factory methods of `Environment`. For example, `Environment.timeout()` creates a `Timeout` event that replaces the `hold` keyword.

The `Process` is now also an event. You can now yield another process and wait for it to finish. For example, think of a car-wash simulation were "washing" is a process that the car processes can wait for once they enter the washing station.

Creating Events via the Environment or Resources

The `Environment` and resources have methods to create new events, e.g. `Environment.timeout()` or `Resource.request()`. Each of these methods maps to a certain event type. It creates a new instance of it and returns it, e.g.:

```
def event(self):
    return Event()
```

To simplify things, we wanted to use the event classes directly as methods:

```
class Environment(object)
    event = Event
```

This was, unfortunately, not directly possible and we had to wrap the classes to behave like bound methods. Therefore, we introduced a `BoundClass`:

```
class BoundClass(object):
    """Allows classes to behave like methods. The ``__get__()`` descriptor
    is basically identical to ``function.__get__()`` and binds the first
    argument of the ``cls`` to the descriptor instance.

    """
    def __init__(self, cls):
        self.cls = cls

    def __get__(self, obj, type=None):
        if obj is None:
            return self.cls
        return types.MethodType(self.cls, obj)


class Environment(object):
    event = BoundClass(Event)
```

These methods are called a lot, so we added the event classes as `types.MethodType` to the instance of `Environment` (or the resources, respectively):

```
class Environment(object):
    def __init__(self):
        self.event = types.MethodType(Event, self)
```

It turned out the the class attributes (the `BoundClass` instances) were now quite useless, so we removed them although it was actually the “right” way to to add classes as methods to another class.

Release Process

This process describes the steps to execute in order to release a new version of SimPy.

Preparations

1. Close all tickets for the next version.
2. Update the *minimum* required versions of dependencies in `setup.cfg`. Update the development dependencies in `requirements-dev.txt`.
3. Run **tox** from the project root. All tests for all supported versions must pass:

```
$ tox
[...]
________ summary ________
py38: commands succeeded
pypy3: commands succeeded
docs: commands succeeded
ruff: commands succeeded
mypy: commands succeeded
congratulations :)
```

4. Build the docs (HTML is enough). Make sure there are no errors and undefined references.

```
$ tox -e sphinx
```

5. Check if all authors are listed in `AUTHORS.txt`.
6. Update the change log (`CHANGES.rst`). Only keep changes for the current major release in `CHANGES.rst` and reference the history page from there.
7. Write a draft for the announcement mail with a list of changes, acknowledgements and installation instructions. Everyone in the team should agree with it.

Build and release

1. Make *local* tag for new release. This tag will be pushed *later*, after checking the build artifacts. If anything is not right, this tag can be deleted and recreated locally *before* pushing to GitLab.

```
$ git tag -a -m "Tag a.b.c release" a.b.c
```

2. Test the build process. Build a source distribution and a wheel package and test them:

```
$ python -m build
$ ls dist/
simpy-a.b.c-py2.py3-none-any.whl simpy-a.b.c.tar.gz
```

   Try installing them:

```
$ rm -rf /tmp/simpy-sdist  # ensure clean state if ran repeatedly
$ virtualenv /tmp/simpy-sdist
$ /tmp/simpy-sdist/bin/pip install dist/simpy-a.b.c.tar.gz
```

   and

```
$ rm -rf /tmp/simpy-wheel  # ensure clean state if ran repeatedly
$ virtualenv /tmp/simpy-wheel
$ /tmp/simpy-wheel/bin/pip install dist/simpy-a.b.c-py2.py3-none-any.whl
```

   It is also a good idea to inspect the contents of the distribution files:

```
$ tar tzf dist/simpy-a.b.c.tar.gz
```

```
$ unzip -l dist/simpy-a.b.c-py2.py3-none-any.whl
```

3. Create or check your accounts for the test server and PyPI. Update your `~/.pypirc` with your current credentials:

```
[distutils]
index-servers =
    pypi
    testpypi

[pypi]
username = <your pypi username>

[testpypi]
repository = https://test.pypi.org/legacy/
username = <your testpypi username>
```

4. Upload the distributions for the new version to the test server and test the installation again:

```
$ twine upload -r testpypi dist/simpy*a.b.c*
$ pip install -i https://test.pypi.org/simple/ simpy
```

5. Check if the package is displayed correctly on the test PyPI: https://test.pypi.org/project/simpy/
6. Push tag for a.b.c release to GitLab. Upon successful build and test, the GitLab CI pipeline will deploy the tagged release to the production PyPI service.

```
$ git push origin master a.b.c
```

7. Check the status of the GitLab CI pipeline: https://gitlab.com/team-simpy/simpy/pipelines
8. Check if the package is displayed correctly on PyPI: https://pypi.org/project/simpy/
9. Finally, test installation from PyPI:

```
$ pip install -U simpy
```

Post release

1. Activate the documentation build for the new version.
2. Send the prepared release announcement to the SimPy group.
3. Update Wikipedia entries.

License

The MIT License (MIT)

Index

A

B

C

D

E

F

G

I

K

L

M

N

O

P

Q

R

S

T

U

V