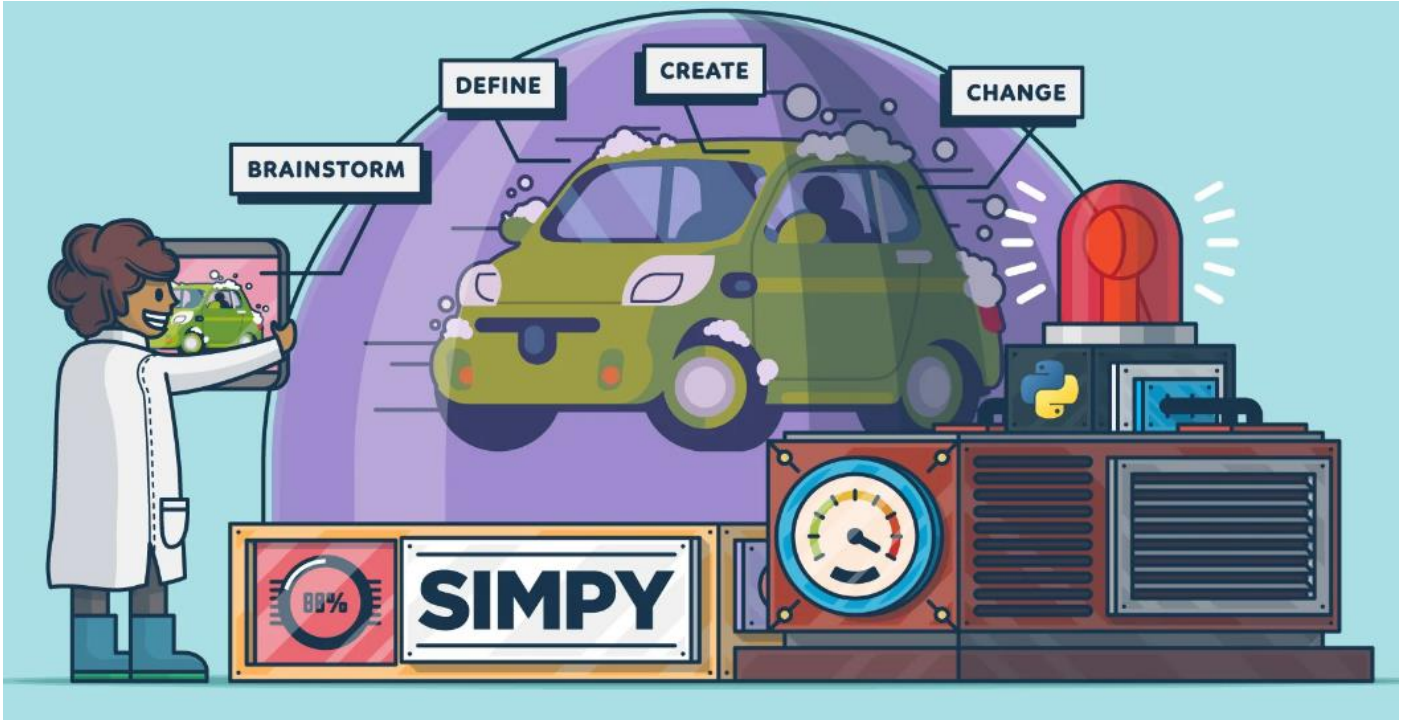



## Обзор фреймворка Simpy (*источник simpy.readthedocs.io* )

**Simpy** – фреймворк для моделирования дискретных процессов, работает в версиях Python >= 3.6.

Устанавливается по типовой схеме: pip install -U simpy

Также можно инсталлировать и в ручном режиме:

```
$ cd where/you/put/simpy/
$ python setup.py install
```

В этом фреймворке реализован сканирующий моделирующий алгоритм – моделирующий движок (engine).

Поведение активных элементов-инициаторов (транспортных средств, клиентов, сообщений и т.п.) моделируется ‘процессами’, происходящими в особом программном окружении (Environment). ‘Процессы’ взаимодействуют с этой средой и друг с другом через ‘события’.

| **Базовые понятия Simpy** | **Реализация** |
|---|---|
| Environment | Цикл событий (Event loop) |
| Process | Со-программа (Task / Coroutine) |
| Event | Фьючерс (Future / Promise / Deferred) |
| Resource | Семафор (Semaphore) |
| RealtimeEnvironment | Цикл в режиме реального времени |

‘Процессы’ описываются функциями-генераторами. Они могут быть реализованы в виде обычной функции или методом класса. ‘Процессы’ создают ‘события’, возвращают (**yield**) их и дожидаются следующего этапа своего выполнения. После генерации ‘процессом’ ‘события’, процесс приостанавливается.

Среда (Environment) Simpy возобновляет процесс, когда выполняется его активное событие, при этом несколько процессов могут ждать одно и то же событие.

Особым типом событий является Timeout. Это событие позволяет процессу находиться как бы в спящем режиме (пассивированый процесс), т.е. сохранять свое состояние в течение заданного модельного времени. Timeout, как и другие события, могут создаваться при вызове специального метода модельной среды, в которой находится сам процесс - ***Environment.timeout()***

Алгоритм действий процесса обычно задаётся в методе класса, называемом методом исполнения процесса или сокращённо PEM - Process Execution Method, обычно в виде метода run(). PEM взаимодействует с моделирующим движком, выдавая одно из нескольких своих ключевых событий (**yield**), определенных в этом модельном процессе.

Собственно моделирование выполняется с помощью функций базового ядра среды. Состояние процессов модели сохраняется в глобальной области параметров. Это упрощает реализацию модели и выполнение в модели с наследованием от процесса и созданием экземпляров процессов перед запуском их PEM. Однако, можно отметить, что наличие глобального состояния всей модели затрудняет распараллеливание модельного алгоритма.

Наиболее заметными особенностями фреймворка SimPy являются:

-- PEM могут быть простыми функциями уровня модуля;

-- состояние модели сохраняется и может быть использовано PEM для взаимодействия со средой моделирования;

-- PEM могут возвращать объекты событий: это интересная возможность, позволяющая расширять модель с помощью новых типов событий.

## Пример модели процесса

Простая модель процесса в Simpy – часы, выводящие в консоль текущее модельное время:

```
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))
env.process(clock(env, 'slow', 1))

env.run(until=2)
>>>
fast 0
slow 0
fast 0.5
slow 1
fast 1.0
fast 1.5
. . .
```

***Environment.now*** – свойство среды с актуальным модельным временем.

Представим более сложную модель: модель электромобиля, который будет поочерёдно ездить и парковаться на некоторое время. Когда машина начинает движение (или паркуется) выводится текущее время. Реализуем функцию, описывающую процесс движения и изменения состояния электромобиля.

```
def car(env):
    while True:
        print(f'Начало парковки в {env.now:d}')
        parking_duration = 5
        yield env.timeout(parking_duration)

        print(f'Начало движения в {env.now:d}')
        trip_duration = 2
        yield env.timeout(trip_duration)
```

Чтобы процесс создавал новые события, необходимо передать в функцию ссылку на среду моделирования – `Environment(env)`. Алгоритм поведения электромобиля прописан в виде бесконечного цикла. Так как PEM-процесс является генератором, он вернет управление в среду моделирования, как только будет достигнута инструкция **yield**. После выполнения активного события выполнение функции возобновится со строчки, следующей за инструкцией **yield**.

Электромобиль поочередно находится в состояние движения или парковки. Он объявляет о своем новом состоянии, выводя в консоль сообщение и текущее модельное время **Environment.now**. Затем процесс вызывает метод **`Environment.timeout()`** для создания события Timeout. Это событие описывает время, когда машина находится на парковке или в движении. Когда процесс возвращает (через **yield**) событие, он сигнализирует среде моделирования, что хочет дождаться выполнения этого события.

```
import simpy
env = simpy.Environment()
env.process(car(env))
env.run(until=15)
>>>
Начало парковки в 0
Начало движения в 5
Начало парковки в 7
Начало движения в 12
Начало парковки в 14
```

Создадим экземпляр модельной среды, который передается функции. Далее вызовем метод среды моделирования ***Environment*.process()**, в который передадим экземпляр процесса. Он добавит в модельную среду новый процесс. Модельный PEM-процесс, возвращаемый методом **process**(), может использоваться для взаимодействия операторных треков модели между собой. Это нужно использовать, если требуется дождаться выполнения другого процесса или прервать другой процесс.

Чтобы PEM-процессы начали свое выполнение, необходимо запустить моделирование, вызывая метод ***Environment*.run()**. Этот метод принимает в качестве параметра время окончания моделирования.

## Взаимодействие процессов

В SimPy процесс может использоваться как событие (технически, процесс тоже является событием). Если функция возвращает процесс, то ее выполнение продолжится после завершения процесса.

Реализуем модель парковки и движения электромобиля, который будет ждать, когда его батарея зарядится, прежде чем он снова сможет начать движение.

Процесс зарядки реализуем с помощью дополнительной функции **charge**(). Для этого необходимо реорганизовать модель электромобиля как класс **Car** с двумя методами: **run**() и **charge**().

Процесс ***run*** будет запускаться при создании экземпляра класса **Car**. Новый процесс ***charge*** будет выполняться каждый раз, когда электромобиль начинает парковку. Процесс **run** ожидает завершения процесса, который вернул метод ***Environment*.process**(), то есть процесса ***charge***.

```
class Car(object):
    def __init__(self, env):
        self.env = env
        # Процесс запускается каждый раз, когда создается объект
        self.action = env.process(self.run())

    def run(self):
        while True:
            print(f'Начало парковки и зарядки в {self.env.now}')
            charge_duration = 5
            # Ожидаем, когда завершится зарядка
            yield self.env.process(self.charge(charge_duration))
            # Зарядка завершена. Начинаем движение
            print(f'Начало движения в {self.env.now}')
            trip_duration = 2
            yield self.env.timeout(trip_duration)

    def charge(self, duration):
        yield self.env.timeout(duration)
```

Чтобы запустить моделирование, создаем среду ***Environment***, процесс ***Car*** и вызываем метод **run**().

```
import simpy
env = simpy.Environment()
car = Car(env)
env.run(until=15)
>>>
Начало парковки и зарядки в 0
Начало движения в 5
Начало парковки и зарядки в 7
Начало движения в 12
Начало парковки и зарядки в 14
```

## Прерывание другого процесса

Предположим, что нет необходимости ждать, когда электромобиль зарядится полностью, то есть надо прервать процесс зарядки и начать движение. SimPy позволяет прервать текущий процесс, вызвав метод ***interrupt***().

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

Пусть процесс **driver** принимает в качестве параметра ссылку на экземпляр класса **Car**. Далее ожидает 3 единицы модельного времени и вызывает метод **interrupt**(), тем самым прерывая процесс электромобиля. Прерывания передаются в функцию процесса как *исключения среды*, которые должны быть обработаны в прерванном процессе.

```
class Car(object):
    def __init__(self, env):
       self.env = env
       self.action = env.process(self.run())
    def run(self):
        while True:
           print(f'Начало парковки и зарядки в {self.env.now}')
           charge_duration = 5
           # Можно прервать зарядку
           try:
              yield self.env.process(self.charge(charge_duration))
           except simpy.Interrupt:
               # Прерываем зарядку и начинаем движение
               print('Зарядка прервана')
           print(f'Начало движения в {self.env.now}')
           trip_duration = 2
           yield self.env.timeout(trip_duration)

    def charge(self, duration):
        yield self.env.timeout(duration)
```

Теперь автомобиль начинает движение в момент времени 3, а не 5.

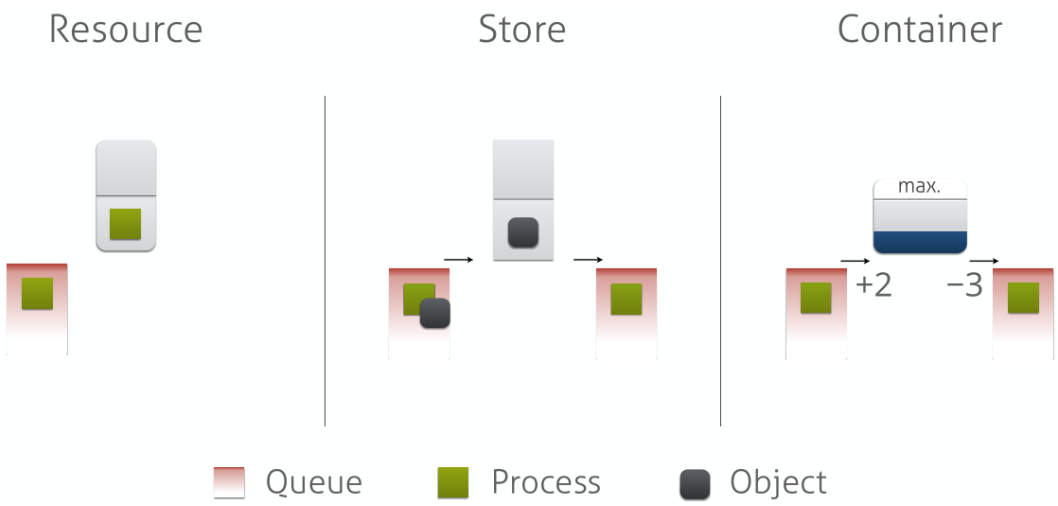
```
env = simpy.Environment()
car = Car(env)
env.process(driver(env, car))
env.run(until=15)
>>>
Начало парковки и зарядки в 0
Зарядка прервана
Начало движения в 3
Начало парковки и зарядки в 5
Начало движения в 10
Начало парковки и зарядки в 12
```

## Общие ресурсы модели

В Simpy имеется несколько типовых ресурсов (Resource), которые позволяют моделировать конфликты доступности, в случае если несколько процессов хотят одновременно использовать некоторый объект системы.

В Simpy определяется три категории ресурсов:

- ✓ **ресурс /Resource** — ресурсы, которые могут использоваться ограниченным числом процессов одновременно (например, заправочная станция с ограниченным числом топливных насосов);
- ✓ **хранилище /Store** — ресурсы, которые позволяют хранить и использовать объекты;
- ✓ **контейнер /Container** — ресурсы, моделирующие производство и потребление однородной недифференцированной массы. Он может быть либо непрерывным (как вода), либо дискретным (как яблоки).




Все ресурсы имеют одну и ту же базовую концепцию: сам ресурс представляет собой своего рода контейнер с ограниченной емкостью. Процессы могут либо попытаться что-то поместить в ресурс, либо попытаться что-то получить из него. Если ресурс полон или пуст, они должны стоять в очереди и ждать. Ресурс в базовой основе выглядит так:

```
BaseResource(capacity):
   put_queue
   get_queue
   put(): event
   get(): event
```

Каждый ресурс имеет максимальную емкость и две очереди: одну для процессов, которые хотят что-то в него поместить, и одну для процессов, которые хотят что-то получить. Оба метода put() и get() возвращают событие, которое запускается при успешном выполнении соответствующего действия.

Система ресурсов является модульной и расширяемой. Ресурсы могут, например, использовать специализированные очереди и типы событий. Это позволяет им использовать сортированные очереди, добавлять приоритеты к событиям или выполнять прерывание с вытеснением.

### *Тип «Ресурс» / Resource*

Resource могут использоваться ограниченным числом процессов одновременно (например, заправочная станция с ограниченным числом топливных насосов). Процессы требуют, чтобы эти ресурсы стали использоваться (или процессы «владели» ими), и должны освободить их после того, как они будут использованы (например, автомобили прибывают на заправочную станцию, используют топливный насос, если он доступен, и уезжают, когда они заправятся).

Запрос ресурса моделируется как «помещение токена процесса в ресурс», а освобождение ресурса, соответственно, как «извлечение токена процесса из ресурса». Таким образом, вызов request() / release() эквивалентен вызову put() / get(). Освобождение ресурса всегда будет успешным, а получение зависит от состояния ресурса.

Resource концептуально выполнен как арифметический семафор. Единственным его параметром, кроме обязательной ссылки на Environment, является его ёмкость: положительное число по умолчанию равное 1:

```
Resource(env, capacity=1)
```

Вместо того, чтобы просто подсчитывать своих текущих пользователей, он сохраняет событие запроса как «токен доступа» для каждого пользователя. Это полезно для добавления прерывания с вытеснением.

Базовый пример использования ресурса:

```
import simpy
def resource_user(env, resource):
     request = resource.request()  # создаем запрос/request
     yield request                 # ждем доступа
     yield env.timeout(1)          # делаем свою работу
     resource.release(request)     # освобождаем ресурс

env = simpy.Environment()
res = simpy.Resource(env, capacity=1)
user = env.process(resource_user(env, res))
env.run()
```

Обратите внимание, что нужно освободить ресурс в любом случае; например, если процесс прервали во время ожидания или использования ресурса. Чтобы избежать большого количества операторов кода, события запросов можно использовать в виде менеджера контекста:

```
try: ... with res.request() as req: ...  finally: ...
```

Методы `Resource` позволяют получить списки использующих процессов – пользователей в очереди к ресурсу, количество пользователей ресурса и емкость ресурса:

```
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: []
```

## Использование базового ресурса

Пусть электромобиль приезжает на электрозарядную станцию (ЭЗС), на которой есть две точки подключения зарядки. Если обе зарядки заняты, электромобиль ждет, когда одна из них освободится. Затем он заряжается и покидает ЭЗС.

```
def car(env, name, bcs, driving_time, charge_duration):
    # Моделирование ЭЗС
    yield env.timeout(driving_time)
    # Запрос одной из зарядок
    print(f'{name} прибыл в {env.now}')
    with bcs.request() as req:
        yield req
    # Зарядка
    print(f'{name} начал зарядку в {env.now}')
    yield env.timeout(charge_duration)
    print(f'{name} покинул ЭЗС в {env.now}')
```

Метод ресурса **request()** создает событие, которое занимает ресурс. Если процесс вернул это событие, то он владеет ресурсом, а все остальные ждут, когда ресурс освободится.

Если использовать из Python формат менеджера контекста **with**, то ресурс освободится автоматически. Но, если метод **request()** был вызван без менеджера контекста, то необходимо вызвать метод **release()**, чтобы закончить использование ресурса.

Когда процесс освобождает ресурс, следующий процесс из очереди занимает освободившееся место. Основной тип ресурса **(Resource)** располагает процессы в очереди с дисциплиной FIFO (первый пришел – первый вышел). При создании ресурса ему необходимо передать ссылку на среду моделирования **(Environment)** и указать ёмкость ресурса.

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

Создадим несколько процессов моделирующих автомобили и передадим ссылку на ресурс и дополнительные параметры.

```
for i in range(4):
    env.process(car(env, 'Электромобиль %d' % i, bcs, i*2, 5))
```

Теперь запустим моделирование. Оно закончится автоматически, как только все автомобили зарядятся.

```
env.run()
>>>
Электромобиль 0 прибыл в 0
Электромобиль 0 начал зарядку в 0
Электромобиль 1 прибыл в 2
Электромобиль 1 начал зарядку в 2
Электромобиль 2 прибыл в 4
Электромобиль 2 начал зарядку в 4
Электромобиль 0 покинул ЭЗС в 5
Электромобиль 3 прибыл в 6
Электромобиль 3 начал зарядку в 6
Электромобиль 1 покинул ЭЗС в 7
Электромобиль 2 покинул ЭЗС в 9
Электромобиль 3 покинул ЭЗС в 11
```

Для расширенного представления объектов в SimPy есть ресурсы дополнительных типов:

- ✓ PriorityResource – ресурс с учетом приоритета,
- ✓ PreemptiveResource – ресурс с возможным прерыванием.

**PriorityResource** – подкласс Resource позволяет запрашивающим процессам устанавливать приоритет для каждого запроса. Более важные запросы получат доступ к ресурсу раньше, чем менее важные. Приоритет выражается целыми числами; ***меньшие числа означают более высокий приоритет***. В остальных случаях он работает как обычный Resource:

```
def resource_user(name, env, resource, wait, prio):
     yield env.timeout(wait)
     with resource.request(priority=prio) as req:
         print(f'{name} запрос в {env.now} с приоритетом={prio}')
         yield req
         print(f'{name} получил ресурс в {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 запрос в 0 с приоритетом =0
1 получил ресурс в 0
2 запрос в 1 с приоритетом =0
3 запрос в 2 с приоритетом =-1
3 получил ресурс в 3
2 получил ресурс в 6
```

Хотя p3 запросил ресурс позже, чем p2 , он смог использовать его раньше, поскольку его приоритет был выше (-1).

**PreemptiveResource** – подкласс Resource с учетом прерывания. **PreemptiveResource** позволяет прерывать работу существующих пользователей из ресурса (это называется вытеснением):

```
def resource_user(name, env, resource, wait, prio):
   yield env.timeout(wait)
   with resource.request(priority=prio) as req:
      print(f'{name} запрос в {env.now} с приоритетом={prio}')
      yield req
      print(f'{name} получил ресурс в {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} был прерван {by} в {env.now} после {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 запрос в 0 с приоритетом=0
1 получил ресурс в 0
2 запрос в 1 с приоритетом=0
3 запрос в 2 с приоритетом=-1
1 был прерван <Process(resource_user) object at 0x...> в 2 после 2
3 получил ресурс в 2
2 получил ресурс в 5
```

**PreemptiveResource** наследуется от **PriorityResource** и добавляет флаг **preempt** к **request**(), по умолчанию установлен в **True**. Установив значение флага в **False** (resource.request(priority=x, preempt=False)), процесс может принять решение не вытеснять другого пользователя ресурса. Однако он все равно будет помещен в очередь в соответствии с его приоритетом.

Реализация **PreemptiveResource** также учитывает более высокий приоритет процессов перед вытеснением. Это означает, что прерывающие запросы не могут перескакивать через запрос с более высоким приоритетом. В примере показано, что вытесняющие запросы с низким приоритетом не могут переходить в очередь через запросы с высоким приоритетом:

```
def user(name, env, res, prio, preempt):
    with res.request(priority=prio, preempt=preempt) as req:
       try:
          print(f'{name} запрос в {env.now}')
          #assert isinstance(env.now, int), type(env.now)
          yield req
          #assert isinstance(env.now, int), type(env.now)
          print(f'{name} получил ресурс в {env.now}')
          yield env.timeout(3)
       except simpy.Interrupt:
          print(f'{name} был прерван в {env.now}')

env = simpy.Environment()
res = simpy.PreemptiveResource(env, capacity=1)
A = env.process(user('A', env, res, prio=0, preempt=True))
B = env.process(user('B', env, res, prio=-2, preempt=False))
C = env.process(user('C', env, res, prio=-1, preempt=True))

env.run(until=9)
>>>
A запрос в 0
B запрос в 0
C запрос в 0
A получил ресурс в 0
B получил ресурс в 3
C получил ресурс в 6
```

Процесс A запрашивает ресурс с приоритетом 0. Он сразу же становится пользователем ресурса. Процесс B запрашивает ресурс с приоритетом -2, но устанавливает вытеснение на **False**. Он встанет в очередь и будет ждать. Процесс C запрашивает ресурс с приоритетом -1, но оставляет приоритет **True**. Обычно он вытесняет A, но в этом случае B ставится в очередь перед C и не позволяет C вытеснить A, процесс C также не может вытеснить B , поскольку его приоритет недостаточно высок.

Таким образом, поведение в примере такое же, как если бы вытеснение вообще не использовалось. При смешанном вытеснении из-за более высокого приоритета процесса B в этом примере не происходит прерывания. Обратите внимание, что дополнительный запрос с приоритетом -3 сможет вытеснить A. Если ваш вариант использования требует другого поведения, например, перехода из очереди или приоритета, можно создать подкласс **PreemptiveResource** и переопределить поведение по умолчанию.

## *Тип ресурса «хранилище» / Store*

С помощью хранилища/Store можно моделировать производство и использование конкретных объектов. Причём, одно хранилище может содержать несколько разных типов объектов.

Пример – типовой сценарий, моделирующий производителя / потребителя:

```
def producer(env, store):
    for i in range(100):
        yield env.timeout(2)
        yield store.put(f'блины_{i}')
        print(f'блины {i} готовы в', env.now)

def consumer(name, env, store):
    while True:
        yield env.timeout(1)
        print(name, 'хочу блины в', env.now)
        item = yield store.get()
        print(name, 'взял', item, 'в', 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 хочу блины в 1
1 хочу блины в 1
блины_0 готовы в 2
0 взял блины_0 в 2
0 хочу блины в 3
блины_1 готовы в 4
1 взял блины_1 в 4
```

Как и в случае с другими типами ресурсов, можно получить вместимость хранилища через атрибут **capacity**. Атрибут **items** указывает на список предметов, доступных в настоящее время в хранилище. Доступ к очередям **put** и **get** можно получить через атрибуты **put_queue** и **get_queue**.

Для типа **Store** есть расширенные типы:

- ✓ **PriorityStore** – хранилище с учетом приоритета.
- ✓ **FilterStore** – хранилище с учетом условий. Этот тип позволяет использовать пользовательскую функцию для фильтрации объектов, которые надо получать из хранилища.

Например, **FilterStore** можно использовать для моделирования механических цехов, в которых станки имеют различные атрибуты. Это может быть полезно, если однородные слоты не нужны:

```
from collections import namedtuple

Machine = namedtuple('Станок', 'size, duration')
m1 = Machine(1, 2)  # малый и медленный станок
m2 = Machine(2, 1)  # большой и быстрый станок
env = simpy.Environment()
```

```
m_s = simpy.FilterStore(env, capacity=2)
m_s.items = [m1, m2]  # собираем цех из станков

def user(name, env, ms, size):
   machine = yield ms.get(lambda machine: machine.size == size)
   print(name, 'получил', machine, 'в', env.now)
   yield env.timeout(machine.duration)
   yield ms.put(machine)
   print(name, 'освободил', machine, 'в', env.now)

users =[env.process(user(str(i+1)+'й',env,m_s,(i%2)+1)) for i in range(4)]
env.run()
>>>
1й получил Станок(size=1, duration=2) в 0
2й получил Станок(size=2, duration=1) в 0
2й освободил Станок(size=2, duration=1) в 1
4й получил Станок(size=2, duration=1) в 1
1й освободил Станок(size=1, duration=2) в 2
4й освободил Станок(size=2, duration=1) в 2
3й получил Станок(size=1, duration=2) в 2
3й освободил Станок(size=1, duration=2) в 4
```

С помощью **PriorityStore** можно моделировать процессы с разными приоритетами. В следующем примере процесс `inspector` (Инспектор) находит и регистрирует проблемы, которые устраняет отдельный процесс `maintainer` (Ремонтник) с учетом приоритета, организуемого PriorityStore.

```
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, 'проверил', issue)
       yield issues.put(issue)

def maintainer(env, issues):
    while True:
       yield env.timeout(3)
       issue = yield issues.get()
       print(env.now, 'починил', issue)

_ = env.process(inspector(env, issues))
_ = env.process(maintainer(env, issues))
env.run()
>>>
1 регистр PriorityItem(priority='P2', item='#0000')
2 регистр PriorityItem(priority='P0', item='#0001')
3 регистр PriorityItem(priority='P3', item='#0002')
3 ремонт PriorityItem(priority='P0', item='#0001')
4 регистр PriorityItem(priority='P1', item='#0003')
```

```
6 ремонт PriorityItem(priority='P1', item='#0003')
9 ремонт PriorityItem(priority='P2', item='#0000')
12 ремонт PriorityItem(priority='P3', item='#0002')
```

## *Тип Контейнер / Container*

Контейнеры помогают моделировать производство и потребление однородной массы материала, который может быть либо непрерывным (как вода), либо дискретным (как яблоки). Можно использовать этот вариант, например, для моделирования бензина на заправочной станции. Автозаправщики увеличивают количество бензина на АЗС, а автомобили-клиенты АЗС - уменьшают. В примере представлена простая модель заправочной станции с ограниченным количеством ТРК /колонок/, смоделированных как Resource, и цистерной /gas_tank/, смоделированной как 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'Вызов заправщика в {env.now}')
            env.process(tanker(env, self))
         yield env.timeout(15)


def tanker(env, gas_station):
    yield env.timeout(10) # заправщику надо ехать 10 мин
    print(f'Заправщик приехал в {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'Клиент {name} приехал в {env.now}')
    with gas_station.fuel_dispensers.request() as req:
       yield req
       print(f'Клиент {name} начал заправку в {env.now}')
       yield gas_station.gas_tank.get(40)
       yield env.timeout(5)
       print(f'Клиент {name} закончил заправку в {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)
```

```
>>>
Клиент 0 приехал в 0
Клиент 0 начал заправку в 0
Клиент 1 приехал в 5
Клиент 0 закончил заправку в 5
Клиент 1 начал заправку в 5
Клиент 2 приехал в 10
Клиент 1 закончил заправку в 10
Клиент 2 начал заправку в 10
Вызов заправщика в 15
Клиент 3 приехал в 15
Клиент 3 начал заправку в 15
Заправщик приехал в 25
Клиент 2 закончил заправку в 30
Клиент 3 закончил заправку в 30
```

Контейнеры позволяют узнавать их текущий уровень/level и емкость/capacity (GasStation. monitor_tank() и tanker()). Можно получить доступ к списку ожидающих событий через атрибуты put_queue и get_queue (аналогично Resource.queue).

Среда **Simpy** - настраиваемая с точки зрения управления выполнением модели.

Можно выполнять модели до тех пор, пока больше не останется событий; или пока не пройдет определенное время моделирования; или до определенного события. Можно провести моделирование процесса по шагам по событиям. Кроме того, можно смешивать эти варианты.

Например, можно запустить моделирование до тех пор, пока не произойдет интересующее событие. Затем можно провести моделирование по событиям в течение некоторого времени; и затем запустить моделирование, пока не останется больше событий и все процессы закончатся.

Главный метод модели - **Environment.run**().

Если вызвать его без каких-либо аргументов ( env.run () ), он будет выполнять моделирование до тех пор, пока не останется больше событий. Если ваши процессы работают вечно ( `while True: yield env.timeout(1))`, то метод никогда не завершится (пока вы не прервёте модель, нажав Ctrl-C).

В большинстве случаев целесообразно остановить моделирование, когда оно достигнет определенного времени моделирования. Для этого можно передать желаемое время через параметр ***until***, например: `env.run(until=10)`. Моделирование остановится, когда внутренние часы достигнут 10, и не будут обработаны события, запланированные на время 10.

Если надо использовать моделирование в GUI, и, например, требуется нарисовать индикатор прогресса (process bar), то можно многократно вызывать эту функцию с увеличением до конечного значения и обновлять индикатор выполнения после каждого вызова:

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

Вместо того, чтобы передавать число run (), также можно передать ему любое событие. Тогда run () завершится, когда событие будет обработано. Например, если текущее время равно 0, `env.run (until=env.timeout(5))` будет эквивалентно `env.run(until=5)`.

Также можно передавать другие типы событий (здесь и процесс является событием):

```
def my_proc(env):
    yield env.timeout(1)
    return 'Python & Anaconda'

env = simpy.Environment()
proc = env.process(my_proc(env))
env.run(until=proc)
print(proc.value)
```

Чтобы моделировать пошагово по событиям, в Simpy предлагаются методы peek() и step():

- ✓ peek () возвращает ***время*** следующего запланированного события.
- ✓ step () обрабатывает следующее запланированное событие, при этом, если событие недоступно, вызывается исключение.

В типичном случае использования эти методы используются в цикле, например:

```
untillimit = 10
while env.peek() < untillimit:
```

```
env.step()
```

Среда моделирования позволяет получить текущее модельное время через свойство **Environment.now.** Время моделирования – это инкрементальный счетчик, который увеличивается через события тайм-аута.

По умолчанию время начинается с 0, но можно передать `initial_time в Environment` для использования другого начала отсчёта. Хотя время моделирования технически абстрактно, вы можете предположить что это, например, в секундах и использовать его как метку времени, возвращаемую `time.time()` для расчета даты или дня недели.

Свойство **Environment.active_process** можно сравнить с os.getpid(), который указывает на текущий активный процесс. Процесс активен, когда выполняется его функция **process**. Он становится неактивным, когда он отдает событие (***yield***) . Таким образом, имеет смысл получить доступ к этому свойству только из функции процесса или функции, вызываемой функцией процесса:

```
def subfunc(env):
  print(env.active_process)

def my_proc(env):
  while True:
    print(env.active_process)
    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...>
```

Примером использования является система ресурсов: если функция процесса вызывает **request**() чтобы запросить ресурс, ресурс определяет источник запроса через **env.active_process**.

Чтобы создать события, обычно надо импортировать **simpy.events**, создать экземпляр класса Event и передать ему ссылку на среду. Чтобы уменьшить количество кода, среда предоставляет некоторые сокращения для обращения. Например, **Environment.event()** эквивалентен **simpy.events.Event(env)**. Есть и другие сокращения:

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

В Simpy функция-генератор может использоваться для предоставления возвращаемых значений для процессов, которые могут быть другими процессами:

```
def other_proc(env):
```

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

или обычно так:

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

Для удобства читаемости, в среде есть метод `exit()`, чтобы сделать то же самое:

```
def my_proc(env):
   yield env.timeout(1)
   env.exit(42) # в SimPy это то же самое "return 42"
```

Диспетчер событий **Simpy** основан на функциях-генераторах и может использоваться как для асинхронной сети процессов, так и для реализации мультиагентных систем с моделируемыми взаимодействиями между ними.

**Simpy** включает набор типов событий для различных целей. Они наследуют **simpy.events.Event.** Иерархию событий в Simpy можно показать так:

```
events.Event
|
+— events.Timeout
|
+— events.Initialize
|
+— events.Process
|
+— events.Condition
| |
| +— events.AllOf
| |
| +— events.AnyOf
⋮
+- [resource events]
```

Это набор основных событий. События являются расширяемыми, и ресурсы могут определять дополнительные события.

Рассмотрим события в модуле `Simpy.events`. События SimPy очень похожи на отсрочки (фьючерсы или обещания), как и события класса, используемые для описания любого вида события. События могут находиться в одном из следующих состояний:

- событие может произойти (не срабатывает),

- произойдет (срабатывает) `Event.triggered`,

- произошло (обработано) `Event.processed`.

События проходят эти состояния ровно по разу в этом порядке. События также плотно привязаны ко времени, и время заставляет события исполняться. Изначально события не запускаются, а создаются

как объекты в памяти. Если событие запускается, оно назначается на заданное время и вставляется в Очередь событий SimPy. Состояние Event.triggered становится истинным (true). Пока событие не обработано, можно добавлять обратные вызовы к событию для своих обработчиков.

Обратные вызовы - это вызываемые объекты, которые принимают событие в качестве параметра и хранятся в Event.callbacks.

Событие обрабатывается, когда SimPy выводит его из очереди событий и вызовет все его обработчики. В этой фазе больше нельзя добавлять обратные вызовы, и для события свойство Event.processed становится истинным (true).

События также могут иметь значение. Значение может быть установлено до или во время срабатывания и может быть извлечено через Event.value или в рамках процесса получения события (value = yield event).

Наиболее распространенным способом добавления обратного вызова к событию является получение его из вашей функции процесса (событие yield ). Этот способ добавит метод процесса _resume () в качестве обратного вызова. Это происходит, если ваш процесс возобновляется, когда он выдал событие.

Однако можно добавить любой вызываемый объект (функцию) в список обратных вызовов, если он принимает экземпляр события в качестве единственного параметра:

```
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...>]
```

Если событие было обработано, все его обработчики были выполнены, то обратные вызовы и атрибут имеет значение None. Это помешает добавлять обратные вызовы - они, конечно, никогда не будут вызваны, потому что событие уже случилось.

Однако процессы в этом отношении умны. Если вы выдаете (yield) обработанное событие, _resume() немедленно возобновит ваш процесс со значением события (потому что ждать нечего).

Когда события срабатывают (triggered), они могут быть успешными или неудачными. Например, если событие должно быть инициировано в конце вычисления, и все работает отлично, то событие будет успешным. Если во время срабатывания происходит исключение, то событие будет неудачным.

Чтобы выполнить событие и отметить его как успешное, можно использовать Event.succeed (value=None). Можно дополнительно передать ему значение (value) (например, результаты вычислений).

Чтобы исполнить событие и отметить его как неудачное (failed), вызовите Event.fail (exception) и передайте ему экземпляр исключения (например, исключение, которое случилось во время неудачного вычисления).

Существует также общий способ исполнения события: Event.trigger(event). В этом случае можно будет принять значение и результат (успех или неудача) события, переданного ему.

Все три метода возвращают экземпляр события, к которому они привязаны, что позволяет в модели сделать так: `yield Event(env).succeed()`

Рассмотрим пример работы с событиями: они могут создаваться процессом или за пределами контекста процесса; они могут передаваться другим процессам и вызывать другие события.

```
import simpy
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(4):
     yield self.env.timeout(45)
     self.class_ends.succeed()
     self.class_ends = self.env.event()
     print("\n_звонок_", i+1,"::",env.now)

 def pupil(self):
   for i in range(3):
    yield self.class_ends
    print('|o/', end=' ')

env = simpy.Environment()
school = School(env)
env.run()
>>>>>>
_звонок_ 1 :: 45
|o/ |o/ |o/
_звонок_ 2 :: 90
|o/ |o/ |o/
_звонок_ 3 :: 135
|o/ |o/ |o/
_звонок_ 4 :: 180
```

Такое поведение событий можно интерпретировать как действия  passivate / reactivate. Ученики пассивизируются, когда занятие начинается и активизируются, когда звенит звонок с урока.

В Simpy процессы (как созданные **Process** или **env.process**()) имеют полезное свойство быть событиями. Это означает, что один процесс может выдать (вернуть **yield**) другой процесс. Затем он будет возобновлен, когда закончится выданный процесс. Значением события будет возвращаемое значение этого процесса:

```
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
```

Когда процесс создается, он планирует инициализировать событие (**Initialize**), которое запустит выполнение процесса при срабатывании (**triggered**). Обычно вам не придется иметь дело с такого рода событиями.

Если нужно, чтобы процесс начался после определенной задержки, можно использовать **simpy.util.start_delayed**(). Этот метод возвращает вспомогательный процесс, использующий тайм-аут перед началом процесса.

В этом примере процесс sub() с отложенным запуском:

```
from simpy.util import start_delayed

 def sub(env):
   yield env.timeout(1)
   return 234

 def parent(env):
   start = env.now
   sub_proc = yield start_delayed(env, sub(env), delay=3)
   assert env.now - start == 3

   ret = yield sub_proc
   return ret

 env.run(env.process(parent(env)))
>>>
234
```

Иногда надо ждать более одного события одновременно. Например, может потребоваться дождаться ресурса, но не более ограниченного времени, или надо подождать, пока несколько событий случится вместе. Для этого Simpy предлагает особые события **AnyOf** и **AllOf**, которые являются событиями типа **Condition**.

В обоих случаях эти события берут список событий в качестве аргумента и выполняют (**triggered**) события: или хотя бы одно, или все из них соответственно:

```
from simpy.events import AnyOf, AllOf, Event
events = [Event(env) for i in range(3)]
a = AnyOf(env, events) # сработает если одно из событий случится
b = AllOf(env, events) # сработает если все события случатся
```

Значение события **Condition** – это словарь с записью для каждого выполненного события. В случае **AllOf** размер этого словаря будет такой же, как и длина списка событий. Значение **dict** в случае **AnyOf** будет иметь хотя бы одну запись. В обоих случаях экземпляры событий используются в качестве ключей, а значения событий будут значениями словаря.

Также с **AnyOf** и **AllOf** можно использовать логические операторы **&** (и), **|** (или):

```
    import simpy
    from simpy.events import AnyOf, AllOf, Event

    def test_condition(env):
      t1, t2 = env.timeout(1, value='вкл'), env.timeout(2, value='выкл')
      ret = yield t1 | t2
      assert ret == {t1: 'вкл'}

      t1, t2 = env.timeout(1, value='вкл'), env.timeout(2, value='выкл')
      ret = yield t1 & t2
      assert ret == {t1: 'вкл', t2: 'выкл'}

      # можно использовать операторы & и |
      e1, e2, e3 = [env.timeout(i) for i in range(3)]
      yield (e1 | e2) & e3
      assert all(e.triggered for e in [e1, e2, e3])
      print(t1, t2, e1, e2, e3)

    env = simpy.Environment()
    proc = env.process(test_condition(env))
    env.run()
-------------------------
```

Для сбора статистики часто используют Монитор – специальную функцию, выполняющийся как процесс с фиксированным таймаутом.

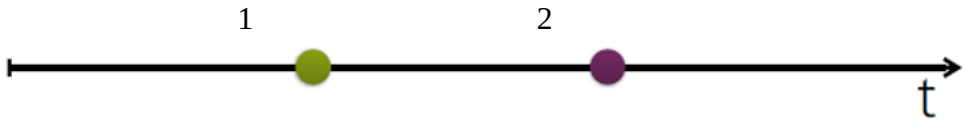
Эта генераторная функция создает итератор, собирающий статистику о состоянии ресурса, очереди, или другого процесса через регулярные промежутки времени. Альтернативным подходом может быть применение свойств пуассоновского процесса, и сбор данных об очереди в моменты непосредственно перед прибытием/выбытием инициатора.

```
    # специальная функция для сбора данных Q_stats
    def monitor(env):
       global Q_stats
       while True:
          Q_stats.count += 1
          Q_stats.cars_waiting += len(queue)
          yield env.timeout(1.0)

    # запускаем процесс мониторинга
    env.process(monitor(env))
```

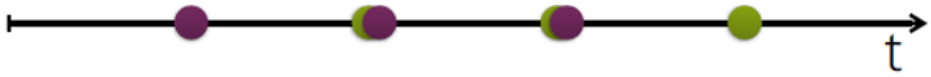
Пример модели взаимодействия «докладчик-модератор»

1) докладчик выступает не более 30 мин



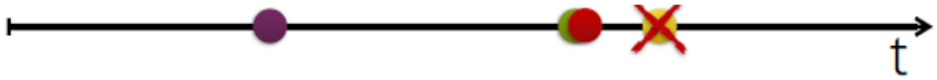

```
def speaker(env, start):
  until_start = start - env.now
  yield env.timeout(until_start) #1
  yield env.timeout(30)          #2
```

2) докладчики выступают по 30 мин



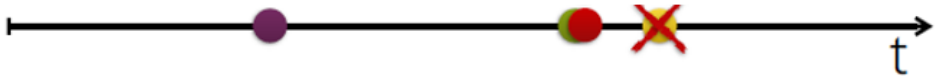

```
def speaker(env):
  yie1d timeout(30)
  return 'handout'
def moderator(env):
  for i in range(3):
      val = yie1d env.process(speaker(env))
      print(val)
```

3) докладчик выступает больше 30 мин, но модератор прерывает его




```
      def speaker(env):
        try:
           yie1d env.timeout(randint(25, 35))
        except simpy.Interrupt as interrupt:
           print (interrupt.cause)
      def moderator(env):
         for i in range(3):
           speaker_proc = env.process(speaker(env))
           yie1d env.timeout(30)
           speaker_proc.interrupt('Время закончилось!')
```

4) докладчик может выступать не больше 30 мин, и модератор может прерывать его




```
      def speaker(env):
        try:
           yie1d env.timeout(randint(25, 35))
        except simpy.Interrupt as interrupt:
           print (interrupt.cause)
      def moderator(env):
         for i in range(3):
           speaker_proc = env.process(speaker(env))
           result = yie1d speaker_proc | env.timeout(30)
           if speaker_proc not in results:
                speaker_proc.interrupt('Время закончилось!')
```

Пример взаимодействия «клиент-сервер»

```
def client(env, client_sock):
   message = Message(env, client_sock)
   reply = yield message.send('hello')
   print (reply)

def server(env, server_sock):
   # новое подключение
   sock = yield server_sock.accept()
   message = Message(env, PacketUTF8(sock))
   # получил сообщение и ответил
   request = yield message.recv()
   print (request.content)
   yield request.succeed('OK')
```

## Полезные ссылки

https://simpy.readthedocs.io/en/latest/
https://simpy.readthedocs.io/en/latest/simpy_intro/index.html
https://simpy.readthedocs.io/en/latest/topical_guides/index.html
https://simpy.readthedocs.io/en/latest/examples/index.html
https://simpy.readthedocs.io/en/latest/api_reference/index.html
https://groups.google.com/forum/#!forum/python-simpy
https://www.youtube.com/watch?v=Bk91DoAEcjY