شبیه سازی رویداد گسسته برای پایتون

مترجمان: مجتبی آل حسینی، پردیس عشقی نژاد

استاد مشاور: جمال زارع پور احمدآبادی

فهرست

بررسی اجمالی

سیمپای[1] یک چارچوب[2] شبیه سازی رویداد گسسته[3] مبتنی بر فرآیند است که مبتنی بر پایتون استاندارد است.

فرایندها در سیمپای توسط توابع مولد[4] پایتون تعریف می شوند و می توانند به عنوان مثال برای مدل سازی اجزای فعال مانند مشتریان ، وسایل نقلیه و غیره استفاده شوند. سیمپای همچنین انواع مختلفی از منابع مشترک[5] را برای مدل سازی نقاط پرازدحام و با ظرفیت محدود (مانند سرورها ، شمارنده های پرداخت و تونل ها) فراهم می کند.

(مترجم: ژنراتور (مولد) یا همان Generator ها به توابعی گفته می‌شوند که به منظور ایجاد یک تابع با رفتاری مشابه اشیا iterator پیاده‌سازی می‌گردند. هنگام فراخوانی یک تابع معمولی، بدنه تابع اجرا می‌شود تا به یک دستور return برسد و خاتمه یابد ولی با فراخوانی یک تابع Generator، بدنه تابع اجرا نمی‌شود بلکه یک شی generator برگردانده خواهد شد که می‌توان با استفاده از متد ()__next__ آن، مقادیر مورد انتظار خود را یکی پس از دیگری درخواست داد.)

شبیه سازی را می توان "با بیشترین سرعت ممکن" ، در زمان واقعی (زمان ساعت دیواری) یا با قدم زدن دستی در وقایع انجام داد.

اگرچه از نظر تئوری می توان شبیه سازی پیوسته[6] را با سیمپای انجام داد، اما هیچ قابلیت دیگری در این زمینه به شما کمک نمی کند. از طرف دیگر ، سیمپای در شبیه سازی هایی با اندازه گام ثابت که فرآیندها برهم کنش ندارند، بیش از حد قوی یا کار نالازم انجام می دهد.

یک مثال کوتاه، شبیه سازی دو ساعت در بازه های زمانی مختلف به شکل زیر است

```
>>> 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
```

---

[1] SimPy
[2] framework
[3] discrete-event
[4] generator functions
[5] shared resources
[6] continuous simulations

در ادامه به نکاتی شامل یک آموزش ، چندین راهنما در توضیح مفاهیم کلیدی ، تعدادی مثال و غیره خواهیم پرداخت.

سیمپای تحت مجوز MIT منتشر می شود. اگه کار شبیه سازی میکنید با جامعه سیمپای به اشتراک بگذارید.

یک سخنرانی مقدماتی وجود دارد که مفاهیم سیمپای را توضیح می دهد و چندین مثال را مطرح می کند، میتوانید آن را تماشا کنید یا اسلایدهای آن را دریافت کنید.

سیمپای همچنین در سایر زبانهای برنامه نویسی مجدداً پیاده سازی شده است. به عنوان مثال در سی شارپ سیمشارپ ، در جولیا ، سیم جولیا و در آر R ، سیمر simmer نام گذاری شده است.

سیمپای در ده دقیقه

در این بخش ، شما فقط در چند دقیقه اصول سیمپای را یاد میگیرید. بعد از آن، میتوانیم یک شبیه سازی ساده را با استفاده از سیمپای پیاده سازی کنیم و اگر سیمپای همان چیزی باشد که شما نیاز دارید ، می توانید تصمیم بگیرید که بهتر آن را یاد بگیرید. در آخرهم نکاتی را در مورد نحوه انجام شبیه سازی های پیچیده تر یاد خواهیم گرفت.

نصب و راه اندازی

سیمپای در پایتون خالص اجرا میشود و وابستگی ندارد، اگر pip را نصب کرده اید، فقط کافی است بنویسید:

```
$ pip install simpy
```

و تمام.

نصب از منبع

همپنین میتوانیم سیمپای را بارگیری کرده و به صورت دستی نصب نماییم. فایل کم حجم شده را استخراج کنید ، یک پنجره ترمینال را که در آن سیمپای را استخراج کرده اید باز کنید و تایپ کنید:

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

اکنون می توانید به صورت اختیاری تست های سیمپای را اجرا کنید تا ببینید آیا همه چیز به درستی کار میکند. شما برای این کار به pytest نیاز دارید. دستور زیر را در مسیر نصب سیمپای اجرا کنید:

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

ارتقا از سیمپای ۲

اگر از قبل با سیمپای ۲ آشنایی دارید ، لطفا راهنمای انتقال از سیمپای ۲ به ۳ را مطالعه کنید.

اکنون که سیمپای را نصب کرده اید ، احتمالاً می خواهید چیزی را شبیه سازی کنید. بخش بعدی شما را با مفاهیم اساسی سیمپای آشنا می کند.

مفاهیم پایه

سیمپای یک کتابخانه شبیه سازی رویداد گسسته است. رفتار اجزای فعال [۷](مانند وسایل نقلیه ، مشتریان یا پیام ها) با فرایندها [۸]مدل سازی می شود. همه فرایندها در یک محیط [۹]هستند. آنها از طریق رویدادها [۱۰]با محیط و یکدیگر تعامل دارند.

فرایندها توسط مولدهای ساده پایتون توصیف می شوند. بسته به اینکه یک تابع یا متد یک کلاس باشد ، می توانید آنها را تابع پردازش[۱۱] یا روش پردازش[۱۲] بنامید. در طول حیات خود ، آنها رویدادهایی را ایجاد کرده و ارائه [۱۳]می دهند تا منتظر راه اندازی شدن آنها بمانند.

هنگامی که یک فرآیند یک رویداد را ارائه می دهد ، فرآیند به حالت تعلیق در می آید. وقتی رویداد رخ می دهد، سیمپای فرآیند را از سر می گیرد (می گوییم این رویداد راه اندازی [۱۴]می شود). چندین فرآیند می توانند منتظر همان رویداد باشند. سیمپای آنها را به همان ترتیب که آن رویداد را ارائه می دهند از سر می گیرد.

یک نوع مهم رویداد Timeout است. وقایع از این نوع پس از گذشت زمان مشخصی (شبیه سازی شده) آغاز می شوند. آنها به فرآیندی اجازه می دهند تا مدت زمان مشخص بخوابد (یا حالت خود را حفظ کند). با فراخوانی روش مناسب محیطی که روند کار در آن زندگی می کند می توان Timeout و سایر رویدادها را ایجاد کرد. (به عنوان مثال Environment.timeout)

اولین روند

اولین مثال ما فرآیند اتومبیل خواهد بود. ماشین به طور متناوب برای مدتی رانندگی و پارک می کند. وقتی شروع به رانندگی (یا پارک کردن) می کند ، زمان شبیه سازی فعلی چاپ می شود.

```
>>> 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)
```

فرایند اتومبیل ما برای ایجاد رویدادهای جدید نیاز به مراجعه به یک محیط [۱۵]دارد. رفتار ماشین در یک حلقه بی نهایت توصیف می شود. به یاد داشته باشید ، این تابع یک مولد است. اگرچه هرگز خاتمه نخواهد یافت ، اما پس از دستیابی به بازده عملکرد، جریان کنترل را به شبیه سازی منتقل می کند. هنگامی که رویداد اتفاق افتاد[۱۶]، شبیه سازی عملکرد این عبارت را از سر می گیرد.

---

[۷] *active components*
[۸] processes
[۹] environment
[۱۰] events
[۱۱] process function
[۱۲] process method
[۱۳] *yield*
[۱۴] triggered
[۱۵] Environment
[۱۶] yield

همانطور که قبلاً گفتیم ، اتومبیل ما بین حالت های پارکینگ و رانندگی سوئیچ می کند. این حالت جدید خود را با چاپ یک پیام و زمان شبیه سازی فعلی (که توسط ویژگی Environment.now برگردانده شده است) اعلام می کند. سپس برای ایجاد یک رویداد Timeout تابع Environment.timeout را فراخوانی می کند. این رویداد نقطه زمانی را که پارکینگ اتومبیل (یا به ترتیب رانندگی) انجام می دهد ، توصیف می کند. با ارائه رویداد ، این شبیه سازی را نشان می دهد که می خواهد منتظر وقوع رویداد باشد.

اکنون که رفتار خودروی ما مدل شده است ، اجازه دهید نمونه ای از آن را ایجاد کرده و نحوه رفتار آن را ببینیم:

```
>>> 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
```

اولین کاری که باید انجام دهیم ایجاد نمونه ای از Environment است. این نمونه به عملکرد فرآیند اتومبیل ما منتقل می شود. فراخوانی آن یک مولد فرآیند ایجاد می کند که باید راه اندازی شود و از طریق Environment.process به محیط اضافه شود. به یاد داشته باشید که در این زمان ، هیچ یک از کد عملکرد فرآیند ما در حال اجرا نیست. اجرای آن صرفاً در زمان شبیه سازی فعلی برنامه ریزی شده است.

فرایند برگشت داده شده توسط processمی تواند برای تعاملات فرآیند مورد استفاده قرار گیرد (ما در بخش بعدی به آن خواهیم پرداخت ، بنابراین فعلاً آن را نادیده خواهیم گرفت).

در آخر ، ما با فراخوانی run و گذراندن زمان پایان به آن ، شبیه سازی را شروع می کنیم.

تعامل(برهم کنش) فرآیند

نمونه فرآیند که توسط Environment.process بازگردانده می شود می تواند برای تعاملات فرآیند استفاده شود. دو مثال متداول برای این امر این است که منتظر بمانیم تا فرآیند دیگری به پایان برسد و روند دیگری را قطع کند در حالی که منتظر یک اتفاق است.

منتظر یک روند بودن

همانطور که اتفاق می افتد ، یک فرآیند سیمپای [۱۷]می تواند مانند یک رویداد استفاده شود (از نظر فنی ، یک فرآیند در واقع یک رویداد است). اگر آن را تسلیم کند (تحویل دهد) ، پس از اتمام روند کار از سر گرفته می شود. شبیه سازی کارواش را تصور کنید که ماشین ها وارد کارواش شوند و منتظر بمانند تا فرآیند شستشو به پایان برسد. یا شبیه سازی فرودگاه که در آن مسافران باید منتظر بمانند تا بررسی امنیتی به پایان برسد.

[۱۷] simPy

فرض کنیم که ماشین آخرین نمونه ما به یک وسیله نقلیه الکتریکی تبدیل شده است. ماشین برقی ها معمولاً پس از سفر مدت زمان زیادی را باتری خود را شارژ می کنند. آنها قبل از شروع دوباره رانندگی باید منتظر بمانند تا باتری آنها شارژ شود.
میتوان این قضیه را با یک فرایند اضافه به اسم charge برای ماشین ها مدل کرد. درنتیجه میتوان ماشین ها را با دوتا متود فرایند کلاس بندی کنیم: یکی ران و دیگری یکیش چارج

فرآیند اجرا به طور خودکار زمانی شروع می شود که ماشین نمونه سازی شود. هر بار که پارک کردن خودرو شروع می شود ، یک فرآیند جدید شارژ آغاز می شود. با بازگرداندن مثال Process که Environment.process برمی گرداند ، فرایند اجرا منتظر می ماند تا پایان یابد:

```
>>> 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)
```

شروع شبیه سازی دوباره ساده است: ما یک محیط ایجاد می کنیم ، یک (یا چند) ماشین و در آخر runرو میزنیم.

```
>>> 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
```

قطع یک روند دیگر

حال فرض کنید که نمی خواهید منتظر بمانید تا وسیله نقلیه الکتریکی شما به طور کامل شارژ شود ، می خواهید روند شارژ را قطع کنید و به جای آن فقط شروع به رانندگی کنید.

سیمپای به شما امکان می دهد با فراخوانی متد intrupt ، یک روند در حال اجرا را قطع کنید:

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

فرآیند driver به روند عملکرد خودرو اشاره دارد. پس از ۳ مرحله انتظار ، این روند را قطع می کند.

وقفه ها رو بوسیله تابع interupt در فرایند قرار می دهیم، بجز زمانی که میتوانند با قطع فرایند مدیریت شوند سپس این فرآیند می تواند تصمیم بگیرد که در مرحله بعد چه کاری انجام دهد (به عنوان مثال ، ادامه انتظار برای رویداد اصلی یا ارائه یک رویداد جدید):

```
>>> 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)
```

هنگامی که خروجی این شبیه سازی را با مثال قبلی مقایسه می کنید ، متوجه خواهید شد که ماشین اکنون به جای ۵ در زمان ۳ شروع به رانندگی می کند:

```
>>> 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
```

منابع مشترک

سیمپای سه نوع منبع را ارائه می دهد که به شما در مدل سازی مسائل کمک می کند ، جایی که چندین فرآیند می خواهند از منبعی با ظرفیت محدود (به عنوان مثال اتومبیل های ایستگاه سوخت با تعداد محدودی پمپ سوخت) یا مسائل تولید کننده-مصرف کننده کلاسیک استفاده کنند.

در این بخش ، ما به طور خلاصه کلاس Resource SimPy را معرفی خواهیم کرد.

پایه ای ترین استفاده ها از منابع

میخواهیم فرایند خودروی الکتریکی را که مورد بحث قرار دادیم اندکی تغییر دهیم.

اتومبیل اکنون به یک ایستگاه شارژ باتری [۱۸]می رود و یکی از دو محل شارژ آن را درخواست می کند. اگر در حال حاضر از هر دو این نقاط استفاده شود ، منتظر می ماند تا دوباره یکی از آنها در دسترس قرار گیرد. سپس شارژ باتری خود را شروع کرده و پس از آن از ایستگاه خارج می شود:

```
>>> 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))
```

متد "request منبع یک رویداد ایجاد می کند که به شما اجازه می دهد صبر کنید تا منبع مجدداً در دسترس قرار گیرد. اگر ادامه دهید ، منبع را مال خودتون می کنید تا زمانی که آن را ولش کنید (آزادش کنید).

اگر از منبع با دستور with استفاده کنید همانطور که در بالا نشان داده شده است ، منبع به طور خودکار آزاد می شود. اگر requestرا بدون with استفاده کنید، پس از اتمام استفاده از منبع ، باید دستور release رو بزنید تا از منابع درست استفاده شود.

وقتی منبعی را آزاد می کنید ، روند انتظار بعدی از سر گرفته می شود و اکنون یکی از اسلات های منبع را مالک می کند. به روش FIFO[۱۹]کار میکند.

---

[۱۸] BCS

[۱۹] First in first out

یک منبع هنگام ایجاد نیاز به ارجاع به یک environment و ظرفیت دارد:

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

حال میتوان فرایندهای اتومبیل را ایجاد کرده و به منابع خود و همچنین برخی پارامترهای اضافی به آنها ارجاع دهیم:

```
>>> 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...>
```

سرانجام ، می توانیم شبیه سازی را شروع کنیم. از آنجا که فرایندهای اتومبیل در این شبیه سازی به خودی خود خاتمه می یابند ، نیازی به تعیین یک زمان نداریم - وقتی دیگر هیچ رویدادی باقی نماند ، شبیه سازی به طور خودکار متوقف می شود:

```
>>> 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
```

توجه داشته باشید که دو ماشین اول می توانند بلافاصله پس از رسیدن به BCS شارژ را شروع کنند ، در حالی که ماشین های ۲ و ۳ باید منتظر بمانند.

چگونه میتوان ادامه داد

اگر هنوز مطمئن نیستید که سیمپای نیازهای شما را برآورده می کند یا می خواهید ویژگی های بیشتری را در عمل مشاهده کنید، باید نگاهی به نمونه های مختلفی که ارائه می دهیم بیاندازید.

اگر به دنبال توضیحات دقیق تری از یک جنبه یا ویژگی خاص سیمپای هستید ، بخش راهنمایی های موضوعی [۲۰]می تواند به شما کمک کند.

سرانجام ، یک مرجع API وجود دارد که تمام توابع و کلاس ها را با جزئیات کامل توصیف می کند.

---

[۲۰] Topival Guides

راهنماهای موضعی

این بخشها جنبه های مختلف سیمپای را بصورت عمیق تری پوشش می دهد. فرض بر این است که شما درک اساسی از قابلیت های سیمپای دارید و می دانید که به دنبال چه چیزی هستید.

اصول سیمپای

حال میخواهیم مفاهیم اصلی سیمپای را مورد بررسی قرار دهیم : چگونه کار می کند؟ فرآیندها ، رویدادها و محیط چیست؟ چه کاری می توانم با آنها انجام دهم؟

سیمپای چگونه کار می کند

اگر سیمپای را بشکنید ، فقط یک توزیع کننده رویداد ناهمزمان (ناهمگام) است. شما رویدادهایی را تولید می کنید و آنها را در یک زمان شبیه سازی مشخص برنامه ریزی می کنید. رویدادها براساس اولویت ، زمان شبیه سازی و شناسه رویداد در حال افزایش مرتب می شوند. یک رویداد همچنین لیستی از پاسخگویی ها را دارد که وقتی این رویداد فعال شده و توسط حلقه رویداد پردازش می شود ، اجرا می شود. رویدادها همچنین ممکن است مقدار بازگشتی داشته باشند.

مولفه هایی که در این امر دخیل هستند ، محیط[۲۱] ، رویدادها[۲۲] و عملکردهای فرایندی[۲۳] هستند.

توابع فرآیند مدل شبیه سازی شما را پیاده سازی می کنند ، یعنی رفتار شبیه سازی شما را تعریف می کنند. آنها توابع ساده تولید کننده پایتون هستند که نمونه هایی از رویداد را ارائه می دهند.

محیط این رویدادها را در لیست رویدادهای خود ذخیره می کند و زمان شبیه سازی فعلی را پیگیری می کند.

(اگر یک تابع فرآیند یک رویداد را به همراه داشته باشد ، سیمپای این فرایند را به تماس های برگشتی رویداد اضافه می کند و روند را به حالت تعلیق در می آورد تا رویداد شروع و پردازش شود. وقتی فرایندی که در انتظار یک رویداد است از سر گرفته شود ، مقدار رویداد را نیز دریافت خواهد کرد.)

در اینجا یک مثال بسیار ساده آورده شده است که همه اینها را نشان می دهد.

```
>>> 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
```

تابع فرآیند example در بالا ابتدا یک رویداد Timeout ایجاد می کند. که از env تاخیر و مقدار رو میگیره.

---

[۲۱] environment

[۲۲] events

[۲۳] process function

Timeout در حال حاضر تاخیر خود را برنامه ریزی می کند (به همین دلیل محیط لازم است). سایر رویدادها معمولاً خود را در زمان شبیه سازی فعلی برنامه ریزی می کنند.

سپس تابع فرآیند رویداد را ارائه می دهد و بعداز آن معلق می شود. وقتی SimPy رویداد Timeout را پردازش می کند ، از سر گرفته می شود. تابع پردازش مقدار رویداد را نیز دریافت می کند (۴۲) - البته این اختیاری است ، بنابراین اگر علاقه ای به این مقدار نداشته باشید یا رویداد اصلاً مقداری نداشته باشد ، yield event مشکلی نخواهد داشت.
سرانجام عملکرد فرآیند ، زمان شبیه سازی فعلی (که از طریق ویژگی اکنون[۲۴] محیط قابل دسترسی است) و مقدار Timeout را چاپ می کند.

اگر تمام توابع پردازش مورد نیاز تعریف شده باشد ، می توانید تمام اشیا را برای شبیه سازی خود نمونه سازی کنید. در بیشتر موارد ، شما با ایجاد یک نمونه از محیط کار خود را شروع می کنید ، زیرا هنگام ایجاد هر چیز دیگری لازم است که آن را بسیار منتقل کنید.

شروع عملکرد فرآیند شامل دو چیز است:
۱ برای ایجاد یک شی تولید کننده باید تابع فرآیند را فراخوانی کنید.
۲ سپس شما یک نمونه از پردازش ایجاد کرده و محیط و شی تولید کننده را به آن منتقل می کنید. این یک رویداد مقداردهی اولیه را در زمان شبیه سازی فعلی برنامه ریزی می کند که اجرای عملکرد فرآیند را شروع می کند. نمونه فرآیند نیز رویدادی است که با بازگشت عملکرد فرآیند ایجاد می شود.

سرانجام ، می توانید حلقه رویداد سیمپای را شروع کنید. به طور پیش فرض ، تا زمانی که رویدادهایی در لیست رویدادها وجود داشته باشد ، اجرا خواهد شد ، اما همچنین می توانید زودتر متوقف کنید.
راهنماهای زیر با جزئیات بیشتری به توصیف محیط و تعاملات آن با رویدادها و عملکردهای فرآیند می پردازند.
نسخه بهتر از کد قبل:

```
>>> 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
```

[۲۴] now

محیط

یک محیط شبیه سازی ، زمان شبیه سازی و همچنین برنامه ریزی و پردازش وقایع را مدیریت می کند. همچنین ابزاری را برای گام برداشتن یا اجرای شبیه سازی فراهم می کند.

در شبیه سازی های معمولی از Environment استفاده می شود. برای شبیه سازی در زمان واقعی ، سیمپای RealtimeEnvironment را فراهم کرده است.

کنترل شبیه سازی

سیمپای از نظر اجرای شبیه سازی بسیار انعطاف پذیر است. شما می توانید شبیه سازی خود را تا زمانی که دیگر هیچ رویدادی وجود ندارد ، تا رسیدن به زمان شبیه سازی خاص یا شروع یک رویداد خاص ، اجرا کنید. شما همچنین می توانید از طریق رویداد شبیه سازی به رویداد گام بردارید. علاوه بر این ، می توانید این موارد را هر طور که دوست دارید باهم دیگه ترکیب کنید.
به عنوان مثال ، شما می توانید شبیه سازی خود را تا زمانی که یک اتفاق جالب رخ دهد ، اجرا کنید. سپس می توانید برای مدتی مرحله به مرحله از شبیه سازی گام بردارید. و سرانجام شبیه سازی را اجرا کنید تا دیگر هیچ رویدادی باقی نماند و فرایندهای شما تمام شود.

مهمترین متد در اینجا Environment.run است:
اگر آن را بدون هیچ آرگومنتی فراخوانی کنید env.run ، مراحل شبیه سازی را طی می کند تا دیگر هیچ رویدادی باقی نماند.
در بیشتر موارد توصیه می شود وقتی شبیه سازی خود را به زمان شبیه سازی خاصی می رسانید ، متوقف شوید. بنابراین ، می توانید زمان مورد نظر را از طریق پارامتر until بگذرانید ، به عنوان مثال: env.run (until = ۱۰).
این شبیه سازی با رسیدن ساعت داخلی به ۱۰ متوقف می شود اما هیچ رویدادی را که برای زمان ۱۰ برنامه ریزی شده باشد پردازش نمی کند. این شبیه به یک محیط جدید است که ساعت ۰ است اما (بدیهی است) هنوز هیچ رویدادی پردازش نشده است.
اگر می خواهید شبیه سازی خود را در GUI ادغام کنید و می خواهید یک نوار فرآیند ترسیم کنید ، می توانید بارها و بارها این تابع را با افزایش مقادیر فراخوانی کرده و نوار پیشرفت خود را پس از هر فراخوانی به روز کنید:

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

به جای گذراندن یک عدد برای اجرا ، می توانید هر رویدادی را نیز به آن منتقل کنید. run پس از آن که این رویداد پردازش شد باز خواهد گشت .

با فرض اینکه زمان فعلی ۰ باشد ، `env.run(until=env.timeout(5))` معادل `env.run(until=5)` است.

همچنین می توانید انواع دیگر رویدادها را نیز منتقل کنید (به یاد داشته باشید که یک فرآیند نیز یک رویداد است):

```
>>> 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'
```

برای قدم زدن در شبیه سازی رویداد به رویداد ، محیط peek و step را ارائه می دهد.

peek :اگر رویدادی در آینده برنامه ریزی نشده باشد ، زمان رویداد برنامه ریزی شده بعدی یا بی نهایت (float ('inf')) را برمی گرداند.

Step: رویداد برنامه ریزی شده بعدی را پردازش می کند. اگر هیچ رویدادی در دسترس نباشد ، یک استثنای EmptySchedule را افزایش می دهد.

در یک حالت معمول استفاده ، شما از این روش ها در یک حلقه مانند موارد زیر استفاده می کنید:

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

دسترسی به حالتها!

این محیط به شما امکان می دهد تا زمان شبیه سازی فعلی را از طریق ویژگی Environment.now دریافت کنید. زمان شبیه سازی یک عدد بدون واحد است و از طریق وقایع Timeout افزایش می یابد.

به طور پیش فرض ، now از ۰ شروع می شود ، اما می توانید زمان اولیه_ را به محیط منتقل کنید تا از چیز دیگری استفاده کنید.

نکته

اگرچه زمان شبیه سازی از نظر فنی بدون واحد است ، شما می توانید وانمود کنید که مثلاً در چند ثانیه است و از آن مانند زمانی استفاده می کنید که توسط time.time برگردانده شده است برای محاسبه یک تاریخ یا روز هفته.

ویژگی Environment.active_process در پردازش فعلی قابل مقایسه با os.getpid است. وقتی یک فرآیند در حال اجرا است ، فعال است. وقتی حادثه ای ایجاد می کند غیرفعال می شود (یا به حالت تعلیق در می آید).

بنابراین ، دسترسی به این ویژگی از درون یک تابع پردازش یا تابعی که توسط تابع پردازش شما فراخوانی می شود ، منطقی است:

```
>>> 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
```

یک مورد استفاده مثال زدنی برای این سیستم منابع است: اگر یک تابع پردازش request منبعی را درخواست کند ، منبع فرآیند درخواست را از طریق env.active_process تعیین می کند. نگاهی به کد بیندازید تا ببینید که چگونه این کار را انجام می دهیم.

ایجاد رویداد

برای ایجاد رویدادها ، شما معمولاً باید simpy.events را ایمپورت کنید ، کلاس رویداد را نمونه کنید و مرجعی را به محیط آن منتقل کنید. برای کاهش میزان تایپ ، محیط زیست میانبرهایی برای ایجاد رویداد ارائه می دهد. به عنوان مثال ، Environment.event معادل (simpy.events.Event (env است.

میانبرهای دیگر عبارتند از:

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

جزئیات بیشتر در مورد آنچه که رویدادها انجام می دهند را می توان در راهنمای رویدادها یافت.

https://simpy.readthedocs.io/en/latest/topical_guides/events.html

متفرقه

از پایتون ۳.۳ ، یک تابع ژنراتور می تواند مقدار بازگشتی داشته باشد:

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

در سیمپای، این می تواند برای ارائه مقادیر برگشتی برای فرآیندهایی که می توانند توسط سایر فرآیندها استفاده شوند ، استفاده شود:

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

رویدادها

سیمپای شامل مجموعه ای گسترده از انواع رویدادها برای اهداف مختلف است. همه آنها simpy.events.Event را به ارث می برند. لیست زیر سلسله مراتب رویدادهای ساخته شده در سیمپای را نشان می دهد:

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

این مجموعه رویدادهای اساسی است. وقایع قابل توسعه هستند و به عنوان مثال منابع ، رویدادهای اضافی را تعریف می کنند. در این راهنما ، ما روی حوادث در ماژول simpy.events تمرکز خواهیم کرد. راهنمای منابع ، وقایع مختلف منابع را توصیف می کند.

لینک راهنمای منابع

https://simpy.readthedocs.io/en/latest/topical_guides/resources.html

اصول اولیه رویداد

رویدادهای سیمپای - اگر یکسان نباشند - به موارد تعویق ، آینده یا وعده ها بسیار شبیه هستند. نمونه های کلاس Event برای توصیف هر نوع رویدادی استفاده می شود. رویدادها می توانند در یکی از حالت های زیر باشند. یک رویداد

ممکن است اتفاق بیفتد (تحریک نشود) ،

قرار است اتفاق بیفتد (تحریک شود) یا

اتفاق افتاده است (پردازش شده).

آنها دقیقاً یک مرتبه به ترتیب ترتیب این حالات را رد می کنند. وقایع نیز کاملاً متصل به زمان هستند و زمان باعث می شود که رویدادها وضعیت خود را پیش ببرند.

در ابتدا ، رویدادها تحریک نمی شوند و فقط اشیا در حافظه هستند.

اگر یک رویداد راه اندازی شود ، در یک زمان مشخص برنامه ریزی شده و در صف رویداد سیمپای قرار می گیرد. خاصیت Event.triggered به True تبدیل می شود.

تا زمانی که رویداد پردازش نشود ، می توانید کال بکس را به یک رویداد اضافه کنید. کال بکس فراخوان هایی هستند که رویدادی را به عنوان پارامتر می پذیرند و در لیست Event.callbacks ذخیره می شوند.

یک رویداد زمانی پردازش می شود که سیمپای آن را از صف رویداد بیرون می آورد و همه کالبک ها را فراخوانی می کند. اکنون دیگر امکان افزودن کالبک وجود ندارد. ویژگی Event.processed ، صحیح True می شود.
رویدادها نیز دارای یک ارزش هستند. مقدار را می توان قبل یا هنگام شروع رویداد تنظیم کرد و از طریق Event.value بازیابی می شود.

افزودن کالبکس به یک رویداد

متداول ترین روش برای اضافه کردن پاسخ به یک رویداد ، بازگرداندن آن از عملکرد فرآیند شما (عملکرد رویداد) است. این روش process_resume را به عنوان پاسخگویی اضافه می کند. به این ترتیب فرآیند شما هنگامی که یک رویداد را ارائه می دهد از سر گرفته می شود.

راه انداز رویدادها

وقتی رویدادها راه اندازی می شوند ، می توانند موفق شوند یا شکست بخورند. به عنوان مثال ، اگر یک رویداد در پایان محاسبه راه اندازی شود و همه چیز خوب کار کند ، این رویداد موفق خواهد شد. اگر در هنگام محاسبه استثنائی رخ دهد ، این رویداد از کار می افتد. چندتا تابع هم برای اینا داریم که میتونین برین بخونین.

مثال مورد استفاده برای رویداد

مکانیک ساده ای که در بالا توضیح داده شد ،انعطاف پذیری زیادی در نحوه استفاده از رویدادها (حتی رویداد[۲۵] اصلی) ایجاد می کند.
یک مثال برای این امر این است که می توان رویدادها را به اشتراک گذاشت. آنها می توانند توسط یک فرآیند یا خارج از متن یک فرآیند ایجاد شوند. آنها می توانند به فرآیندهای دیگر منتقل شده و زنجیر شوند:

```
>>> 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/
```

[۲۵] event

بگذارید زمان بگذرد[۲۶]

برای جلو بردن زمان در شبیه سازی از ایونت timeout استفاده میکنیم، دو پارامتر وجود دارد: تاخیر[۲۷] و ارزش[۲۸] که در اینجا ارزش به صورت اختیاری می باشد. بعدم نحوه استفاده ازش رو توضیح داده

فرایندها نیز یک رویداد هستند

فرایندهای سیمپای ویژگی های خوبی دارند که می توانند یک رویداد باشند، به این صورت که یک فرایند میتواند حاصل فرایندی دیگر باشد. این بدان معناست که یک فرآیند می تواند فرآیند دیگری را بوجود آورد، سپس با پایان یافتن، فرآیند دیگر از سر گرفته خواهد شد .مقدار رویداد مقدار بازگشتی آن فرآیند خواهد بود.

```
>>> 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)))
۲۳
```

هنگامی که یک فرایند ایجاد می شود initialize آن رویدادی را برنامه ریزی می کند که هنگام شروع، اجرای فرایند را شروع می کند. اگر نمی خواهید بلافاصله فرایندی شروع شود اما پس از تاخیری خاص[۲۹] ، می توانید استفاده کنید، این روش یک فرایند کمکی را برمی گرداند که قبل از شروع واقعی یک فرایند از *وقفه* استفاده می کند.

منتظر ماندن برای چندین رویداد همزمان

بعضی اوقات اتفاق می افتد که همزمان منتظر نتیجه چندین رویداد باشیم به عوان مثال منتظر یک منبعی هستیم اما زمان نامحدودی نداریم یااینکه ممکن است منتظر باشیم که مجموعه ای از اتفاقات رخ دهد و نتیجه آن حاصل شود. در اینجا راه کاری که سیمپای ارائه می دهد ایونت های anyof و allof می باشد، هر دو لیستی از رویدادهارا به عنوان آرگومان درنظر می گیرند و درصورت تحریک هر یک (حداقل یک مورد) یا همه آنها فعال می شوند.

---

[۲۶] timeout

[۲۷] delay

[۲۸] value

[۲۹] SimPy.until.start_delayed

تعامل فرآیند

شبیه سازی رویداد گسسته تنها با کنش متقابل بین فرآیندها جالب توجه می شود. این قسمت در مورد سه موضوع آموزشی داده می شود:

منتظر ماندن برای فعال شدن دوباره ی فرایندها

در انتظار خاتمه روند دیگری

قطع یک روند دیگر

دو موضوع اول در قسمت راهنمای رویدادها بیان شده اند، اما برای اینکه مطلبی شهید نشود در اینجا هم به آنها اشاره کردیم.

یک احتمال دیگر برای تعامل فرآیندها در قسمت منابع وجود دارد. در یک راهنمای جداگانه در مورد آن بحث کرده ایم.

منتظر ماندن برای فعال شدن دوباره ی فرایندها

تصور کنید که می خواهید یک وسیله نقلیه الکتریکی را با یک کنترل کننده هوشمند شارژ باتری مدل کنید .در حین رانندگی خودرو ، کنترل کننده می تواند منفعل باشد اما پس از اتصال خودرو به شبکه برق برای شارژ باتری ، باید دوباره فعال شود.

در سیمپای ۲ این الگو به عنوان *منفعل شدن/دوباره فعال کردن*[۳۰] شناخته میشد. در سیمپای ۳ میتوانید به سادگی با استفاده از `Event` آنرا انجام دهید:

```
>>> 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)
```

چون `bat_ctrl()` فقط برای رویدادهای معمولی صبر میکند، ما در سیمپای ۳ دیگر آنرا *منفعل شدن/دوباره فعال کردن* *passivate / reactivate* نمینامیم

```
Bat. ctrl. passivating at 0
Start parking at 29
Bat. ctrl. reactivated at 29
Bat. ctrl. passivating at 60
Stop parking at 131
```

[۳۰] *passivate / reactivate*

منابع اشتراکی

منابع اشتراکی یک راه دیگر برای مدل کردن تعامل بین فرایندها است. سیمپای، سه دسته بندی برای منابع معرفی میکند:

منابع: که به مقدار محدودی میتوانند استفاده بشوند.

ظروف: منابعی که تولید و مصرف یک حجم [۳۱]همگن[۳۲](مشابه) و چندبخشی نشده [۳۳]را مدل می کنند. ممکن است پیوسته باشد (مانند آب) یا گسسته باشد (مانند سیب).

انبار: منابعی که اجازه دارند از اشیاء پایتون تولید و مصرف داشته باشند.

ادامه مطالب را میتوانید با کلیک بر روی تیترها بخوانید.

شبیه سازی در زمان واقعی

گاهی اوقات شما نمیخواهید که شبیه سازی در سریع ترین زمان ممکن باشد، میخواهید مانند زمان ساعت دیواری جلو برود. به این نوع شبیه سازی، *شبیه سازی در زمان واقعی* [۳۴]گفته میشود.

شبیه سازی در زمان واقعی ممکن در شرایط زیر لازم باشد:

- اگر یک سخت افزار در حلقه داشته باشید،
- اگر یک واکنش انسانی در شبیه سازی باشد،یا
- اگر بخواهید رفتار الگوریتم را در زمان واقعی آنالیز کنید.

برای خواندن مطالب بیشتر روی تیتر کلیک کنید.

نظارت[۳۵]

مانیتور کردن نسبتاً یک مبحث پیچیده است که کاربردهای فراوان متفاوتی دارد.

برای خواندن مطالب بیشتر میتوانید روی تیتر آن موضوع کلیک کنید.

نظارت بر فرآیندهایتان[۳۶]

استفاده از منابع[۳۷]

ردیابی رویداد[۳۸]

---

[۳۱] bulk
[۳۲] homogeneous
[۳۳] undifferentiated
[۳۴] *real-time simulation*
[۳۵] Monitoring
[۳۶] Monitoring your processes
[۳۷] Resource usage
[۳۸] Event tracing

زمان و برنامه ریزی

هدف از این قسمت این است که شما مفهوم عمیق تری از زمان را در سیمپای درک کنید و بفهمید که زمان در سیمپای چگونه برنامه ریزی شده و رویدادها را پردازش میکند.
با کلیک بر روی تیتر به صفحه مربوطه هدایت خواهید شد.

انتقال از سیمپای ۳ به ۴

انتقال از سیمپای ۲ به ۳

مثال ها

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

منابع API

درباره سیمپای

منبع:

https://simpy.readthedocs.io/en/latest/contents.html

مترجمان:

مجتبی آل حسینی

پردیس عشقی نژاد

استاد مشاور:

جمال زارع پور احمدآبادی

راه های ارتباطی:

Alehoseini.Mojtaba@gmail.com
pardis.esh@gmail.com
https://yazd.ac.ir/people/jzarepour

تابستان ۱۴۰۰