Методические указания к семинару
по теме «Построение программы имитационного моделирования»

В этом примере рассматривается процесс поездок нескольких машин такси с момента выезда из гаража и до момента возвращения в гараж.

Структурно модель состоит из двух логических блоков (рисунок 1): контроллера генератора и процессора, обрабатывающего инициаторы в цикле. Количество циклов задаётся при генерации инициатора. Каждое такси совершает своё фиксированное количество поездок и возвращается в гараж. Такси выезжает из гаража и начинает искать пассажира. Поиск продолжается, пока пассажир не сядет в такси, в этот момент начинается поездка. Когда пассажир выходит из машины, такси возвращается в режим поиска. По завершении своих циклов инициатор уничтожается. Время поиска и поездок имеют экспоненциальное распределение. Для простоты отображения время измеряется в минутах (для моделирования можно применять и интервалы типа float).

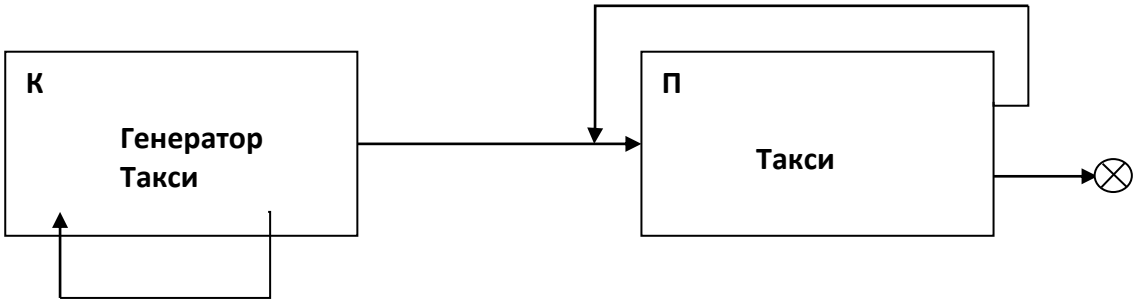

Рисунок 1 – Блочная схема логики модели

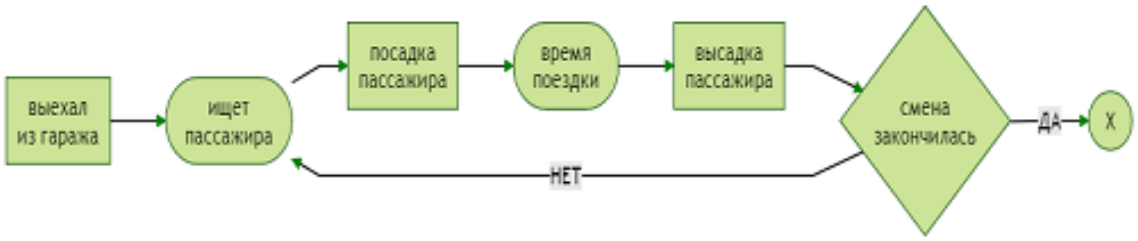

Рисунок 2 – ОПС блока-процессор «Такси»

В программе моделирования **taxi_sim** создаётся несколько экземпляров инициаторов, представляющих в модели машины такси. Всякое изменение состояния любого такси выводится как событие.

На рисунке 3 показан пример результата прогона программы. На рисунке видно наглядное чередование поездок всех такси. На рисунке вручную добавлены стрелки, чтобы поездки было лучше видно: стрелка начинается в момент посадки пассажира и кончается в момент высадки. Это помогает представить, как используются сопрограммы (особая форма функции в Python) для управления (квази)параллельными действиями (с ключевым словом yield).

Обратите внимание, что:

- Интервал между выездами такси из гаража составляет 5 минут.
- В такси 0 первый пассажир сел через 1 минуту после выезда, в момент времени 1; в такси 1 через 4 минуты после выезда (время=9), а в такси 2 — через 3 минуты (время=13).
- Водитель такси 0 сделал четыре поездки: первая началась в момент времени 1; вторая началась в момент времени 24; третья началась в момент времени 45; четвертая началась в момент времени 72; и вернулся в гараж в момент времени 94.

- Такси 1 сделало 3 поездки, после чего вернулось в гараж в момент времени 68.
- Такси 2 сделало 3 поездки и вернулось в гараж в момент времени 89.
- Такси 3 сделало 3 поездки и вернулось в гараж в момент времени 91.
- В момент времени 45 совпало начало у двух поездок – для такси 0 и такси 1.
- В момент времени 88 совпало завершение для двух поездок – для такси 0 и такси 3. (короткие стрелки).

Рисунок 3 - пример прогона программы **taxi_sim**

В этом прогоне все запланированные события завершились в отведенное время моделирования (180 минут); последнее событие произошло в момент time=94. Но может случиться и так, что время моделирование закончилось, а события еще остались. В таком случае последнее сообщение выглядело бы так:

```
*** время закончилось: 2 события остались ***
```

В этом коде важны две функции: **taxi_process** (типа сопрограмма - coroutine) и метод **Simulator.run**, в котором выполняется главный цикл моделирования.

Ниже показан код функции **taxi_process**. В ней используются два объекта, определенных далее: функция **compute_delay**, которая возвращает временной интервал в минутах, и класс **Event** — именованный кортеж, определенный как:

```
Event = collections.namedtuple('событие', 'время ID состояние')
```

В экземпляре event атрибут время - это модельное время события, ID - идентификатор процесса

такси, а состояние — строка, описывающая действие. Рассмотрим код taxi_process.

```
# начало функции TAXI_PROCESS
def taxi_process(ident, trips, start_time=0):  # <1>
    # выдать текущее состояние объекта-такси планировщику модели
    time = yield Event(start_time, ident, 'выехал из гаража') # <2>
    for i in range(trips):  # <3>
        time = yield Event(time, ident, 'посадил пассажира') # <4>
        time = yield Event(time, ident, 'высадил пассажира') # <5>

    yield Event(time, ident, 'уехал в гараж')  # <6>
# конец TAXI_PROCESS # <7>
```

Пояснения к коду:

1. **taxi_process** вызывается один раз для каждой машины такси и создает объект-генератор, представляющий его действия. **ident** - это номер такси (в нашем примере 0, 1, 2), **trips** - сколько поездок должно совершить такси, перед тем как вернуться в гараж; **start_time** — когда такси выезжает из гаража.
2. Первый отданный объект **Event** — (выехал из гаража). Выполнение сопрограммы приостанавливается, так что главный цикл моделирования может перейти к следующему запланированному событию. Когда настанет время возобновить этот процесс, главный цикл отправит (методом **send**) текущее модельное время, которое будет присвоено в переменную **time**.
3. Этот блок кода повторяется по одному разу для каждой поездки.
4. Отдается событие посадки пассажира. Здесь сопрограмма приостанавливается. Когда настанет время возобновить этот процесс, главный цикл снова отправит текущее время.
5. Отдается событие высадки пассажира. Сопрограмма снова приостанавливается и ждет, когда главный цикл отправит ее время возобновления.
6. Цикл **for** заканчивается после заданного числа поездок и выдается последнее событие (уехал в гараж). Сопрограмма приостанавливается в последний раз. При возобновлении она получит от главного цикла модельное время, но его не присваиваем никакой переменной, потому что оно не будет использоваться.
7. Когда сопрограмма доходит до конца, объект-генератор возбуждает исключение StopIteration.

Изучите работу модели, вызывая из консоли функцию **taxi_process** с другими параметрами. Например, так:

```
>>>from taxi_simmod import taxi_process
>>>taxi =taxi_process(13,2,0)
>>>next(taxi)
событие(время=0, ID=13, состояние='выехал из гаража')
>>>taxi.send(_.time+7)
событие(время=7, ID=13, состояние='посадил пассажира')
>>>taxi.send(_.time+23)
событие(время=30, ID=13, состояние='высадил пассажира')
>>>taxi.send(_.time+5)
событие(время=35, ID=13, состояние='посадил пассажира')
>>>taxi.send(_.time+48)
событие(время=83, ID=13, состояние='высадил пассажира')
>>>taxi.send(_.time+2)
событие(время=85, ID=13, состояние='уехал в гараж')
```

Пояснения к коду:

1. Создаём объект-генератор, представляющий такси ident=13, которое сделает две поездки и начнет работать в момент t=0.
2. Инициализируем сопрограмму, она выдаст начальное событие.

3. Теперь можно отправить ей текущее время. В оболочке переменная **_** (нижнее подчеркивание) связана с последним результатом; например, прибавим 7 к текущему времени, т.е. такси потратит на поиск первого пассажира 7 минут.
4. Это событие выдается циклом for в начале первой поездки.
5. Отправка _.time+23 означает, что поездка с первым пассажиром займет 23 минуты.
6. Затем такси будет 5 минут искать пассажира.
7. Последняя поездка займет 48 минут.
8. После завершения двух поездок цикл заканчивается и выдаётся событие ‘going home’.
9. Следующая попытка послать что-то сопрограмме приводит к естественному возврату из нее. В этот момент интерпретатор возбуждает исключение StopIteration.

В примере используем оболочку для имитации с главным циклом моделирования. Получаем атрибут **time** объекта **Event**, отданного сопрограммой **taxi**, прибавим к нему произвольное число, и отправим сумму методом **taxi.send** для возобновления сопрограммы. При моделировании все сопрограммы, представляющие такси, управляются главным циклом в методе **Simulator.run**. Часы модельного времени хранятся в переменной **sim_time** и обновляются временем каждого выданного события.

Чтобы создать экземпляр класса **Simulator**, функция **main** из модели **taxi_sim** строит словарь **taxis**:

```
taxis = {i: taxi_process(i, int(random.uniform(9,15)), i*DEPARTURE_INTERVAL) \
            for i in range(num_taxis)}
sim = Simulator(taxis)
```

Количество поездок во втором параметре задается распределением в интервале 9 .. 15. Поэтому значениями в словаре taxis будут объекты-генераторы с разными параметрами. Например, такси 1 совершит 4 поездки и начнет поиск пассажиров в момент start_time=5. Этот словарь — единственный аргумент, необходимый для создания объекта **Simulator**.

Перечислим структуры данных класса **Simulator**:

self.events

Очередь с приоритетами **PriorityQueue** для хранения объектов **Event**. Структура **PriorityQueue** позволяет помещать элементы методом **put** и извлекать их методом **get** в порядке, определяемом элементом **item[0];** в случае именованных кортежей **Event** это атрибут **time**.

self.procs

Словарь **dict** отображающий номер процесса на активный процесс моделирования — объект-генератор, представляющий одно такси. Он будет связан с копией словаря **taxis**, показанного выше.

```
# BEGIN TAXI_SIMULATOR
class Simulator:

    def __init__(self, procs_map):
        self.events = queue.PriorityQueue()
        self.procs = dict(procs_map)
```

Очередь **PriorityQueue** для хранения запланированных событий упорядочена по возрастанию времени.

Мы получаем аргумент **procs_map** в виде словаря (или произвольного отображения), но строим из него другой словарь, чтобы иметь локальную копию, потому что по ходу моделирования каждое такси, возвращающееся в гараж, удаляется из **self.procs**, а мы не хотим изменять объект, переданный пользователем.

PriorityQueue – полезная структура данных при моделировании дискретных событий: события создаются в произвольном порядке, помещаются в очередь, и позже извлекаются в порядке

запланированного времени события. Например, мы могли бы в самом начале поместить в очередь такие два события:

```
Event(time=14, proc=0, action=’pick up passenger’)
Event(time=11, proc=1, action=’pick up passenger’)
```

В данном случае это означает, что такси 0 потребуется 14 минут для поиска первого пассажира, а такси 1, которое выехало из гаража в момент time=10 – всего 1 минуту, первый пассажир сядет в момент time=11. Если эти два события находятся в очереди, то первым главный цикл извлечёт событие Event(time=11, proc=1, action= ’посадил пассажира’).

Теперь рассмотрим основной алгоритм моделирования - метод **Simulator.run**. Он вызывается из функции **main** сразу после создания объекта **Simulator**:

```
sim= Simulator (taxis)
sim.run (end_time)
```

Ниже приведен текст класса **Simulator** с аннотациями, а здесь дадим общий обзор алгоритма:

1. Цикл по процессам, представляющим такси.

а) Инициализировать сопрограмму для каждого такси, вызвав для нее функцию **next()**. В ответ будет выдано первое событие для такси.
б) Поместить каждое событие в очередь **self.events** объекта **Simulator**.

2. Выполнять главный цикл моделирования, пока **sim_time < end_time**.

а) Проверить, пуста ли очередь **self.events**; если да, выйти из цикла.
б) Получить из **self.events** текущее событие **current_event.** Это будет объект **Event** с наименьшим временем.
в) Вывести **Event**.
г) Обновить модельное время, присвоив ему значение атрибута **time** объекта **current_event.**
д) Отправить время сопрограмме, определяемой атрибутом **proc** объекта **current_event.** Сопрограмма отдаст следующее событие **next_event.**
е) Запланировать **next_event**, поместив его в очередь **self.events.**

```
class Simulator:

    def __init__(self, procs_map):
        self.events = queue.PriorityQueue()
        self.procs = dict(procs_map)

    def run(self, end_time):  # <1>
        # планировать и показывать события до конца модельного периода
        # запланировать первое событие для такси
        for _, proc in sorted(self.procs.items()):  # <2>
            first_event = next(proc)  # <3>
            self.events.put(first_event)  # <4>

        # основной цикл модели
        sim_time = 0  # <5>
        while sim_time < end_time:  # <6>
            if self.events.empty():  # <7>
                print(' ***события закончились!***')
                break

            current_event = self.events.get()  # <8>
```

```
                sim_time, proc_id, previous_action = current_event  # <9>
                print('такси:', proc_id, proc_id * '   ', current_event)  # <10>
                active_proc = self.procs[proc_id]  # <11>
                next_time = sim_time + compute_duration(previous_action)  # <12>
                try:
                    next_event = active_proc.send(next_time)  # <13>
                except StopIteration:
                    del self.procs[proc_id]  # <14>
                else:
                    self.events.put(next_event)  # <15>
            else:  # <16>
                msg = '*** время закончилось: {} события остались ***'
                print(msg.format(self.events.qsize()))
    # конец класса TAXI_SIMULATOR
```

Пояснения к коду:

1. Окончание модельного времени **end_time** — единственный обязательный аргумент **run**.
2. Используем функцию **sorted** для выборки элементов **self .procs**, упорядоченных по ключу; сам ключ нам не важен, поэтому присваиваем его переменной _.
3. Вызов **next(proc)** инициализирует каждую сопрограмму, заставляя ее дойти до первого предложения **yield**, после чего ей можно посылать данные. Отдается объект **Event**.
4. Помещаем каждое событие в очередь с приоритетами **self.events**. Первым событием для каждого такси является 'leave garage', как видно из распечатки прогона.
5. Обнуляем часы модельного времени **sim_time**.
6. Главный цикл моделирования: выполнять, пока **sim_time** меньше **end_time**.
7. Выход из главного цикла производится и тогда, когда в очереди не осталось событий.
8. Получаем из очереди объект **Event** с наименьшим значением **time;** присваиваем его переменной **current_event**.
9. Распаковываем кортеж **Event**. В этой строке часы модельного времени **sim_time** приводятся в соответствии с временем события.
10. Распечатываем объект **Event**: выводим идентификатор такси и соответствующий ему отступ.
11. Извлекаем сопрограмму для активного такси из словаря **self.procs**.
12. Вычисляем время следующего возобновления, складывая **sim_time** и результат вызова функции **compute_duration(..)** для предыдущего действия (т.е. ‘посадил пассажира’, 'высадил пассажира’ и т.д.)
13. Отправляем **time** сопрограмме такси. Сопрограмма отдаст **next_event** или возбудит исключение **StopIteration** по завершении.
14. Если возникло исключение **StopIteration**, удаляем сопрограмму из словаря **self.procs**.
15. В противном случае помещаем **next_event** в очередь.
16. Если произошел выход из цикла в связи с истечением времени, печатаем количество оставшихся в очереди событий (иногда оно может оказаться равным нулю).

Обратите внимание, что в методе **Simulator.run** в двух местах используются блоки **else**, не имеющие отношение к предложению **if:**

- в главном цикле **while** часть **else** выполняется, когда моделирование завершилось из-за превышения **end_time**, а не потому, что в очереди не осталось событий.

- в предложении **try** в конце цикла **while** мы пытаемся получить **next_event**, отправляя **next_time** процессу активного такси, и если не возникло ошибки, то в блоке **else** помещаем **next_event** в очередь **self.events**.

Целью этого примера было показать способ построения цикла обработки событий, который управляет сопрограммами, отправляя им данные. На этом примере видно, как можно использовать генераторы вместо потоков или обратных вызовов для моделирования параллельных процессов.

В событийно-ориентированных программах на основе сопрограмм каждая параллельная операция выполняется сопрограммой, которая периодически уступает управление главному циклу, давая возможность поработать другим сопрограммам. Это вариант невытесняющей многозадачности – сопрограммы добровольно и явно уступают управление центральному планировщику.

[Здесь принято широкое неформальное определение сопрограммы: генераторная функция, управляемая клиентским кодом, который посылает ей данные с помощью метода send или предложения yield from. Такое определение используется в документе PEP342 Coroutines…]

(пример кода из книги «Лучано Рамальо. Python.К вершинам мастерства /Пер.с англ. Слинкин А.А.-М.:ДМК Пресс, 2016) [электронный ресурс https://ru.scribd.com/document/342687030/Python-K-Vershinam-Masterstva]

Задание:

1) Добавьте в модель для каждого такси счетчики -
   - количества поездок,
   - суммарной длительности поездок с пассажиром,
   - суммарного времени поиска пассажира.
   Выведите их значения на консоль при завершении работы такси (при статусе 'уехал в гараж').
2) Изменяйте параметры создания такси (меняйте i от 3 до 9, параметры trips и start_time):

```
taxis ={i: taxi_process(i,(i+1)*3,i*DEPARTURE_INTERVAL) for i in range(num_taxis)}
```

   Понаблюдайте изменения (т.е. сравните 3-4 варианта) в получаемой статистике.

3) Отобразите на графике разными маркерами моменты начала и завершения поездок.