Многопоточность в Python

Многопоточность - это основная концепция программирования, которую поддерживают почти все языки программирования высокого уровня. В этом уроке по многопоточности в Python мы рассмотрим различные методы создания потоков и реализации синхронизации.

Давайте сначала узнаем, что такое поток и что означает многопоточность в компьютерных науках.

Что такое Thread в информатике?

В программировании поток - это наименьшая единица выполнения с независимым набором инструкций. Он является частью процесса и работает в таких же исполняемых ресурсах программы совместного использования контекста, как память. Поток имеет начальную точку, последовательность выполнения и результат. Он имеет указатель инструкций, который хранит текущее состояние потока и контролирует, что будет выполнено в следующем порядке.

Что такое Multithreading в информатике?

Способность процесса выполнять несколько потоков параллельно называется многопоточностью. В идеале многопоточность может значительно улучшить производительность любой программы. И механизм многопоточности Python довольно удобен для пользователя, который вы можете быстро освоить.

Плюсы многопоточности:

- Multithreading может значительно повысить скорость вычислений в многопроцессорных или многоядерных системах, поскольку каждый процессор или ядро обрабатывает отдельный поток одновременно.
- Multithreading позволяет программе оставаться отзывчивой, пока один поток ожидает ввода, а другой одновременно запускает графический интерфейс. Это утверждение справедливо как для многопроцессорных, так и для однопроцессорных систем.
- Все потоки процесса имеют доступ к его глобальным переменным. Если глобальная переменная изменяется в одном потоке, она видна и другим потокам. Поток также может иметь свои собственные локальные переменные.

Минусы многопоточности:

- В однопроцессорной системе многопоточность не влияет на скорость вычислений. Фактически производительность системы может снизиться из-за накладных расходов на управление потоками.
- Синхронизация необходима, чтобы избежать взаимного исключения при доступе к общим ресурсам процесса. Это напрямую ведет к увеличению использования памяти и процессора.
- Многопоточность увеличивает сложность программы, что также затрудняет отладку. Это повышает вероятность потенциальных ошибок.
- Это может вызвать голод, когда поток не получает регулярный доступ к общим ресурсам. Тогда он не сможет возобновить свою работу.

Python модули

Python предлагает два модуля для реализации потоков в программах.

- ***thread***
- ***threading***

Для вашей информации, модуль ***thread*** устарел в Python 3 и переименован в модуль **_thread** для обратной совместимости. Но мы объясним оба метода, потому что многие пользователи все еще используют устаревшие версии Python.

Основное различие между этими двумя модулями состоит в том, что модуль реализует поток как функцию. С другой стороны, модуль предлагает объектно-ориентированный подход, позволяющий создавать потоки.

Как использовать модуль Thread для создания потоков?

Если вы решили применить модуль *thread* в своей программе, используйте следующий метод для создания потоков.

```
thread.start_new_thread ( function, args[, kwargs] )
```

Этот метод является довольно простым и эффективным способом создания потоков. Вы можете использовать его для запуска программ в Linux и Windows.

Этот метод запускает новый поток и возвращает его идентификатор. Он вызовет функцию, указанную в качестве параметра «function» с переданным списком аргументов. Когда возвращается функция, поток молча завершает работу.

Здесь args - это кортеж аргументов; используйте пустой кортеж для вызова без каких-либо аргументов. Необязательный аргумент указывает словарь аргументов с ключевыми словами.

Если функция завершается с необработанным исключением, выводится трассировка стека, а затем поток выходит (это не влияет на другие потоки, они продолжают работать). Используйте приведенный ниже код, чтобы узнать больше о многопоточности.

Базовый пример Multithreading Python

```
# Пример многопоточности Python.
# 1. Рассчитать факториал с помощью рекурсии.
# 2. Вызовите факториальную функцию, используя поток.

from thread import start_new_thread

threadId = 1

def factorial(n):
  global threadId
  if n < 1:
      print "%s: %d" % ("Thread", threadId )
      threadId += 1
      return 1
  else:
      returnNumber = n * factorial( n - 1 )  # рекусрсивыний вызов
```

```
        print(str(n) + '! = ' + str(returnNumber))
        return returnNumber

start_new_thread(factorial,(5, ))
start_new_thread(factorial,(4, ))

c = raw_input("Waiting for threads to return...\n")
```

Вы можете запустить приведенный выше код в своем локальном терминале или использовать любой онлайн-терминал. Как только вы запустите эту программу, она выдаст следующее.

```
Waiting for threads to return...
Thread: 1
1! = 1
2! = 2
3! = 6
4! = 24
Thread: 2
1! = 1
2! = 2
3! = 6
4! = 24
5! = 120
```

Как использовать модуль Threading для создания потоков?

Последний модуль ***threading*** предоставляет богатые возможности и большую поддержку потоков, чем устаревший модуль *thread*, описанный в предыдущем разделе.
Модуль ***threading*** является отличным примером многопоточности Python.

Модуль объединяет все методы модуля *thread* и предоставляет несколько дополнительных методов.

- **threading.activeCount()**: находит общее число активных объектов потока.
- **threading.currentThread()**: его можно использовать для определения количества объектов потока в элементе управления потоком вызывающей стороны.
- **threading.enumerate()**: он предоставит вам полный список объектов потока, которые в данный момент активны.

Помимо описанных выше методов, модуль также представляет класс ***Thread***, который вы можете попробовать реализовать в потоках. Это объектно-ориентированный вариант многопоточности Python.

Класс имеет следующие методы.

Методы класса	Описание метода
run():	Это функция точки входа для любого потока.

start():	запускает поток при вызове метода **run**.
join([time]):	позволяет программе ожидать завершения потоков.
isAlive():	проверяет активный поток.
getName():	извлекает имя потока.
setName():	обновляет имя потока.

При желании вы можете обратиться к родной документации Python, чтобы глубже изучить функциональность модуля *threading*.

Шаги для реализации потоков с помощью модуля Threading

Вы можете выполнить следующие шаги для создания нового потока с помощью модуля .

- Создайте класс наследовав его от **Thread**.
- Переопределите метод __ **init** __ **(self [, args])** для предоставления аргументов в соответствии с требованиями.
- Затем переопределите метод **run(self [, args])**, чтобы создать бизнес-логику потока.

Как только вы определили новый подкласс *Thread*, вы должны создать его экземпляр, чтобы начать новый поток. Затем вызовите метод для его запуска. В конечном итоге он вызовет метод для выполнения бизнес-логики.

Пример - создание класса потока для печати даты

```
# Пример многопоточности Python для печати текущей даты.
# 1. Определите подкласс, используя класс Thread.
# 2. Создайте подкласс и запустите поток.

import threading
import datetime

class myThread (threading.Thread):
   def __init__(self, name, counter):
      threading.Thread.__init__(self)
      self.threadID = counter
      self.name = name
      self.counter = counter
   def run(self):
      print("Starting " + self.name)
      print_date(self.name, self.counter)
      print("Exiting " + self.name)

def print_date(threadName, counter):
   datefields = []
   today = datetime.date.today()
   datefields.append(today)
   print(
```

```
        "%s[%d]: %s" % ( threadName, counter, datefields[0] )
    )

# Создать треды
thread1 = myThread("Thread", 1)
thread2 = myThread("Thread", 2)

# Запустить треды
thread1.start()
thread2.start()

thread1.join()
thread2.join()
print("Exiting the Program!!!")
```

Python Multithreading - синхронизация потоков

Модуль имеет встроенную функциональность для реализации блокировки, которая позволяет синхронизировать потоки. Блокировка необходима для контроля доступа к общим ресурсам для предотвращения повреждения или пропущенных данных.

Вы можете вызвать метод **Lock()**, чтобы применить блокировки, он возвращает новый объект блокировки. Затем вы можете вызвать метод захвата (блокировки) объекта блокировки, чтобы заставить потоки работать синхронно.

Необязательный параметр блокировки указывает, ожидает ли поток получения блокировки.

- В случае, если блокировка установлена на ноль, поток немедленно возвращается с нулевым значением, если блокировка не может быть получена, и 1, если блокировка получена.
- В случае, если для блокировки задано значение 1, поток блокируется и ожидает снятия блокировки.

Метод **release()** объекта блокировки используется для снятия блокировки, когда она больше не требуется.

Просто для вашей информации, встроенные в Python структуры данных, такие как списки, словари, являются поточно-ориентированными, что является побочным эффектом наличия атомарных байт-кодов для управления ими. Другие структуры данных, реализованные в Python или базовые типы, такие как целые числа и числа с плавающей запятой, не имеют такой защиты. Для защиты от одновременного доступа к объекту мы используем объект **Lock**.

Пример блокировки в многопоточности

```
Пример многопоточности Python для демонстрации блокировки.
# 1. Определите подкласс, используя класс Thread.
# 2. Создайте подкласс и запустите поток.
# 3. Реализуйте блокировки в методе выполнения потока.
```

```
import threading
import datetimeexit

Flag = 0

class myThread (threading.Thread):
   def __init__(self, name, counter):
       threading.Thread.__init__(self)
       self.threadID = counter
       self.name = name
       self.counter = counter

   def run(self):
       print("Starting " + self.name)

       # Получить блокировку для синхронизации потока
       threadLock.acquire()
       print_date(self.name, self.counter)

       # Снять блокировку для следующего потока
       threadLock.release()
       print("Exiting " + self.name)

def print_date(threadName, counter):
   datefields = []
   today = datetime.date.today()
   datefields.append(today)
   print(
      "%s[%d]: %s" % ( threadName, counter, datefields[0] )
   )

threadLock = threading.Lock()
threads = []

# создать треды
thread1 = myThread("Thread", 1)
thread2 = myThread("Thread", 2)

# Запустить треды
thread1.start()
thread2.start()
```

```
# Добавить треды в список
threads.append(thread1)
threads.append(thread2)

# Дождитесь завершения всех потоков
for t in threads:
   t.join()

print "Exiting the Program!!!"
```