

Объектно-ориентированное программирование на Python

Осипов Никита Алексеевич



ЛЕКЦИЯ 10. МНОГОПОТОЧНОСТЬ В PYTHON

Учебные вопросы:

1. Основы многопоточности.
2. Создание потоков.

Понятие потока (thread)

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

Компоненты потока →

Счетчик программ
Стек
Набор регистров
Уникальный идентификатор

Понятие многопоточности

- Способность процесса выполнять несколько потоков параллельно называется многопоточностью
- Основная идея многопоточности заключается в достижении параллелизма путем разделения процесса на несколько потоков

Преимущества

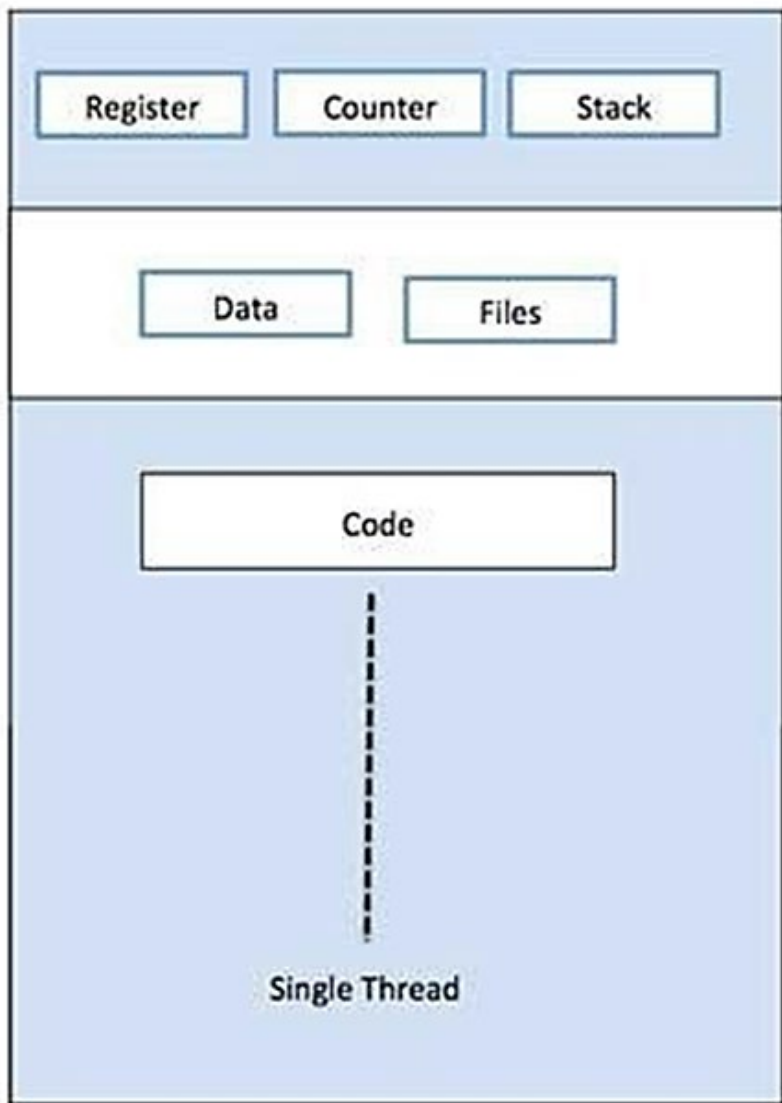
Скорость обмена данными — многопоточность повышает скорость вычислений, поскольку каждое ядро или процессор обрабатывает отдельные потоки одновременно.

Программа остается отзывчивой — она позволяет программе оставаться отзывчивой, поскольку один поток ожидает ввода, а другой одновременно запускает графический интерфейс.

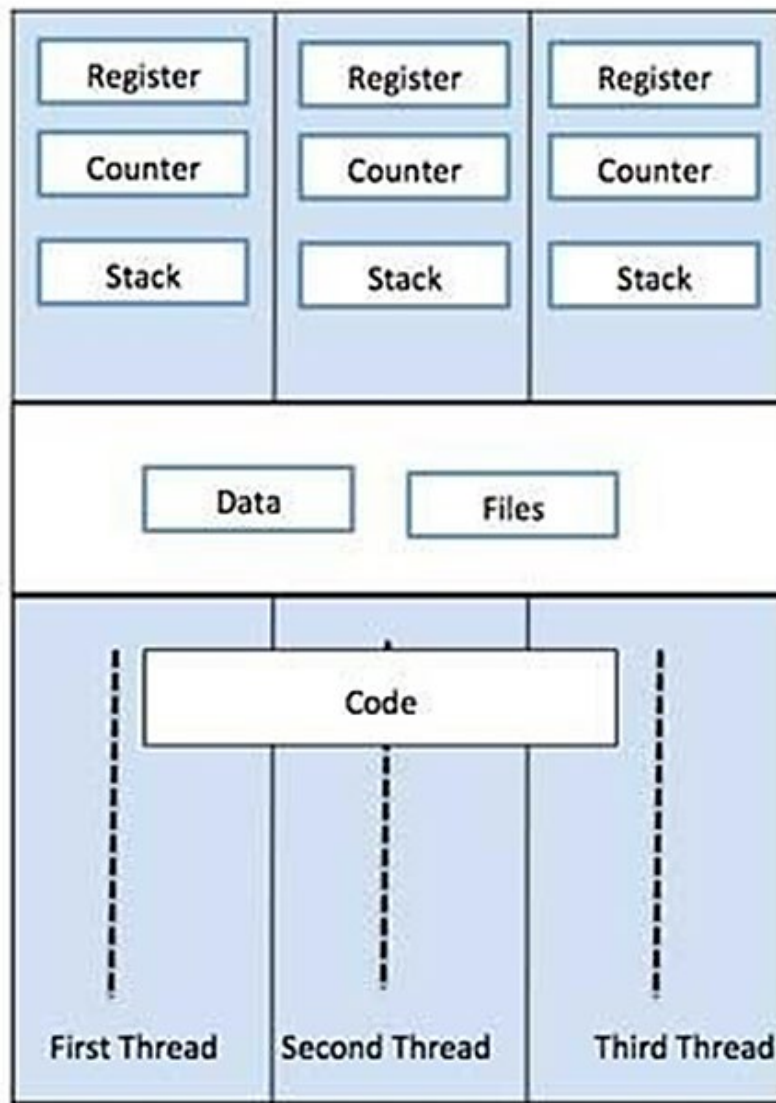
Доступ к глобальным переменным — в многопоточности все потоки определенного процесса могут получить доступ к глобальным переменным, и если есть какое-либо изменение в глобальной переменной, то это видно и другим потокам.

Использование ресурсов — запуск нескольких потоков в каждой программе улучшает использование процессора, а время простоя процессора уменьшается.

Понятие многопоточности



Single Process P with single thread



Single Process P with three threads

- Процесс может иметь только один поток или несколько потоков, имеющих свой собственный набор регистров, счетчик программ и стек

Реализация потоков

- В Python есть два модуля, которые реализуют потоки:
 - `_thread`
 - `threading`
- Основное различие:
 - модуль **`_thread`** обрабатывает поток как функцию, а модуль **`threading`** обрабатывает каждый поток как объект и реализует его объектно-ориентированным способом

В более ранней версии Python был модуль `thread`, но он считается устаревшим и был переименован в `_thread` для обратной несовместимости в Python 3

Модуль `_thread`

`_thread_example.py`

- Для создания потока нужно вызвать метод *start_new_thread*:
 `_thread.start_new_thread (function, args[, kwargs])`
 - `args` – кортеж аргументов
 - `Kwargs` – необязательный словарь аргументов ключевых слов

Модуль threading

hello_threads_example.py

- Модуль threading 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

Модуль threading

hello_threads_example.py

- Модуль threading также представляет класс **Thread** (объектно-ориентированный вариант многопоточности Python):
 - run() является точкой входа для потока.
 - start() запускает поток, вызывая метод run.
 - join([время]) ожидает завершения потоков.
 - isAlive() проверяет, выполняется ли еще поток.
 - getName() возвращает имя потока.
 - setName() устанавливает имя потока.

Создание потоков – функциональный подход

threads_1fun.py

■ Функция Thread() для создания потоков

□ Синтаксис:

variable = Thread(target=function_name, args=(arg1, arg2,))

- параметр target — это «целевая» функция, которая определяет поведение потока и создаётся заранее

```
from threading import Thread
def prescript(thefile, num):
    ...
thread1 = Thread(target=prescript, args=('f1.txt', 200,))
thread2 = Thread(target=prescript, args=('f2.txt', 1000,))
thread1.start()
thread2.start()
thread1.join()
thread2.join()
```

start() запускает ранее созданный поток

join() останавливает поток, когда тот выполнит свои задачи

Создание потоков – классовый подход

threads_1class.py


threads_2class.py

- Определить новый подкласс класса *Thread*
- Для добавления дополнительных аргументов переопределить метод `__init__ (self [, args])`
- Переопределить метод `run (self [, args])`, чтобы реализовать то, что поток должен делать при запуске

```
class myThread (threading.Thread):  
    def __init__(self, ...):  
        threading.Thread.__init__(self)  
  
    ...  
  
    def run(self):  
        ...
```

```
thread1 = myThread("Thread", 1)  
thread1.start()
```

После создания подкласса **Thread** создать его экземпляр и затем запустить новый поток, вызвав **start()**, который, в свою очередь, вызывает метод **run()**



ЛЕКЦИЯ 11. АСИНХРОННОЕ ПРОГРАММИРОВАНИЕ

Учебные вопросы:

1. Основы асинхронного программирования.
2. Модуль Asyncio.

Стратегии минимизации задержек блокирования

Многопроцессорная
обработка
(multiprocessing)

Многопоточность
(threading)

Асинхронность

Понятие асинхронности

Пример asyncio01.py

- Асинхронное программирование – совместная многозадачность (способность одновременно обрабатывать задачи, но контекст переключения “внутри”)

В стандартной (синхронной) программе все инструкции, передаваемые интерпретатору, будут выполняться одна за другой

Асинхронный подход – переключение между задачами для минимизации времени простоя

Модуль для реализации асинхронности

asyncio

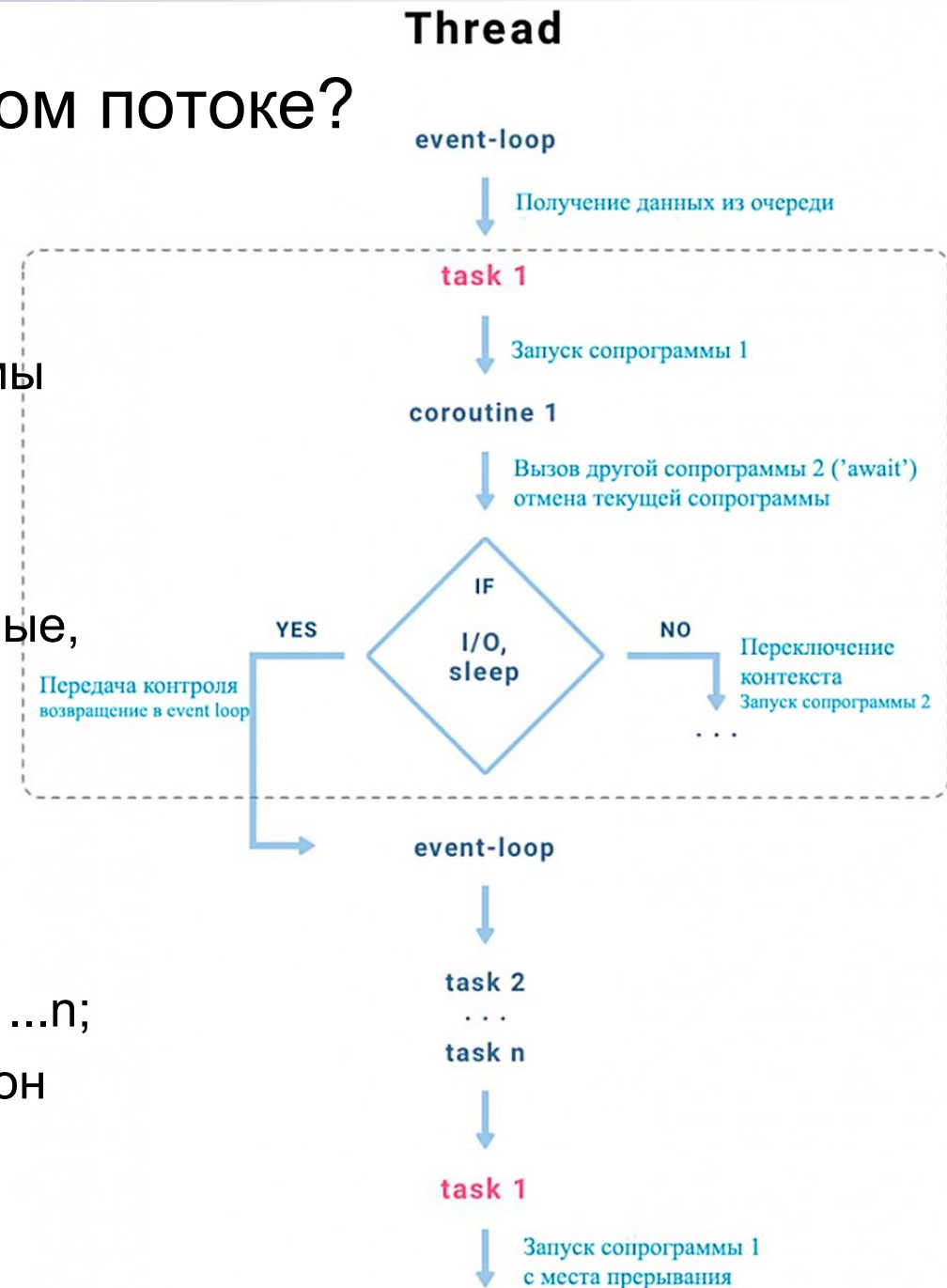
- Модуль для организации конкурентного программирования
 - появился в Python 3.4
 - предназначен для упрощения создания асинхронного кода — чтобы код выглядел как синхронный (без обратных вызовов (коллбэков))
- Модуль предоставляет инфраструктуру для написания однопоточного параллельного кода с использованием
 - сопрограмм, мультиплексирования доступа ввода-вывода через сокеты и другие ресурсы, запуска сетевых клиентов и серверов и других связанных примитивов

Основные понятия asyncio

- event loop (цикл событий) управляет выполнением различных задач: регистрирует поступление и запускает в подходящий момент
- coroutine (cooperative subroutine, корутины, сопрограммы) — для добровольной упреждающей многозадачности: активно уступает свои ресурсы другим подпрограммам и процессам, а не принудительно вытесняется ядром. Необходимо, чтобы они были запущены именно через цикл событий
- Task (задача) – планировщики для сопрограмм, это awaitable (ожидаемый) объект, который оборачивается вокруг подпрограммы (coroutine)
- Future (футуры) — объекты, в которых хранится текущий результат выполнения какой-либо задачи: полученный результат или информация о том, что задача ещё не обработана (может быть информацией об исключении)

Как асинхронность выполняется в одном потоке?

- event loop выполняется в потоке
- выбираются данные из очереди – `get_event_loop()`
- каждая задача вызывает следующий шаг сопрограммы
- если сопрограмма вызывает другую сопрограмму (`await<имя_сопрограммы>`), текущая сопрограмма приостанавливается, и происходит переключение контекста. Контекст текущей сопрограммы (переменные, состояние) сохраняется и загружается контекст вызванной сопрограммы
- если сопрограмма встречает блокирующий код (I/O, `sleep`), текущая сопрограмма приостанавливается, и управление возвращается в event loop;
- event loop получает следующие задачи из очереди 2, ...n;
- затем event loop возвращается к задаче 1, с которой он был прерван



Сопрограммы

- Сопрограммы в asyncio – это генераторы, которые отвечают определённым требованиям.
 - К сопрограммам, основанным на генераторах должен быть применён декоратор `@asyncio.coroutine`.

```
@asyncio.coroutine
```

```
def time_consuming_computation(x):  
    print('Computing {0} ** 2...'.format(x))  
    yield from asyncio.sleep(1)  
    return x ** 2
```

```
@asyncio.coroutine
```

```
def process_data(x):  
    result = yield from time_consuming_computation(x)  
    print('{0} ** 2 = {1}'.format(x, result))
```

Сопрограммы в новом стиле

async_01.py

- Ключевое слово *async*, используемое перед оператором *def*, определяет новую сопрограмму:
 - Coroutine в *asyncio* — это любая функция Python, в определении которой указан префикс ***async***

```
async def async_hello():  
    print("hello, world!")
```

Начиная с версии 3.5
рекомендуется применение *async def*

Объект сопрограммы (coroutine object) ничего не делает, пока его выполнение не запланировано в цикле событий

```
loop = asyncio.get_event_loop()  
loop.run_until_complete(async_hello())
```

Сопрограмма
отправляется на
выполнение в цикл
событий

Применение задач

async_02.py

async_await_01.py

■ Новые задачи можно добавить в цикл

- предоставив другой объект для ожидания использования функции

asyncio.wait():

```
async def print_number(number):  
    print(number)
```

```
loop = asyncio.get_event_loop()  
  
loop.run_until_complete(  
    asyncio.wait([  
        print_number(number)  
        for number in range(10)  
    ])  
)  
loop.close()
```

Функция *asyncio.wait()*
принимает список объектов
сопрограмм и немедленно
возвращается

Результатом является генератор, который
выдает объекты, представляющие будущие
результаты (Future)

- или вызвав метод `loop.create_task()`

task_01.py

task_02.py

task_03.py

Применение задач

■ Новые задачи можно добавить в цикл

- предоставив другой объект для ожидания использования функции `asyncio.wait()`
- или вызвав метод **`loop.create_task()`**

Чтобы создать и сразу
запланировать задачу
(вернет объект задачи)

```
asyncio.create_task(coro(args...))
```

```
import asyncio
```

```
async def my_coro(n):  
    print(f"The answer is {n}.")
```

```
async def main():  
    mytask = asyncio.create_task(my_coro(42))  
    await mytask
```

Создав задачу, вы
запланировали ее запуск
по усмотрению цикла
событий

Программа
останавливается пока
задача не будет завершена

Инициализация главного Event Loop

- Точкой входа в программу asyncio является **asyncio.run(main())**, где `main()` — подпрограмма (coroutine) верхнего уровня
- Вызов `asyncio.run()` неявно создает и запускает event loop (цикл обработки событий)

До Python 7

```
loop = asyncio.get_event_loop()  
loop.run_until_complete(main()) ]
```

Python 7 и старше

```
asyncio.run(main())
```

Функция `asyncio.run()` не может быть вызвана из существующего цикла событий, поэтому, возможны ошибки, если вы запускаете программу в контролирующей среде, такой как Anaconda или Jupyter, которая выполняет собственный цикл обработки событий

- Чтобы отменить запущенную задачу, используйте метод ***cancel()***
 - Его вызов приведет к тому, что задача выдаст исключение **CancelledError** в обернутую сопрограмму.
 - Если сопрограмма ожидает объекта Future во время отмены, объект Future будет отменен.
- ***cancelled()*** можно использовать для проверки того, была ли задача отменена
 - Метод возвращает True, если обернутая сопрограмма не подавила исключение CanceledError и была фактически отменена