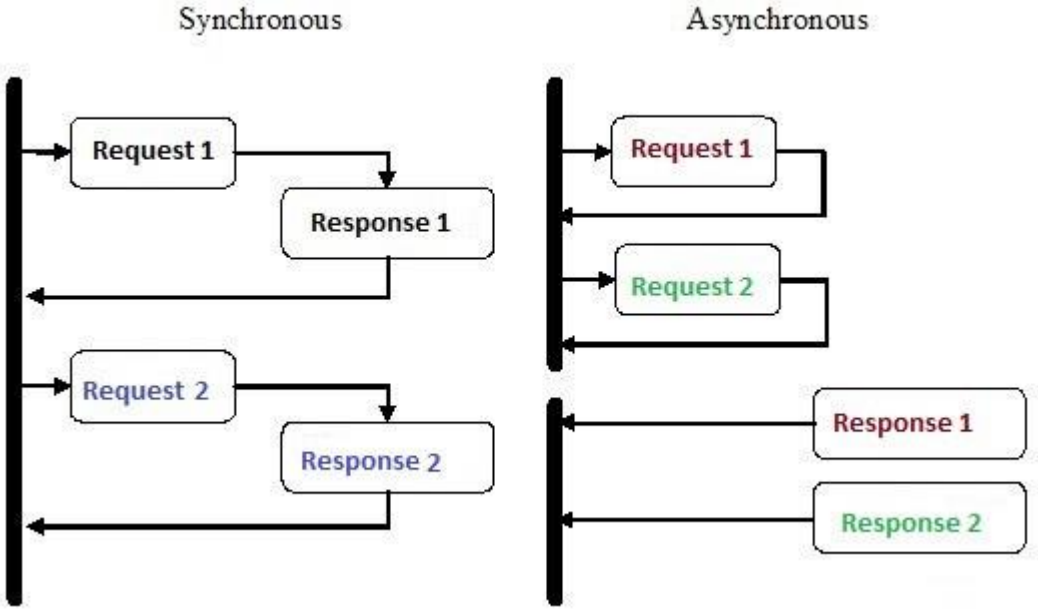
Асинхронное программирование в Python

Асинхронное программирование – это вид параллельного программирования, в котором какая-либо единица работы может выполняться отдельно от основного потока выполнения приложения. Когда работа завершается, основной поток получает уведомление о завершении рабочего потока или произошедшей ошибке. У такого подхода есть множество преимуществ, таких как повышение производительности приложений и повышение скорости отклика.

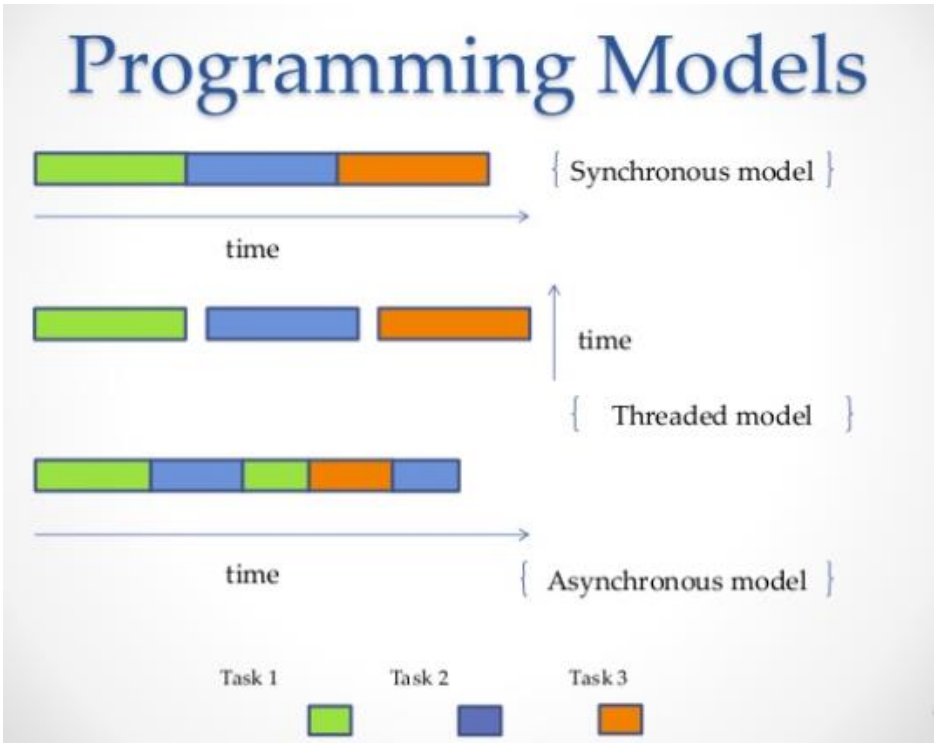


Несмотря на то, что этот вид программирования может быть сложнее традиционного последовательного выполнения, он гораздо более эффективен.

Например, вместо того, что ждать завершения HTTP-запроса перед продолжением выполнения, вы можете отправить запрос и выполнить другую работу, которая ждет своей очереди, с помощью асинхронных корутин в Python.

Асинхронность – это одна из основных причин популярности выбора Node.js для реализации бэкенда. Большое количество кода, который мы пишем, особенно в приложениях с тяжелым вводом-выводом, таком как на веб-сайтах, зависит от внешних ресурсов. В нем может оказаться все, что угодно, от удаленного вызова базы данных до POST-запросов в REST-сервис. Как только вы отправите запрос в один из этих ресурсов, ваш код будет просто ожидать ответа. С асинхронным программированием вы позволяете своему коду обрабатывать другие задачи, пока ждете ответа от ресурсов.

Как Python может делать несколько вещей одновременно?



1. Множественные процессы

Самый очевидный способ – это использование нескольких процессов. Из терминала вы можете запустить свой скрипт два, три, четыре, десять раз, и все скрипты будут выполняться независимо и одновременно. Операционная система сама позаботится о распределении ресурсов процессора между всеми экземплярами. В качестве альтернативы вы можете воспользоваться библиотекой multiprocessing, которая умеет порождать несколько процессов.

```
from multiprocessing import Process

def print_func(continent='Asia'):
    print('The name of continent is : ', continent)

if __name__ == "__main__":  # confirms that the code is under main function
    names = ['America', 'Europe', 'Africa']
    procs = []
    proc = Process(target=print_func)  # instantiating without any argument
    procs.append(proc)
    proc.start()

    # instantiating process with arguments
    for name in names:
        # print(name)
        proc = Process(target=print_func, args=(name,))
        procs.append(proc)
        proc.start()

    # complete the processes
```

```
    for proc in procs:
        proc.join()
```

Вывод:

```
The name of continent is :  Asia
The name of continent is :  America
The name of continent is :  Europe
The name of continent is :  Africa
```

2. Множественные потоки

Еще один способ запустить несколько работ параллельно – это использовать потоки. Поток – это очередь выполнения, которая очень похожа на процесс, однако в одном процессе вы можете иметь несколько потоков, и у всех них будет общий доступ к ресурсам. Однако из-за этого написать код потока будет сложно. Аналогично, все тяжелую работу по выделению памяти процессора сделает операционная система, но глобальная блокировка интерпретатора (GIL) позволит только одному потоку Python запускаться в одну единицу времени, даже если у вас есть многопоточный код. Так GIL на CPython предотвращает многоядерную конкурентность. То есть вы насильно можете запуститься только на одном ядре, даже если у вас их два, четыре или больше.

```
import threading

def print_cube(num):
    """
    function to print cube of given num
    """
    print("Cube: {}".format(num * num * num))

def print_square(num):
    """
    function to print square of given num
    """
    print("Square: {}".format(num * num))

if __name__ == "__main__":
    # creating thread
    t1 = threading.Thread(target=print_square, args=(10,))
    t2 = threading.Thread(target=print_cube, args=(10,))

    # starting thread 1
    t1.start()
    # starting thread 2
    t2.start()

    # wait until thread 1 is completely executed
    t1.join()
    # wait until thread 2 is completely executed
    t2.join()

    # both threads completely executed
```

```
    print("Done!")
```

Вывод:

```
Square: 100
Cube: 1000
Done!
```

3. Корутины и yield:

Корутины – это обобщение подпрограмм. Они используются для кооперативной многозадачности, когда процесс добровольно отдает контроль (yield) с какой-то периодичностью или в периоды ожидания, чтобы позволить нескольким приложениям работать одновременно. Корутины похожи на генераторы, но с дополнительными методами и небольшими изменениями в том, как мы используем оператор yield. Генераторы производят данные для итерации, в то время как корутины могут еще и потреблять данные.

```
def print_name(prefix):
    print("Searching prefix:{}".format(prefix))
    try :
        while True:
                # yeild used to create coroutine
                name = (yield)
                if prefix in name:
                    print(name)
    except GeneratorExit:
            print("Closing coroutine!!")

corou = print_name("Dear")
corou.__next__()
corou.send("James")
corou.send("Dear James")
corou.close()
```

Вывод:

```
Searching prefix:Dear
Dear James
Closing coroutine!!
```

4. Асинхронное программирование

Четвертый способ – это асинхронное программирование, в котором не участвует операционная система. Со стороны операционной системы у вас останется один процесс, в котором будет всего один поток, но вы все еще сможете выполнять одновременно несколько задач.

Asyncio – модуль асинхронного программирования, который был представлен в Python 3.4. Он предназначен для использования корутин и future для упрощения написания асинхронного кода и делает его почти таким же читаемым, как синхронный код, из-за отсутствия callback-ов.

Asyncio использует разные конструкции: event loop, корутины и future.

- event loop управляет и распределяет выполнение различных задач. Он регистрирует их и обрабатывает распределение потока управления между ними.
- Корутины (о которых мы говорили выше) – это специальные функции, работа которых схожа с работой генераторов в Python, с помощью await они возвращают поток управления обратно в event loop. Запуск корутины должен быть запланирован в event loop. Запланированные корутины будут обернуты в Tasks, что является типом Future.
- Future отражает результат таска, который может или не может быть выполнен. Результатом может быть exception.

С помощью asyncio вы можете структурировать свой код так, чтобы подзадачи определялись как корутины и позволяли планировать их запуск так, как вам заблагорассудится, в том числе и одновременно. Корутины содержат точки yield, в которых мы определяем возможные точки переключения контекста. В случае, если в очереди ожидания есть задачи, то контекст будет переключен, в противном случае – нет.

Переключение контекста в asyncio представляет собой event loop, который передает поток управления от одной корутины к другой.

В следующем примере, мы запускаем 3 асинхронных таска, которые по-отдельности делают запросы к Reddit, извлекают и выводят содержимое JSON. Мы используем aiohttp – клиентскую библиотеку http, которая гарантирует, что даже HTTP-запрос будет выполнен асинхронно.

Использование Redis и Redis Queue RQ

Использование asyncio и aiohttp не всегда хорошая идея, особенно если вы пользуетесь более старыми версиями Python. К тому же, бывают моменты, когда вам нужно распределить задачи по разным серверам. В этом случае можно использовать RQ (Redis Queue). Это обычная библиотека Python для добавления работ в очередь и обработки их воркерами в фоновом режиме. Для организации очереди используется Redis – база данных ключей/значений.

В примере ниже мы добавили в очередь простую функцию count_words_at_url с помощью Redis.

```
from mymodule import count_words_at_url
from redis import Redis
from rq import Queue


q = Queue(connection=Redis())
```

```
job = q.enqueue(count_words_at_url, 'http://nvie.com')


******mymodule.py******

import requests

def count_words_at_url(url):
    """Just an example function that's called async."""
    resp = requests.get(url)

    print( len(resp.text.split()))
    return( len(resp.text.split()))
```

Вывод:

```
15:10:45 RQ worker 'rq:worker:EMPID18030.9865' started, version 0.11.0
15:10:45 *** Listening on default...
15:10:45 Cleaning registries for queue: default
15:10:50 default: mymodule.count_words_at_url('http://nvie.com') (a2b7451e-731f-4f31-
9232-2b7e3549051f)
322
15:10:51 default: Job OK (a2b7451e-731f-4f31-9232-2b7e3549051f)
15:10:51 Result is kept for 500 seconds
```

На практике асинхронность определяется как стиль параллельного программирования, в котором одни задачи освобождают процессор в периоды ожидания, чтобы другие задачи могли им воспользоваться. В Python есть несколько способов достижения параллелизма, отвечающих вашим требованиям, потоку кода, обработке данных, архитектуре и вариантам использования.