13. Multiprocessing

Общие сведения​

Входит в стандартную библиотеку Python​

Позволяет запускать задачи в разных процессах​

Процессы управляются операционной системой​

Каждый процесс имеет свою копию интерпретатора и всех ресурсов​

Позволяет получить прирост производительности на многоядерных системах​

Создание нескольких процессов​

import os​

from multiprocessing import Process ​

 ​

def doubler(number):​

    result = number \* 2​

    proc = os.getpid()​

    print('{0} doubled to {1} by process id: {2}'.format(​

        number, result, proc)) ​

 ​

if \_\_name\_\_ == '\_\_main\_\_':​

    numbers = [5, 10, 15, 20, 25]​

    procs = []​

    ​

    for index, number in enumerate(numbers):​

        proc = Process(target=doubler, args=(number,))​

        procs.append(proc)​

        proc.start()​

for proc in procs:​

        proc.join()

Использование замков

from multiprocessing import Process, Lock ​

 ​

def printer(item, lock):​

    """ Выводим то что передали​

    """​

    lock.acquire()​

    try:​

        print(item)​

    finally:​

        lock.release() ​

 ​

if \_\_name\_\_ == '\_\_main\_\_':​

    lock = Lock()​

    items = ['tango', 'foxtrot', 10]​

    ​

    for item in items:​

        p = Process(target=printer, args=(item, lock))​

        p.start()

Создание пула процессов

from multiprocessing import Pool​

 ​

 ​

def doubler(number):​

    return number \* 2​

 ​

 ​

if \_\_name\_\_ == '\_\_main\_\_':​

    numbers = [5, 10, 20]​

    pool = Pool(processes=3)​

    print(pool.map(doubler, numbers))

Использование очереди

from multiprocessing import Process, Queue  ​

sentinel = -1 ​

def creator(data, q):​

    print('Creating data and putting it on the queue')​

    for item in data:​

        q.put(item)  ​

def my\_consumer(q):​

    while True:​

        data = q.get()​

        print('data found to be processed: {}'.format(data))    ​

        processed = data \* 2​

        print(processed)    ​

        if data is sentinel:​

            break ​

if \_\_name\_\_ == '\_\_main\_\_':​

    q = Queue()​

    data = [5, 10, 13, -1]    ​

    process\_one = Process(target=creator, args=(data, q))​

    process\_two = Process(target=my\_consumer, args=(q,))    ​

    process\_one.start()​

    process\_two.start()    ​

    q.close()​

    q.join\_thread()    ​

    process\_one.join()​

    process\_two.join()

## Модуль ****multiprocessing**** был добавлен в Python версии 2.6. Изначально он был определен в PEP 371 Джесси Ноллером и Ричардом Одкерком. Модуль multiprocessing позволяет вам создавать процессы таким же образом, как при создании потоков при помощи модуля threading. Суть в том, что, в связи с тем, что мы теперь создаем процессы, вы можете обойти GIL (Global Interpreter Lock) и воспользоваться возможностью использования нескольких процессоров на компьютере. Пакет ****multiprocessing**** также включает ряд API, которых вообще нет в модуле threading. Например, есть очень удобный ****класс Pool****, который вы можете использовать для параллельного выполнения функции между несколькими входами. Особенности организации многопроцессорной программы в Python. Это использование двух или более процессорных блоков в одной компьютерной системе. Это лучший способ получить полный потенциал от нашего оборудования, используя полное количество процессорных ядер, доступных в нашей компьютерной системе.

## C:\Users\Александр\Desktop\Сайт\H_jh26Afv6k.jpg

## Устранение влияния глобальной блокировки интерпретатора (GIL)

При работе с параллельными приложениями в Python есть ограничение, называемое GIL (Global Interpreter Lock) . GIL никогда не позволяет нам использовать несколько ядер CPU, и поэтому мы можем сказать, что в Python нет настоящих потоков. GIL – мьютекс – блокировка взаимного исключения, которая делает вещи безопасными. Другими словами, мы можем сказать, что GIL препятствует параллельному выполнению кода Python несколькими потоками. Блокировка может удерживаться только одним потоком за раз, и если мы хотим выполнить поток, он должен сначала получить блокировку.

Используя многопроцессорность, мы можем эффективно обойти ограничение, вызванное GIL –

* Используя многопроцессорность, мы используем возможности нескольких процессов и, следовательно, мы используем несколько экземпляров GIL.
* В связи с этим нет ограничений на выполнение байт-кода одного потока в наших программах одновременно.

Используя многопроцессорность, мы используем возможности нескольких процессов и, следовательно, мы используем несколько экземпляров GIL.

В связи с этим нет ограничений на выполнение байт-кода одного потока в наших программах одновременно.

14. Asyncio  
Асинхронное программирование является некоторой моделью программирования, которая сосредоточена на координации различных задач в некотором приложении. Её цель состоит в предоставлении гарантии того, что данное приложение завершит исполнять такие задачи за наименьший возможный промежуток времени. С этой точки зрения асинхронное программирование состоит в переключении с одной задачи на другую когда это соответствует созданию совпадению между временем ожидания и временем обработки, а следовательно, сокращению общего времени, требуемого для завершения данной программы в целом.

Общие сведения  
Введен в Python с версии 3,5​

Позволяет писать конкурентные корутины​

Включает собственную реализацию блокирующих операций​

Позволяет писать ясный код​

Цикл выполнения контролируется интерпретатором​

Позволяет явно прописывать места переключения потока выполнения​

High-level и low-level API

Основные понятия  
корутины — специальные функции, похожие на генераторы python, от которых ожидают (await), что они будут отдавать управление обратно в цикл событий. Необходимо, чтобы они были запущены именно через цикл событий​

task в asyncio – это объект, который оборачивает coroutine, предоставляя методы для контроля ее выполнения и запроса ее статуса. task может быть создан с помощью asyncio.create\_task() или asyncio.gather().​

В asyncio event loop управляет планированием и передачей ожидаемых объектов. Каждая программа asyncio имеет как минимум один event loop. Можно иметь несколько циклов, но в Python 3.7 настоятельно рекомендуется использовать только один.

Awaitable

Любой объект, который можно ожидать прерывание своего процесса выполнения, называется awaitable.​

Ключевое слово await приостанавливает выполнение текущей подпрограммы (coroutine) и вызывает указанное ожидание awaitable.​

​

async def my\_coro():  ​

    pass

Цикл обработки событий  
  
event loop выполняется в потоке;​

получает данные из очереди;​

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.