



Backend разработка на Python

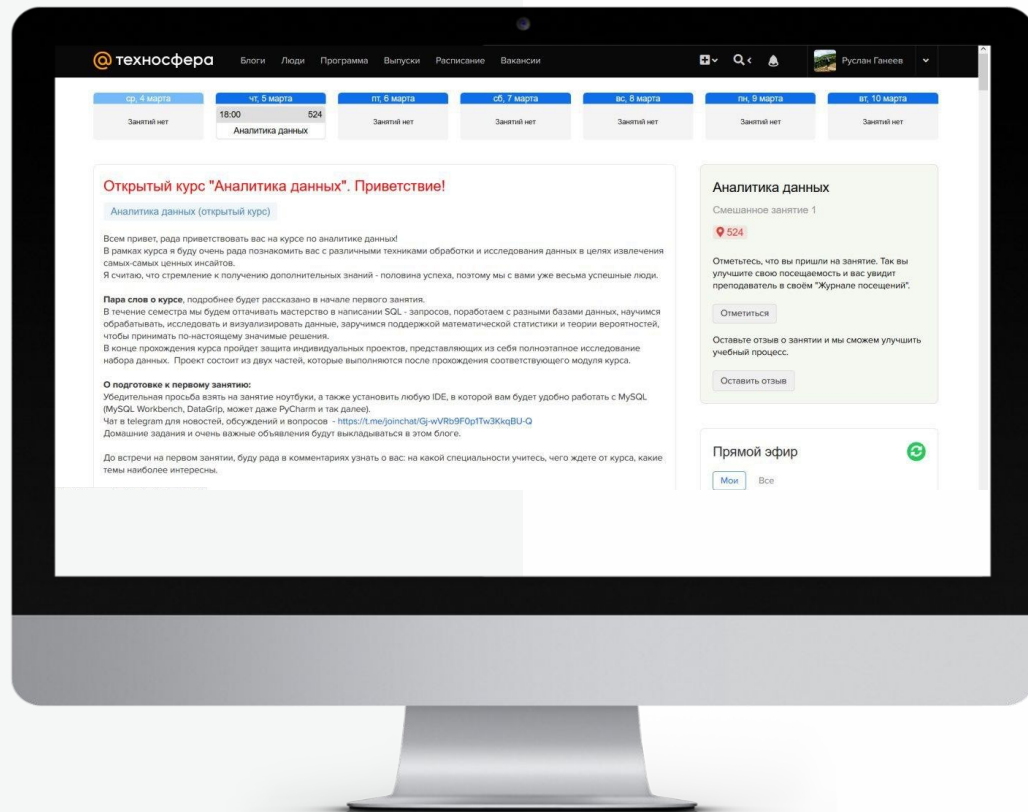
Лекция 9

Потоки, процессы, GIL

Асинхронное программирование

Кандауров Геннадий





Напоминание отметиться на портале

+ отзывы после лекции

Квиз по прошлой лекции




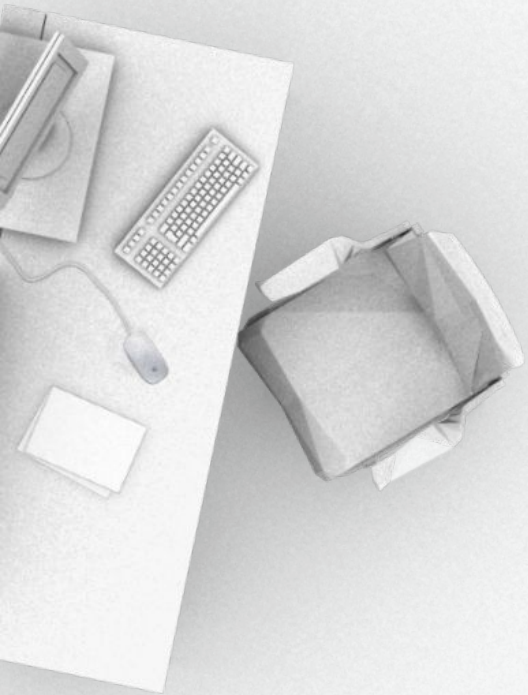
Результаты квиза #7





Содержание занятия


- Потоки
 - GIL
 - Процессы
 - Механизмы синхронизации
 - IPC
 - Асинхронное программирование
 - Event loop
 - Корутины, нативные корутины
 - asyncio
- 



Threads (потоки)




Threads



Thread (поток) - это сущность операционной системы, процесс выполнения на процессоре набора инструкций, а именно программного кода.



Threads: создание и запуск



```
1. class CustomThread(threading.Thread):  
    def __init__(self):  
        pass  
    def run(self):  
        func()
```

```
    th = CustomThread()
```


```
2. th = threading.Thread(target=func)
```

```
th.start()
```

```
th.join()
```




Threads: local



```
import threading

my_data = threading.local()
my_data.x = 42
```



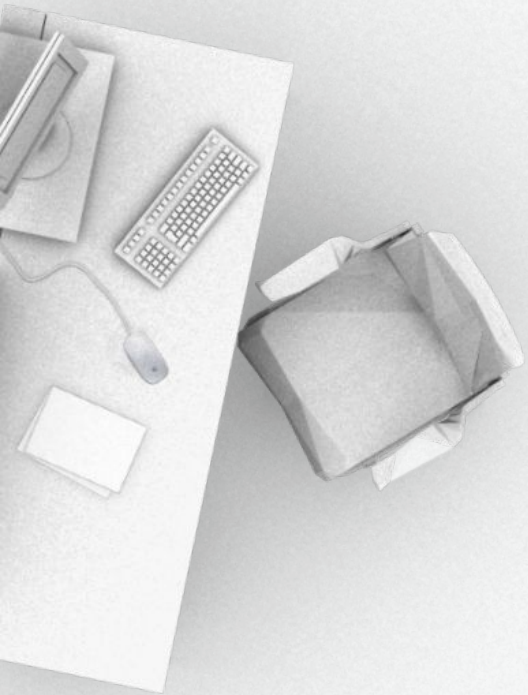
Thread: синхронизация

`import threading`

- `threading.Lock`
- `threading.RLock`
- `threading.Semaphore`
- `threading.BoundedSemaphore`
- `threading.Event`
- `threading.Timer`
- `threading.Barrier`

Дополнительно:


`queue (Queue, LifoQueue, PriorityQueue)`



GIL



GIL




Global Interpreter Lock (GIL) — это способ синхронизации потоков, который используется в некоторых интерпретируемых языках программирования.

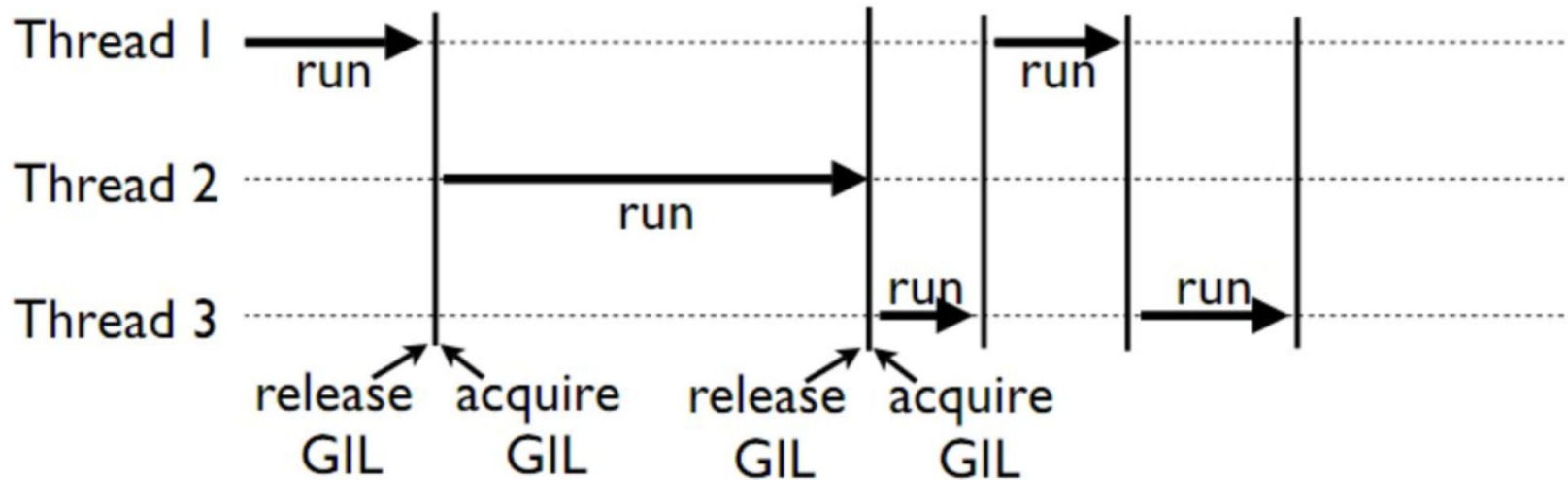
Mutex, который разрешает только одному потоку использовать интерпретатор python

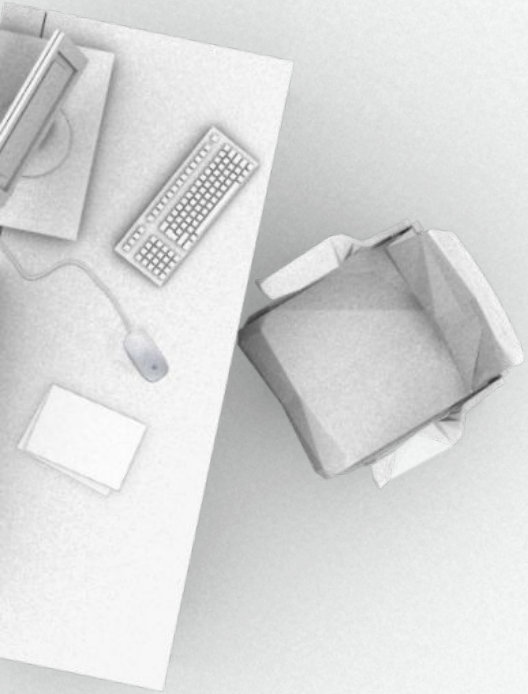


GIL

- 
- Что решает? - Race conditions
 - Почему глобальный? - Deadlocks, производительность
 - Выбран в качестве решения из-за C extensions
 - Изначально вводился для I/O bound потоков

GIL





Multiprocessing



Процесс (process)


Процесс - абстракция, которая инкапсулирует в себе все ресурсы процесса: открытые файлы, отображенные в память файлы, дескрипторы, потоки и тд.

Составные части:

1. Образ машинного кода;
2. Область памяти, в которую включается исполняемый код, данные процесса (входные и выходные данные, стек вызовов и куча (для хранения динамически создаваемых данных));
3. Дескрипторы ОС, например, файловые;
4. Состояние процесса.



multiprocessing



```
import os
from multiprocessing import Process

def print_info(name):
    print(f"Process {name}, pid={os.getpid()}, parent pid={os.getppid()}")

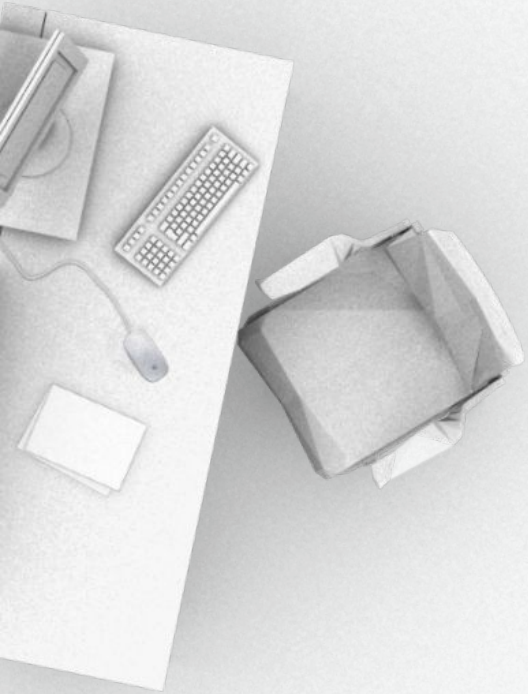
if __name__ == "__main__":
    print_info("main")
    processes = [
        Process(target=print_info, args=(f"child{i}",))
        for i in range(1, 5)
    ]
    for proc in processes:
        proc.start()
    for proc in processes:
        proc.join()
```

multiprocessing: Pool

```
import multiprocessing
import time

def countdown(n):
    while n > 0:
        n -= 1

if __name__ == '__main__':
    t1 = time.time()
    with multiprocessing.Pool(2) as p:
        p.apply_async(countdown, (100000000,))
        p.apply_async(countdown, (100000000,))
        p.close()
        p.join()
    t2 = time.time()
    print(t2 - t1)
```



IPC




IPC

ОС предоставляют механизмы для IPC:

- механизмы обмена сообщениями
- механизмы синхронизации
- механизмы разделения памяти
- механизмы удаленных вызовов (RPC)



IPC: виды

- 
- файл
 - сигнал
 - сокет
 - каналы (именованные/неименованные)
 - семафор
 - разделяемая память
 - обмен сообщениями
 - проецируемый в памяти файл
 - очередь сообщений
 - почтовый ящик



IPC: сигналы

```
import os
import time
import signal


def signal_handler(signal_num, frame):
    print(f"Handle signal {signal_num}")

if __name__ == "__main__":
    signal.signal(signal.SIGUSR1, signal_handler)
    signal.signal(signal.SIGUSR2, signal_handler)

    print(f"pid={os.getpid()}")
    while True:
        time.sleep(0.5)
```



IPC: coker



```
import socket

client = socket.socket(socket.AF_UNIX, socket.SOCK_DGRAM)
client.connect("/tmp/py_unix_example")
client.send(data.encode())

server = socket.socket(socket.AF_UNIX, socket.SOCK_DGRAM)
server.bind("/tmp/py_unix_example")
data = server.recv(1024)
```



IPC: каналы (pipe)



```
# sender.py
```

```
import os
```

```
fpath = "/tmp/example.fifo"  
os.mkfifo(fpath)
```

```
fifo = open(fpath, "w")  
fifo.write("Hello!\n")  
fifo.close()
```

```
# receiver.py
```


```
import os  
import sys
```

```
fpath = "/tmp/example.fifo"
```

```
fifo = open(fpath, "r")  
for line in fifo:  
    print(f"Recv: {line}")  
fifo.close()
```




IPC: mmap



```
import mmap

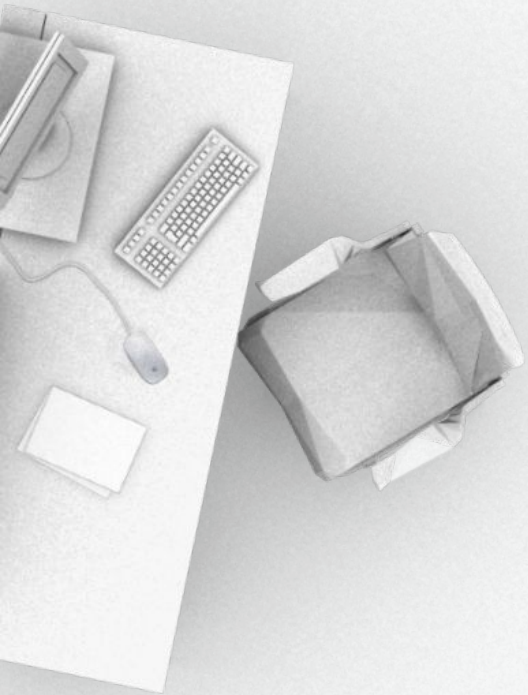
with open("data.txt", "w") as f:
    f.write("Hello, python!\n")

with open("data.txt", "r+") as f:
    map = mmap.mmap(f.fileno(), 0)
    print(map.readline()) # Hello, python!
    print(map[:5]) # Hello
    map[7:] = "world!\n"
    map.seek(0)
    print(map.readline()) # Hello, world!
    map.close()
```



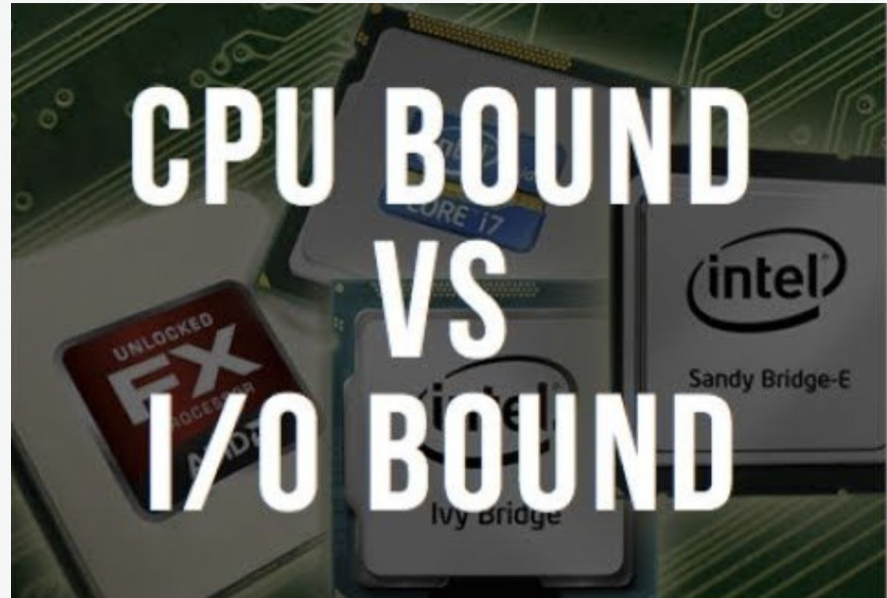
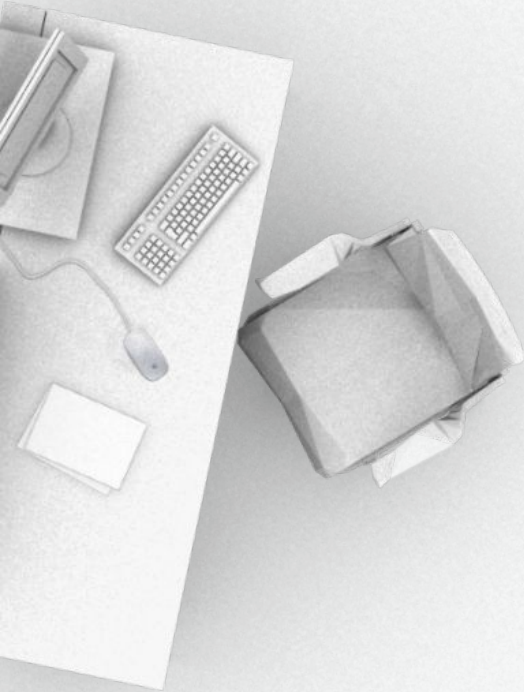
IPC: multiprocessing

- **Value**
`result = multiprocessing.Value("i")`
- **Array**
`result = multiprocessing.Array("i", 4)`
- **Manager**
`with multiprocessing.Manager() as manager:`
`records = manager.list([])`
- **Queue**
`q = multiprocessing.Queue()`
- **Pipe**
`Parent_conn, child_conn = multiprocessing.Pipe()`



Асинхронное программирование

IO bound vs CPU bound



Блокирующие операции

```
import socket
```

```
server_sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
server_sock.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
server_sock.bind(('localhost', 15000))
server_sock.listen()
```

```
while True:
```

```
    client_sock, addr = server_sock.accept()
```

```
        while True:
```

```
            data = client_sock.recv(4096)
```

```
            if not data:
```

```
                break
```

```
            else:
```

```
                client_sock.send(data.decode().upper().encode())
```

```
        client_sock.close()
```



Блокирующие операции

- connect, accept, recv, send - блокирующие операции
- C10k problem, <http://kegel.com/c10k.html>
- Потоки дорого стоят (CPU & RAM)
- Потоки простаивают часть времени



Неблокирующие операции

Системные вызовы:


- select (man 2 select)
- poll (man 2 poll)
- epoll (man 7 epoll)
- kqueue

python:

- select
- selectors



select




```
def event_loop():
    while True:
        ready_to_read, _, _ = select(to_monitor, [], [])

        for sock in ready_to_read:
            if sock is server_sock:
                accept_conn(sock)
            else:
                respond(sock)
```




selectors



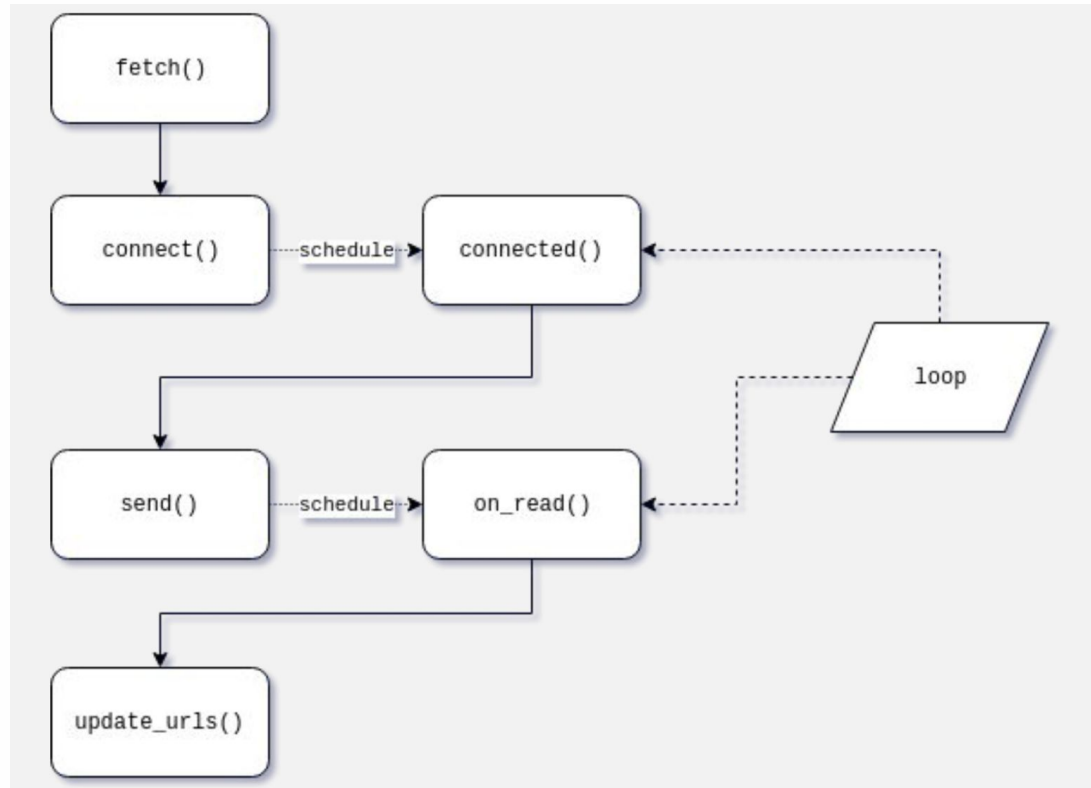
```
import selectors

selector = selectors.DefaultSelector()
selector.register(server_sock, selectors.EVENT_READ, accept_conn)

def event_loop():
    while True:
        events = selector.select() # (key, events_mask)

        for key, _ in events:
            # key: NamedTuple(fileobj, events, data)
            callback = key.data
            callback(key.fileobj)
            # selector.unregister(key.fileobj)
```

Callback hell



generator based event loop

Дэвид Бизли (David Beazley), "Python Concurrency From the Ground Up: LIVE!"

```
def event_loop():
    while any([tasks, to_read, to_write]):
        while not tasks:
            ready_to_read, ready_to_write, _ = select(to_read, to_write, [])
            for sock in ready_to_read:
                tasks.append(to_read.pop(sock))
            for sock in ready_to_write:
                tasks.append(to_write.pop(sock))
        try:
            task = tasks.pop(0)
            op_type, sock = next(task)
            if op_type == 'read':
                to_read[sock] = task
            elif op_type == 'write':
                to_write[sock] = task
        except StopIteration:
            pass
```

Корутины

```
def grep(pattern):  
    print('start grep for', pattern)  
    while True:  
        s = yield  
        if pattern in s:  
            print('found!', s)  
        else:  
            print('no %s in %s' % (pattern, s))
```

```
g = grep('python')  
next(g)  
g.send('data')  
g.send('deep python')
```

```
$ python generator_socket.py  
start grep for python  
no python in data  
found! deep python
```



Корутины



- использование ***yield*** более обобщенно определяет корутину
- не только генерируют значения
- потребляют данные, отправленные через ***.send***
- отправленные данные возвращаются через ***data = yield***



Нативные корутины



coroutine

Coroutines are a more generalized form of subroutines. Subroutines are entered at one point and exited at another point. Coroutines can be entered, exited, and resumed at many different points. They can be implemented with the `async def` statement. See also [PEP 492](#).

Нативные корутины

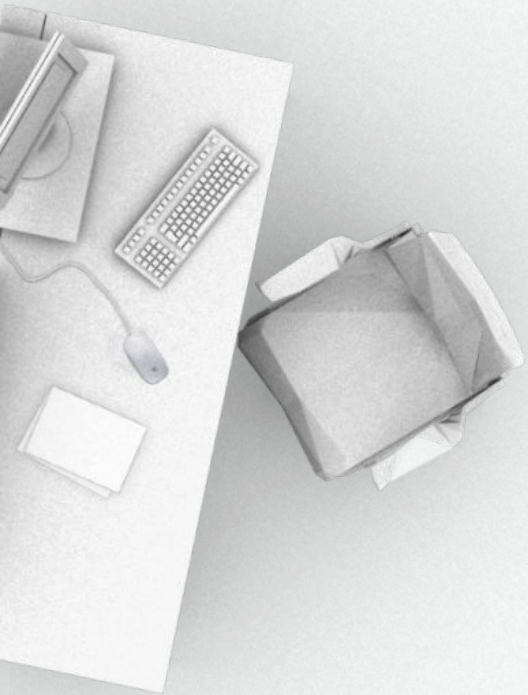
```
import asyncio, time

async def say_after(delay, what):
    await asyncio.sleep(delay)
    print(what)

async def main():
    print(f"started at {time.strftime('%X')}")
    await say_after(1, 'hello')
    await say_after(2, 'world')
    print(f"finished at {time.strftime('%X')}")

asyncio.run(main())


>run.py
started at 16:42:46
hello
world
finished at 16:42:49
```



asyncio



asyncio

- 
- 1 процесс
 - 1 поток
 - кооперативная многозадачность (vs вытесняющая)
 - передача управления в event loop на ожидающих операциях
 - async/await это API Python, а не часть asyncio



asyncio

Event loop:

coroutine > Task (Future)

- **Future** представляет ожидаемый в будущем (eventual) результат асинхронной операции;
- **Task** это *Future-like* объект, запускающий корутины в событийном цикле;
- **Task** используется для запуска нескольких корутин в событийном цикле параллельно.



asyncio

High-level APIs

- Coroutines and Tasks
- Streams
- Synchronization Primitives
- Subprocesses
- Queues
- Exceptions



asyncio

Low-level APIs

- Event Loop
- Futures
- Transports and Protocols
- Policies
- Platform Support



asyncio

Вспомогательное API

- `asyncio.create_task`
- `asyncio.sleep`
- `asyncio.gather`
- `asyncio.shield`
- `asyncio.wait_for`
- `asyncio.wait`
- `asyncio.Queue`
- `asyncio.Lock`
- `asyncio.Event`



Домашнее задание по лекции #9

ДЗ #9

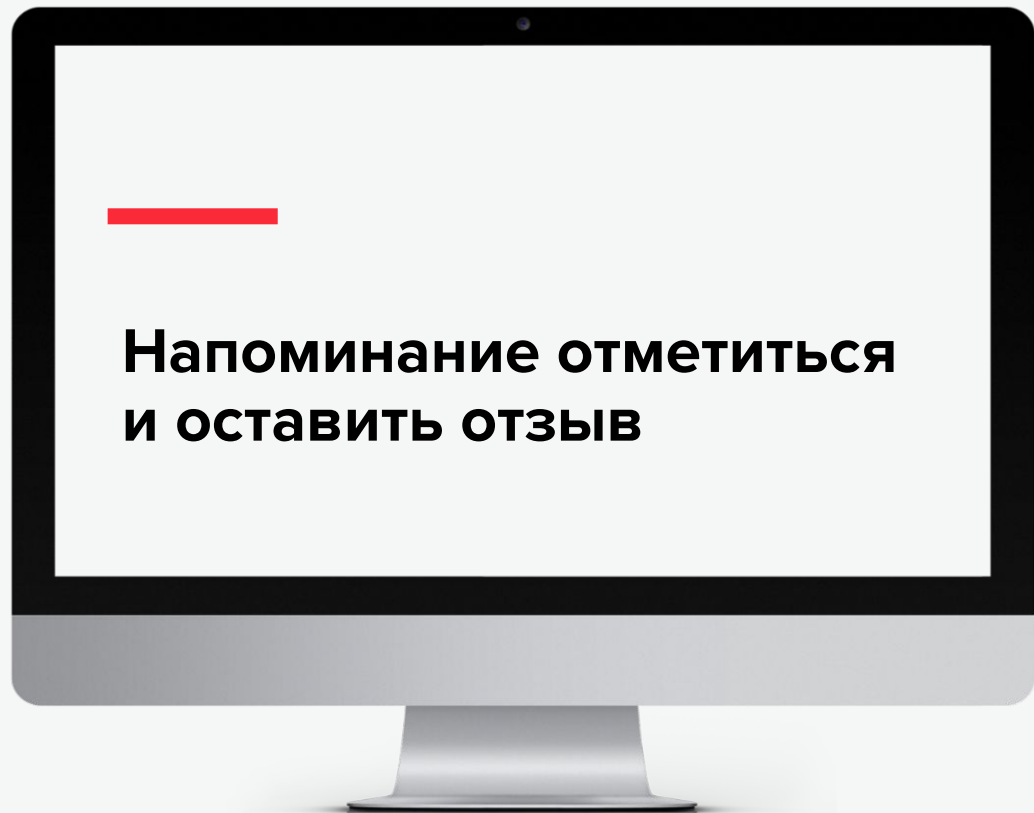
8

баллов за
задание

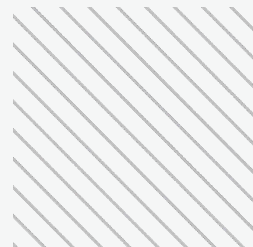
02.06.2021

срок сдачи

- Сервер с воркерами для равномерной обкатки и парсинга веб-страниц



**Напоминание отметить
и оставить отзыв**



**СПАСИБО
ЗА ВНИМАНИЕ**

