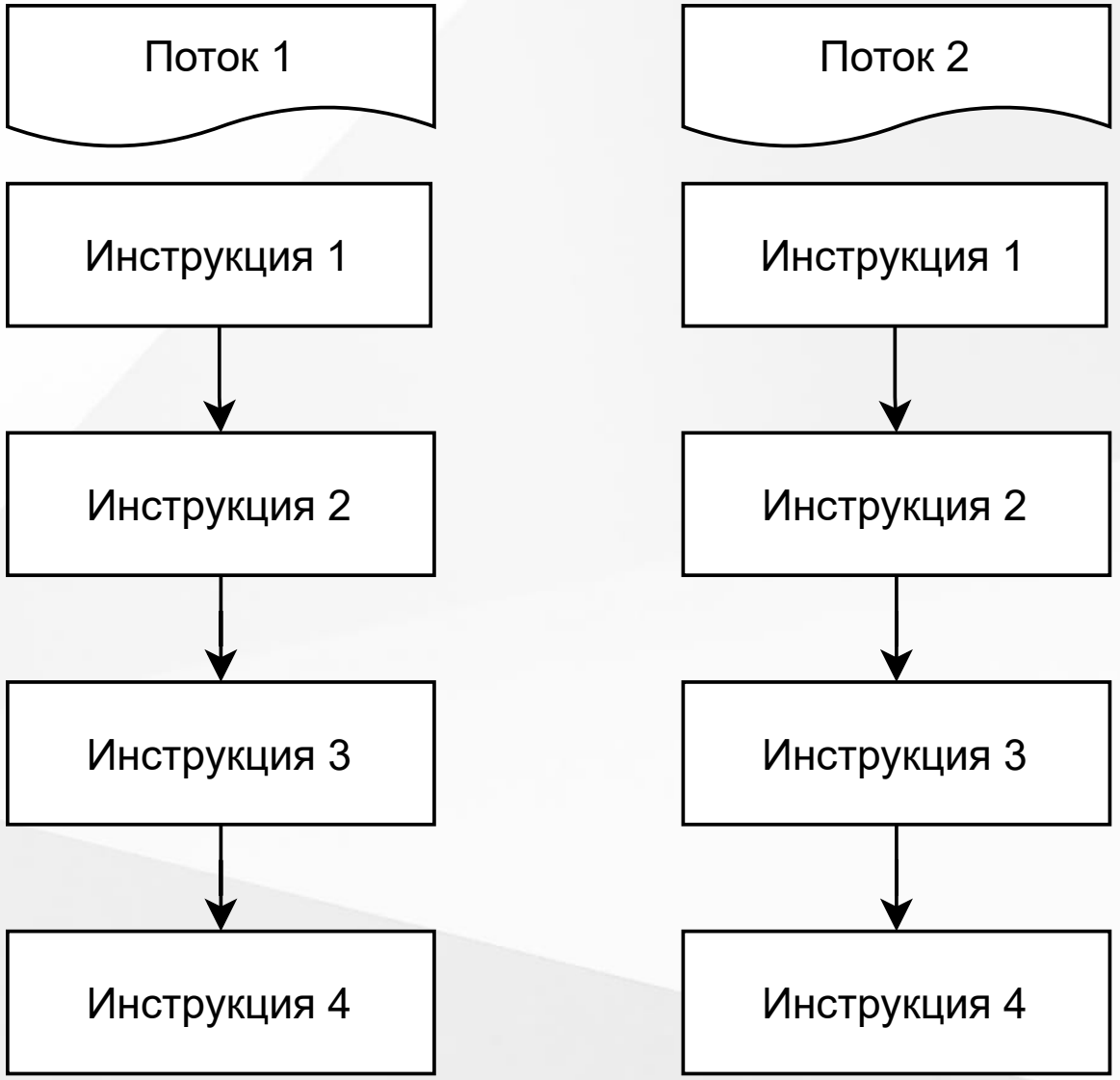
# Асинхронное программирование

SkillFactory
Вебинар 16 января
ментор Егор Закутей

# Синхронное выполнение программы

Инструкции выполняются последовательно.
Когда программа ожидает ответа извне, выполнение блокируется, пока ответ не будет получен.



# Знакомые примеры блокирования выполнения программы:

- Пользовательский ввод

```
name = input("Введите своё имя:")
age = input("Введите свой возраст:")
```

- Обращение к сторонним API

```
import requests
r = requests.get("https://api.exchangeratesapi.io/latest")
```

- Модуль `time`

```
import time
time.sleep(5)
```

# Моделируем внешнее API

```
from random import randint
import time

class Task:
    def __init__(self, name):
        self.completed = False
        self.response = None
        self.name = name
        self.__end = time.time() + randint(1, 3)

    def ready(self):
        return time.time() > self.__end

    def process(self):
        if self.ready and not self.completed:
            self.completed = True
            self.response = f"Выполнил задачу {self.name}"

    def sync_complete(self):
        while not self.ready():
            time.sleep(0.1)
        self.process()
```

Полный код в файле 01_simple_sync.py

# Проблема синхронного кода

К нам поступило сразу несколько задач и нужно их всех обработать:

```
queue = [
    "Подготовить бумаги",
    "Проверить почту",
    "Консультация по проекту",
    "Заполнить форму"
]

for task_name in queue:
    task = Task(task_name)
    task.sync_complete()
    print(task.response)
```

Для выполнения задач не требуется проводить какие-либо вычисления в программе, нужно просто ждать. Пока мы ждём, выполнение блокируется, и мы тратим время зря.

Полный код в файле 01_simple_sync.py

# Асинхронное решение

```
tasks = []

for task_name in queue:
    tasks.append(Task(task_name))

completed = 0
while True:
    for task in tasks:
        if task.ready() and not task.completed:
            task.process()
            completed += 1
            print(task.response)

    if completed == len(tasks):
        break
```

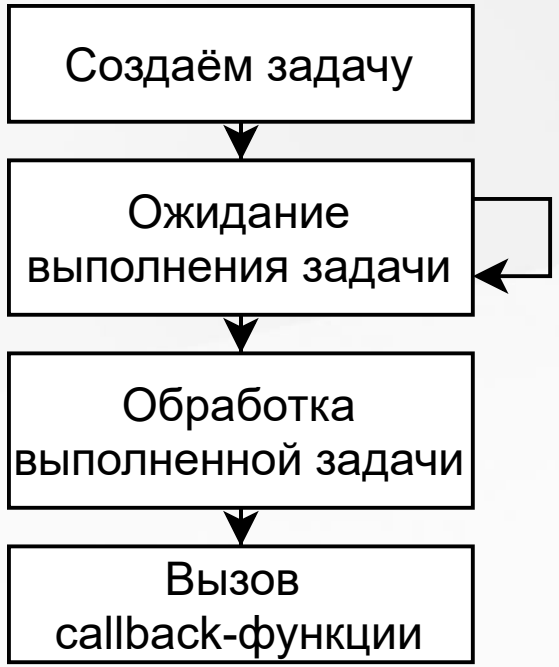
Полный код в файле 02_simple_async.py

# Асинхронность на callback'ах

## Обобщим понятие задачи.

Для каждой задачи известно:

- Готова ли задача к выполнению
- Выполнена ли задача
- Результат выполнения задачи
- Функция, которой предаётся результат после выполнения



# Базовый класс

```
class Task:
    def __init__(self, callback):
        self.completed = False
        self.response = None
        self.callback = callback

    @property
    def ready(self):
        raise NotImplementedError()

    @property
    def need_process(self):
        return self.ready and not self.completed

    def process(self):
        self.completed = True
        self.callback(self.response)
```

Полный код в файле 03_simple_callbacks.py

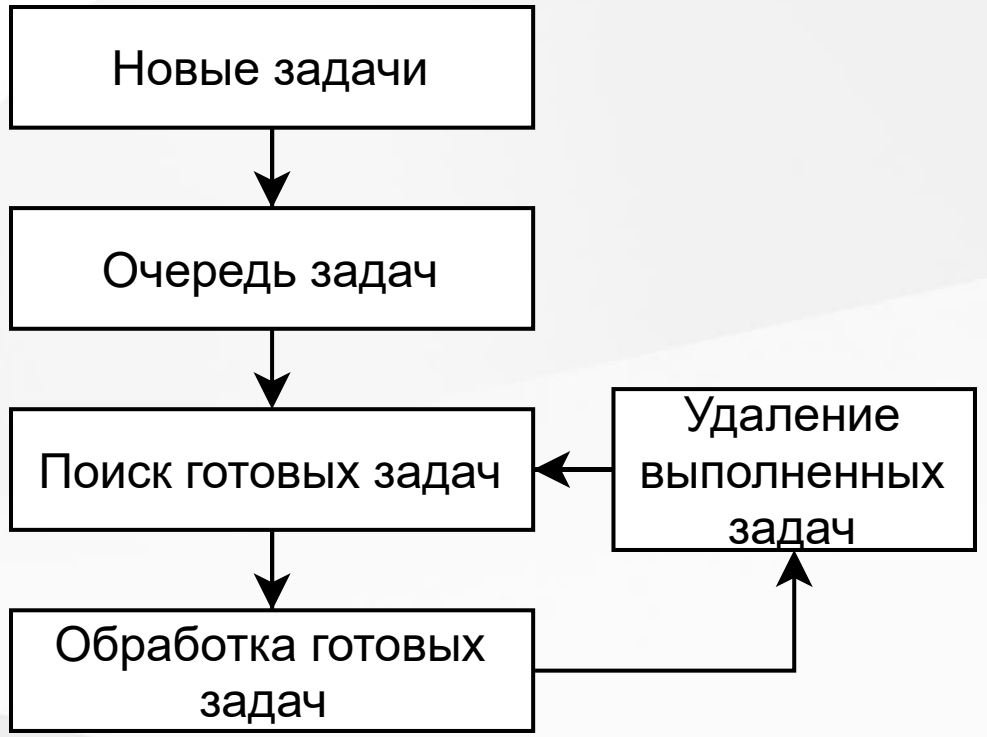
# Задачи из предыдущего примера

```
class TimeoutTask(Task):
    def __init__(self, name, callback):
        super().__init__(callback)
        self.name = name
        self.__end = time.time()+randint(1, 5)

    @property
    def ready(self):
        return time.time() > self.__end

    def process(self):
        self.response = f'Выполнена задача {self.name}'
        super().process()
```

Полный код в файле 03_simple_callbacks.py

# Управление задачами

```
class TaskManager:
    def __init__(self):
        self.tasks = []

    def add(self, task):
        self.tasks.append(task)

    def loop(self):
        while self.tasks:
            task = self.tasks.pop(0)
            if task.need_process:
                task.process()
            else:
                self.tasks.append(task)
```




Полный код в файле 03_simple_callbacks.py

# Тот же пример:

```
tm = TaskManager()

def simple_handler(response):
    print(response)

queue = [
    "Подготовить бумаги",
    "Проверить почту",
    "Консультация по проекту",
    "Заполнить форму"
]

for task_name in queue:
    tm.add(TimeoutTask(task_name, simple_handler))

tm.loop()
```

Полный код в файле 03_simple_callbacks.py

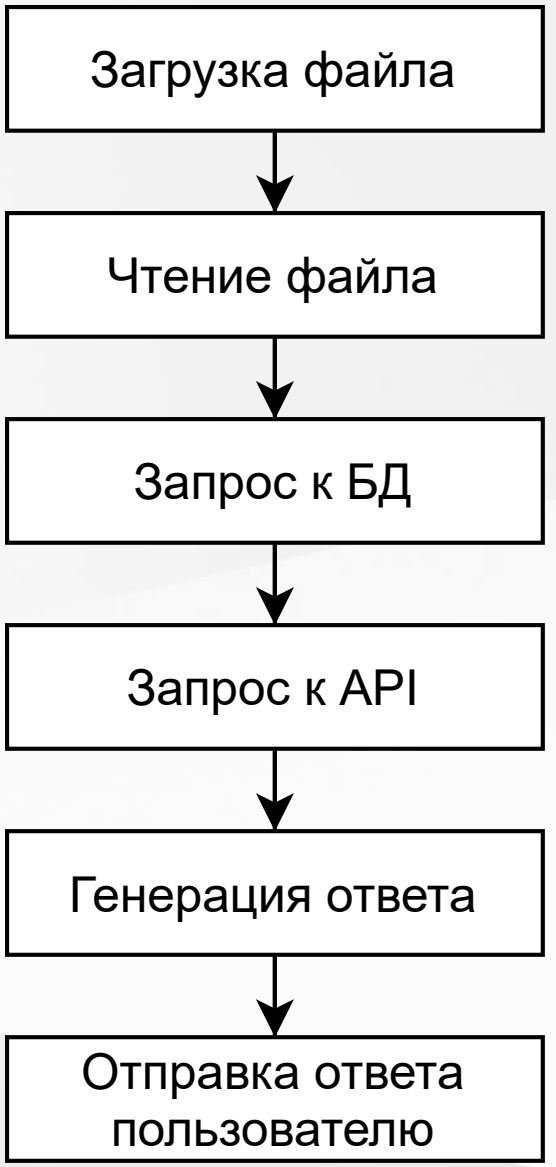
# Добавляем функционал

```
tm = TaskManager()

def check(response):
    print(f"Отправили на проверку: -- {response} --")
    tm.add(TimeoutTask(f"Проверка: < {response} >", show))

def show(response):
    print(response)

for task_name in queue:
    tm.add(TimeoutTask(task_name, check))

tm.loop()
```


Полный код в файле 04_complex_callbacks.py

# Сложная цепочка и callback hell

```
tm = TaskManager()

def on_load(response):
    print("Скачали файл")
    tm.add(TimeoutTask("Чтение файла", on_read))

def on_read(response):
    print("Прочли файл")
    tm.add(TimeoutTask("Запрос к БД", on_db))

def on_db(response):
    print("Получили результат из БД")
    tm.add(TimeoutTask("Запрос к стороннему API", on_api))

def on_api(response):
    print("Получили результат из API")
    tm.add(TimeoutTask("Сгенерировать ответ", on_resp))

def on_resp(response):
    print("Вернули ответ пользователю")

tm.add(TimeoutTask("Скачивание файла", on_load))

tm.loop()
```




Полный код в файле 05_callback_hell.py

# Асинхронность на `asyncio`

**`async def *имя*`** - асинхронная функция, сущность аналогичная нашей `Task`

**`await`** - внутри `async` функции асинхронно выполняет "задачу"

**`asyncio.sleep`** - аналог `time.sleep`, но без блокировки выполнения

**`asyncio.run`** - запускает задачу асинхронно

**`asyncio.gather`** - выполнение нескольких задач асинхронно

Первый пример будет выглядеть так в случае с одной задачей:

```python
import asyncio
from random import randint

async def task(name):
    duration = randint(1, 3)
    await asyncio.sleep(duration)
    response = f"Выполнил задачу {name}"
    print(response)

asyncio.run(task("Подготовить бумаги"))
```

Полный код в файле 06_simple_asyncio.py

# В случае нескольких задач:

```
import asyncio
from random import randint

async def task(name):
    duration = randint(1, 3)
    await asyncio.sleep(duration)
    response = f"Выполнил задачу {name}"
    print(response)

queue = [
    "Подготовить бумаги",
    "Проверить почту",
    "Консультация по проекту",
    "Заполнить форму"
]

async def main():
    tasks = []
    for task_name in queue:
        tasks.append(task(task_name))
    await asyncio.gather(*tasks)

asyncio.run(main())
```

Полный код в файле 07_queue_asyncio.py

`asyncio` позволил коротко и логично записать наш длинный код с `TaskManager`

Так же он не требует создания множества callback'ов на каждую задачу.

```python
import asyncio
from random import randint

async def task(name):
    duration = randint(1, 3)
    await asyncio.sleep(duration)
    response = f"Выполнил задачу {name}"
    return response

async def main():
    await task("Скачивание файла")
    print("Скачали файл")
    await task("Чтение файла")
    print("Прочли файл")
    await task("Запрос к БД")
    print("Получили результат из БД")
    await task("Получили результат из API")
    print("Получили результат из API")
    await task("Сгенерировать ответ")
    print("Вернули ответ пользователю")

asyncio.run(main())
```

Полный код в файле 08_multitask_asyncio.py

# Обращение к реальному API

## Синхронные запросы:

```
import requests
import json
import time

t_0 = time.time()
rates = []
for day in range(1, 31):
    r = requests.get(f"https://api.exchangeratesapi.io/2020-10-{day}?symbols=RUB")
    rates.append(json.loads(r.content)["rates"]["RUB"])

dt = time.time() - t_0

print(f"Затрачено {dt:.2f} секунды")
print(rates)
```

Полный код в файле 09_sync_api.py

# Асинхронные запросы с `aiohttp`

```
import aiohttp
import asyncio
import time

t_0 = time.time()

async def day_rate(session, day):
    async with session.get(f"https://api.exchangeratesapi.io/2020-10-{day}?symbols=RUB") as resp:
        r = await resp.json()
        return r["rates"]["RUB"]

async def main():
    async with aiohttp.ClientSession() as session:
        resps = [day_rate(session, day) for day in range(1, 31)]
        rates = await asyncio.gather(*resps)

    dt = time.time() - t_0
    print(f"Затрачено {dt:.2f} секунды")
    print(rates)


event_loop = asyncio.get_event_loop()
event_loop.run_until_complete(main())
```

Полный код в файле 10_async_api.py

# Синхронный сервер

```
from flask import Flask
import requests
import time

app = Flask(__name__)

def get_todos():
    todos_list = []
    for todo_id in range(1, 10):
        r = requests.get(f"https://jsonplaceholder.typicode.com/todos/{todo_id}")
        todos_list.append(r.json()['title'])
    return todos_list

@app.route('/')
def hello_world():
    t_0 = time.time()
    todos_list = get_todos()
    dt = time.time()-t_0
    print(f"Затрачено {dt:.2f} секунд")
    return "<br>".join(todos_list)
```

Полный код в файле 11_sync_server.py

# Асинхронный сервер

```
import aiohttp
import asyncio
from aiohttp import web
import time

async def get_todo(session, todo_id):
    async with session.get(f"https://jsonplaceholder.typicode.com/todos/{todo_id}") as resp:
        r = await resp.json()
        return r['title']

async def get_todos():
    async with aiohttp.ClientSession() as session:
        resps = [get_todo(session, todo_id) for todo_id in range(1, 10)]
        rates = await asyncio.gather(*resps)
        return rates

async def todos(request):
    t_0 = time.time()
    rates = await get_todos()
    dt = time.time()-t_0
    print(f"Затрачено {dt:.2f} секунд")
    return web.Response(text="\n".join(rates))

app = web.Application()
app.add_routes([web.get('/', todos)])

web.run_app(app, host="localhost")
```

Полный код в файле 12_async_server.py

# Заключение

- **Синхронный** подход нужно использовать, когда запросов к сторонним службам не выполняется. Когда программа не простаивает в ожидании запросов, синхронный код будет работать немного быстрее. Ускорить его можно, запуская код в несколько потоков.
- **Асинхронный** подход нужно использовать при выполнении длительных запросов к сторонним службам.

Замечание: **В левом нижнем углу есть ссылка на полный код примеров.**