

Технологии программирования

Лекция 10 | Применимость Python



Основная концепция языка

Возникновение, характеристики языка и применимость

Python для ML и DataScience

Экосистема и библиотеки Numpy, Numba, Pandas, Scipy

Python для автоматизации

Написание мелких утилит

Современный подход и правила

Улучшаем Python-код внутренними идиомами

Интеграция с другими языками

Добавление C++, Rust, C-вставок в Python-код



Мы здесь

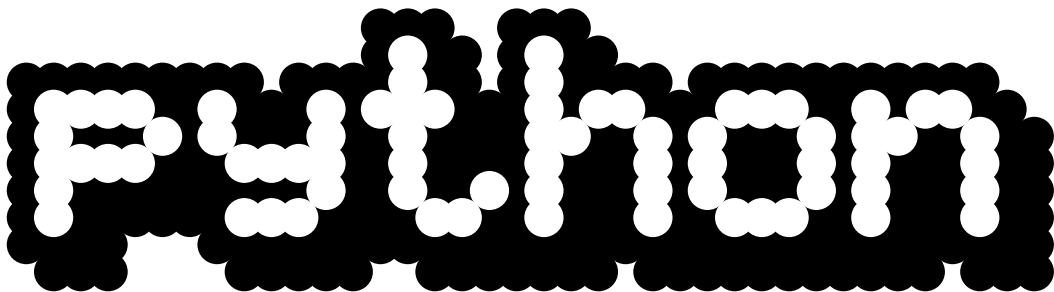


Возникновение Python

В 1980 годах разрабатывался язык ABC, призванный сместить сложные низкоуровневые языки. Но один из разработчиков ABC был недоволен малым комьюнити языка и отсутствием предложений по улучшению и решил создать свой.

Так в 1991 году был представлен Python 0.9.0

Первые версии Python были без привычного сборщика мусора, но уже имели базовый функционал ООП, а первая версия появилась в 1994, вторая в 2000, а третья в 2008.



Первый вариант
логотипа

Особенности Python

1

Простой и читаемый синтаксис

- Нет точек с запятой, фигурных скобок, явного объявления типов
- Тип переменной определяется в момент присваивания, а не при объявлении

2

Сборщик мусора

Отсутствует ручное управление памятью
Объекты удаляются автоматически, когда на них нет ссылок

3

Стандартная библиотека и экосистема

Встроенные модули для работы с JSON, HTTP, файлами, многопоточностью и т.д.

```
a = 10      # int
a = "Python" # str

# Python
if x > 5:
    print("Hello")
else:
    print("World")

def func(x: int, y: int) -> int:
    return x + y

class SumForInts:
    _x: int = 0 # Комментарий к "Protected" члену класса
    __x: int = 0 # Приватный член класса
    x: int = 0 # Обычный член класса

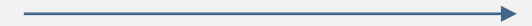
    def __init__(self, x_: int): # Конструктор
        self.x = self._x = self.__x = x_

    @staticmethod
    def static_function(cls, args):
        # ... Какой-то код
```

Особенности Python

4

**Высокая скорость разработки и написания кода,
даже жертвуя производительностью**



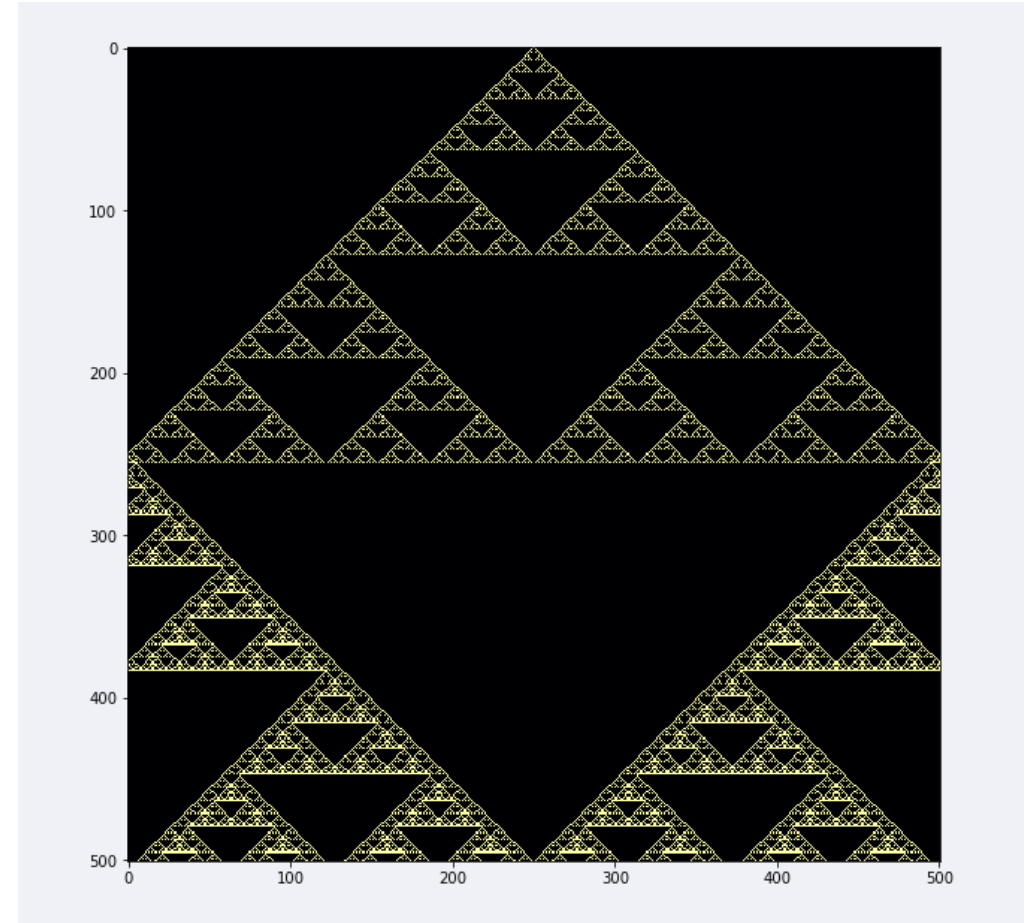
Быстрая разработка, низкая производительность

Для примера сравнения скорости работы Python и C++ возьмем задачу вычисления модели Wolfram

Модели Wolfram – тип одномерной модели клеточного автомата

Эти типы моделей, как правило, состоят из сетки ячеек. Ячейки могут находиться в одном из конечного числа состояний, и для поиска следующего состояния на сетке используется правило обновления

Для работы данного алгоритма необходимо большое количество итераций и проверок



Быстрая разработка, низкая производительность

Python

```
@nb.jit #numba the function
def Rule30_code():
    Rule30 = np.zeros((1000,100000)) #initilize an array to run on (timesteps, width)
    Rule30[0,50] = 1
    for y in range(Rule30.shape[0]-1): #iterate through grid
        for x in range(Rule30.shape[1]):
            #update the next rows values accoring to neighbor & self value
            right = x + 1
            down = y + 1
            left = x - 1
            if right >= Rule30.shape[1]:
                right = 0
            if Rule30[y,right] == 1 and Rule30[y,left] == 0:
                Rule30[down,x] = 1
            elif Rule30[y,x] == 0 and Rule30[y,right] == 0 and Rule30[y,left] == 1:
                Rule30[down,x] = 1
            elif Rule30[y,x] == 1 and Rule30[y,right] == 0 and Rule30[y,left] == 0:
                Rule30[down,x] = 1
            else:
                Rule30[down,x] = 0
```

Быстрая разработка, низкая производительность

C++

```
int main() {  
    const size_t sz = 100000;  
    const int iter = 1000;  
    cout << boolalpha << std::fixed <<  
    std::setprecision(5);  
  
    int val;  
    long av_time = 0;  
    std::bitset<sz> v(0);  
    std::bitset<sz> tmp(0);  
    v[sz/2] = 1;  
    using time_unit = std::chrono::microseconds;
```

```
    for(int run = 0; run < 10; run++){  
        auto start = std::chrono::steady_clock::now();  
        for(int i=0; i<iter; ++i){  
            for(size_t i = v.size(); i>=1; --i){  
                val = (int)v[i-1] << 2 | (int)v[i] << 1 | (int)v[i+1];  
                tmp[i] = (val == 3 || val == 5 || val == 6);  
            }  
            v = tmp;  
        }  
        auto duration =  
        std::chrono::duration_cast<time_unit>(std::chrono::  
        steady_clock::now() - start);  
        av_time += duration.count();  
    }  
    cout << "msec = " << av_time/10.0 << endl;  
}
```


Быстрая разработка, низкая производительность

Code/Width	100	300	1000	3000	10000	30000	100000
Python	0.1330	0.3990	1.3565	4.2989	15.4906	44.7538	148.8903
Numba	0.0193	0.0185	0.0231	0.0324	0.0617	0.1400	0.4058
C++ Naive	0.0390	0.1120	0.3350	1.0170	3.3470	9.9960	33.3370
C++ Naive -O2	0.0100	0.0220	0.0590	0.1590	0.5180	1.5270	5.1140
C++ Optimized	0.0150	0.0330	0.0960	0.2720	0.8810	2.6120	8.7240
C++ Optimized -O2	0.0001	0.0003	0.0009	0.0023	0.0067	0.0149	0.0365

Опции, сильно упрощающие написание кода

Функции можно передавать в аргументы, возвращать из функций

```
def apply(func, x):  
    return func(x)  
  
print(apply(lambda x: x * 2, 5)) # 10
```

Генераторы и списковые включения

Списковые включения для компактного создания списков

```
`squares = [x**2 for x in range(10)] # [0, 1, 4, ..., 81]`
```

Генераторы для ленивых вычислений

```
`gen = (x**2 for x in range(10)) # Не хранит весь список в памяти`
```

Декораторы

Модификация поведения функций без изменения их кода

```
def logger(func):  
    def wrapper(*args):  
        print(f"Calling {func.__name__}")  
        return func(*args)  
    return wrapper
```

```
@logger  
def add(a, b):  
    return a + b
```

ООП также имеет простую синтаксическую реализацию

Датаклассы (dataclasses)

Автоматическая генерация boilerplate-кода
(`init`, `repr`)

```
from dataclasses import dataclass

@dataclass
class User:
    name: str
    age: int
    is_active: bool = True

user = User("Alice", 25) # Не нужно писать __init__
print(user) # User(name='Alice', age=25, is_active=True)
```

Абстрактные классы (ABC)

Строгое определение интерфейсов с помощью
компактного кода

```
from abc import ABC, abstractmethod

class Shape(ABC):
    @abstractmethod
    def area(self) -> float: ...

class Circle(Shape):
    def __init__(self, radius: float):
        self.radius = radius

    def area(self) -> float:
        return 3.14 * self.radius ** 2
```

ООП также имеет простую синтаксическую реализацию

Композиция вместо наследования

Python поощряет использование композиции и миксинов для избежания сложных иерархий

```
class LoggerMixin:
    def log(self, message: str) -> None:
        print(f"Log: {message}")

class DataProcessor(LoggerMixin):
    def process(self) -> None:
        self.log("Processing data...")
```

Магические методы и перегрузка операторов

Лаконичная и простая реализация вектора

```
from abc import ABC, abstractmethod

class Vector:
    def __init__(self, x: float, y: float):
        self.x = x
        self.y = y

    def __add__(self, other: "Vector") -> "Vector":
        return Vector(self.x + other.x, self.y + other.y)

    def __repr__(self) -> str:
        return f"Vector({self.x}, {self.y})"

v1 = Vector(1, 2)
v2 = Vector(3, 4)
print(v1 + v2) # Vector(4, 6)
```

Поэтому, язык стал популярен | Другие причины:

- **Низкий порог входа**
Подходит для новичков и быстрого прототипирования
- **Большое сообщество**
Огромное количество библиотек (NumPy, Pandas, TensorFlow, Django)
- **Универсальность:**

Веб-разработка (Django, Flask)

Data Science (анализ данных, машинное обучение)

Автоматизация (скрипты для CI/CD, обработки файлов)

DevOps (написание утилит для управления инфраструктурой)

Для чего больше всего подходит? | Когда применять

Выбирайте Python, если:

Скорость разработки

- Нужно быстро создать прототип
- Требуется минимальный код для скриптов или автоматизации
- Проект имеет частые изменения требований

Библиотеки и экосистема

- Проект связан с библиотекой, написанной на более производительном языке, например C++
- Нужны готовые решения для веб-разработки (Flask) или DevOps (Ansible)

Производительность

- Производительность не критична (например, построение логики)
- Задача связана с I/O-операциями, где асинхронные фреймворки (например, Numpy) компенсируют недостатки

Примеры областей применения Python

Веб-разработка

- Серверная часть и API (Django, Flask, FastAPI)
- Генерация HTML/CSS через шаблонизаторы, интеграция JavaScript-фреймворков

Управление инфраструктурой

- Ansible (конфигурация серверов)
- Terraform (инфраструктура как код)

Обработка файлов

- Переименование файлов, конвертация, работа с Excel/PDF

Пример:

Скрипт для массового добавления водяных знаков в изображения (Pillow)

Симуляции и моделирование

- Решение дифференциальных уравнений (SciPy), физические эксперименты

Компьютерное зрение

- Распознавание объектов, обработка изображений
- Библиотеки: OpenCV, YOLO

Робототехника

- Управление роботами через ROS (Robot Operating System)

Примеры областей применения Python

Data Science



Machine Learning



Автоматизация и DevOps



Прототип | Подходит ли Python?

Прототип – это упрощённая модель будущего продукта, созданная для тестирования гипотез, демонстрации базовой функциональности или сбора обратной связи

! Он не предназначен для реального использования, а служит инструментом для визуализации идеи и проверки её жизнеспособности

Особенности прототипов:

- **Быстрая разработка:** Создаётся за дни или часы
- **Минимальные затраты:** Нет сложной архитектуры, тестирования, документации
- **Фокус на ключевых фичах:** Проверяет одну идею (например, «Как пользователь будет заказывать еду?»)
- **Низкая детализация:** Может не иметь дизайна, анимаций, обработки ошибок
- **Гибкость:** Легко вносить изменения на основе фидбека

Прототип | Подходит ли Python?

Python отлично подходит для быстрого прототипирования

С его помощью можно легко проверить гипотезу или новый функционал, а затем интегрировать готовое решение в проект на других языках (например, C++ или Rust) через C-расширения

Важно:

Даже если финальный проект пишется на том же языке, что и прототип (например, на C++), прототип можно (и часто нужно) разрабатывать отдельно

Это позволяет:

- Сфокусироваться на идее, а не на оптимизации
- Быстро вносить изменения
- Упростить тестирование

MVP | Подходит ли Python?

Minimal Viable Product (минимально жизнеспособный продукт) – тестовая версия товара, услуги или сервиса с минимальным набором функций (иногда даже одной), которая несет ценность для конечного потребителя

Минимально жизнеспособный продукт позволяет:

- Проверить гипотезу на основе реальных данных и доказать жизнеспособность идеи
- Снизить возможность финансовых убытков при запуске неудачного продукта
- Уменьшить стоимость разработки за счёт отказа от ненужных функций
- Выявить неучтённые потребности клиентов, собрать начальную базу клиентов
- Оптимизировать тестирование продукта и ускорить поиск ошибок
- Выйти на рынок и привлечь инвесторов

Задача – сократить время и усилия на тестирование идеи до начала разработки полноценного продукта

MVP | Подходит ли Python?

Python не подходит для MVP такого проекта, в конечной реализации которого предполагаются другие языки

Особенно, если разрабатывается средне- или высоконагруженные системы)

Конечно, если проект предполагает Python в качестве основного языка, то и MVP должен быть соответствующим

! Но не стоит прибегать к лишнему использованию Python лишь для быстроты разработки, если придется мигрировать на другие языки

Прототип vs MVP

Критерий	Прототип	MVP
Цель	Проверить идею, собрать фидбек	Проверить спрос на рынке, получить первых пользователей
Аудитория	Внутренняя команда, инвесторы, фокус-группы	Реальные пользователи
Функциональность	Частичная (только ключевые сценарии)	Минимально достаточная для решения проблемы пользователя
Качество кода	Неважно (возможны «костыли»)	Чистая архитектура, тесты, документация
Готовность к релизу	Нет	Да
Пример	Интерактивный макет интерфейса в Figma	Приложение с базовым функционалом (например, Uber без оплаты в приложении)

Прототип менеджера бюджета

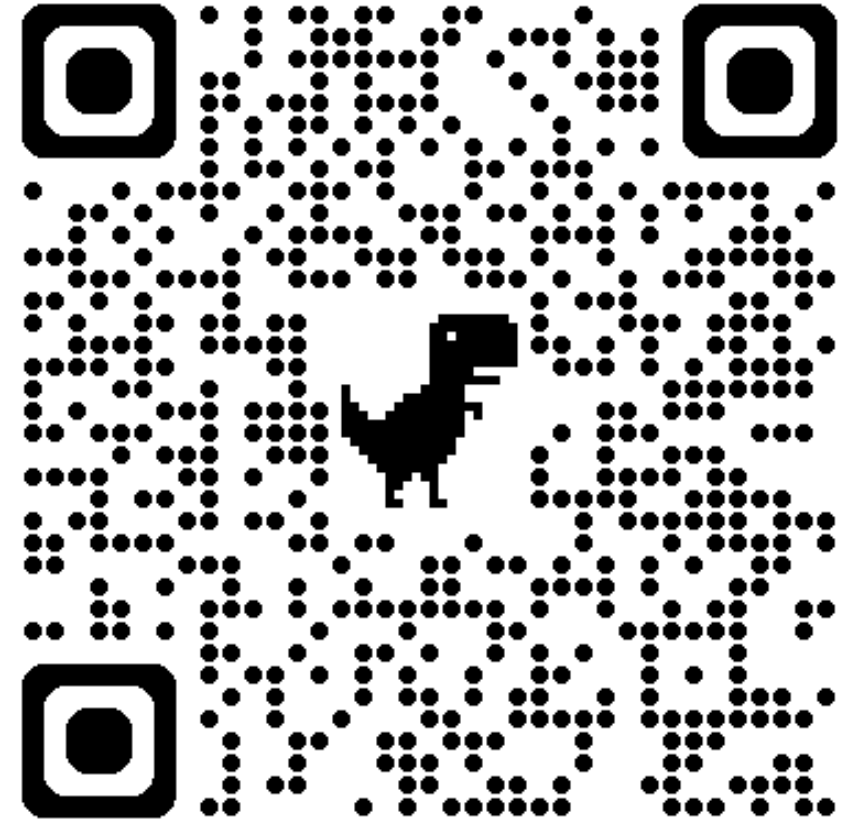
Цель – определена исходя из пожеланий:

ТГ бот для добавления и удаления трат по категориям в Google Sheet

Таблица заранее подготовлена и требуется реализовать ограниченную функциональность

Стек:

- Готовых подобных реализаций, которые можно было бы удобно встроить нет, поэтому пишем сами
- Из подходящих библиотек для взаимодействие с ТГ берем python-telegram-bot, а для Google Sheets – oauth2client



Gitlab прототипа



Основная концепция языка

Возникновение, характеристики языка и применимость

Python для ML и DataScience

Экосистема и библиотеки Numpy, Numba, Pandas, Scipy

Python для автоматизации

Написание мелких утилит

Современный подход и правила

Улучшаем Python-код внутренними идиомами

Интеграция с другими языками

Добавление C++, Rust, C-вставок в Python-код



Мы здесь

Python один из лучших языков для ML и DS

Богатая Экосистема

Библиотеки и Фреймворки:
Разнообразие библиотек и фреймворков,
таких как TensorFlow, PyTorch и Scikit-Learn

Портативность и Гибкость

Python можно легко интегрировать с другими языками программирования и платформами, что делает его идеальным для разработки комплексных систем машинного обучения

Сообщество и Поддержка

Сильное и активное сообщество Python состоит из миллионов разработчиков по всему миру, которые постоянно работают над улучшением и развитием языка

Библиотеки Python для ML и DataScience

Обработка данных

Pandas, NumPy

Визуализация

Matplotlib, Seaborn, Plotly

Машинное обучение

Scikit-learn, TensorFlow / Keras,
PyTorch

Научные вычисления

SciPy

Дополнительные инструменты

Statsmodels, XGBoost /
LightGBM, Dask

Обработка естественного языка (NLP)

NLTK, SpaCy

NumPy

NumPy – фундаментальная библиотека для работы с многомерными массивами и математическими функциями

Основной инструмент в Data Science для операций линейной алгебры, преобразования Фурье и генерации случайных чисел

```
import numpy as np
```

```
arr = np.array([[1, 2], [3, 4]])  
print(arr.sum(axis=0)) # [4 6]  
print(arr.dot(arr.T)) # [[ 5 11] [11 25]]
```

```
I = np.eye(3) # единичная матрица 3x3  
D = np.diag([1, 2, 3]) # диагональная матрица  
n = np.repeat(3, 4) # массив из троек длины 4  
seq = np.arange(1, 11) # то же самое, что range(1, 11)  
grid = np.linspace(1, 10, 50) # массив из 50 точек  
разбиения отрезка [1; 10] с равномерным шагом
```

Polars и Pandas

Polars и Pandas – библиотеки для обработки табличных данных

Polars, написанный на Rust, быстрее обрабатывает большие датасеты благодаря multithreading и векторизации, а Pandas остается стандартом для ETL-задач с поддержкой сложных операций

Polars

чтение CSV, фильтрация через `filter`, группировка `group_by` с агрегацией, ленивые вычисления для оптимизации памяти

```
import polars as pl

df = pl.DataFrame({
    "A": [1, 2, 3],
    "B": ["x", "y", "z"]
})
result = df.filter(pl.col("A") >
1).group_by("B").agg(pl.mean("A"))
print(result)
```

Pandas

объединение `DataFrame` через `merge`, применение пользовательских функций `apply`, работа с временными рядами `resample`

```
import pandas as pd

df = pd.DataFrame({
    "A": [1, 2, 3],
    "B": ["x", "y", "z"]
})
result = df[df["A"] > 1].groupby("B").mean()
print(result)
```

```
# Файл: mymodule.pyx
def sum_cython(int n):
    cdef int i, total = 0
    for i in range(n):
        total += i
    return total
```

```
# Использование в Python после компиляции
from mymodule import sum_cython
print(sum_cython(1000000)) # 499999500000
```

Numba

Numba – JIT-компилятор для ускорения Python-кода, особенно в связке с NumPy

Применяется для оптимизации циклов, мат. алгоритмов без переписывания на C

JIT-компиляция – динамическая компиляция кода по ходу работы приложения

Пример декорирования функции @njit для автоматической компиляции в машинный код, ускорение вычисления числа π через метод Монте-Карло в 100 раз

```
from numba import njit
import numpy as np

@njit
def monte_carlo_pi(n_samples):
    count = 0
    for _ in range(n_samples):
        x, y = np.random.random(), np.random.random()
        if x**2 + y**2 <= 1:
            count += 1
    return 4 * count / n_samples

print(monte_carlo_pi(1_000_000)) # ~3.141
```

PyTorch и TensorFlow

PyTorch и TensorFlow – фреймворки для машинного обучения и глубокого обучения

- PyTorch часто используется в исследовательских задачах благодаря динамическим графам вычислений и гибкости
- TensorFlow популярен в промышленном внедрении из-за оптимизации под распределенные вычисления и поддержки TensorFlow Lite для мобильных устройств

PyTorch и TensorFlow

```
import tensorflow as tf

# Создание модели
model = tf.keras.Sequential([
    tf.keras.layers.Dense(10, activation='relu'),
    tf.keras.layers.Dense(2, activation='softmax')
])
model.compile(optimizer='adam',
              loss='sparse_categorical_crossentropy')

# Обучение
X_train = tf.random.normal((100, 10))
y_train = tf.random.uniform((100,), maxval=2, dtype=tf.int32)
model.fit(X_train, y_train, epochs=10)
```

```
import torch
import torch.nn as nn
import torch.optim as optim

# Создание модели
class Net(nn.Module):
    def __init__(self):
        super().__init__()
        self.fc = nn.Linear(10, 2)
    def forward(self, x):
        return self.fc(x)

model = Net()
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters(), lr=0.001)
```

```
# Обучение
inputs = torch.randn(5, 10)
labels = torch.tensor([1, 0, 1, 0, 1])
for epoch in range(100):
    outputs = model(inputs)
    loss = criterion(outputs, labels)
    optimizer.zero_grad()
    loss.backward()
    optimizer.step()
```

SciPy

SciPy – библиотека для научных вычислений, включающая модули для оптимизации, линейной алгебры, статистики и обработки сигналов

Используется в инженерных расчетах, физическом моделировании и анализе данных

Пример применения:

- Решение системы дифференциальных уравнений с помощью `integrate.Solve_ivp`
- Интерполяция данных `interpolate.Interp1d`
- Работа с разреженными матрицами `sparse`

```
from scipy.optimize import minimize
from scipy import stats
import numpy as np
```

Минимизация функции

```
result = minimize(lambda x: x**2 + 5, x0=2)
print(result.x) # [-0.0]
```

Генерация случайных данных

```
data = stats.norm.rvs(size=1000)
print(stats.describe(data))
```



Основная концепция языка

Возникновение, характеристики языка и применимость

Python для ML и DataScience

Экосистема и библиотеки Numpy, Numba, Pandas, Scipy

Python для автоматизации

Написание мелких утилит

Современный подход и правила

Улучшаем Python-код внутренними идиомами

Интеграция с другими языками

Добавление C++, Rust, C-вставок в Python-код



← Мы здесь

Автоматизация

Автоматизация – процесс внедрения технологий и систем, позволяющих максимально оптимизировать и упростить рутинные процессы путем делегирования их выполнения программам и технике

Программы небольшого объема и состоящие из одного файла, называют **скриптом**

Лучше всего для этого подходит:

- семейство Shell языков (используются в Unix подобных системах)
- Python
- GoLang

Проще всего писать на Shell, так как в некоторых задачах он разительно лаконичнее даже чем Python, и такие скрипты будут запускаться на любой Unix Системе

Bash shell

Bash – командный язык и оболочка для Unix-систем, предназначенный для управления процессами, файлами и автоматизации задач

Сам по себе этот язык сценариев очень прост и выполняется построчно, что даёт систему управления, похожую на скрипты Python

Или же просто последовательное выполнение команд в терминале оболочки сервера, если в скрипте нет ветвления или обработки ошибок

Весь функционал языка довольно ограничен:

Выполнение команд	Работа с файлами	Аргументы командной строки
Переменные	Перенаправление ввода / вывода	Управление процессами
Управление потоком выполнения	Конвейеры (Pipes)	Обработка ошибок
Циклы	Функции	Взаимодействие с пользователем

Анализ логов веб-сервера

(подсчет уникальных IP, статус-кодов и популярных URL)

```
#!/bin/bash

# Функция для вывода помощи
usage() {
    echo "Usage: $0 <logfile> [top-ips-limit]"
    exit 1
}

# Функция обработки ошибок
error_exit() {
    echo "Error: $1" >&2
    exit 1
}

# Проверка аргументов
[ $# -eq 0 ] && usage
LOG_FILE="$1"
TOP_IPS_LIMIT=${2:-10} # Дефолтное значение: 10

# Проверка существования файла
[ -f "$LOG_FILE" ] || error_exit "File $LOG_FILE not found"
```

```
# Проверка зависимостей
for cmd in awk sed sort uniq; do
    if ! command -v "$cmd" &> /dev/null; then
        error_exit "Command '$cmd' not found"
    fi
done

# Основная обработка через pipe
analyze_logs() {
    echo -e "\nTop $TOP_IPS_LIMIT IPs:"
    awk '{print 1}' "$LOG_FILE" | sort | uniq -c | sort -nr | head -n
    "$TOP_IPS_LIMIT"
    echo -e "\nStatus code distribution:"
    awk '{print 9}' "$LOG_FILE" | sort | uniq -c | sort -nr |
    awk '{printf "Status %s: %s times\n", 2, 1}'
    echo -e "\nMost frequent URLs:"
    awk '{print 7}' "$LOG_FILE" | sort | uniq -c | sort -nr | head -n 5
}

# Обработка ошибок выполнения
if ! analyze_logs; then
    error_exit "Failed to process logs"
fi

echo -e "\nTotal requests: (wc -l < \"$LOG_FILE\")"
```

Fish shell

(модификация Bash Shell)

```
#!/usr/bin/fish
# Функция для вывода помощи
function usage
    echo "Usage: "(status filename)" <logfile> [top-ips-limit]"
    exit 1
end

# Функция обработки ошибок
function error_exit
    echo "Error: $argv[1]" >&2
    exit 1
end

# Проверка аргументов
if test (count $argv) -eq 0
    usage
end
set LOG_FILE $argv[1]
set TOP_IPS_LIMIT (math "default $argv[2] 10")

# Проверка файла
if not test -f $LOG_FILE
    error_exit "File $LOG_FILE not found"
end
```

```
# Основная обработка
function analyze_logs
    echo -e "\nTop $TOP_IPS_LIMIT IPs:"
    awk '{print 1}' $LOG_FILE | sort | uniq -c | sort -nr | head -n TOP_IPS_LIMIT

    echo -e "\nStatus code distribution:"
    awk '{print 9}' $LOG_FILE | sort | uniq -c | sort -nr | awk '{printf "Status %s: %s times\n", $2, 1}'

    echo -e "\nMost frequent URLs:"
    awk '{print 7}' LOG_FILE | sort | uniq -c | sort -nr | head -n 5
end

# Запуск анализа
if not analyze_logs
    error_exit "Failed to process logs"
end

echo -e "\nTotal requests: "(wc -l < $LOG_FILE)
```

Python

```
#!/usr/bin/env python3
import argparse
from collections import defaultdict

def main():
    parser = argparse.ArgumentParser(description='Log analyzer')
    parser.add_argument('logfile', help='Path to log file')
    parser.add_argument('-t', '--top', type=int, default=10, help='Top entries limit')
    args = parser.parse_args()

    run_algo(args)

if __name__ == "__main__":
    main()
```

```
def run_algo(args, counters):
    counters = defaultdict(lambda: defaultdict(int))
    try:
        with open(args.logfile) as f:
            for line in f:
                parts = line.strip().split()
                if len(parts) < 9: continue
                counters['ip'][parts[0]] += 1
                counters['status'][parts[8]] += 1
                counters['url'][parts[6]] += 1
            for category in ['ip', 'status', 'url']:
                print(f"\nTop {args.top if category != 'url' else 5} {category.upper()}s:")
                items = sorted(counters[category].items(), key=lambda x: -
x[1][:args.top if category != 'url' else 5])
                [print(f'{k}: {v}') for k, v in items]
                print(f"\nTotal requests: {sum(counters['status'].values())}")
    except Exception as e:
        print(f"Error: {type(e).__name__} - {e}")
        exit(1)
```

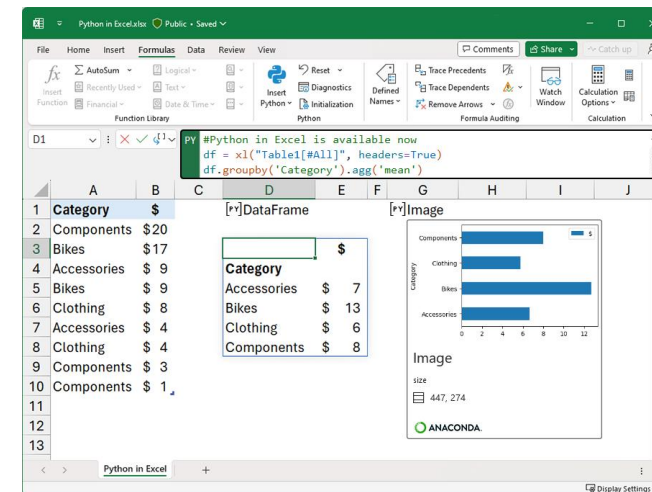
Python есть везде

Python в микроконтроллерах

MicroPython, чистый Python
в кофемашине

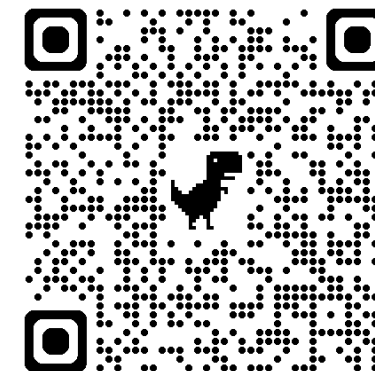
Python в Excel

Доступны формулы и библиотеки



Даже внутри JavaScript:
rascript – Open Source платформа для Python в браузере

Пример проекта о получении данных с сервера



Пример проекта о получении данных с сервера

index.html

```
<!DOCTYPE html>
<html lang="en">
<head>
  <title>API Proxy Tutorial</title>

  <!-- Recommended meta tags -->
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width,initial-scale=1.0">

  <!-- PyScript CSS -->
  <link rel="stylesheet" href="https://pyscript.net/releases/2024.5.2/core.css">

  <!-- This script tag bootstraps PyScript -->
  <script type="module" src="https://pyscript.net/releases/2024.5.2/core.js"></script>
</head>
<body>
  <script type="py" src="./main.py" config="./pyscript.toml" async terminal></script>
</body>
</html>
```

main.py

```
from pyscript import fetch

response = await fetch(
    "https://examples.pyscriptapps.com/api-proxy-
tutorial/api/proxies/status-check",
    method="GET"
).json()

print(response)
```

pyscript.toml

```
name = "API Proxy Tutorial"
description = "Introduction to using API Proxies"
tags = ["features", "tutorial"]
```

Python для DevOps

Популярность Python в DevOps обусловлена его простотой, читабельностью и мощными библиотеками, что делает его идеальным для:

Автоматизации

Python упрощает повторяющиеся задачи – от развертывания до мониторинга

Кроссплатформенной совместимости

Скрипты, написанные на Python, могут работать в любой операционной системе

Интеграции инструментов

Python работает как с Jenkins, Docker, Kubernetes, так и облачными платформами (AWS, GCP, Azure)

Пример | Сбор метрик компонентов системы роботов

Изначальная задача:

Написать модуль по обработке и анализу метрик для всех компонент системы с минимальными корректировками по обработке данных для каждого модуля

Целевая платформа:

C++ со своими классами и полями

Для запуска метрик мы не имеем доступ к данному C++ коду

Требуется повторить классы с данными, которые могут быть получены из JSON строки

Пример | Сбор метрик компонентов системы роботов

Базовый класс для сбора метрик

```
from dataclasses import dataclass, field
from typing import Dict, List
import time
import logging

@dataclass
class ComponentMetrics:
    component_name: str
    metrics: Dict[str, List[float]] = field(default_factory=dict)

    def log_metric(self, metric_name: str, value: float):
        if metric_name not in self.metrics:
            self.metrics[metric_name] = []
        self.metrics[metric_name].append(value)
        logging.info(f"[{self.component_name}] {metric_name}: {value}")

    def get_summary(self) -> Dict[str, float]:
        return {
            metric: sum(values) / len(values)
            for metric, values in self.metrics.items()
        }
```

Локализация

```
class LocalizationMetrics(ComponentMetrics):
    def __init__(self):
        super().__init__("Localization")

    def track_error(self, x_error: float, y_error: float):
        self.log_metric("position_error_x", x_error)
        self.log_metric("position_error_y", y_error)
```

Навигация

```
class NavigationMetrics(ComponentMetrics):
    def __init__(self):
        super().__init__("Navigation")

    def track_path_deviation(self, deviation: float):
        self.log_metric("path_deviation", deviation)
```

Уведомления

```
class NotificationMetrics(ComponentMetrics):
    def __init__(self):
        super().__init__("Notifications")

    def track_latency(self, latency_ms: float):
        self.log_metric("notification_latency", latency_ms)
```

Пример | Сбор метрик компонентов системы роботов

```
def main():  
    # Инициализация  
    loc_metrics = LocalizationMetrics()  
    nav_metrics = NavigationMetrics()  
    notif_metrics = NotificationMetrics()  
  
    # Симуляция работы системы  
    for _ in range(10):  
        # Локализация  
        loc_metrics.track_error(x_error=0.1, y_error=0.2)  
  
        # Навигация  
        nav_metrics.track_path_deviation(deviation=0.05)  
  
        # Уведомления  
        notif_metrics.track_latency(latency_ms=150)  
        time.sleep(0.1)  
  
    # Анализ и экспорт метрик  
    print("Локализация:", loc_metrics.get_summary())  
    print("Навигация:", nav_metrics.get_summary())  
    print("Уведомления:", notif_metrics.get_summary())
```

```
# Результат выполнения  
[Localization] position_error_x: 0.1  
[Localization] position_error_y: 0.2  
[Navigation] path_deviation: 0.05  
[Notifications] notification_latency: 150  
...  
  
Локализация: {'position_error_x': 0.1, 'position_error_y': 0.2}  
Навигация: {'path_deviation': 0.05}  
Уведомления: {'notification_latency': 150.0}
```

Пример | Сбор метрик компонентов системы роботов

Почему это удобно?

Универсальность:

Единая кодовая база для всех компонентов

Минимальные правки:

Добавление нового компонента требует только наследования от `ComponentMetrics`

Интеграция:

Легко подключить экспорт в веб-сервисы по агрегации метрик

Логирование:

Стандартизированный формат логов для анализа в ELK-стеках

Кроссплатформенность:

Работает на Linux (роботы), MacOS (тесты), Docker / Kubernetes (продакшен)



Основная концепция языка

Возникновение, характеристики языка и применимость

Python для ML и DataScience

Экосистема и библиотеки Numpy, Numba, Pandas, Scipy

Python для автоматизации

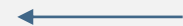
Написание мелких утилит

Современный подход и правила

Улучшаем Python-код внутренними идиомами

Интеграция с другими языками

Добавление C++, Rust, C-вставок в Python-код



Мы здесь

Улучшаем производительность Python

Основные две причины проблем производительности Python: **типизация** и **многопоточность**. Но язык развивается и появляются новые функции (как внутренние, так и внешние), которые позволяют оптимизировать код минимум в 2 раза

Некоторые правила для написания более производительного кода:

Работа с коллекциями

- Списковые включения
- Генераторы
- Распаковка

Функции

- Аргументы по умолчанию
- Функции высшего порядка

Управление ресурсами

- Менеджеры контекста

ООП

- Перегрузка операторов
- Свойства

Прочие идиомы

- Обработка исключений
- enumerate для индексов
- f-строки



Улучшаем производительность Python

Например, мы можем заменить модуль создания структур из JSON и обратно в модуле подготовки данных для задачи анализа и вывода метрик работы робота

Путем смены библиотеки по преобразованию JSON в DATACLASS получили прирост в **10 раз**

```
from dataclasses import dataclass
from dataclasses_json import dataclass_json
```

```
@dataclass_json
@dataclass
class SimpleExample:
    str_field: str
```

```
SimpleExample.from_dict({'str_field': 'howdy!'})
SimpleExample.from_json('{"str_field": "howdy!"}')
# SimpleExample(str_field='howdy!')
```

```
from dataclasses import dataclass
from fastclasses_json import dataclass_json
```

```
@dataclass_json
@dataclass
class SimpleExample:
    str_field: str
```

```
SimpleExample.from_dict({'str_field': 'howdy!'})
SimpleExample.from_json('{"str_field": "howdy!"}')
# SimpleExample(str_field='howdy!')
```

Плюсы объявления типов, даже если язык не требует

! Аннотация типов не влияет на производительность, а язык динамически типизирован
Однако в языке есть ряд инструментов для указания типа переменной

Читаемость

код становится ближе к статически типизированному языку

Инструменты

IDE (PyCharm, VSCode) лучше подсказывают и находят ошибки

Совместимость

аннотации используются в FastAPI для валидации данных

```
def greet(name: str, age: int) -> str:  
    return f"Hello {name}, you are {age} years old."
```

```
from typing import List, Dict, Optional  
  
User = Dict[str, str]  
def get_users() -> List[Optional[User]]:  
    return [{"name": "Alice"}, None]
```

```
from typing import Protocol
```

```
class Drawable(Protocol):  
    def draw(self) -> None: ...
```

```
class Circle:  
    def draw(self) -> None:  
        print("Drawing circle")
```

```
def render(item: Drawable) -> None:  
    item.draw()
```


Списковые включения (List Comprehensions)

Замена циклов for при создании списков

```
squares = list()
for x in range(10):
    squares.append(x**2)
```



```
squares = [x**2 for x in range(10)] # [0, 1, 4, ..., 81]
```

Генераторы (Generators)

Ленивые вычисления для экономии памяти

```
squares = list()
for x in range(10):
    squares.append(x**2)
```



```
gen = (x**2 for x in range(10)) # Не хранит весь список
print(next(gen)) # 0
```

Распаковка (Unpacking)

Извлечение элементов из коллекций в переменные

```
c = [1, 2, 3, 4]  
a = c[0]  
b = c[1]  
rest = c[2:]
```



```
a, b, *rest = [1, 2, 3, 4] # a=1, b=2, rest=[3,4]  
first, *_ = (10, 20, 30)  # first=10
```

Аргументы по умолчанию

Использование None для изменяемых аргументов

```
# Только для существующих массивов  
def add_item(item, lst):  
    lst.append(item)  
    return lst
```



```
# для любых массивов  
def add_item(item, lst=None):  
    lst = lst or []  
    lst.append(item)  
    return lst
```

Функции высшего порядка

Передача функций как аргументов

```
numbers = [1, 2, 3]
doubled = list()
for x in doubled:
    doubled.append(x * 2)
```



```
numbers = [1, 2, 3]
doubled = list(map(lambda x: x * 2, numbers)) # [2, 4, 6]
```

Менеджеры контекста (with)

Автоматическое закрытие файлов и соединений (аналог RAII в C++)

```
f = open("file.txt", "r")  
try:  
    data = f.read()  
finally:  
    f.close()
```



```
with open("file.txt", "r") as f:  
    data = f.read()
```

Перегрузка операторов

Использование `__add__`, `__repr__` для классов

```
class Vector:
```

```
    x: int = 0
```

```
    y: int = 0
```

```
v1 = Vector(1, 2)
```

```
v2 = Vector(3, 4)
```

```
v3 = Vector(0,0)
```

```
v3.x = v1.x + v2.x
```

```
v3.y = v1.y + v2.y
```



```
class Vector:
```

```
    x: int = 0
```

```
    y: int = 0
```

```
    def __init__(self, x, y):
```

```
        self.x = x
```

```
        self.y = y
```

```
    def __add__(self, other):
```

```
        return Vector(self.x + other.x, self.y + other.y)
```

```
v1 = Vector(1, 2)
```

```
v2 = Vector(3, 4)
```

```
v3 = v1 + v2 # Vector(4, 6)
```

Свойства (@property)

Контроль доступа к полям класса

```
class Circle:  
    _radius: float = 0.0
```

```
    def __init__(self, radius):  
        self._radius = radius
```

*# Внутри User кода идут проверки
А не внутри класса*



```
class Circle:  
    def __init__(self, radius):  
        self._radius = radius
```

```
    @property  
    def radius(self):  
        return self._radius
```

```
    @radius.setter  
    def radius(self, value):  
        if value > 0:  
            self._radius = value
```


Обработка исключений (try-except-else)

Выполнение кода при отсутствии ошибок

```
result = 10 / 0.0
```



```
x = 0.0
# ... some code
try:
    result = 10 / x
except ZeroDivisionError:
    print("Ошибка деления!")
else:
    print(f"Результат: {result}") #
    Выполнится, если нет исключения
```

f-строки

Форматирование строк через f"{"}

```
name = "Alice"  
print("Hello, {0}!".format(name))
```



```
name = "Alice"  
print(f"Hello, {name}!") # Hello, Alice!
```

Итерация (enumerate для индексов)

Замена перебора по индексам в стиле C++

```
names = ["Alice", "Bob", "Charlie"]  
idx = 0  
for name in names:  
    print(f"{idx}: {name}")  
    idx += 1
```



```
names = ["Alice", "Bob", "Charlie"]  
for idx, name in enumerate(names):  
    print(f"{idx}: {name}")
```



Основная концепция языка

Возникновение, характеристики языка и применимость

Python для ML и DataScience

Экосистема и библиотеки Numpy, Numba, Pandas, Scipy

Python для автоматизации


Написание мелких утилит

Современный подход и правила

Улучшаем Python-код внутренними идиомами

Интеграция с другими языками

Добавление C++, Rust, C-вставок в Python-код



Мы здесь



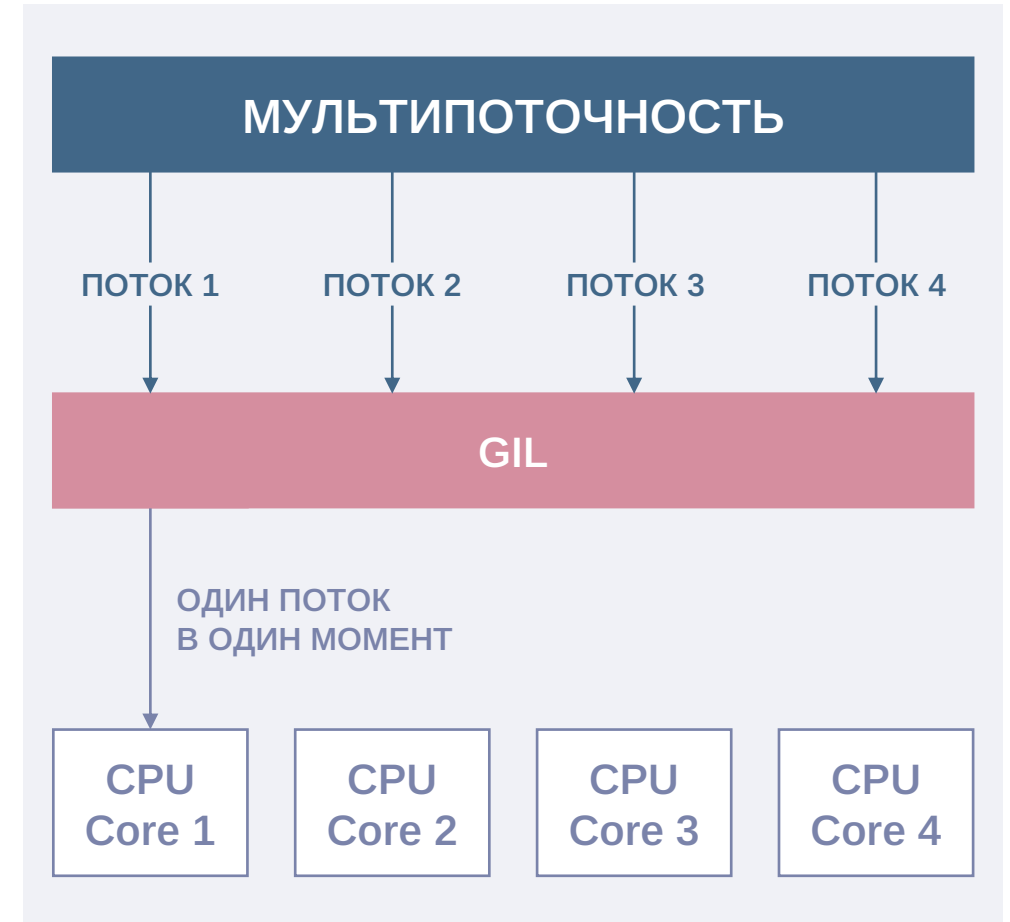
Python и «недомногопоточность»

GIL, или Global Interpreter Lock – своеобразная блокировка, которая позволяет только одному потоку управлять интерпретатором Python

Таким образом корректно работать в один момент времени можно только с одним потоком

- Защищает доступ к объектам Python, предотвращая одновременное выполнение байт-кодов несколькими потоками
- Работает даже в многопоточном приложении

Такая блокировка необходима, поскольку управление памятью в Python не считается потокобезопасным



Многопоточность (Threading)

! Подходит для задач с ожиданием I/O

Пример:

Веб-скрапинг с параллельными запросами

```
import threading

def fetch_url(url):
    # Запрос к сайту
    print(f"Fetching {url}...")

urls = ["https://site1.com", "https://site2.com"]
threads = []
for url in urls:
    thread = threading.Thread(target=fetch_url, args=(url,))
    thread.start()
    threads.append(thread)

for thread in threads:
    thread.join()
```

Асинхронность (Asyncio)

! Эффективнее потоков
для высоконагруженных I/O-операций

```
import asyncio

async def fetch_url(url):
    print(f"Fetching {url}...")
    await asyncio.sleep(1) # Имитация I/O-ожидания

async def main():
    urls = ["https://site1.com", "https://site2.com"]
    tasks = [fetch_url(url) for url in urls]
    await asyncio.gather(*tasks)

asyncio.run(main())
```

Мультипроцессинг (Multiprocessing)

! Обход GIL для CPU-bound задач
через процессы

Для каждого процесса выделяется
свой GIL со своей памятью

```
from multiprocessing import Pool

def compute(x):
    return x ** 2

with Pool(4) as p:
    results = p.map(compute, [1, 2, 3]) # [1, 4, 9]
```


Как добавить другой язык

! Основное – выбрать языки, которые оптимально решают задачи проекта

Стоит учесть:



Есть ли ограничения по ресурсам?



Какая команда и сроки задачи?



Какая тематика приложения?



Какая примерная архитектура будущего приложения?

Грамотно komponуем архитектуру проекта

Задача архитектуры чётко разделить модули по языкам и ответственности
Каждый модуль должен решать **одну конкретную задачу**

Пример:

- Модуль auth отвечает только за аутентификацию и авторизацию
- Модуль payment обрабатывает платежи, но не занимается генерацией чеков
- Модуль notifications отправляет уведомления, но не определяет, кому их отправлять

Как реализовать:

- Разбивайте код на директории / пакеты по функциональности

```
src/  
├── auth/      # Вход, регистрация, JWT  
├── payments/  # Интеграция с Stripe, PayPal  
├── notifications/ # Email, SMS, push-уведомления  
└── core/      # Общая бизнес-логика
```

Добавление другого языка

Задача сводится к разработке Python модуля на другом языке программирования, например C++, Rust, Zig, C и т.д.

```
// Rust Lib
use pyo3::prelude::*;

/// Formats the sum of two numbers as string.
#[pyfunction]
fn sum_as_string(a: usize, b: usize) -> PyResult<String> {
    Ok((a + b).to_string())
}

/// A Python module implemented in Rust. The name of this function must match
/// the `lib.name` setting in the `Cargo.toml`, else Python will not be able to
/// import the module.
#[pymodule]
fn string_sum(m: &Bound<'_, PyModule>) -> PyResult<()> {
    m.add_function(wrap_pyfunction!(sum_as_string, m)?);
    Ok(())
}
```

```
$ python
>>> import string_sum
>>> string_sum.sum_as_string(5, 20)
'25'
```

C++: pybind11 для Python

```
// -----
// pure C++ code
// -----

std::vector<int> multiply(const std::vector<double>& input)
{
    std::vector<int> output(input.size());

    for ( size_t i = 0 ; i < input.size() ; ++i )
        output[i] = 10*static_cast<int>(input[i]);

    return output;
}
```

```
namespace py = pybind11;

// wrap C++ function with NumPy array IO
py::array_t<int> py_multiply(py::array_t<double, py::array::c_style |
py::array::forcecast> array)
{
    // allocate std::vector (to pass to the C++ function)
    std::vector<double> array_vec(array.size());

    // copy py::array -> std::vector
    std::memcpy(array_vec.data(), array.data(), array.size()*sizeof(double));

    // call pure C++ function
    std::vector<int> result_vec = multiply(array_vec);

    // allocate py::array (to pass the result of the C++ function to Python)
    auto result = py::array_t<int>(array.size());
    auto result_buffer = result.request();
    int *result_ptr = (int *) result_buffer.ptr;

    // copy std::vector -> py::array
    std::memcpy(result_ptr, result_vec.data(), result_vec.size()*sizeof(int));

    return result;
}
```

C++: pybind11 для Python

```
// wrap as Python module
PYBIND11_MODULE(example,m)
{
    m.doc() = "pybind11 example plugin";

    m.def("multiply", &py_multiply, "Convert all entries of
an 1-D NumPy-array to int and multiply by 10");
}
```

```
# main.py

import numpy as np
import example # Это наш модуль

A = [0,1,2,3,4,5]
B = example.multiply(A)

print('input list = ',A)
print('output    = ',B)

A = np.arange(10)
B = example.multiply(A)

print('input list = ',A)
print('output    = ',B)
```

Взаимодействие с «С»

Преобразовывает struct / union C
в классы Python с методами
сериализации и десериализации

Все делается внутри Python
без дополнительных библиотек

```
class Position(cstruct.MemCStruct):
    __def__ = """
        struct {
            unsigned char head;
            unsigned char sector;
            unsigned char cyl;
        }
    """

    @property
    def lba(self):
        return (self.cyl * 16 + self.head) * 63 + (self.sector - 1)

pos = Position(cyl=15, head=15, sector=63)
print(f"head: {pos.head} sector: {pos.sector} cyl: {pos.cyl} lba: {pos.lba}")

packed = pos.pack()
```

Спасибо за внимание!
Ваши вопросы?

