

---

**toolchain**

*Выпуск 1.0*

**Artem Vesnin**

мая 20, 2024



---

## Культура и инструменты разработки

---

<b>1 Терминология</b>	<b>1</b>
1.1 Терминология . . . . .	1
1.1.1 Приложение (англ. Application) . . . . .	1
1.1.2 Модуль/Пакет/Библиотека/Фреймворк (англ. Module/Package/Library/Framework) . . . . .	2
1.1.3 Модули приносят множество преимуществ в наш код; . . . . .	2
1.1.4 Парадигма программирования (англ. programming paradigm) . . . . .	4
1.1.5 Статическое и динамическое типизирование . . . . .	5
1.1.6 Тестирование . . . . .	6
1.1.7 Логирование (англ. Logging) . . . . .	7
1.1.8 Язык разметки (англ. Markup Language) . . . . .	7
1.1.9 Система контроля версий (англ. Version Control System) . . . . .	8
1.1.10 Ошибка (англ. bug) . . . . .	8
1.1.11 Задачи/Проблемы (англ. Issues) . . . . .	9
1.1.12 Fail-fast . . . . .	9
1.1.13 Исключение (англ. Exceptions) . . . . .	9
<b>2 Командная строка</b>	<b>11</b>
2.1 Работа в командной строке . . . . .	11
2.1.1 Мотивация . . . . .	11
2.1.2 cd - change directory . . . . .	12
2.1.3 ls - list . . . . .	14
2.1.4 clear . . . . .	16
2.1.5 mkdir - make directory . . . . .	17
2.1.6 touch . . . . .	17
2.1.7 cat - concatenate . . . . .	18
2.1.8 echo . . . . .	20
2.1.9 export . . . . .	20
2.1.10 \$ . . . . .	21
2.1.11 rm — remove . . . . .	24
2.1.12 rmdir — remove directory . . . . .	25
2.1.13 cp — copy . . . . .	26
2.1.14 mv — move . . . . .	26
2.1.15 chmod — change mode . . . . .	27
2.1.16 sudo — superuser do . . . . .	28
2.1.17 chown — change owner . . . . .	28
2.1.18 history . . . . .	29

2.1.19	grep — global regular expression print . . . . .	29
2.1.20	top — table of process . . . . .	31
2.1.21	Практические примеры . . . . .	31
2.2	Задачи . . . . .	34
2.2.1	Задача 1 . . . . .	34
2.2.2	Задача 2 . . . . .	34
2.2.3	Задача 3 . . . . .	35
2.2.4	Задача 4 . . . . .	36
2.2.5	Задача 5 . . . . .	36
2.3	Вопросы . . . . .	37
<b>3</b>	<b>Система контроля версий Git</b>	<b>39</b>
3.1	Git . . . . .	39
3.1.1	Система контроля версий Git . . . . .	39
3.1.2	Краткий исторический экскурс . . . . .	40
3.1.3	Приступаем к использованию Git . . . . .	40
3.1.4	Тестовый код программы . . . . .	44
3.1.5	Описание тестовой программы . . . . .	45
3.1.6	Оформление рабочего каталога . . . . .	45
3.1.7	Настройка конфигурации git . . . . .	46
3.1.8	Первый пример работы с git . . . . .	48
3.1.9	Вот несколько шагов, которые помогут начать работу с Git: . . . . .	54
3.1.10	Что такое публичный и приватный репозиторий? . . . . .	54
3.1.11	Зафиксируйте изменения git . . . . .	54
3.1.12	Немного статистики о системе git . . . . .	55
<b>4</b>	<b>Git как инструмент командной работы. Работа с удаленными репозиториями</b>	<b>57</b>
4.1	Git как инструмент командной работы. Работа с удаленными репозиториями . . . . .	57
4.1.1	Знакомство с методом SCRUM . . . . .	59
4.1.2	Знакомство с канбан-доской . . . . .	61
4.1.3	Задача: Планирование и организация работы над разработкой чат-бота с использованием Канбан доски . . . . .	63
4.1.4	Заключение . . . . .	64
4.1.5	Справочная информация . . . . .	65
4.1.6	Роли каждого участника команды . . . . .	65
4.1.7	Краткая история методологии SCRUM . . . . .	66
4.1.8	Управление коммуникациями . . . . .	67
4.1.9	Планирование коммуникаций: . . . . .	67
4.1.10	Мониторинг и контроль коммуникаций: . . . . .	67
4.1.11	Процесс управления коммуникациями в стартапе, использующем SCRUM, может выглядеть следующим образом: . . . . .	67
4.2	Задача: Планирование и организация работы над разработкой чат-бота с использованием Канбан доски . . . . .	70
4.3	Вопросы . . . . .	71
<b>5</b>	<b>Знакомство с платформами размещения исходного кода программ на примере GitFlic</b>	<b>73</b>
5.1	Знакомство с платформами размещения исходного кода программ на примере Gitflick . . . . .	73
5.1.1	Знакомство с отечественным сервисом GitFlic . . . . .	75
5.1.2	Типовой процесс совместной работы с кодом . . . . .	77
5.1.3	Задание для команды разработчиков: Создание чат-бота для мессенджера Telegram с использованием GitFlick . . . . .	81
5.1.4	Заключение . . . . .	82
5.1.5	Справочная информация . . . . .	83

5.2	Задание для команды разработчиков: Создание чат-бота для мессенджера Telegram с использованием GitFlick . . . . .	84
5.3	Вопросы . . . . .	86
<b>6</b>	<b>Среда разработки</b>	<b>87</b>
6.1	Среда разработки . . . . .	87
6.1.1	Подсветка синтаксиса . . . . .	88
6.1.2	Автодополнение кода (завершение кода) . . . . .	89
6.1.3	Рефакторинг . . . . .	92
6.1.4	Компиляция . . . . .	93
6.1.5	Настройка окружения под себя . . . . .	94
<b>7</b>	<b>Стили оформления кода</b>	<b>95</b>
7.1	Оформление кода . . . . .	95
7.1.1	Используйте отступы . . . . .	96
7.1.2	Соблюдайте принятую длину строки . . . . .	98
7.1.3	Используйте фигурные скобки правильно . . . . .	99
7.1.4	Именуйте объекты правильно . . . . .	100
7.1.5	Придерживаться определенного подхода . . . . .	100
7.1.6	Избегайте вложенности (Flat is better than nested) . . . . .	101
7.1.7	Не перегружайте код (Sparse is better than dense) . . . . .	102
7.1.8	Выводы . . . . .	102
7.2	Задачи . . . . .	102
7.2.1	Задача 1 . . . . .	102
7.2.2	Задача 2 . . . . .	103
7.2.3	Задача 3 . . . . .	103
7.2.4	Задача 4 . . . . .	104
7.2.5	Задача 5 . . . . .	106
7.3	Вопросы . . . . .	109
<b>8</b>	<b>Польза документации на примере поиска проекта на GitHub</b>	<b>111</b>
8.1	Сервис GitHub . . . . .	111
8.1.1	Учимся читать и использовать в своем проекте чужой код с GitHub . . . . .	111
8.1.2	Где и как искать код? . . . . .	112
8.1.3	Работа с GitHub . . . . .	122
<b>9</b>	<b>Раздел 2. Работа над проектом</b>	<b>133</b>
<b>10</b>	<b>Стадии жизни проекта</b>	<b>135</b>
10.1	Стадии жизни проекта . . . . .	135
10.1.1	От человека: . . . . .	135
10.1.2	Поехали . . . . .	136
10.1.3	Идея . . . . .	137
10.1.4	Визуализация . . . . .	138
10.1.5	Архитектура . . . . .	138
10.1.6	Разработка . . . . .	139
10.1.7	Отладка . . . . .	140
10.1.8	Документация . . . . .	141
10.1.9	Релиз . . . . .	142
10.1.10	Сборка . . . . .	143
10.1.11	Контейнеризация . . . . .	144
<b>11</b>	<b>Минимально полезный продукт</b>	<b>145</b>
11.1	Работа над MVP . . . . .	145
11.1.1	Этапы разработки и проверка гипотез . . . . .	145

11.1.2	Вот пример MVP для новой службы доставки еды: . . . . .	146
11.1.3	Вот пример MVP для нового мобильного приложения: . . . . .	147
11.1.4	Как оценить качество MVP, какие показатели, критерии? . . . . .	148
11.1.5	Задание для проверки . . . . .	149
<b>12</b>	<b>UML диаграммы</b>	<b>151</b>
12.1	Факультатив. Стандартные графические схемы при разработке ПО . . . . .	151
12.1.1	Возможности использования UML (Unified Modeling Language) . . . . .	151
12.1.2	Что такое UML? . . . . .	152
12.1.3	Каковы основные типы диаграмм в UML? . . . . .	154
12.2	Задачи . . . . .	178
12.2.1	1. Построение диаграммы классов для чат-бота Telegram: . . . . .	178
12.2.2	2. Создание диаграммы последовательности для обработки запросов от пользователя: . . . . .	179
12.2.3	3. Построение диаграммы состояний для управления состоянием чат-бота: . . . . .	179
12.2.4	4. Разработка диаграммы компонентов для чат-бота Telegram: . . . . .	179
12.2.5	5. Построение диаграммы прецедентов для чат-бота Telegram: . . . . .	179
12.3	Вопросы . . . . .	179
<b>13</b>	<b>Код vs. Библиотека</b>	<b>181</b>
13.1	Код -> Библиотека . . . . .	181
13.1.1	Мотивация . . . . .	181
13.1.2	Определяем функциональность библиотеки . . . . .	181
13.1.3	Собираем проект . . . . .	186
13.1.4	Выводы . . . . .	188
13.2	Задачи . . . . .	188
13.2.1	Задача 1 . . . . .	188
13.2.2	Задача 2 . . . . .	191
13.2.3	Задача 3 . . . . .	192
13.2.4	Задача 4 . . . . .	193
13.3	Вопросы . . . . .	195
<b>14</b>	<b>Основы Open Source</b>	<b>197</b>
14.1	Основы Open Source, базовые определения и принципы устройства . . . . .	197
14.1.1	Что такое Open Source? . . . . .	197
14.1.2	Как развивалось сообщество открытого исходного кода? . . . . .	198
14.1.3	Как открытый исходный код помогает развивать технологии и ускорять прогресс? 198	198
<b>15</b>	<b>Лицензии Open Source</b>	<b>201</b>
15.1	Лицензии Open Source . . . . .	201
15.1.1	Самые распространенные лицензии Open Source . . . . .	201
15.1.2	Какая лицензия ПО позволяет использовать исходный код для коммерческой разработки ПО? . . . . .	203
15.1.3	Какая лицензия ПО запрещает использование кода в коммерческих целях, для получения прибыли? . . . . .	204
15.1.4	Какие есть варианты использования лицензии GPL? . . . . .	205
15.1.5	Какая лицензия у ОС Android? . . . . .	205
15.1.6	Какую лицензию выбрать, если я хочу использовать чужой код в своем проекте? 206	206
15.1.7	Несколько общих советов по лицензированию и защите прав разработчиков программного обеспечения . . . . .	207
15.1.8	Интересный вопрос: кому принадлежат права на результат работы искусственного интеллекта? . . . . .	207
<b>16</b>	<b>GitHub Pages</b>	<b>209</b>
16.1	Github Pages . . . . .	209

16.1.1	С какой целью используют сервис GitHub Pages? . . . . .	215
16.1.2	Преимущества и недостатки сервиса GitHub Pages . . . . .	215
16.2	Задачи . . . . .	216
16.3	Вопросы . . . . .	218
<b>17</b>	<b>Раздел 3. Инструменты разработки</b>	<b>219</b>
<b>18</b>	<b>Виртуальное окружение</b>	<b>221</b>
18.1	Знакомство с рабочим окружением . . . . .	221
18.1.1	Добавить директорию в PATH . . . . .	224
18.1.2	Знакомство с virtualenv . . . . .	225
18.1.3	Справочный материал . . . . .	230
18.2	Задачи для практики . . . . .	232
18.2.1	Задача: настроить виртуальную среду с помощью venv для управления зависимостями проекта . . . . .	232
18.2.2	Задача: настроить виртуальную среду с помощью venv и отправить код в репозиторий Git . . . . .	232
18.3	Вопросы: . . . . .	233
<b>19</b>	<b>Инструменты для создания документации</b>	<b>235</b>
19.1	Документирование . . . . .	235
19.1.1	Мотивация . . . . .	235
19.1.2	Markdown . . . . .	236
19.1.3	reStructuredText . . . . .	242
19.1.4	Sphinx . . . . .	242
19.1.5	Markdown и Sphinx . . . . .	244
19.1.6	Документация из кода . . . . .	245
19.1.7	Оформление документации . . . . .	246
19.2	Задачи . . . . .	252
19.2.1	Задача 1 . . . . .	252
19.2.2	Задача 2 . . . . .	253
19.2.3	Задача 3 . . . . .	256
19.3	Вопросы . . . . .	256
<b>20</b>	<b>Основы тестирования</b>	<b>259</b>
20.1	Тестирование . . . . .	259
20.1.1	Мотивация . . . . .	259
20.1.2	Виды тестов . . . . .	260
20.1.3	Интеграционное тестирование . . . . .	265
20.1.4	Больше тестов . . . . .	265
20.1.5	Разработка через тестирование . . . . .	265
20.2	Задачи: . . . . .	268
20.2.1	Задача 1 . . . . .	268
20.2.2	Задача 2 . . . . .	269
20.2.3	Задача 3 . . . . .	272
20.2.4	Задача 4 . . . . .	273
20.3	Вопросы . . . . .	275
<b>21</b>	<b>Логгирование работы приложения</b>	<b>277</b>
21.1	Логирование. Что это и в чем его польза? . . . . .	277
21.1.1	Мотивация . . . . .	277
21.1.2	Зачем нужно логирование? . . . . .	277
21.1.3	Уровни логирования . . . . .	278
21.1.4	Основы грамотного логирования . . . . .	279
21.1.5	Библиотеки логирования для языка программирования Python . . . . .	280

21.1.6 Централизация логов . . . . .	284
21.1.7 Вывод . . . . .	284
21.2 Задача . . . . .	284
21.2.1 Задача 1 . . . . .	284
21.2.2 Задача 2 . . . . .	285
21.2.3 Задача 3 . . . . .	287
21.3 Вопросы . . . . .	289
<b>22 Сборка проекта в pip пакет</b>	<b>291</b>
22.1 Менеджеры пакетов . . . . .	291
22.1.1 Ctrl-C/Ctrl-V . . . . .	291
22.1.2 Клонируем и указываем путь через sys . . . . .	292
22.1.3 Ставим через pip, используя git репозиторий . . . . .	293
22.1.4 Делаем сборку библиотеки для публикации. pip . . . . .	294
22.1.5 Публикация в PyPI . . . . .	299
22.2 Задачи . . . . .	299
22.2.1 Задача 1 . . . . .	299
22.2.2 Задача 2 . . . . .	300
22.3 Вопросы . . . . .	301
<b>23 Менеджеры служб и планировщики задач</b>	<b>303</b>
23.1 Менеджеры и планировщики . . . . .	303
23.1.1 Мотивация . . . . .	303
23.1.2 Планировщик crontab . . . . .	304
23.1.3 Менеджер systemctl . . . . .	307
23.2 Задачи . . . . .	311
23.2.1 Задача 1 . . . . .	311
23.2.2 Задача 2 . . . . .	312
23.3 Вопросы . . . . .	313
<b>24 Контейнеризация с Docker</b>	<b>315</b>
24.1 Контейнеризация на примере Docker . . . . .	315
24.1.1 Введение . . . . .	315
24.1.2 Рассмотрим пример: как Docker может помочь в разработке ПО . . . . .	316
24.1.3 Концепции Docker . . . . .	318
24.1.4 Установка и использование . . . . .	318
24.1.5 Файл Dockerfile . . . . .	320
24.1.6 Практическое задание: Развертывание Python кода в Docker контейнере . . . . .	325
24.1.7 Заключение . . . . .	327
24.1.8 Ссылки на дополнительные материалы по Docker . . . . .	328
24.2 Практическое задание 1: Развертывание Python кода в Docker контейнере . . . . .	328
24.2.1 Практическое задание 2: Развертывание Python кода в Docker контейнере с использованием образа Alpine . . . . .	329
24.3 Вопросы . . . . .	330
<b>25 Дополнительные материалы</b>	<b>333</b>
<b>26 Работа с сетью в Docker</b>	<b>335</b>
26.1 Концепция сети в Docker . . . . .	335
26.1.1 Познакомимся с тем, как объединять несколько контейнеров в сеть Docker . . . . .	335
26.1.2 Сети: вводная часть . . . . .	335
26.1.3 Познакомимся с сетями Bridge и Host на практике . . . . .	338
<b>27 Реестр Docker</b>	<b>341</b>
27.1 Реестр Docker . . . . .	341

27.1.1	Создание первого Docker репозитория . . . . .	343
27.1.2	Поиск Docker образа . . . . .	345
27.1.3	Загрузка Docker образа из реестра Docker Hub . . . . .	347
27.1.4	Создание Docker образа . . . . .	348
27.1.5	Как это работает? . . . . .	348
27.1.6	Создайте Dockerfile, который определяет конфигурацию нашего приложения . . . . .	348
27.1.7	Образ на Docker Hub . . . . .	349



## Терминология

---

### 1.1 Терминология

В данном курсе мы будем использовать терминологию, связанную с разработкой программного обеспечения, поэтому полезно собрать все термины в одном месте. Ниже мы будем приводить русский и английский варианты терминов, так как последние будут полезны при поиске информации. Мы не претендуем на полноту списка терминов, используемых для разработки приложений, но постарались собрать те из них, которые важны в рамках данного курса, и по возможности снабдить их примерами.

#### 1.1.1 Приложение (англ. Application)

По-английски это Application, или app. Последнее часто используется при написании приложений. Например, при написании **бэкэнда** используется именно это имя для того, чтобы обозначить главное приложение.

(код):

```
from fastapi import FastAPI

app = FastAPI()

@app.get("/")
async def root():
    return {"message": "Hello World"}
```

### 1.1.2 Модуль/Пакет/Библиотека/Фреймворк (англ. Module/Package/Library/Framework)

Эти термины связаны с организацией кода и с тем, как он функционирует. Если мы хотим, чтобы наш проект можно было читать и, как следствие, он был интересен другим разработчикам, которые могли бы его развивать, мы должны позаботиться о структуре проекта. Все начинается с **модуля**, модуль — это файл с кодом, который объединен общим смыслом. Например, если у нас есть код для генерации данных и их визуализации, будет неплохой идеей разделить его на два файла-модуля. *Примечание: если модулей генерации данных и визуализации несколько, мы уже будем объединять их в пакеты, о которых речь пойдет ниже.* Python-модули имеют расширение .py.

(код):

```
"""From module generating data"""
def generate_arithmetic_series(n):
    return list(range(n))

def generate_geometric(b, n):
    return list([b**i for i in range(n)])
```

(код):

```
"""From module plotting data"""
import matplotlib.pyplot as plt

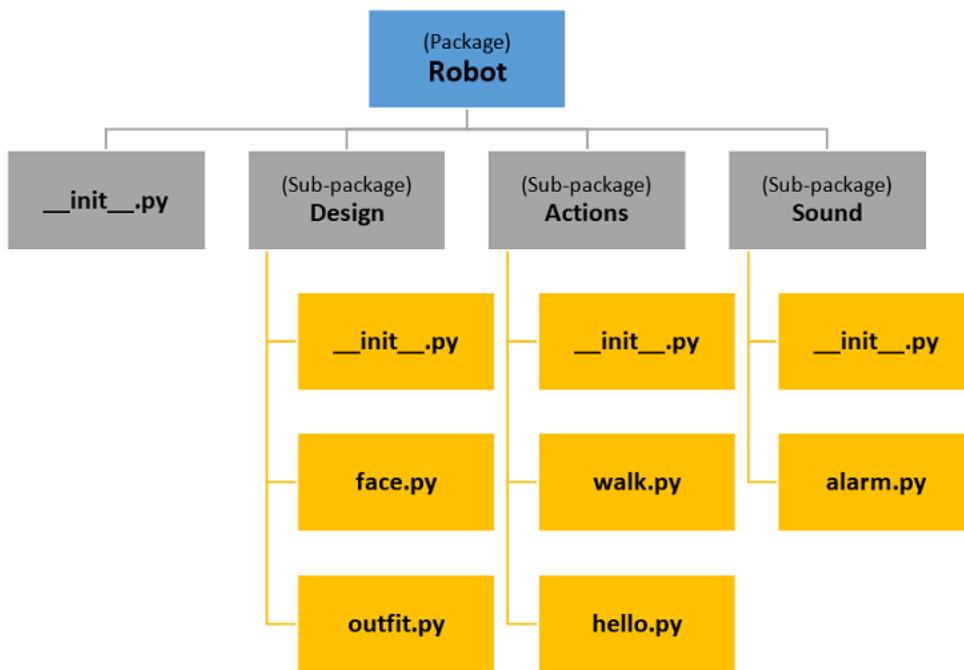
def plot_lines(data):
    plt.plot(data)
    plt.show()

def plot_dots(data):
    x = [x for x in range(len(data))]
    plt.scatter(x, data)
    plt.show()
```

### 1.1.3 Модули привносят множество преимуществ в наш код:

- Улучшенный процесс разработки. Модули Python помогают вам сосредоточиться на одной небольшой части задачи, не держа в голове весь контекст проекта. Кроме того, модули обычно пишутся таким образом, чтобы свести к минимуму зависимость друга от друга. Почти всегда, когда возникает желание использовать код одного модуля в другом и наоборот, можно выделить взаимный код в третий модуль и использовать его в двух первых. Также мы уменьшаем количество конфликтов при слиянии кода (*merge conflict*).
- Уменьшение количества кода. Функциональность, которую мы определяем в одном модуле, можем использовать в разных частях приложения, сводя к минимуму дублирование кода.
- Отдельные пространства имен. С модулями Python вы можете определить отдельные пространства имен, чтобы избежать конфликтов между идентификаторами в разных частях вашего приложения.

Термин **пакет** является достаточно перегруженным, и его значение может меняться в зависимости от контекста. Так как в курсе мы в основном работаем с Python, будем придерживаться терминологии, принятой в нем. **Пакет** — это следующий уровень в организации кода после модулей. Если модули содержали функции и классы, то пакет уже содержит сами модули. Для того, чтобы каталог был воспринят как пакет, он должен содержать файл `__init__.py`. *Пример* пакета:



Пакеты могут содержать внутри себя каталоги, которые, в свою очередь, тоже являются пакетами.

**Библиотека** — это общий термин для кода, который можно переиспользовать. Мы все пишем код и собираем его в модули и пакеты, но это не значит, что наш код можно считать библиотекой. Для этого он должен обладать некоторой долей универсальности, определенным оформлением и другими элементами, например, покрытием тестами. Т.е. библиотека обязательно является пакетом, но не каждый пакет можно считать библиотекой. Например, на сайте [numpy](#) указано, что это пакет:

NumPy is the fundamental Python package for scientific computing

При этом на сайте другого популярного проекта [matplotlib](#) говорится, что это библиотека:

Matplotlib is a comprehensive library for creating static, animated, and interactive visualizations in Python

Поэтому есть мнение, что библиотека — это *пакет с пакетами*, т.е. что-то, что обладает более разносторонней функциональностью, чем пакет.

Итак, мы подобрались к самым сложным объектам — **фреймворкам**. Наиболее подходящий перевод с английского — *основа* или *каркас*, т.е. что-то, на чем мы строим или на что навешиваем свой функционал. Идея фреймворков появилась, когда программисты поняли, что часто используют библиотеки почти одинаковым способом. Например, нам нужно принять данные от пользователя, валидировать их и сохранить в базе данных. Можно «поднять» все необходимые компоненты и написать все это вручную, а можно взять готовый **фреймворк**, где уже будет реализован базовый поток данных и архитектура приложения, и нам нужно будет только написать код, который относится к нашей задаче. Поэтому распространенное определение, что фреймворк — это набор библиотек, не совсем верно. Можно думать о **различии библиотеки и фреймворка** так: если мы не напишем ни одной строчки кода, **фреймворк** все равно будет работать и поддерживать базовый поток. Поэтому тот код, который напишет программист, используя фреймворк, будет выполняться и контролироваться этим фреймворком. Еще один способ думать о фреймворках — это насколько полным является его функционал для решения задачи, т.е. мы можем ли решить бизнес-задачу, не выходя за пределы какого-то инструмента. Мы часто слышим про фреймворки в контексте [web-приложений](#), но этим не ограничивается их применение.

### 1.1.4 Парадигма программирования (англ. programming paradigm)

Способ, как мы смотрим на то, что происходит при разработке. Например, с точки зрения водителя улица, через которую он едет *транзитом*, это ширина проезжей части дороги, количество светофоров и прочее. Но улица, на которой расположен пункт назначения, уже рассматривается водителем с точки зрения удобства парковки, хотя на транзитной улице тоже есть парковочные места. Т.е. на одну и туже сущность *улица* мы можем смотреть под разными углами, в зависимости от того, что сейчас для нас важно. *Парадигма программирования* — это способ описать задачу, например, данные и действия над ними. Рассмотрим на примере процедурной и объектно-ориентированной парадигм. **Процедурное** программирование описывает список действий (процедур), которые мы предпринимаем для достижения цели. **Объектно-ориентированное** программирование строится вокруг объектов и их свойств, здесь мы применяем такие термины как Класс, Экземпляр (Объект), Поле (Атрибут), Метод. **Объектно-ориентированное** программирование — наиболее часто используемый подход, хотя существуют задачи (1, 2), где решение лучше описывается в других парадигмах. Подробнее об ООП на примере Python читайте в [статье](#). На практике мы можем смешивать парадигмы, если это не вредит поддерживаемости кода. Ниже приведен пример решения одной и той же задачи с использованием разных парадигм:

(код):

```
# PROCEDURAL
def average_age(table):
    average = 0
    count = 0
    for item in table:
        average = average + item[1]
        count = count + 1
    return average / count

people = [("Ivan", 24), ("Mary", 20), ("Alex", 21), ("Sara", 29)]
print("Average age:", average_age(people))
```

(код):

```
# OOP
class Person:

    def __init__(self, name, age):
        self.name = name
        self.age = age

class Workers:

    def __init__(self):
        self.workers = []

    def add(self, worker):
        self.workers.append(worker)

    def get_average_age(self):
        age_sum = 0
        for worker in self.workers:
            age_sum = age_sum + worker.age
        return age_sum / len(self.workers)
```

(continues on next page)

(продолжение с предыдущей страницы)

```

workers = Workers()
workers.add(Person(name="Ivan", age = 24))
workers.add(Person(name="Mary", age = 20))
workers.add(Person(name="Alex", age = 21))
workers.add(Person(name="Sara", age = 29))

print("Average age:", workers.get_average_age())

```

Что такое? Почему в ООП больше кода, когда это такая классная парадигма? Все верно, кода больше, но он лучше структурирован, и нужно думать не о количестве кода, а о том, насколько его легко читать и насколько его легко поддерживать. Например, средний возраст — это характеристика группы людей, поэтому мы пишем метод в `Workers`, а если мы хотим написать метод, который увеличит возраст сотрудника в день его рождения, мы идем в класс `Person`. Также в **процедурном** программировании часто возникают конструкции типа `item[1]`, и мы должны помнить, что возраст — это индекс 1 (а не 0, индексы нумеруются от нуля). В **объектно-ориентированном** мы не работаем с большой таблицей данных, а структурируем информацию в том виде, с которым принято работать в той или иной сфере. Подумайте, как бы вы решили следующие задачи:

- храним не возраст, а дату рождения, и нужно по-прежнему знать возраст;
- хотим уволить :( человека.

### 1.1.5 Статическое и динамическое типизирование

В Python мы можем объявить переменную без указания типа, интерпретатор сам определит тип переменной. Это называется **динамическая типизация**.

```

if __name__ == "__main__":
    magic_number = 42.
    print(type(s))

```

С другой стороны, когда в описании переменной необходимо указывать тип, мы имеем дело со статической типизацией. Она применяется как в C++, так и во многих других языках.

```

#include <iostream>

int main() {
    float magic_number = 42.;
    std::cout << "Hello " << magic_number;
    return 0;
}

```

Теперь посмотрим не со стороны языка программирования, а со стороны разработчика. При объявлении переменной ее тип может быть понятен из значения, которое в ней хранится. Но при использовании функций мы не можем понять, что именно мы должны передать без использования документации. Поэтому мы можем использовать аннотацию типов и в таких языках как Python:

(код):

```

def fib_nohint(n):
    a, b = 0, 1
    while a < n:

```

(continues on next page)

(продолжение с предыдущей страницы)

```

yield a
a, b = b, a+b

def fib_hint(n: int) -> Iterator[int]:
    a, b = 0, 1
    while a < n:
        yield a
        a, b = b, a+b

```

Функцию, где указаны типы аргументов и тип возвращаемого значения, проще использовать. Кроме удобства при чтении кода, можно добавить также и [валидацию](#) значений переменных, используя библиотеку [pydantic](#). Проводить проверку своего кода на наличие аннотирования можно, используя такие инструменты как [шурп](#).

### 1.1.6 Тестирование

Оно должно быть. Рассмотрим здесь несколько важных терминов: **Покрытие тестами** (англ. Test Coverage) — доля кода, которая покрыта тестами, т.е. та часть кода, которую мы действительно выполняем с использованием тестовых данных. Ниже приведен пример на C++: код скомпилируется, но при этом его нельзя выполнить без ошибки. Если бы функция `foo` была покрыта тестами, мы бы увидели ошибку сразу.

(код):

```

#include <iostream>

int main() {
    std::cout << "Hello World!";
    return 0;
}

float foo(int a) {
    a = 0;
    return 42. / a;
}

```

**Разработка через тестирование** (англ. Test-driven development) — подход, при котором мы сначала пишем тесты, и только потом разрабатываем саму функциональность. На практике это не всегда достижимо и не всегда удобно, поэтому в написании тестов после разработки функциональности нет ничего плохого.

**Юнит-тестирование** (англ. Unit Testing) — разбиение программы на мелкие блоки, каждый из которых может быть протестирован отдельно. Пример ниже содержит простой класс, который умеет только добавлять 10 и умножать на 10. Мы можем подумать над тестами заранее (**Разработка через тестирование**) или написать тесты потом, в любом случае, нам нужно добиться прохождения тестов. На практике вы будете использовать библиотеки типа `unittest`, пример ниже специально написан без использования сторонних библиотек, чтобы показать концепцию тестирования и не отвлекаться на изучение конкретных инструментов. Позже в курсе тестированию будут посвящены отдельные занятия.

(код):

```

class Calculator:

    def add_10(a):
        return a + 10

    def multiply_by_10(a):
        return a * 10

if __name__ == "__main__":
    # testing code
    assert Calculator.add_10(12) == 22           #test 1
    print("Pass test 1")
    assert Calculator.multiply_by_10(12) == 120 #test 2
    print("Pass test 2")
    try:                                         #test 3
        Calculator.add_10("asda")
        print("Fail test 3")
    except:
        print("Pass test 3")

    try:                                         #test 4
        Calculator.multiply_by_10("asda")
        print("Fail test 4")
    except:
        print("Pass test 4")

```

*Подумайте, почему четвертый тест не прошел и как сделать так, чтобы он отработал. Оцените, насколько удобно иметь тесты.*

### 1.1.7 Логирование (англ. Logging)

Ведение журнала событий, в котором записывается информация, облегчающая отслеживание событий, которые нельзя отнести к нормальной работе кода или системы в целом.

### 1.1.8 Язык разметки (англ. Markup Language)

**Язык разметки** позволяет снабдить текст специальными символами, которые регулируют его отображение. Например, этот документ написан с помощью языка разметки *Markdown*, а GitHub или Gitlab собрали из него красивый документ, который вы сейчас и читаете. Популярность *Markdown* породила множество инструментов написания текста в нем и компиляции итогового документа. Данный раздел выглядит вот так (редактор ReText):

#### ### Язык разметки (англ. - Markup Language)

**\*\*Язык разметки\*\*** позволяет снабдить текст специальными символами, которые регулирует его отображение. Например, этот документ написан с помощью языка разметки *\*Markdown\**, а GitHub или Gitlab собрали из него красивый документ, который вы сейчас и читаете. Популярность *\*Markdown\** породила [множество] (<https://habr.com/ru/post/672266/>) инструментов написания текста в нем и компиляции итогового документа. Данный раздел выглядит вот так (редактор ReText):

### 1.1.9 Система контроля версий (англ. Version Control System)

**Система контроля версий** позволяет облегчить работу над приложением, даже при работе в одиночку. При работе в команде она практически незаменима. В курсе мы будем использовать **git**, которому посвящено несколько занятий. На основе git построены такие системы как GitLab и GitHub, которые предоставляют удаленный репозиторий git и привносят дополнительную функциональность, такую как отслеживание багов через \*Задачи/Проблемы (англ. Issues), визуализацию и многое другое.

### 1.1.10 Ошибка (англ. bug)

Ошибка или непонятное поведение программы, в общем, все, что нельзя отнести к нормальному поведению программы. Часто используется прямая транскрипция с английского — **баг**. Баг может приводить к сбоям при использовании определенного места в программе, и это хороший сценарий, поскольку сразу понятно, где ошибка. *В примере ниже в print\_value() идет обращение к переменной, которую ранее нигде не инициализировали, это можно исправить, написав self.value вместо value*

(код):

```
class ClassWithBugFailFast:

    def print_value(self):
        print(value)

    def set_value(self, value):
        self.value = value

if __name__ == "__main__":
    cls = ClassWithBugFailFast()
    cls.set_value(123)
    cls.print_value()
```

Часто бывает, что ошибки не происходит, но поведение программы неочевидно.

(код):

```
class ClassWithBugNoFail:

    def print_value(self, value):
        print(self.value)

    def set_value(self, value):
        self.value = value

if __name__ == "__main__":
    cls = ClassWithBugNoFail()
    cls.set_value(123)
    cls.print_value(456)
```

### 1.1.11 Задачи/Проблемы (англ. Issues)

Одна из целей, которые преследуют Issues, — документирование багов и отслеживание их решения. Однако они используется гораздо шире и, как многие другие инструменты, призваны повысить эффективность разработки приложений.

### 1.1.12 Fail-fast

В примере выше мы посмотрели, что неправильно написанный код, который сразу приводит к ошибке, лучше, чем код, который приводит не к ошибке, а к непонятным результатам. Понятно, что мы не планируем писать баги, но поведение, при котором мы вызываем ошибку (кидаем исключение) при непонятной ситуации, получило название **fail-fast**. Подход **fail-fast** является предпочтительным при написании кода, так как позволяет определить место, где произошла ошибка. Другим подходом при обработке непредвиденной ситуации является подход **forgive**, при котором мы пытаемся уменьшить негативные последствия и стараемся не кидать ошибки. Подробнее про два подхода можно почитать в [статье](#)

### 1.1.13 Исключение (англ. Exceptions)

Исключительная ситуация требует исключительных мер. При работе приложения мы стараемся предусмотреть возможные варианты его использования, например, варианты входных данных, которые приходят в приложение от пользователя. Данные от пользователя могут быть недопустимыми, и мы можем попробовать исправить это. Например, конвертировать строку, содержащую число, в целое число, используя `int()`, а если не получится, то `float()`. Для данного примера считаем, что это приемлемый вариант.

(код):

```
class Validator:

    def get_int_number(string):
        result = 0
        try:
            result = int(string)
        except ValueError:
            print('Could convert to int, try via float:')
            result = int(float(string))
        return result

if __name__ == "__main__":
    print(Validator.get_int_number('10.'))
    try:
        user_input = 'aaa'
        Validator.get_int_number(user_input)
    except ValueError:
        print(f'{user_input} is not a number, please enter proper data')
    except Exception as e:
        print(f'Unknow error ask developers {e}')
```

Если не получается конвертировать в число в принципе, как в случае со строкой 'aaa', мы можем попросить пользователя повторить ввод. Если же происходит что-то, что мы не можем идентифицировать, то тоже должны «поймать» это на верхнем уровне (в примере выше это `except Exception as e`) и зафиксировать: можно, например, записать сообщение об ошибке в лог.



## Командная строка

---

### 2.1 Работа в командной строке

Команды `bash` являются неотъемлемой частью разработки. Командная строка, не только `bash`, но и другие, дает большие возможности по сравнению с любым графическим интерфейсом. Любой интерфейс полагается на эти команды и вызывает их с соответствующими параметрами.

#### 2.1.1 Мотивация

1. Относитесь к `bash` как универсальному скриптовому языку, который покажет одинаковое поведение на разных системах Линукс. С помощью `bash` и различных утилит мы можем получать доступ к «железу» машины, на которой работаем, и настраивать автоматические сценарии. Это можно осуществить и с использованием языков программирования, но сделать это будет сложнее, и реализация будет зависеть от конкретного языка программирования. Но если мы напишем `bash`, то «дернуть» его из любого языка программирования не составит труда, так как большинство из них поддерживает системные вызовы. Если же вызов системных функций невозможен или неудобен, обмен информацией всегда доступен через файлы, куда можно записать информацию из `bash`-скрипта, а прочитать из `python`.
2. При работе на своей компьютере вы можете не оценить все достоинства командной строки, однако она точно понадобится, чтобы настроить работу вашего приложения на удаленном сервере, физическом или виртуальном, к которому вы будете подключаться по `ssh` и на котором может быть в принципе не установлен оконный интерфейс.

Ниже будут описаны команды, которые помогут ходить по файловой системе, создавать, копировать и перемещать файлы, изменять разрешения для файлов, передавать владельца у файла, просматривать запущенные процессы и открытые порты. Предположим, у нас есть пользователь с именем **артем**, на машине, которая называется **pc**. Тогда при запуске командной строки мы увидим предложение ввода команды:

```
артем@pc:/$
```

Мы видим, что здесь указаны имя пользователя и имя компьютера через символ @. Далее после двоеточия (:) указан текущий путь, сейчас мы находимся в корневой директории, далее символ \$ для отделения команды. Начнем изучение командной строки с команды, которая позволяет ходить по файловой системе.

**Hint:** при нажатии на стрелку вверх в командную строку будет подставлена предыдущая команда. Стрелку можно нажимать несколько раз

**Hint:** Дополнительные информацию и опции по каждой команде можно увидеть вызвав man command или command --help, man сокращенно от manual

Мы надеемся, что это было достаточно убедительно, чтобы не пройти мимо темы bash.

### 2.1.2 cd - change directory

Команда для смены директории. При вызове происходит переход в директорию, которая указана после команды cd /path/to/destination. Предположим, у нас есть пользователь с именем artem. Следующие команды можно выполнить, чтобы перейти сначала в директорию home, а затем в домашнюю директорию пользователя /home/artem:

```
artem@pc:/$ cd /home  
artem@pc:/home$ cd ./artem  
artem@pc:~$
```

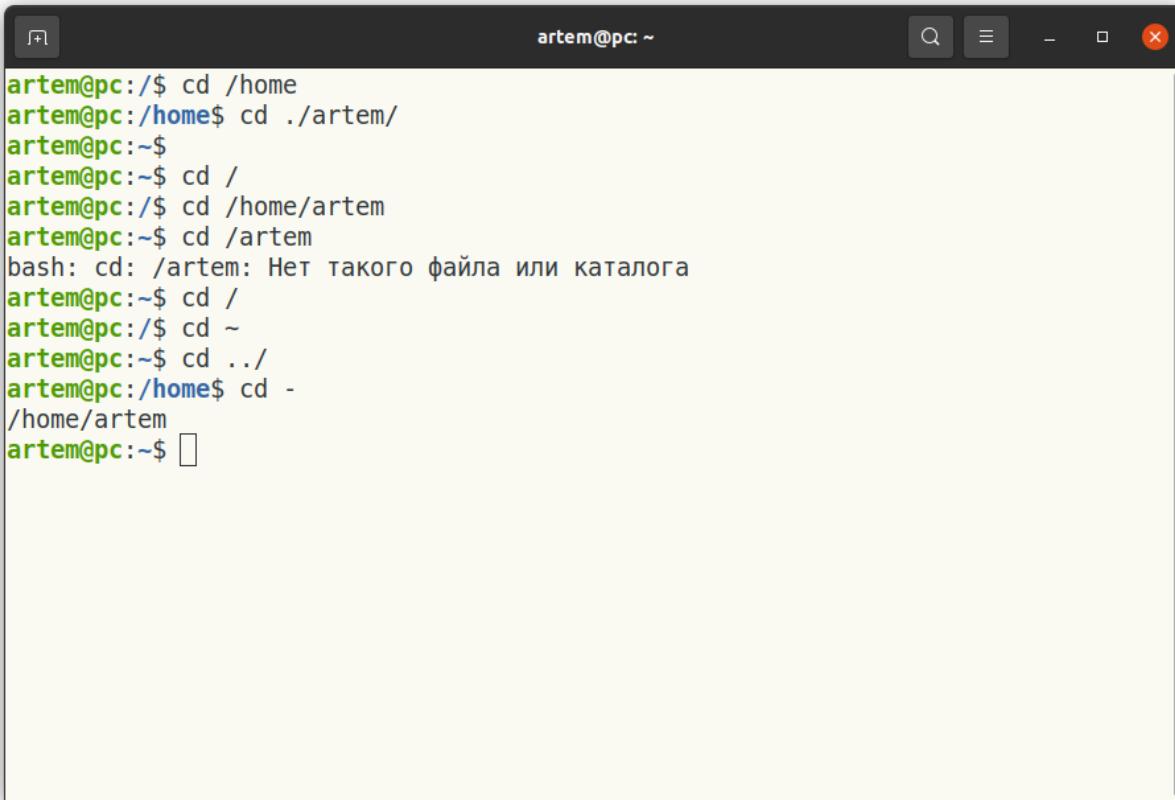
Заметьте, что во втором случае мы использовали ./ для перехода по относительному пути. Также видно, что меняется текущий путь, в котором мы находимся: в первой строке это была корневая директория (/), во второй — директория, в которой содержатся все домашние директории всех пользователей данной системы (/home). В третьей строке вместо пути мы видим символ ~, который является сокращением для домашней папки конкретного пользователя. Две команды выше эквиваленты одной команде с указанием абсолютного пути:

```
artem@pc:/$ cd /home/artem
```

Команда cd /artem вызовет ошибку, так как будет происходить поиск в корневой директории, а в ней **НЕ** содержится директории artem. При использовании cd можно использовать сокращения, которые позволяют переходить по определенным путям, не прописывая полный путь (в примерах ниже # означает начало комментария):

```
cd ~      # переход в домашнюю папку  
cd ..    # переход на один уровень выше  
cd ../../ # переход на два уровня выше, чтобы уйти еще выше, можно добавлять ../  
cd -     # возврат в предыдущую директорию  
cd .     # текущий каталог
```

На рисунке ниже представлено использование описанных команд.



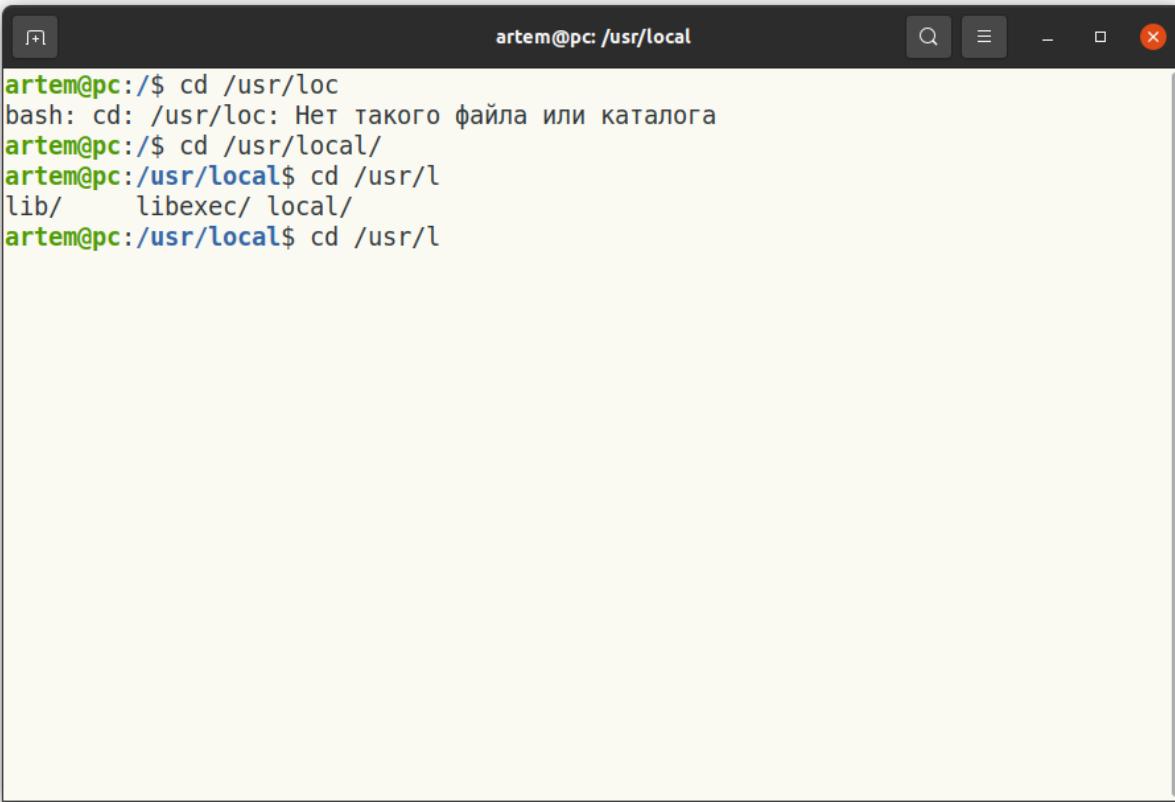
```
artem@pc:~/home$ cd ./artem/
artem@pc:~/home$ cd /home/artem
artem@pc:~/home$ cd /home/artem
bash: cd: /artem: Нет такого файла или каталога
artem@pc:~/home$ cd ~
artem@pc:~/home$ cd ../
artem@pc:~/home$ cd -
/home/artem
artem@pc:~$ █
```

При использовании команды `cd` необязательно помнить полное наименование директории. Можно начать писать путь и затем нажать на TAB для автодополнения.

```
artem@pc:~/home$ cd /usr/loc      # нажимаем TAB
artem@pc:~/home$ cd /usr/local/
```

Если написанного пользователем будет недостаточно для того, чтобы однозначно определить путь, командная строка выдаст все возможные варианты:

```
user#pc: cd /usr/l      # нажимаем TAB
lib/    libexec/    local/
user#pc: cd /usr/l
```



The screenshot shows a terminal window with a dark header bar. The header bar contains the text "artem@pc: /usr/local" on the left and standard window control icons (minimize, maximize, close) on the right. The main area of the terminal displays the following command-line session:

```
artem@pc:/$ cd /usr/loc
bash: cd: /usr/loc: Нет такого файла или каталога
artem@pc:/$ cd /usr/local/
artem@pc:/usr/local$ cd /usr/l
lib/    libexec/  local/
artem@pc:/usr/local$ cd /usr/l
```

Вопросы для проверки:

Что можно указать после команды `cd`?

- Абсолютный путь (верно)
- Относительный путь (верно)
- Адрес веб-сайта (НЕверно)
- Один из специальных наборов символов, например, `../` (верно)

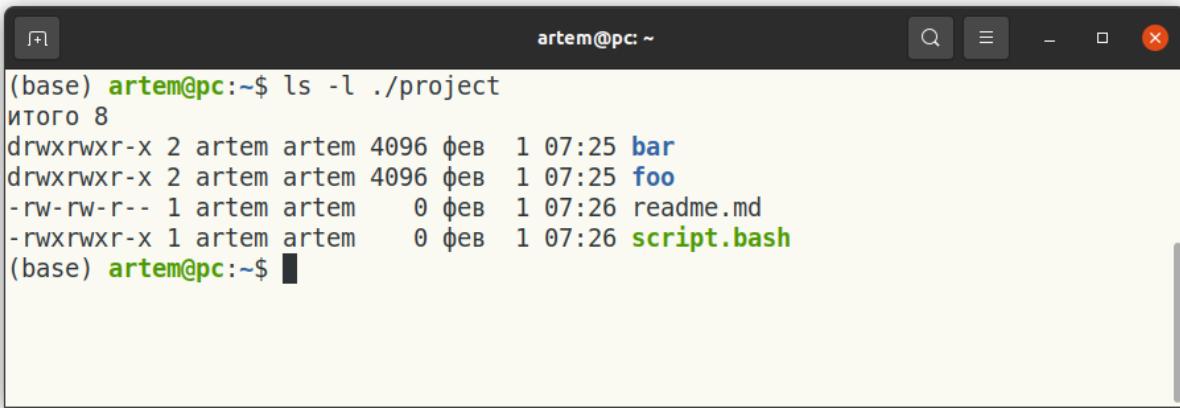
### 2.1.3 `ls` - list

Для просмотра содержимого директории можно использовать `ls`. При вызове без параметров будет показано содержимое *текущей директории*. Также можно указать путь `ls /path/to/dir`, в этом случае будет показано содержимое *указанной директории*.



```
artem@pc: ~
(base) artem@pc:~/project$ ls
bar  foo  readme.md  script.bash
(base) artem@pc:~/project$ cd ~
(base) artem@pc:~$ ls ./project
bar  foo  readme.md  script.bash
(base) artem@pc:~$
```

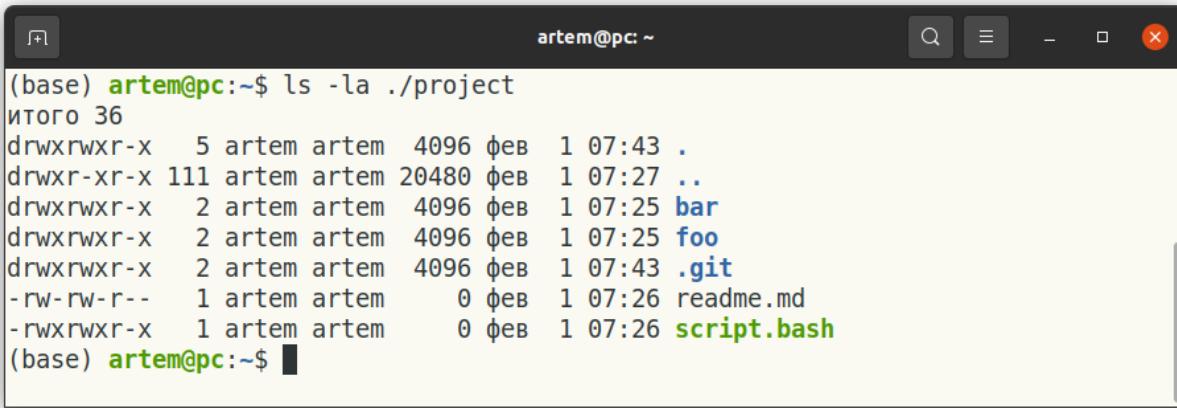
Мы видим, что разные имена подкрашены по-разному, на картинке выше три вида разметки. В вашей командой строке цвета могут отличаться. На картинке синий обозначает директорию, черный — обычный файл и зеленый — исполняемый файл. Но как понять это без цветовой индикации? Для этого вызовите `ls -l`:



```
artem@pc: ~
(base) artem@pc:~/project$ ls -l .
итого 8
drwxrwxr-x 2 artem artem 4096 фев  1 07:25 bar
drwxrwxr-x 2 artem artem 4096 фев  1 07:25 foo
-rw-rw-r-- 1 artem artem    0 фев  1 07:26 readme.md
-rwxrwxr-x 1 artem artem    0 фев  1 07:26 script.bash
(base) artem@pc:~$
```

Теперь мы видим больше информации по каждому файлу и директории в `project`. Среди этой информации: разрешение, владелец, группа, размер, дата последнего изменения, имя файла. Как видите, директория имеет букву `d` в разрешениях, а исполняемый файл, в отличие от неисполнимого, имеет атрибут `x`.

Для просмотра скрытых файлов нужно указать параметр `a`. К скрытым файлам относятся директории, которые создаются при инициализации репозитория `git`.



```
artem@pc:~$ ls -la ./project
итого 36
drwxrwxr-x  5 artem artem  4096 фев  1 07:43 .
drwxr-xr-x 111 artem artem 20480 фев  1 07:27 ..
drwxrwxr-x  2 artem artem  4096 фев  1 07:25 bar
drwxrwxr-x  2 artem artem  4096 фев  1 07:25 foo
drwxrwxr-x  2 artem artem  4096 фев  1 07:43 .git
-rw-rw-r--  1 artem artem     0 фев  1 07:26 readme.md
-rwxrwxr-x  1 artem artem     0 фев  1 07:26 script.bash
(base) artem@pc:~$
```

Вопросы для проверки:

*Можно ли указывать ls без параметров?*

- Да (верно)
- Нет (НЕверно)

#### 2.1.4 clear

Командой **clear** можно очистить терминал от вывода. При этом история команд останется, и их по-прежнему можно доставать, нажимая на стрелку вверх.



```
artem@pc:~$ ls -la ./project
итого 36
drwxrwxr-x  5 artem artem  4096 фев  1 07:43 .
drwxr-xr-x 111 artem artem 20480 фев  1 07:27 ..
drwxrwxr-x  2 artem artem  4096 фев  1 07:25 bar
drwxrwxr-x  2 artem artem  4096 фев  1 07:25 foo
drwxrwxr-x  2 artem artem  4096 фев  1 07:43 .git
-rw-rw-r--  1 artem artem     0 фев  1 07:26 readme.md
-rwxrwxr-x  1 artem artem     0 фев  1 07:26 script.bash
artem@pc:~$ clear #not pressed
```

При использовании clear мы увидим пустой терминал:



PS: для скриншотов данного материала `clear` использовалось ооочень много раз :)

### 2.1.5 `mkdir - make directory`

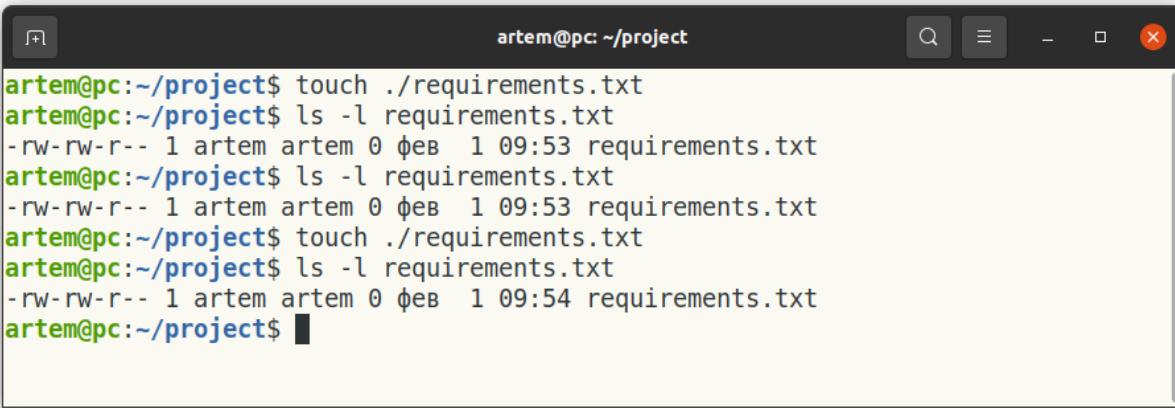
Для создания директорий мы используем команду `mkdir`. Так же, как в случае с `cd` и `ls`, мы можем использовать относительный путь. Если директория уже существует, будет выдано сообщение об ошибке.

```
artem@pc:~$ cd ./project
artem@pc:~/project$ mkdir baz
artem@pc:~/project$ mkdir baz
mkdir: невозможно создать каталог «baz»: Файл существует
artem@pc:~/project$
```

A screenshot of a terminal window titled "artem@pc: ~/project". The user runs the command "cd ./project" to change the directory to "/project". Then they run "mkdir baz", which fails because a directory named "baz" already exists. The terminal has a dark header bar and a light gray body.

### 2.1.6 `touch`

Представьте ситуацию: нужно узнать время, когда последний раз запускалось то или иное приложение, которое запускается в фоне. Существует множество способов это сделать: запись в базе данных, анализ логов и так далее. Самой простой способ для bash-скрипта — изменение времени последнего изменения какого-либо файла. Это можно сделать, записав что-то в файл. Если такой вариант не подходит, воспользуйтесь командой `touch`, которая как раз предназначена для изменения времени обращения к файлу на текущее, при этом открытие файла не происходит.



```
artem@pc:~/project$ touch ./requirements.txt
artem@pc:~/project$ ls -l requirements.txt
-rw-rw-r-- 1 artem artem 0 фев 1 09:53 requirements.txt
artem@pc:~/project$ ls -l requirements.txt
-rw-rw-r-- 1 artem artem 0 фев 1 09:53 requirements.txt
artem@pc:~/project$ touch ./requirements.txt
artem@pc:~/project$ ls -l requirements.txt
-rw-rw-r-- 1 artem artem 0 фев 1 09:54 requirements.txt
artem@pc:~/project$
```

Если файл не существует, он будет создан пустым, если не указан аргумент `-c`.



```
artem@pc:~/project$ ls
bar baz foo readme.md script.bash
artem@pc:~/project$ touch ./requirements.txt
artem@pc:~/project$ ls
bar baz foo readme.md requirements.txt script.bash
artem@pc:~/project$ touch -c ./some.txt
artem@pc:~/project$ ls
bar baz foo readme.md requirements.txt script.bash
artem@pc:~/project$
```

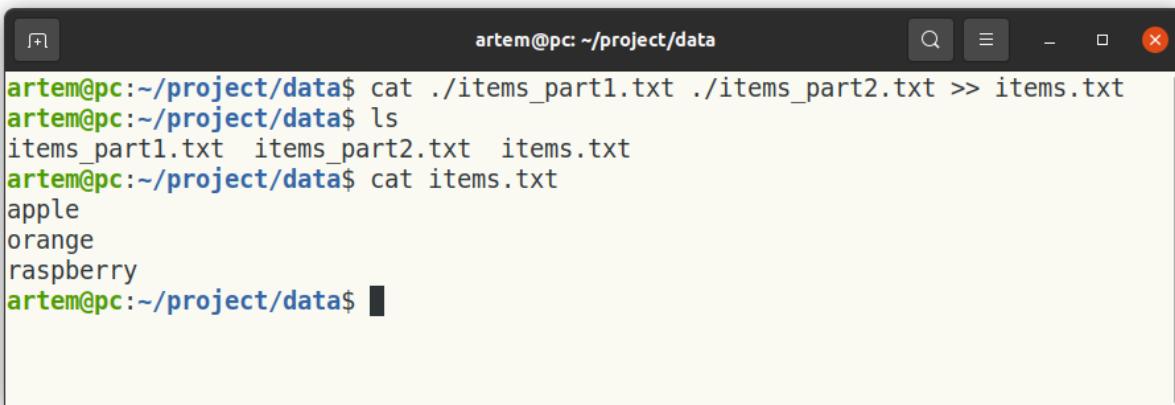
### 2.1.7 cat - concatenate

Читает содержимое файлов и выводит его. Добавим в проект директорию `data` и два файла `items_part1.txt` и `items_part2.txt`.



```
artem@pc:~/project/data$ ls
items_part1.txt items_part2.txt
artem@pc:~/project/data$ cat ./items_part1.txt
apple
orange
artem@pc:~/project/data$ cat ./items_part2.txt
raspberry
artem@pc:~/project/data$
```

Можно использовать `cat` для склеивания содержимого нескольких файлов: укажите несколько файлов после `cat`, затем `>>` и имя файла, в который нужно слить данные. Символы `>>` означают перенаправление потока, т.е. мы делаем вывод не в стандартный поток, а, например, в файл.



```
artem@pc:~/project/data$ cat ./items_part1.txt ./items_part2.txt >> items.txt
artem@pc:~/project/data$ ls
items_part1.txt items_part2.txt items.txt
artem@pc:~/project/data$ cat items.txt
apple
orange
raspberry
artem@pc:~/project/data$
```

С помощью `cat` можно пронумеровать строки в файлах сквозной нумерацией.



```
artem@pc:~/project/data$ cat -n ./items_part1.txt ./items_part2.txt
 1 apple
 2 orange
 3 raspberry
artem@pc:~/project/data$
```

**Hint:** посмотрите также команды `head` и `tail`, они выводят начало и конец файлов. `tail` будет удобен для просмотра журнала логов.

**Hint:** `>>` и `>` перенаправляют поток и создают файл, если его нет. Отличие состоит в том, что `>>` добавляет новое содержимое в конец файла, а `>` перезаписывает файл, удаляя предыдущее содержимое этого файла.

### 2.1.8 echo

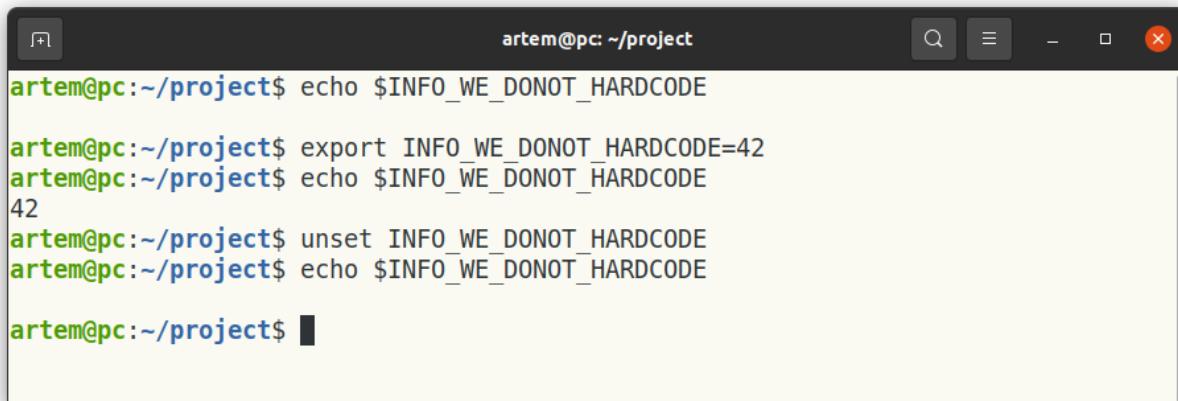
Обычно используется в скриптах для вывода информации (как `print` в `python`). Также с помощью `echo` можно записать строку в файл, если файла не было, он будет создан.



```
artem@pc:~/project$ cat ./requirements.txt
artem@pc:~/project$ echo "matplotlib" >> ./requirements.txt
artem@pc:~/project$ cat ./requirements.txt
matplotlib
artem@pc:~/project$ echo "Hello"
Hello
artem@pc:~/project$
```

### 2.1.9 export

Используется для установки переменных среды. Команда `export` отмечает переменную окружения для экспорта с любым новым дочерним процессом, и это позволяет дочернему процессу наследовать все отмеченные переменные. Они нам понадобятся для того, чтобы использовать ту информацию, которую мы не хотим прописывать в коде приложения, например, параметры доступа. Хардкодить такие значения неудобно, так как можно случайно закоммитить их, и придется все менять. Например, так сделано в проекте по созданию [чат-бота](#), посмотрите секцию **Setup** в `README`. Ниже приведен пример использования `export`.



```
artem@pc:~/project$ echo $INFO_WE_DONOT_HARDCODE
artem@pc:~/project$ export INFO_WE_DONOT_HARDCODE=42
artem@pc:~/project$ echo $INFO_WE_DONOT_HARDCODE
42
artem@pc:~/project$ unset INFO_WE_DONOT_HARDCODE
artem@pc:~/project$ echo $INFO_WE_DONOT_HARDCODE

artem@pc:~/project$
```

Подробнее с вариантами использования `export` можно ознакомиться [здесь](#).

## 2.1.10 \$

Есть несколько конструкций, в которые входит \$. Это может путать, но если относиться к этим конструкциям, как к обособленными элементам `bash`, так их легче воспринимать.

Символ \$ ставится перед переменной для того, чтобы получить ее значение. По умолчанию любое имя считается строкой.



```
artem@pc:~$ name=Artem
artem@pc:~$ echo name
name
artem@pc:~$ echo $name
Artem
artem@pc:~$ echo "Hello, $name"
Hello, Artem
artem@pc:~$ echo "Hello, name"
Hello, name
artem@pc:~$
```

При этом использование \$ внутри двойных кавычек и без них может вызвать разное поведение, если в значении переменной содержатся специальные символы. Если \$ используется в двойных кавычках, то значение переменной будет выдено как есть. Если \$ используется без кавычек, то `bash` проведет интерпретацию специальных символов, например, как ниже с \*, которую мы использовали для поиска файлов по шаблону.

Другое использование \$ — запись вывода команды в переменную. В этом случае \$ используется совместно со скобками.



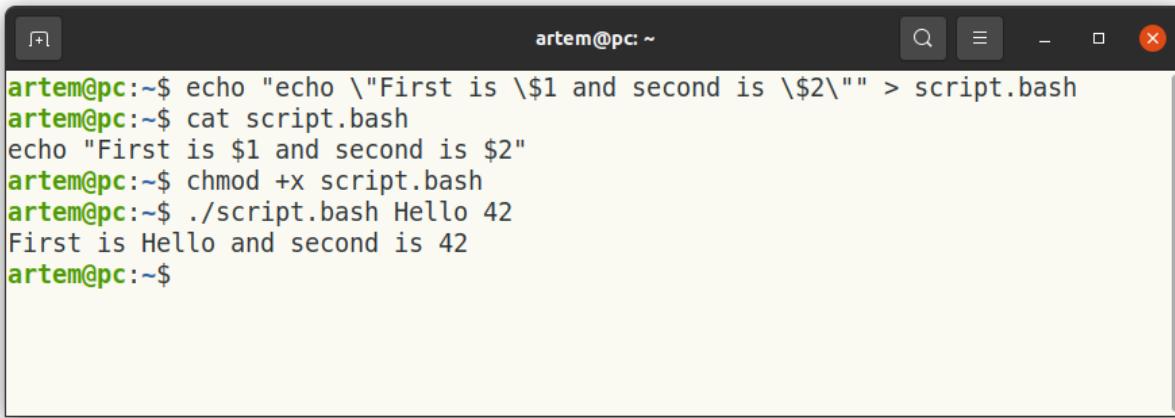
```
artem@pc:~$ echo "some content" > test.txt
artem@pc:~$ content=$(cat test.txt)
artem@pc:~$ echo $content
some content
artem@pc:~$
```

В `bash` нельзя просто так взять и выполнить математическую операцию. Для этого используется \$ совместно с двойными скобками.



```
artem@pc:~$ result=1+2
artem@pc:~$ echo $result
1+2
artem@pc:~$ result=$((1+2))
1+2: команда не найдена
artem@pc:~$ result=$((1+2))
artem@pc:~$ echo $result
3
artem@pc:~$
```

Также \$ используется для получения значений аргументов, переданных в скрипт. Можно считать, что сама оболочка присваивает номера переменным вместо того, чтобы именовать их. Это похоже на то, как мы используем `sys.argv[N]` в `python`, когда хотим получить то, что передано при вызове нашей программы.



```
artem@pc:~$ echo "echo \"First is \$1 and second is \$2\"" > script.bash
artem@pc:~$ cat script.bash
echo "First is $1 and second is $2"
artem@pc:~$ chmod +x script.bash
artem@pc:~$ ./script.bash Hello 42
First is Hello and second is 42
artem@pc:~$
```

Первая команда на скриншоте выше используется для того, чтобы создать код скрипта. Вы также можете взять код из двойных кавычек `echo "First is $1 and second is $2"` и сохранить его в файл `script.bash`, используя любой текстовый редактор. Описание `chmod` смотрите ниже.

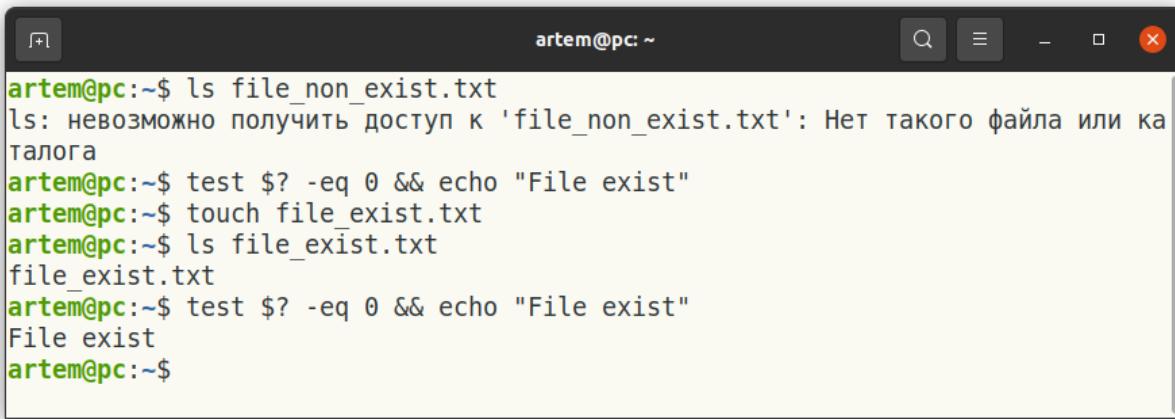
Есть еще один вариант использования \$, но на этот раз с фигурными скобками. Это называется parameter expansion и позволяет произвести операции над значениями переменной и передать результат дальше. Ниже пример со строкой `name=ArtemVesnin` и срезами, а также со списком `name=(Artem Vesnin)` и обращением к элементам списка.



```
artem@pc:~$ name=ArtemVesnin
artem@pc:~$ echo ${name:0:5}
Artem
artem@pc:~$ echo ${name:5:6}
Vesnin
artem@pc:~$ name=(Artem Vesnin)
artem@pc:~$ echo ${name[0]}
Artem
artem@pc:~$ echo ${name[1]}
Vesnin
artem@pc:~$
```

Также существует специальная переменная `$?`, куда записывается статус выполнения последней команды.

```
ls non_existent_file.txt
if test $? -ne 0 ; then
    echo "Error: File not found or other issue occurred."
fi
```

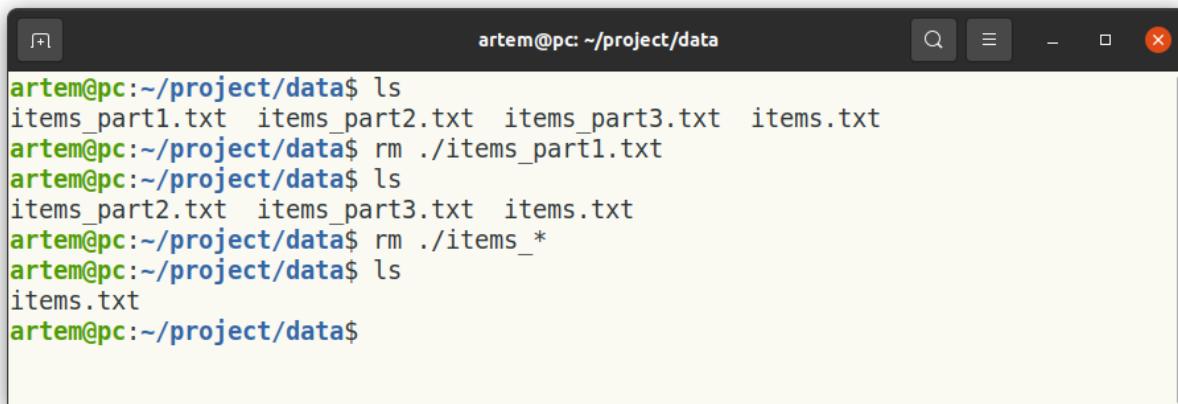


```
artem@pc:~$ ls file_non_exist.txt
ls: невозможно получить доступ к 'file_non_exist.txt': Нет такого файла или каталога
artem@pc:~$ test $? -eq 0 && echo "File exist"
artem@pc:~$ touch file_exist.txt
artem@pc:~$ ls file_exist.txt
file_exist.txt
artem@pc:~$ test $? -eq 0 && echo "File exist"
File exist
artem@pc:~$
```

Здесь мы использовали `test` для проведения операций сравнения. `-eq` сокращенно от `equal`.

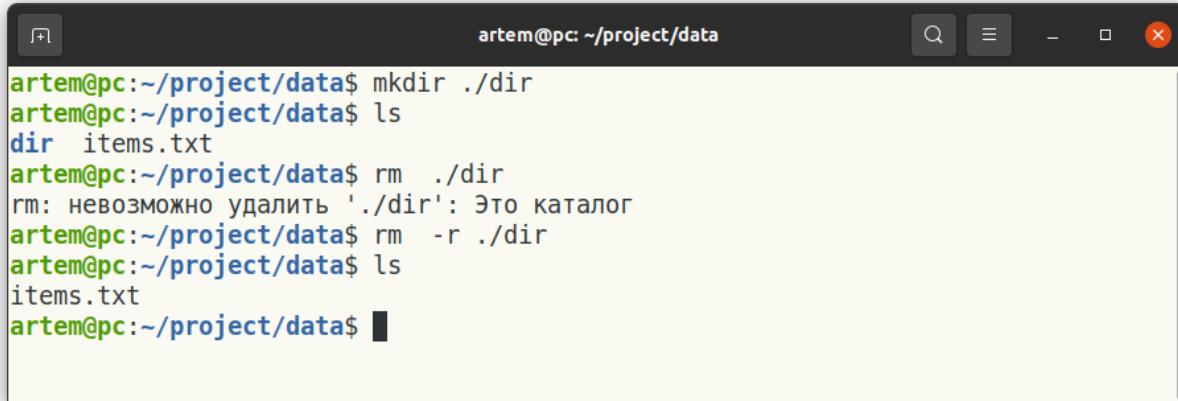
### 2.1.11 rm — remove

Для удаления файлов используется команда **rm**. Удалить можно один файл, указав путь до него, или несколько, указав ту часть пути, которая может меняться через \*, т.е. задав шаблон, в который должны вписаться пути.



```
artem@pc:~/project/data$ ls
items_part1.txt items_part2.txt items_part3.txt items.txt
artem@pc:~/project/data$ rm ./items_part1.txt
artem@pc:~/project/data$ ls
items_part2.txt items_part3.txt items.txt
artem@pc:~/project/data$ rm ./items_*
artem@pc:~/project/data$ ls
items.txt
artem@pc:~/project/data$
```

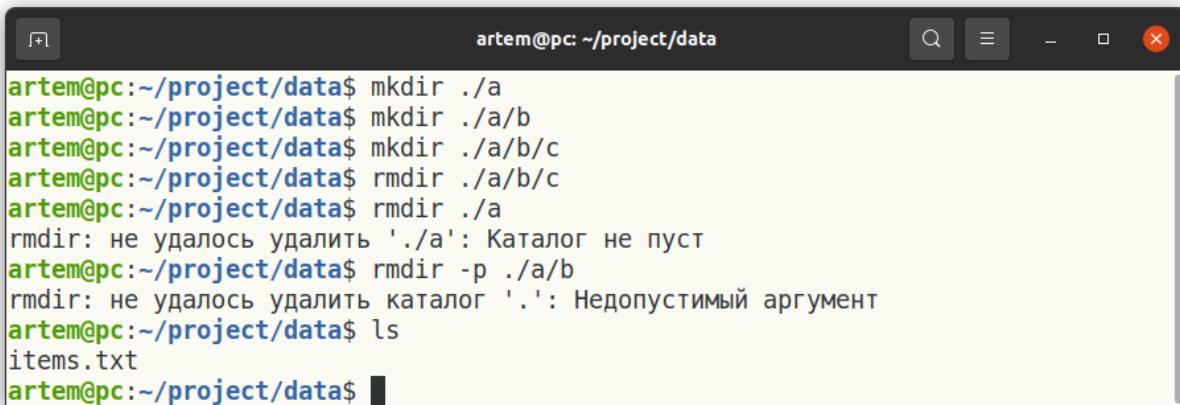
Удаление каталогов требует дополнительных опций, так как удаляются не отдельные файлы, а все содержимое каталога, поэтому нужно делать это рекурсивно, заходя во все вложенные каталоги.



```
artem@pc:~/project/data$ mkdir ./dir
artem@pc:~/project/data$ ls
dir items.txt
artem@pc:~/project/data$ rm ./dir
rm: невозможно удалить './dir': Это каталог
artem@pc:~/project/data$ rm -r ./dir
artem@pc:~/project/data$ ls
items.txt
artem@pc:~/project/data$
```

### 2.1.12 rmdir — remove directory

Удаление пустых директорий или группы вложенных пустых директорий.



```
artem@pc:~/project/data$ mkdir ./a
artem@pc:~/project/data$ mkdir ./a/b
artem@pc:~/project/data$ mkdir ./a/b/c
artem@pc:~/project/data$ rmdir ./a/b/c
artem@pc:~/project/data$ rmdir ./a
rmdir: не удалось удалить './a': Каталог не пуст
artem@pc:~/project/data$ rmdir -p ./a/b
rmdir: не удалось удалить каталог '.' Недопустимый аргумент
artem@pc:~/project/data$ ls
items.txt
artem@pc:~/project/data$
```

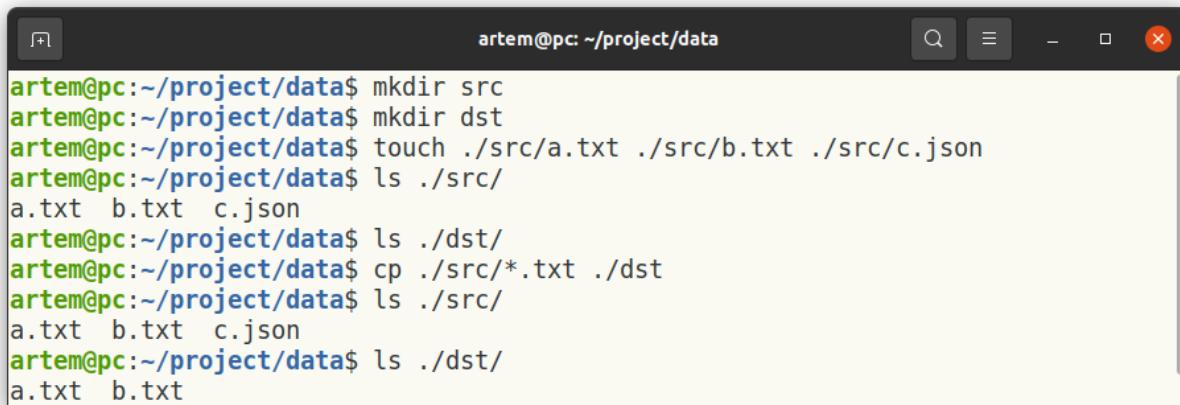
Обратите внимание на сообщение об ошибке (вообще всегда смотрите на результат ваших действий, чтобы понимать, выполнилась ли команда). Часть работы была сделана: удалены директории `./a/b` и `./a`. А самый верхний уровень этого пути `(.)` — это, как мы помним, текущая директория, т.е. `~/project/data`, и `rmdir` сам не дает пользователю удалить директорию, в которой он находится. Правильным будет `mkdir -p a/b`, хотя в большинстве случаев `./a/b` и `a/b` равнозначны.



```
artem@pc:~/project/data$ rmdir -p a/b
artem@pc:~/project/data$
```

### 2.1.13 cp — copy

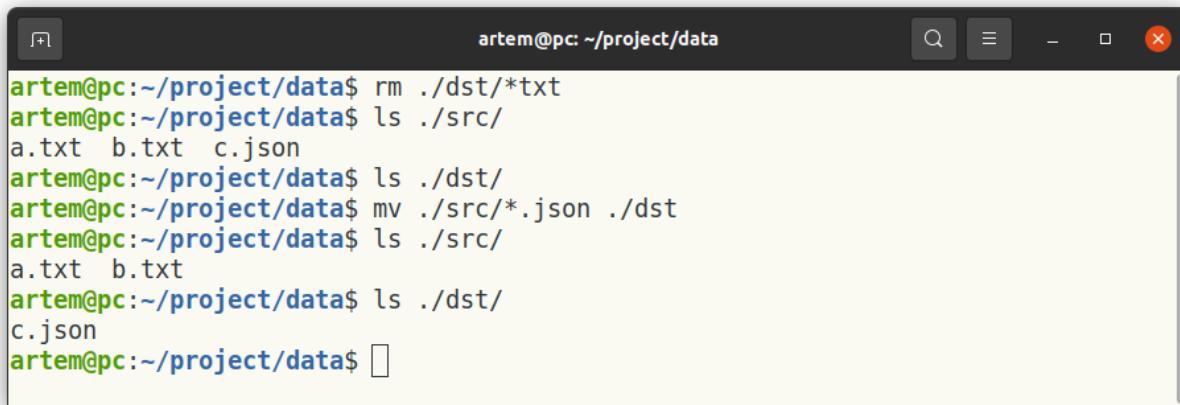
Копирует один или несколько файлов из одного места в другое. Так же, как и в случае с `rm`, можно указывать шаблон, которому должны соответствовать пути.



```
artem@pc:~/project/data$ mkdir src
artem@pc:~/project/data$ mkdir dst
artem@pc:~/project/data$ touch ./src/a.txt ./src/b.txt ./src/c.json
artem@pc:~/project/data$ ls ./src/
a.txt b.txt c.json
artem@pc:~/project/data$ ls ./dst/
artem@pc:~/project/data$ cp ./src/*.txt ./dst
artem@pc:~/project/data$ ls ./src/
a.txt b.txt c.json
artem@pc:~/project/data$ ls ./dst/
a.txt b.txt
```

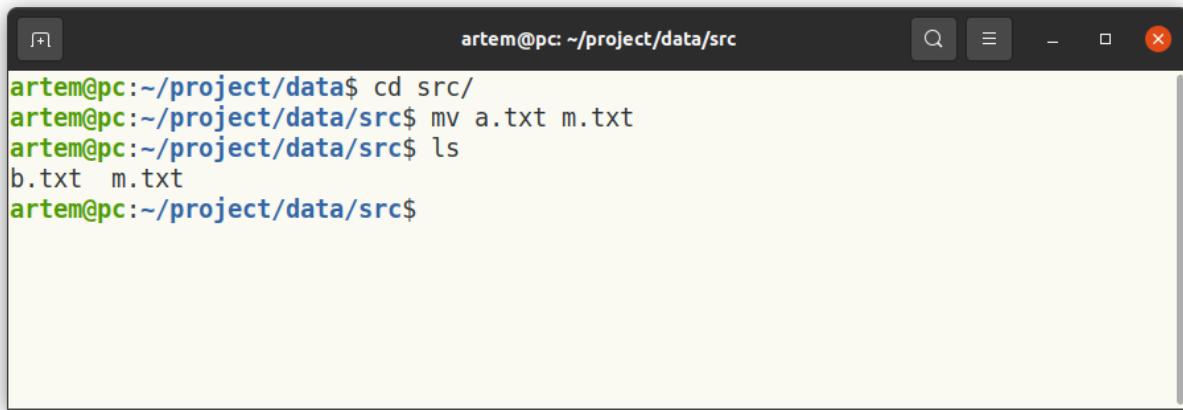
### 2.1.14 mv — move

Перемещает один или несколько файлов из одного места в другое. Так же, как и в случае с `rm`, можно указывать шаблон, которому должны соответствовать пути.



```
artem@pc:~/project/data$ rm ./dst/*txt
artem@pc:~/project/data$ ls ./src/
a.txt b.txt c.json
artem@pc:~/project/data$ ls ./dst/
artem@pc:~/project/data$ mv ./src/*.json ./dst
artem@pc:~/project/data$ ls ./src/
a.txt b.txt
artem@pc:~/project/data$ ls ./dst/
c.json
artem@pc:~/project/data$ 
```

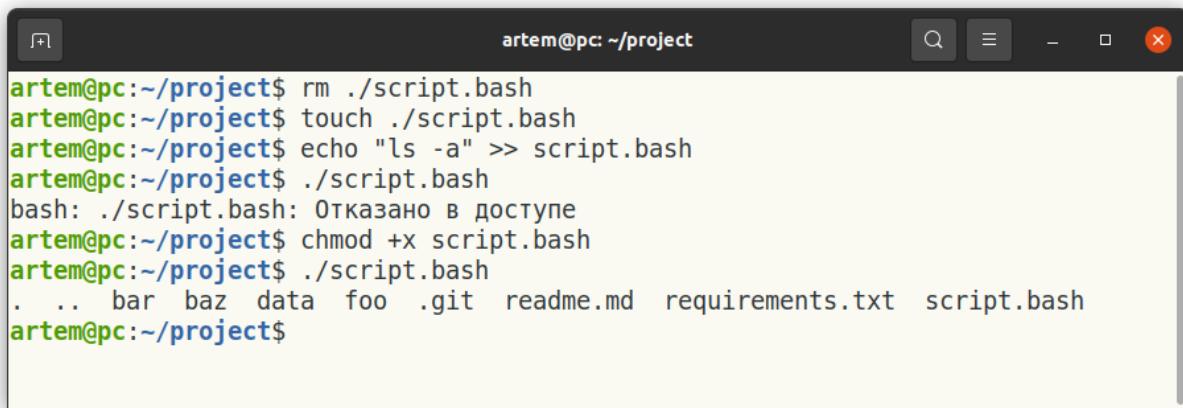
`mv` можно использовать для переименования файлов.



```
artem@pc:~/project/data$ cd src/
artem@pc:~/project/data/src$ mv a.txt m.txt
artem@pc:~/project/data/src$ ls
b.txt m.txt
artem@pc:~/project/data/src$
```

### 2.1.15 chmod — change mode

Изменяет разрешения файлов и директорий. Помните, на рисунке, который уже был выше, `script.bash` подсвечивался зеленым? Это означало, что у него есть разрешение на исполнение. Давайте удалим его. Создадим скрипт заново и запишем туда строку `ls -l` с помощью `echo`, попробуем выполнить, получим ошибку, потом попробуем дать разрешение на исполнение и выполним еще раз.



```
artem@pc:~/project$ rm ./script.bash
artem@pc:~/project$ touch ./script.bash
artem@pc:~/project$ echo "ls -a" >> script.bash
artem@pc:~/project$ ./script.bash
bash: ./script.bash: Отказано в доступе
artem@pc:~/project$ chmod +x script.bash
artem@pc:~/project$ ./script.bash
. . . bar baz data foo .git readme.md requirements.txt script.bash
artem@pc:~/project$
```

Разрешения делятся на три категории:

- `r` (*read*) — чтение;
- `w` (*write*) — запись;
- `x` (*execute*) — исполнение.

А также на три уровня:

1. Владелец файла или директории.
2. Группа пользователей, к которой принадлежит владелец.
3. Все остальные.

К примеру, права доступа `[drwxrwxrwx]` дают полный доступ на чтение, редактирование и запуск абсолютно всем пользователям. Рассмотрим этот случай подробнее. Ключи доступа из примера можно поделить на 4 секции: 1 — `[d]`, 2 — `[rwx]`, 3 — `[rwx]`, 4 — `[rwx]`.

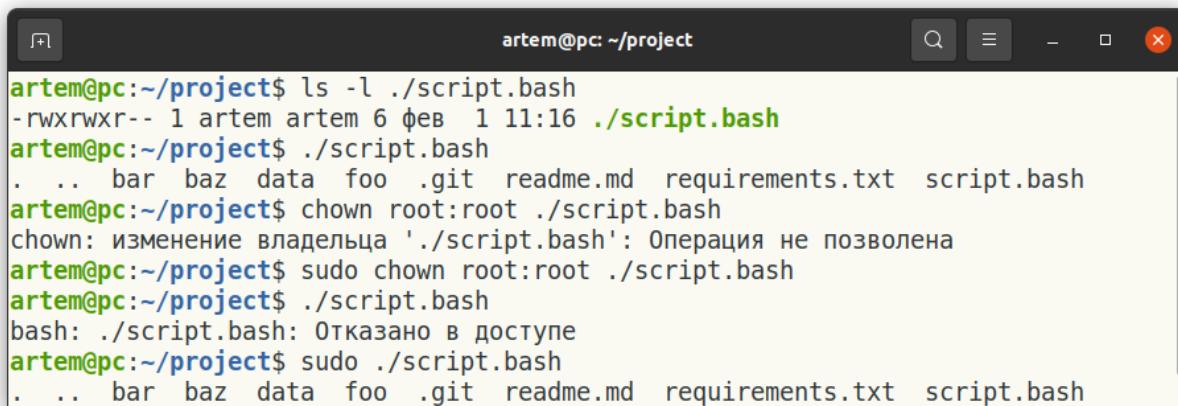
1 секция может быть либо «d» — директория, либо «-» — файл. 2 секция показывает права доступа владельца файла или директории (r — чтение, w — запись, x — выполнение). 3 секция показывает права доступа группы, в которой находится владелец, rwx — аналогично. 4 секция показывает права доступа всех остальных пользователей системы.

Например, **-rwxr-xr--** означает, что владелец может читать, писать и исполнять файл, пользователи из группы — только читать и исполнять, все остальные — только читать. Можно вообще не давать разрешений, например, **-rwxr-x---** запретит доступ к файлу всем пользователям, кроме его владельца (того, кто создал этот файл) и участникам группы, к которой он относится.

Все группы, созданные в системе, находятся в файле `/etc/group`. Посмотрев содержимое этого файла, вы увидите список групп `linux`, которые уже есть в вашей системе. Этот список может быть большим и малоинформативным для рядового пользователя. Чтобы вывести список всех групп, членом которых является текущий пользователь, можно использовать команду `groups`.

### 2.1.16 sudo — superuser do

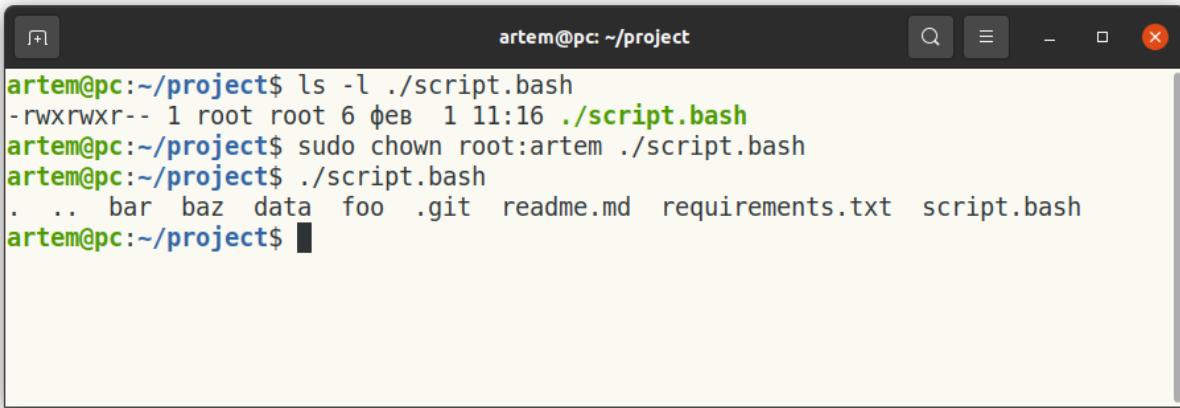
В большинстве случаев для выполнения команд описанных, выше достаточно обычных прав, которые имеются у пользователя. Но в ряде случаев, при использовании системных утилит или смене владельца у файла (см. `chown` ниже), необходимо использовать права суперпользователя. Первый способ — авторизоваться, используя аккаунт суперпользователя в системе. Но такой сценарий не является предпочтительным, так может привести к случайному выполнению команды, которая может нанести вред системе. Более широкое применение получил способ, при котором для выполнения команды с правами суперпользователя перед командой добавляется `sudo`. Ниже представлен пример передачи владения скрипта пользователю `root`, после этого мы сможем выполнять скрипт только от имени нового владельца, т.е. `root` (для текущих разрешений скрипта **-rwxrwxr--**).



```
artem@pc:~/project$ ls -l ./script.bash
-rwxrwxr-- 1 artem artem 6 фев 1 11:16 ./script.bash
artem@pc:~/project$ ./script.bash
. . . bar baz data foo .git readme.md requirements.txt script.bash
artem@pc:~/project$ chown root:root ./script.bash
chown: изменение владельца './script.bash': Операция не позволена
artem@pc:~/project$ sudo chown root:root ./script.bash
artem@pc:~/project$ ./script.bash
bash: ./script.bash: Отказано в доступе
artem@pc:~/project$ sudo ./script.bash
. . . bar baz data foo .git readme.md requirements.txt script.bash
```

### 2.1.17 chown — change owner

Изменяет владельца файла или директории. При этом, если вы хотите сделать это рекурсивно, для всех файлов внутри директории, нужно указать специальный флаг `-R` для рекурсивной работы. При использовании мы указываем имя пользователя и группу через двоеточие `username:group`. Каждый пользователь может состоять в одной или нескольких группах. Можно менять владельца и не менять группу. В примере ниже мы вернули владение группе `artem`, и теперь можем запускать файл от обычного пользователя (для текущих разрешений скрипта **-rwxrwxr--**).



```
artem@pc:~/project$ ls -l ./script.bash
-rwxrwxr-- 1 root root 6 фев 1 11:16 ./script.bash
artem@pc:~/project$ sudo chown root:artem ./script.bash
artem@pc:~/project$ ./script.bash
. . . bar baz data foo .git readme.md requirements.txt script.bash
artem@pc:~/project$
```

### 2.1.18 history

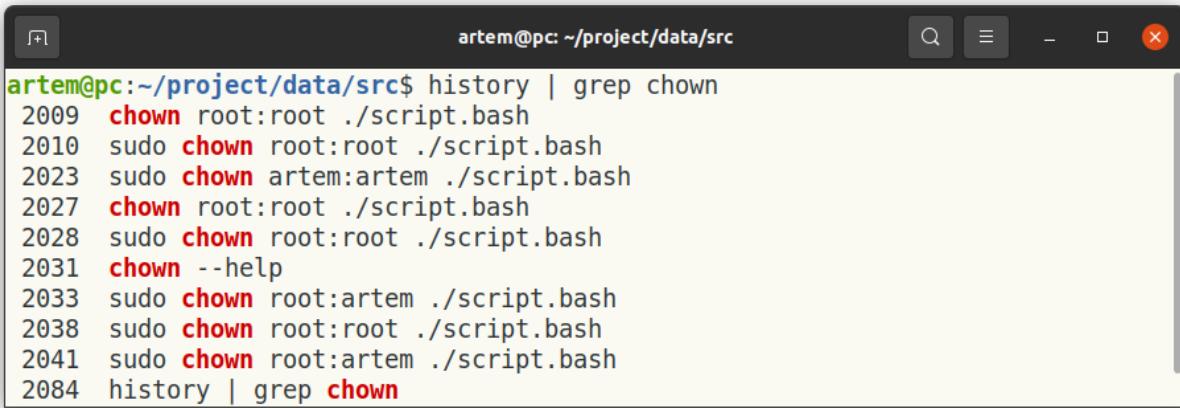
Выводит историю выполненных команд.



```
2073 cd src/
2074 cd ../
2075 clear
2076 cd src/
2077 mv a.txt m.txt
2078 ls
2079 clear
2080 hstory
2081 clear
2082 history
artem@pc:~/project/data/src$
```

### 2.1.19 grep — global regular expression print

Производит поиск в тексте строки, содержащей определенную подстроку. При этом текст можно получить прямо из стандартного вывода или из файла. Бывает полезным, чтобы вспомнить, как выполнялась та или иная команда в bash.



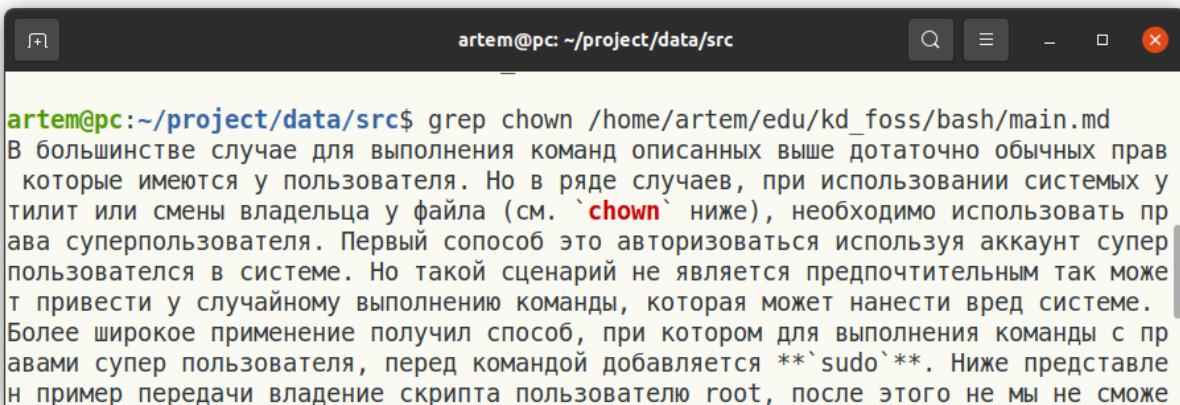
```
artem@pc:~/project/data/src$ history | grep chown
2009 chown root:root ./script.bash
2010 sudo chown root:root ./script.bash
2023 sudo chown artem:arTEM ./script.bash
2027 chown root:root ./script.bash
2028 sudo chown root:root ./script.bash
2031 chown -- help
2033 sudo chown root:arTEM ./script.bash
2038 sudo chown root:root ./script.bash
2041 sudo chown root:arTEM ./script.bash
2084 history | grep chown
```

Также можно облегчить себе поиск документации по `man`, если примерно помнить термин.



```
artem@pc:~/project/data/src$ man chown | grep recurs
      fail to operate recursively on '/'
      -R, --recursive
          operate on files and directories recursively
artem@pc:~/project/data/src$
```

Или найти что-нибудь в документе, который вы читаете:



```
artem@pc:~/project/data/src$ grep chown /home/arTEM/edu/kd_foss/bash/main.md
В большинстве случаев для выполнения команд описанных выше достаточно обычных прав
которые имеются у пользователя. Но в ряде случаев, при использовании системных утилит или смены владельца у файла (см. `chown` ниже), необходимо использовать права суперпользователя. Первый способ это авторизоваться используя аккаунт суперпользователя в системе. Но такой сценарий не является предпочтительным так может привести к случайному выполнению команды, которая может нанести вред системе.
Более широкое применение получил способ, при котором для выполнения команды с правами суперпользователя, перед командой добавляется **`sudo`**. Ниже представлен пример передачи владение скрипта пользователю root, после этого не сможем
```

**Hint:** | перенаправляет выходной поток от одной команды к входу другой. Можно использовать, чтобы миновать запись в файл на диске.

## 2.1.20 top — table of process

Выводит активные процессы, а также информацию по загрузке процессора и памяти. Информация будет обновляться в терминале. Чтобы выйти из этого режима, нажмите **Ctrl-C**.

```

top - 08:17:31 up 17:20,  1 user,  load average: 0,49, 0,48, 0,45
Tasks: 273 total,   1 running, 272 sleeping,   0 stopped,   0 zombie
%Cpu(s): 4,3 us, 2,9 sy, 0,0 ni, 92,8 id, 0,0 wa, 0,0 hi, 0,0 si, 0,0 st
Mem Mem : 7875,7 total, 1530,5 free, 3039,1 used, 3306,2 buff/cache
Mem Swap: 2048,0 total, 2048,0 free, 0,0 used. 3798,8 avail Mem

PID USER      PR  NI    VIRT    RES    SHR S %CPU %MEM TIME+ COMMAND
2936 artem    20   0 312088  9368  7316 S 6,2  0,1 1:35.09 ibus-da+
10824 artem    20   0 2727036 201136 108440 S 6,2  2,5 0:18.30 Isolate+
  1 root      20   0 170056 13340  8312 S 0,0  0,2 0:13.28 systemd
  2 root      20   0      0      0      0 S 0,0  0,0 0:00.01 kthreadd

```

**Hint:** htop дополняет функциональность top. Поставьте htop через менеджер пакетов, который используется в вашем дистрибутиве (`sudo apt install htop` в Ubuntu).

## 2.1.21 Практические примеры

Ниже мы рассмотрим несколько примеров использования `bash` и еще раз повторим написанное в мотивации к этому уроку: рассматривайте `bash` как дополнительный инструмент, который может облегчить решение некоторых задач. Ради тренировки можно написать, например, калькулятор, но его легко сделать на вашем любимом языке программирования гораздо быстрее, поэтому ниже мы рассмотрим примеры, которые используют доступ к системным утилитам и позволяют нам получить информацию о подключенных устройствах и их состоянии.

### Мониторинг подключения usb-устройств

Ниже приведены примеры скрипта, отслеживающего подключение и отключение usb-устройства (код):

```

#!/bin/bash

# Define the log file path
LOG_FILE="./usb-monitor.log"

# Get the initial list of connected USB devices
initial_devices=$(lsusb)

# Function to log the events
log_event() {
    local event_time=$(date +'%Y-%m-%d %H:%M:%S')
    local event_type="$1"
    local device_id="$2"

    echo "$event_time | $event_type | $device_id" >> "$LOG_FILE"
}


```

(continues on next page)

(продолжение с предыдущей страницы)

```

# Function to check for new devices
check_devices() {
    while true
    do
        current_devices=$(lsusb)
        echo "$initial_devices" | sort > "./idev.txt"
        echo "$current_devices" | sort > "./cdev.txt"
        new_devices=$(comm -13 "./idev.txt" "./cdev.txt")
        removed_devices=$(comm -13 "./cdev.txt" "./idev.txt")
        if test "$new_devices" != ""
        then
            for device in "$new_devices"; do
                log_event "connected" "$device"
            done
        fi
        if test "$removed_devices" != ""
        then
            for device in "$removed_devices"; do
                log_event "disconnected" "$device"
            done
        fi
        initial_devices=$current_devices
        sleep 1 # Adjust the update interval as needed
    done
}

# Start monitoring, just call check_devices()
check_devices

```

Мы видим ряд конструкций, которые доступны в обычном языке программирования, например, условные операторы, циклы, функции. Реализация их в `bash` имеет свои особенности, но понять их синтаксис несложно, если вы знаете хотя бы один язык программирования. Вызов функции также имеет особенности, например, мы передаем аргументы не в скобках, а после вызова функции. Особенности использования `$` мы рассмотрели выше.

В скрипте мы используем новую команду `comm`, которая сравнивает два файла и выводит результат в три колонки, вторая из которых — строки, уникальные для второго файла (что находится в других колонках, можно посмотреть в документации `man comm`). Аргументы `-13` удаляют колонки 1 и 3, оставляя только вторую. Поэтому мы меняем файлы местами, чтобы определить `new_devices` и `removed_devices`. Функция `log_event()` записывает информацию о новых или отключенных устройствах в файл. Обратите внимание на вызов `check_devices` в конце скрипта.

Мы проверяем устройства каждую секунду `sleep 1`, поэтому при быстром подключении и отключении можем потерять информацию о событии. Далее мы рассмотрим пример того, как этого избежать.

Также мы использовали команду `test`, которая производит сравнение аргументов с использованием логических операторов. Вы можете встретить `["$new_devices" != ""]` вместо `test "$new_devices" != ""`, это одно и то же.

## Мониторинг подключения устройств с помощью dmesg

Выше мы рассмотрели скрипт для мониторинга usb-устройств с помощью `lsusb`. В Линукс есть команда для мониторинга всех сообщений, которые появляются в системе. Мы можем настроить шаблоны, которые позволят нам отфильтровать нужные нам сообщения. Ниже приведен пример, которые позволит отловить все сообщения, где встречается подстрока «usb». При этом, в отличие от скрипта с `lsusb`, мы лучше защищены от потери события, так как `dmesg` выводит сообщения из буфера, который хранит не только последнее сообщение, но и историю сообщений (код):

```
#!/bin/bash
prev_records=$(dmesg | tail -20)

monitor_devices_records() {

    while true
    do
        curr_records=$(dmesg | tail -20)
        new_records=$(comm -13 <(echo "$prev_records") <(echo "$curr_records"))
        usb_records=$(echo "$new_records" | grep usb)
        if test "$usb_records" != ""
        then
            echo "$usb_records"
        fi
        prev_records=$curr_records
    done
}

# Start monitoring
monitor_devices_records
```

## Мониторинг оставшегося места на диске

Скрипт, приведенный ниже мониторит доступное на диске место и предупреждает, когда процент занятого места превышен. Конструкцию `${}{}` мы описывали выше. Из нового мы видим конструкцию `($df_record)`, которая позволяет создать список из значения переменной `df_record`, например, «one two three» → (`«one»`, `«two»`, `«three»`) (код):

```
#!/bin/bash
monitor_device=$1 #device to monitor eg. /dev/sdc1
alert_level=$2      #available percentage to alert eg. 1

monitor_space_usage() {

    while true
    do
        df_record=$(df | grep "$monitor_device")
        df_record_array=($df_record)
        avail=${df_record_array[3]}
        total=${df_record_array[1]}
        if test $($(( 100 * $avail / $total)) -le $alert_level
        then
            echo "Alert disk is almost full"
        fi
    done
}
```

(continues on next page)

(продолжение с предыдущей страницы)

```

    sleep 1
done
}

monitor_space_usage

```

## 2.2 Задачи

В описании задач дается название утилит, которые можно использовать для ее решения. Предполагается, что вы самостоятельно найдете правильные параметры, используя документацию `man имя_утилиты` или поиск в сети.

### 2.2.1 Задача 1

Напишите скрипт, который будет делать резервную копию файла. Резервный файл должен содержать дату создания в имени.

**Ответ**

(код):

```

#!/bin/bash
file_to_backup="myfile.txt"
backup_file="backup_$(date +'%Y%m%d%H%M%S').txt"
cp "$file_to_backup" "$backup_file"
echo "Backup created: $backup_file"

```

### 2.2.2 Задача 2

Напишите скрипт, который удалит файлы старше 30 дней в выбранной директории. Используйте утилиту `find` для поиска файлов. В пути до файлов, включая имя файлов, не должно быть пробелов и специальных символов. В этой задаче предполагается, что пути содержат только латинские буквы, цифры, тире и подчеркивания.

**Ответ**

(код):

```

#!/bin/bash

# Define the directory to search in and the days threshold
directory="/path/to/directory"
days_threshold=30

# Store the output of the find command in a variable
found_paths=$(find "$directory" -type f -mtime +$days_threshold)

# Iterate through the paths and remove each file
for path in $found_paths; do
    rm "$path"

```

(continues on next page)

(продолжение с предыдущей страницы)

```
echo "Deleted: $path"
done
```

### 2.2.3 Задача 3

Напишите скрипт, который будет мониторить использование памяти определенным процессом, например, вашей программой, и ежесекундно записывать в файл объем использованной памяти. Используйте утилиту `ps` для получения информации о процессе и `awk` для получения суммарного объема использованной памяти.

Скрипт `memory_monitor.sh` написать так, чтобы его можно было использовать, как показано ниже:

```
# Launch your program (replace "./my_program" with the actual command to run your
# program)
./my_program &

# Capture the PID of the launched program
program_pid=$!

# Start the monitoring script with the captured PID as an argument
./memory_monitor.sh "$program_pid"
```

#### Ответ

(код):

```
#!/bin/bash

# File to store the memory usage data
output_file="memory_usage_log.txt"

# Header for the output file (if it doesn't exist)
if [ ! -f "$output_file" ]; then
    echo "Datetime Memory_Usage(KB)" > "$output_file"
fi

# Function to get memory usage of the process by PID
get_memory_usage() {
    local pid=$1
    local memory_usage=$(ps -o rss= -p "$pid" | awk '{ sum+=$1 } END { print sum }')
    echo "$memory_usage"
}

# Check if a process ID was provided as an argument
if test -z "$1" ; then
    echo "Usage: $0 <process_id>"
    exit 1
fi

# Main loop to monitor and log memory usage
while true; do
    datetime=$(date +"%Y-%m-%d %H:%M:%S")
    usage=$(get_memory_usage $1)
    echo "$datetime $usage" >> "$output_file"
    sleep 1
done
```

(continues on next page)

(продолжение с предыдущей страницы)

```

pid="$1"
memory_usage=$(get_memory_usage "$pid")

# Append data to the output file
echo "$datetime $memory_usage" >> "$output_file"

# Wait for 1 second before the next iteration
sleep 1
done

```

#### 2.2.4 Задача 4

Напишите скрипт, который будет проверять доступность сайта. Это может пригодиться во многих приложениях, когда происходит отправка данных или работоспособность одного приложения зависит от другого, расположенного на другом сервере. Используйте утилиту curl, которая позволяет делать запросы.

##### Ответ

(код):

```

#!/bin/bash
website="https://example.com"
response=$(curl -Is "$website" | head -n 1)
if [[ "$response" == *"200 OK"* ]]; then
    echo "Website is reachable."
else
    echo "Website is down or unreachable."
fi

```

#### 2.2.5 Задача 5

Напишите скрипт для мониторинга портов, открытых на удаленной машине. Для этого используйте утилиту nc.

##### Ответ

(код):

```

#!/bin/bash

# Remote website to check (replace example.com and 80 with the appropriate values)
remote_host="example.com"
port="80"

# Function to check if the port is open on the remote site
check_port_open() {
    local host=$1
    local port=$2
    nc -z -w5 "$host" "$port" >/dev/null 2>&1
    if [ $? -eq 0 ]; then

```

(continues on next page)

(продолжение с предыдущей страницы)

```

    echo "Port $port is open on $host"
else
    echo "Port $port is closed or unreachable on $host"
fi
}

# Call the function to check the port
check_port_open "$remote_host" "$port"

```

## 2.3 Вопросы

1. Что означает `#!/bin/bash` в начале bash-скрипта?
2. Как вы объявляете и используете переменные в bash?
3. Какова цель оператора `if` в bash?
4. Как написать цикл `for` в bash для перебора списка элементов?
5. Каково значение `$0`, `$1`, `$2` и т. д. в bash-скрипте?
6. Как вы выполняете арифметические операции в bash?
7. Как использовать условные операторы (например, `if`, `elif`, `else`) в bash?
8. Как вы перенаправляете ввод и вывод в bash?
9. В чем разница между `>` и `>>`?
10. Что такое подстановка команд и как вы их используете в bash? **Ответ:** способ записать вывод команды в переменную, например: `current_date=$(date +"%Y-%m-%d")` или `current_date=$(date +"%Y-%m-%d")`.
11. Как вы используете grep для обработки текста в bash?
12. Что такое функции в bash и как их определить и вызвать?
13. Как вы обрабатываете аргументы командной строки в сценарии bash?
14. Как вы можете проверить, существует ли файл или каталог в bash?
15. Как вы создаете и используете массивы в bash?
16. Как запустить сценарий bash в фоновом режиме и отключить его от терминала?



## Система контроля версий Git

---

### 3.1 Git

#### 3.1.1 Система контроля версий Git

Трудно определить точное количество людей, использующих Git. Тем не менее, Git широко применяется разработчиками и организациями по всему миру и считается самой популярной системой контроля версий.



**Git** — это система управления версиями (*Version Control System, VCS*) с распределенным контролем состояния репозитория. Это значит, что каждый разработчик имеет полную копию репозитория (директория, содержащая все файлы проекта и историю изменений этих файлов) на локальном компьютере. Это позволяет команде разработчиков работать с кодом проекта одновременно и без конфликтов.

### 3.1.2 Краткий исторический экскурс

Git — это система контроля версий, созданная Линусом Торвальдсом в 2005 году. Она была разработана как средство для обслуживания ядра Linux, но в настоящее время широко используется в различных проектах и доменах. Основная цель Git — предоставить быстрый и надежный способ хранения и контроля версий кода, чтобы разработчики могли легко отслеживать изменения, откатывать их и работать в команде.

Согласно отчету GitHub Octoverse за 2019 год, на GitHub было более 40 миллионов пользователей, и большинство из них используют Git в качестве своей системы контроля версий. Кроме того, в отчете говорится, что на GitHub было более 100 миллионов репозиториев, и только в 2019 году в эти репозитории было внесено более 2,1 миллиарда материалов.

Помимо **GitHub**, **Git** также используется на других платформах, таких как **GitLab** и **Bitbucket**, а также разработчиками, которые размещают свои собственные репозитории **Git**. Поэтому можно с уверенностью сказать, что миллионы разработчиков используют Git для контроля версий и совместной работы. В целом, использование **Git** постоянно растет, поскольку он широко применяется разработчиками, различными организациями и корпорациями по всему миру, став негласным стандартом.

Выполним несколько простых действий, чтобы начать работать с этим крутым инструментом. Для начала нужно определиться с целью: поскольку **Git** — это прежде всего инструмент для удобной и эффективной работы команды разработчиков с кодом, мы сформулируем следующую задачу:

*Мы команда IT стартапа. У нас есть исходный код нашего продукта. Мы планируем его развивать, а также добавлять новый функционал, расширяя его возможности. Также мы планируем постепенно подключать к работе новых разработчиков. Делиться кодом в соцсетях, как и с помощью облачных сервисов, ненадежно и опасно как для нашего проекта, так и для пользователей нашего продукта. В таком случае всегда есть риск «порчи кода», кражи интеллектуальной собственности, роста ошибок и уязвимостей в коде из-за отсутствия единой системы контроля версий. Мы будем использовать тестовый код, представленный ниже.*

### 3.1.3 Приступаем к использованию Git

Скачаем приложение **Git** с официального сайта: <https://git-scm.com> и установим его на компьютер.

 **git** --local-branching-on-the-cheap

Git is a [free and open source](#) distributed version control system designed to handle everything from small to very large projects with speed and efficiency.

Git is [easy to learn](#) and has a [tiny footprint](#) with [lightning fast performance](#). It outclasses SCM tools like Subversion, CVS, Perforce, and ClearCase with features like [cheap local branching](#), [convenient staging areas](#), and [multiple workflows](#).



**About**  
The advantages of Git compared to other source control systems.

**Documentation**  
Command reference pages, Pro Git book content, videos and other material.

**Downloads**  
GUI clients and binary releases for all major platforms.

**Community**  
Get involved! Bug reporting, mailing list, chat, development and more.

**Pro Git** by Scott Chacon and Ben Straub is available to read online for free. Dead tree versions are available on [Amazon.com](#).

Latest source Release  
**2.39.1**  
Release Notes (2022-12-13)

**Download for Windows**











**Companies & Projects Using Git**

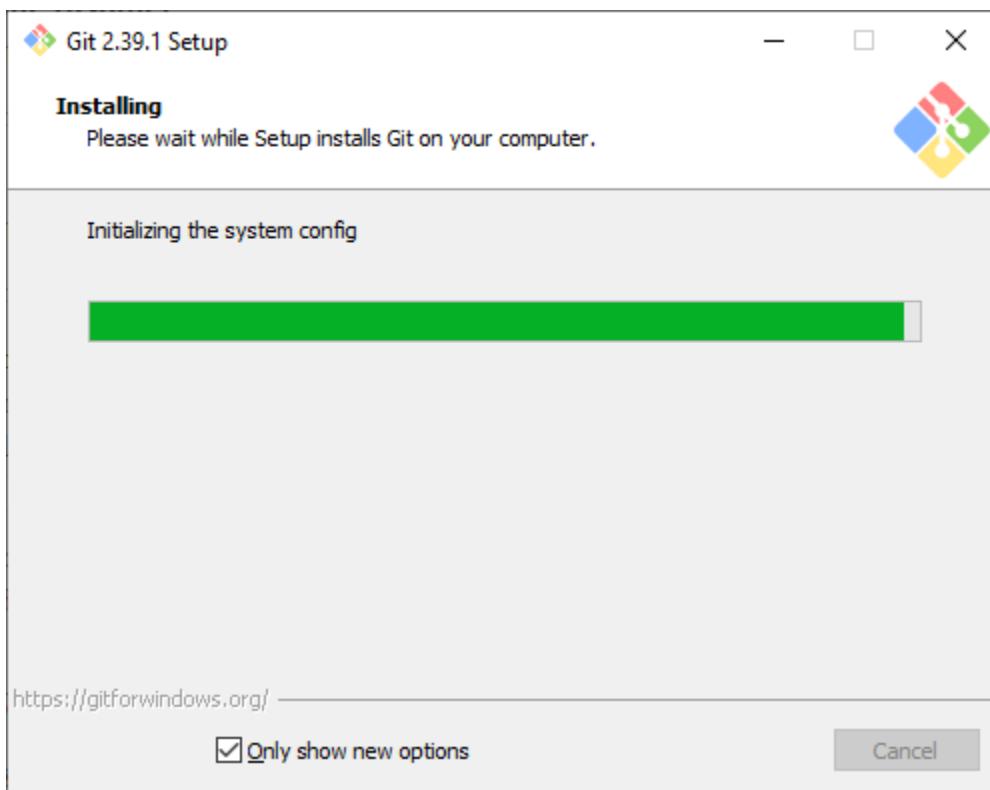
Google Microsoft LinkedIn NETFLIX PostgreSQL Android

В случае, если вы используете ОС Windows, выберите подходящий вариант, соответствующий разрядности вашей ОС (x64, x32).

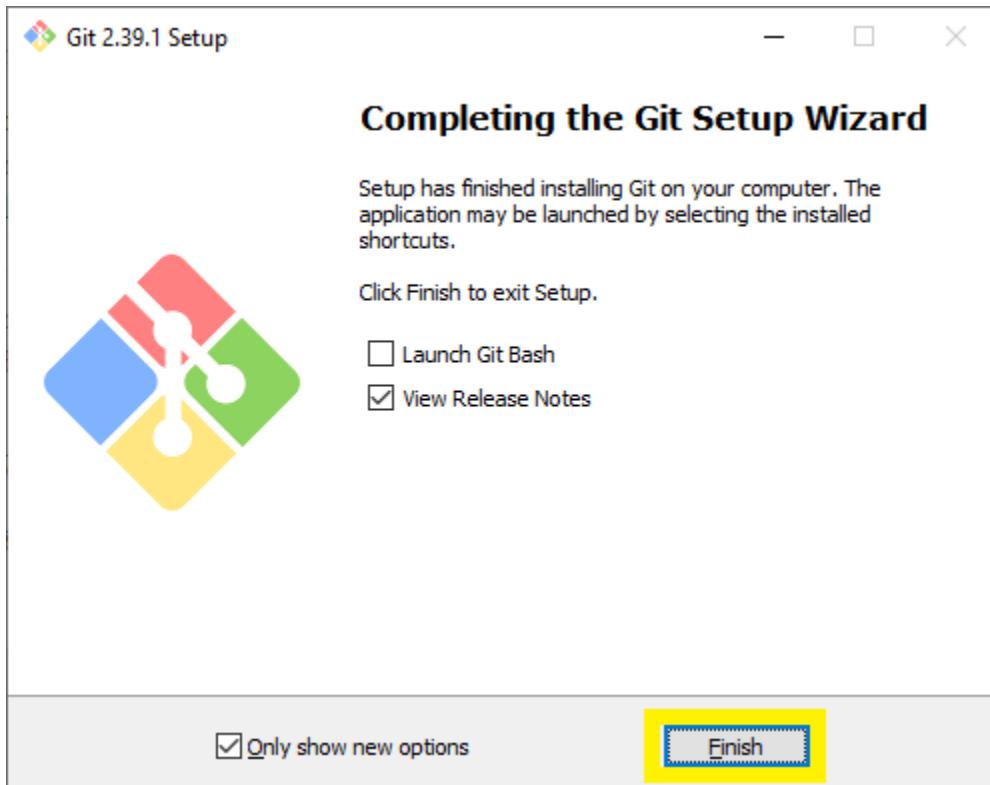
The screenshot shows the official Git website at [git-scm.com](https://git-scm.com/). The main navigation menu includes 'About', 'Documentation', 'Downloads' (selected), 'Community', and a sidebar with links to 'GUI Clients' and 'Logos'. A callout box highlights the 'Downloads' section. The central content area is titled 'Download for Windows' and features a prominent yellow button with the text 'Click here to download the latest (2.39.1) 64-bit version of Git for Windows. This is the most recent maintained build. It was released 7 days ago, on 2023-01-17.' Below this are links for 'Standalone Installer', '32-bit Git for Windows Setup.', '64-bit Git for Windows Setup.', 'Portable ("thumbdrive edition")', '32-bit Git for Windows Portable.', and '64-bit Git for Windows Portable.'. A section for 'Using winget tool' includes a command line snippet: `winget install --id Git.Git -e --source winget`.

Запустите установочный файл и следуйте инструкциям.



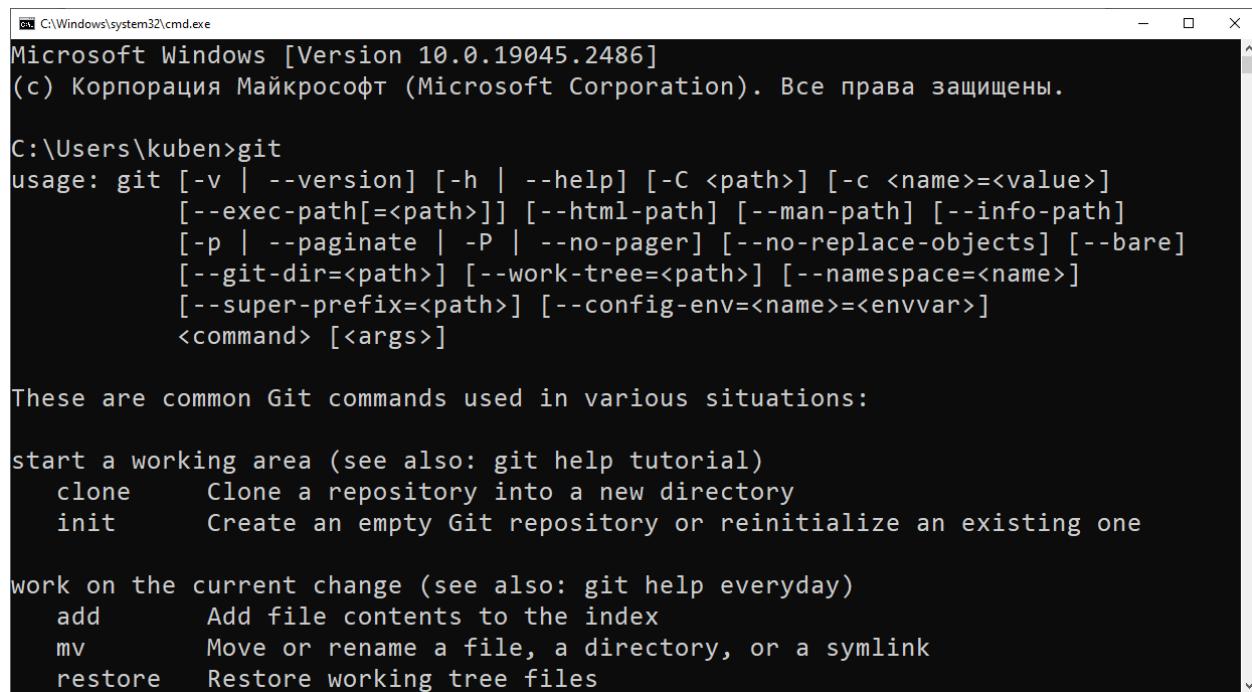


В результате у вас должно отобразиться следующее окно, сообщающее об успешной установке Git на ваш компьютер.



После завершения установки убедимся в том, что *Git* корректно установился на компьютер и мы

можем приступать к работе. Для этого откроем командную строку в Windows и введем команду **git**, как это показано на рисунке 2.



```
C:\Windows\system32\cmd.exe
Microsoft Windows [Version 10.0.19045.2486]
(c) Корпорация Майкрософт (Microsoft Corporation). Все права защищены.

C:\Users\kuben>git
usage: git [-v | --version] [-h | --help] [-C <path>] [-c <name>=<value>]
           [--exec-path[=<path>]] [--html-path] [--man-path] [--info-path]
           [-p | --paginate | -P | --no-pager] [--no-replace-objects] [--bare]
           [--git-dir=<path>] [--work-tree=<path>] [--namespace=<name>]
           [--super-prefix=<path>] [--config-env=<name>=<envvar>]
           <command> [<args>]

These are common Git commands used in various situations:

start a working area (see also: git help tutorial)
  clone      Clone a repository into a new directory
  init       Create an empty Git repository or reinitialize an existing one

work on the current change (see also: git help everyday)
  add        Add file contents to the index
  mv         Move or rename a file, a directory, or a symlink
  restore    Restore working tree files
```

Рисунок 2 — Командная строка Windows с введенной командой **git**

В окне консоли должна отобразиться информация о командах **Git**, как показано на рисунке 2.

Если у вас возникли проблемы, попробуйте удалить **git** из системы и повторить процесс установки, внимательно следя инструкциям и указаниям программы-установщика **git**.

### 3.1.4 Тестовый код программы

В качестве примера будем использовать код тестовой программы, разработанной на языке Python, для наглядной работы с системой контроля версий **git**. Тестовая программа выполняет запрос на сервис погоды *OpenWeatherMap*, используя обращение к сервису по API, затем полученный результат записывается в базу данных **SQLite**.

```
import requests
import sqlite3
# Connect to the SQLite database
conn = sqlite3.connect("example.db")
cursor = conn.cursor()
# Create a table to store the data
cursor.execute("""
CREATE TABLE IF NOT EXISTS weather (
id INTEGER PRIMARY KEY,
city TEXT,
temperature REAL
)
""")
# Make a request to the API
url = """https://api.openweathermap.org/data/2.5/weather?
```

(continues on next page)

(продолжение с предыдущей страницы)

```

q=London&appid=your_api_key"""

response = requests.get(url)
data = response.json()
# Extract the relevant data
city = data["name"]
temperature = data["main"]["temp"]
# Insert the data into the table
cursor.execute("INSERT INTO weather (city, temperature) VALUES (?, ?)",
               (city, temperature))
# Commit the changes to the database
conn.commit()
# Close the connection to the database
conn.close()

```

### 3.1.5 Описание тестовой программы

Эта программа сначала подключается к базе данных SQLite с именем *example.db* и создает таблицу с именем *weather*, если она еще не существует. Затем она делает запрос к общедоступному API (в данном случае API OpenWeatherMap), используя библиотеку запросов *requests*. Ответ от API анализируется как JSON, и из него извлекаются необходимые данные. Наконец, данные вставляются в таблицу *weather* в базе данных SQLite с помощью оператора вставки SQL, и изменения фиксируются.

Обратите внимание, что вам нужно заменить *your\_api\_key* на ваш собственный ключ API из *OpenWeatherMap* API, иначе данный код не будет работать. Кроме того, на вашем компьютере должна быть установлена библиотека *sqlite3*, она используется для работы с базой данных SQLite.

**Важно иметь в виду, что этот код является примером, и в нем отсутствуют многие важные детали, которые должны учитываться перед выпуском проекта в виде готового продукта.** Например, обработка ошибок и исключительных ситуаций, таких как сбои соединения, ошибки API и возможные ошибки выполнения SQL-запроса к СУБД. Также данный код не учитывает должным образом вопросы безопасности. Используя данный тестовый код, разработчики должны понимать, какие недостатки в нем присутствуют.

### 3.1.6 Оформление рабочего каталога

Для удобства работы с файлами проекта создадим директорию с названием *my\_project*, например, на диске D:\. Можно выбрать другое расположение папки с проектом. В качестве примера далее будем считать, что путь к файлам проекта выглядит следующим образом: «D:\my\_project».

Перейдем в директорию с нашим проектом (*my\_project*) и вызовем командную строку, как показано на рисунке 3. Для этого нужно нажать на строку, содержащую путь к текущей папке, ввести команду *cmd* и нажать на клавиатуре кнопку *Enter*.

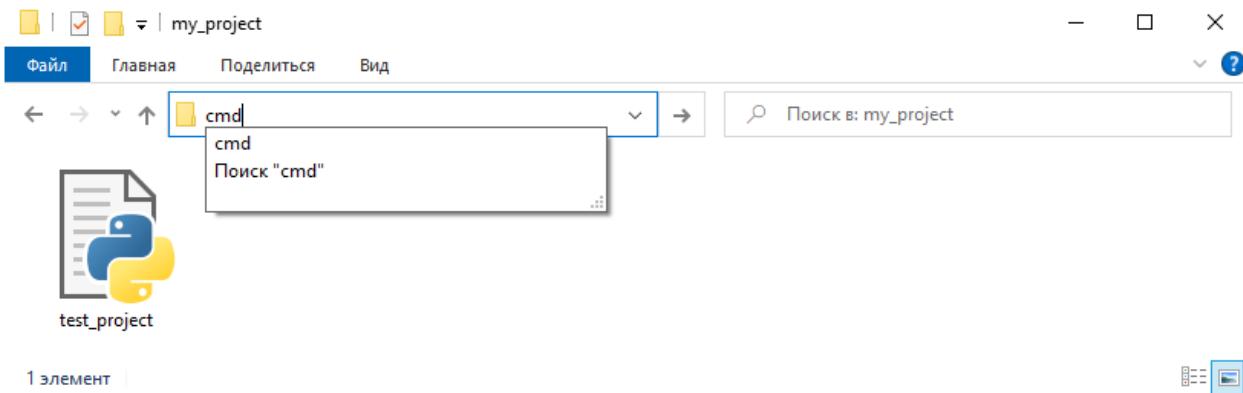


Рисунок 3 — Вызов командной строки в текущей папке

После выполнения данной команды должно отобразиться окно консоли (рисунок 4). В этом окне мы продолжим работу с Git.

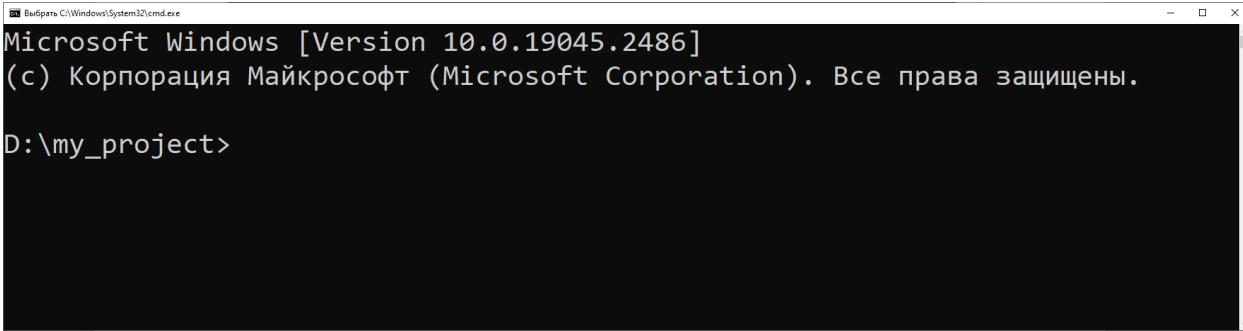


Рисунок 4 — Командная строка в текущей папке проекта

### 3.1.7 Настройка конфигурации git

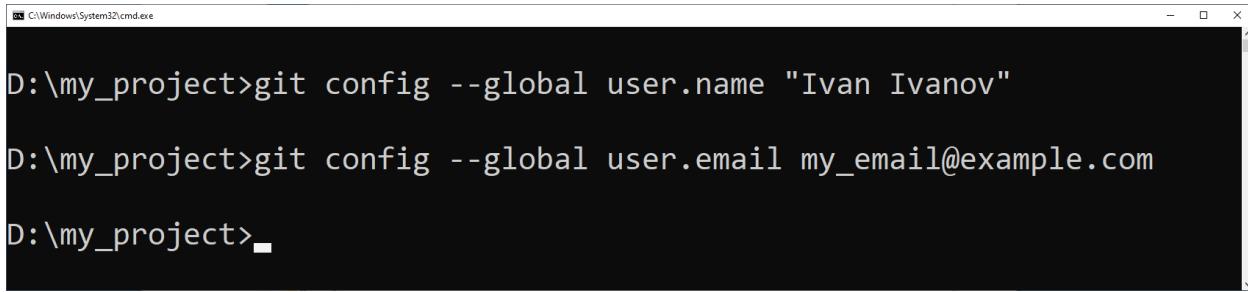
Перед тем как приступить к работе с кодом, нужно выполнить краткую настройку Git, а именно — указать имя пользователя и почтовый адрес. Эта информация используется системой Git для отслеживания операций всех пользователей проекта в репозитории. Это нужно сделать только один раз, при работе с другим проектом вам не придется повторно указывать имя пользователя и почтовый адрес.

Команда git	Описание
git config --global user.name «Ivan Ivanov»	Изменяем имя пользователя в git
git config --global user.email my_email@example.com	Изменяем название почты пользователя

Ниже представлен пример выполнения данных команд в консоли Git:

```
git config --global user.name "Ivan Ivanov"
git config --global user.email my_email@example.com
```

Репозиторий — это каталог, в котором располагаются все файлы одного проекта. В нашем случае у нас есть каталог «**my\_project**», его мы и будем использовать. В итоге у вас должно получиться так же, как показано на рисунке 5.



```
D:\my_project>git config --global user.name "Ivan Ivanov"
D:\my_project>git config --global user.email my_email@example.com
D:\my_project>
```

Если вы хотите проверить используемую конфигурацию, можете применить команду `git config --list`, чтобы показать все настройки Git текущего репозитория:

```
user.name= Ivan Ivanov
user.email= my_email@example.com
color.status=auto
color.branch=auto
color.interactive=auto
color.diff=auto
```

Также вы можете проверить значение конкретного ключа, выполнив `git config <key>`:

```
git config user.name
Ivan Ivanov
```

Если вам нужна помощь при использовании Git, есть три способа открыть страницу руководства по любой команде:

```
git help <команда>
git <команда> --help
```

Например, так можно открыть руководство по команде `git config`:

```
git help config
```

После ввода имени пользователя и адреса электронной почты можно закончить с настройкой и перейти к знакомству с основными командами Git.

Для базового варианта использования Git достаточно знать всего несколько команд для ведения истории изменений.

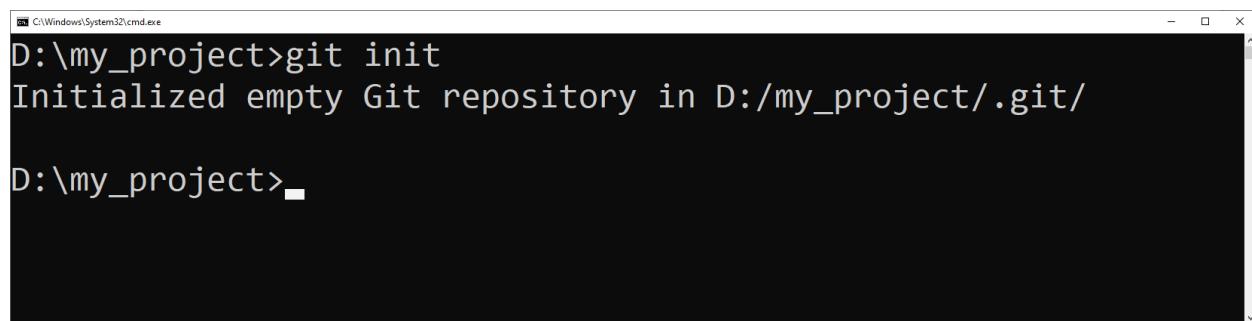
- Команда `git init` создает в текущем каталоге новый подкаталог с именем `.git`, содержащий все необходимые файлы и структуру git репозитория. На этом этапе ваш проект еще не находится под версионным контролем.
- Команда `git add` добавляет содержимое рабочего каталога в индекс (staging area) для последующего коммита: `git add <название файла_1, название файла_2>`. По умолчанию `git commit` использует лишь этот индекс, так что вы можете использовать `git add` для сборки слепка вашего следующего коммита. Это одна из ключевых команд Git.
- Команда `git status` показывает состояния файлов в рабочем каталоге и индексе: какие файлы изменены, но не добавлены в индекс; какие ожидают коммита в индексе. В добавок к этому выводятся подсказки о том, как изменить состояние файлов.
- Команда `git commit` берет все данные, добавленные в индекс с помощью `git add`, и сохраняет их слепок во внутренней базе данных, а затем сдвигает указатель текущей ветки на этот слепок.

- Команда `git push` используется для установления связи с удаленным репозиторием, определения локальных изменений, отсутствующих в текущей версии кода, и собственно их передачи на удаленный репозиторий, например, *GitHub* или *Bitbucket*.
- Команда `git pull` работает как комбинация команд `git fetch` и `git merge`, т.е. Git вначале забирает изменения из указанного удаленного репозитория, а затем пытается объединить их (выполнить слияние) с текущим кодом (веткой).

### 3.1.8 Первый пример работы с git

Отлично! Теперь, когда мы настроили параметры нашего `git` и познакомились с основными командами, сделаем следующее:

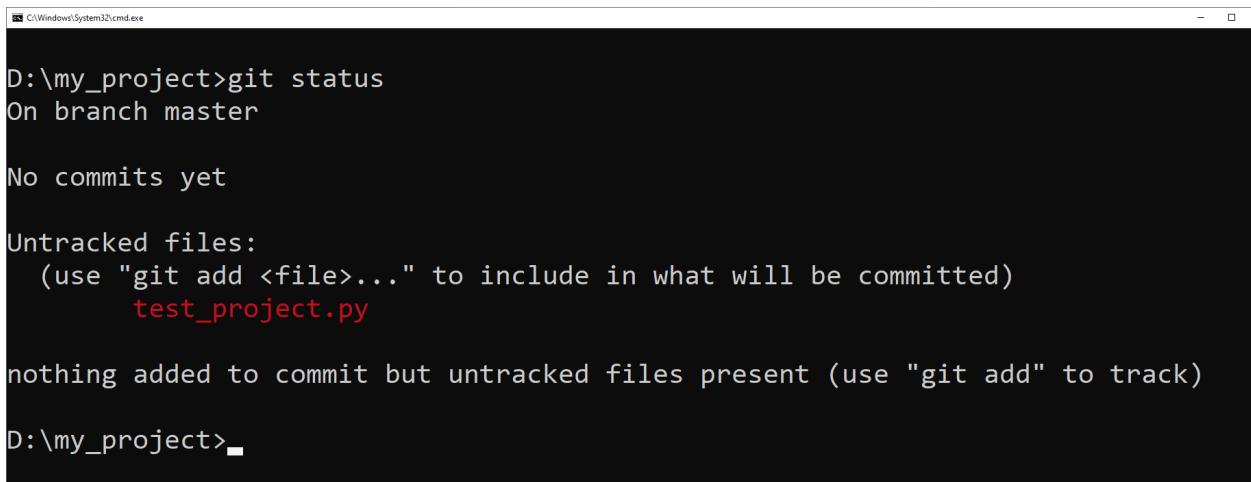
1. Выполним инициализацию репозитория в папке с проектом.
  2. Проверим статус нашего репозитория.
  3. Добавим файл с тестовым кодом `test_project.py` для отслеживания его изменений в `git`.
  4. Создадим первый коммит и посмотрим, как изменится статус `git`.
- **Шаг 1: Инициализация нашего рабочего каталога (репозитория)** Когда вы инициализируете репозиторий командой `git init`, Git создает ветку с именем `master` по умолчанию. В результате мы настроили рабочее окружение Git и выполнили инициализацию репозитория (рисунок 6).



```
D:\my_project>git init
Initialized empty Git repository in D:/my_project/.git/
D:\my_project>
```

Рисунок 6 — Вызов команды `git init`

- **Шаг 2: Проверка статуса репозитория** После того как был создан репозиторий, мы можем отслеживать все изменения (статус) в рабочей папке нашего проекта, вызвав команду `git status`. Что это значит? Дело в том, что после выполнения команды `git init` все изменения в директории `my_project` будут контролироваться (отслеживаться) системой управления версиями Git. В директории `my_project` есть файл `test_project.py` с тестовым кодом на языке Python. Давайте выполним команду `git status`, как показано на рисунке 7.



```
D:\my_project>git status
On branch master

No commits yet

Untracked files:
  (use "git add <file>..." to include in what will be committed)
    test_project.py

nothing added to commit but untracked files present (use "git add" to track)

D:\my_project>
```

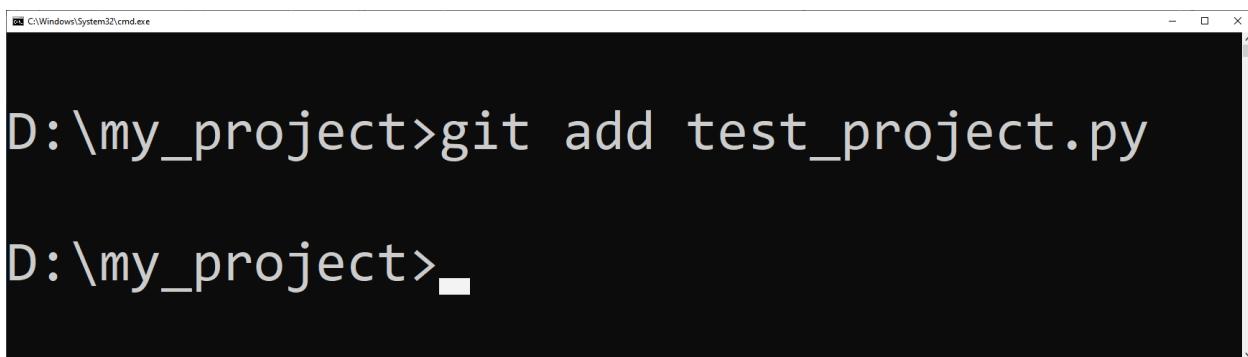
Рисунок 7 — Вызов команды `git status`

В результате работы команды `git status` мы видим несколько сообщений. Давайте разберемся, что они означают.

Сообщение Git	Описание
On branch master	Сообщение, в котором содержится информация о том, что мы находимся в ветке master.
No commits yet	В текущий момент нет коммитов, готовых для их фиксации, внесения каких-либо подготовленных изменений в проект.
Untracked files: use «git add ...» to include in what will be committed test_project.py	Найден файл «test_project.py», который в данный момент находится в папке проекта, но его состояние пока не контролируется системой git. И дается справка о том, что для добавления этого файла нужно использовать команду <code>git add</code> .
nothing added to commit but untracked files present (use «git add» to track)	Отсутствуют коммиты, но присутствуют неотслеживаемые файлы.

С этого места давайте поподробней. Команда `git status` вывела нам подробную информацию о том, в каком состоянии в данный момент находится наш проект с точки зрения системы контроля версий Git. В текущей директории есть файл `test_project.py` с тестовым кодом, и пока что изменения этого файла никак не отслеживаются. Поскольку мы планируем работать с этим кодом и хотим знать, какие изменения были внесены нами, либо кем-то из разработчиков нашей команды, нам нужно добавить его под контроль Git.

- **Шаг 3: Добавление файла для отслеживания изменений git** Чтобы добавить файл для отслеживания его изменений, выполним команду `git add test_project.py`, как показано на рисунке 8.

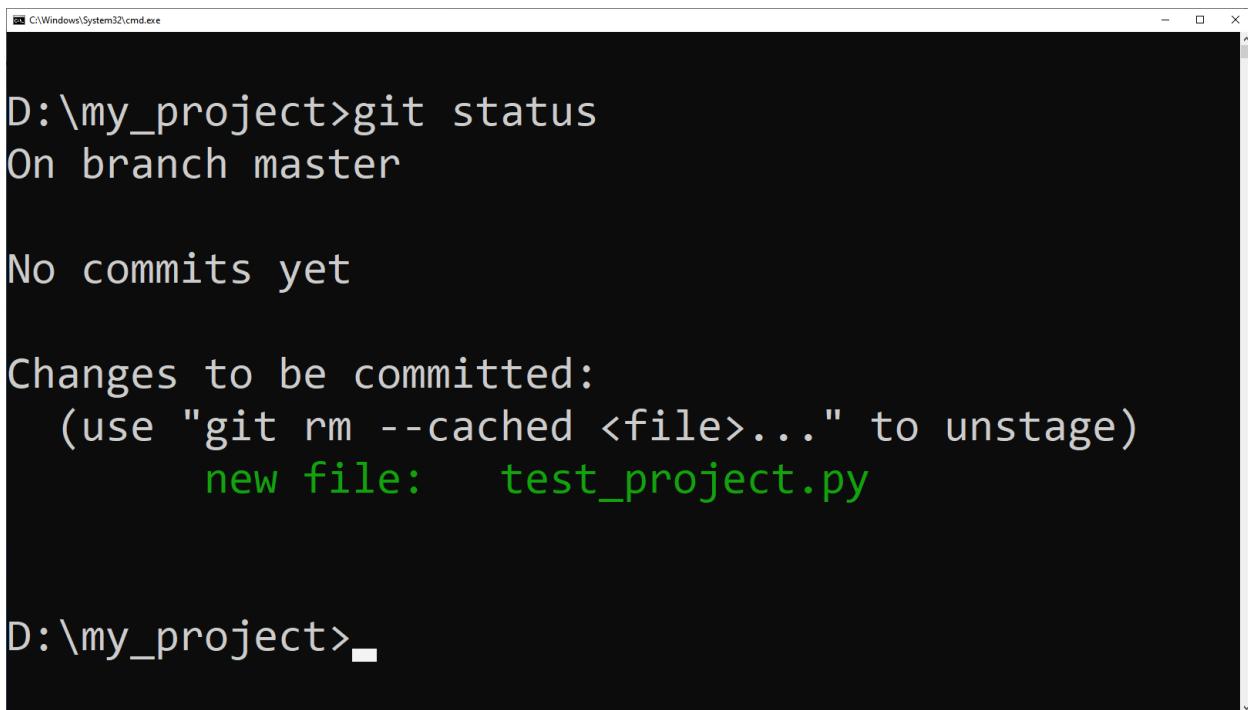


```
D:\my_project>git add test_project.py
```

```
D:\my_project>
```

Рисунок 8 — Вызов команды `git add`

Выполним повторно вызов команды `«git status»` и посмотрим, что изменилось (рисунок 9).



```
D:\my_project>git status
```

```
On branch master
```

```
No commits yet
```

```
Changes to be committed:
```

```
(use "git rm --cached <file>..." to unstage)
```

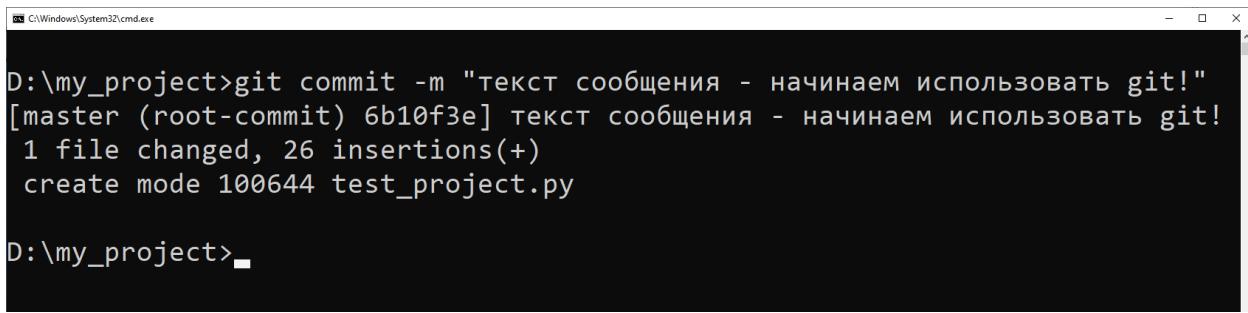
```
  new file:   test_project.py
```

```
D:\my_project>
```

Рисунок 9 — Вызов `git status` после выполнения команды `git add`

Видим, что теперь текст сообщения несколько изменился. Изменилась строка с названием нашего тестового файла. Ранее она была выделена красным цветом (рисунок 6). Сейчас же эта строка выделена зеленым цветом, и добавилась приставка `«new file: test_project.py»`. Это означает, что теперь Git начал следить за изменениями этого файла, но пока мы их не зафиксировали. Другими словами, Git увидел файл и готов зафиксировать это изменение. Для этого нам надо выполнить коммит (commit).

- **Шаг 4: Создаем коммит** Чтобы сделать коммит (сохранить изменения и зафиксировать текущую версию), выполним следующую команду: `git commit -m "текст сообщения - начинаям использовать git!"`. На рисунке 10 видим, что Git обработал нашу команду и вернул результат, в котором сказано, что в ветке `master` изменен 1 файл, в который добавлено 26 строк.



```
D:\my_project>git commit -m "текст сообщения - начинаем использовать git!"  
[master (root-commit) 6b10f3e] текст сообщения - начинаем использовать git!  
 1 file changed, 26 insertions(+)  
 create mode 100644 test_project.py  
  
D:\my_project>
```

Рисунок 10 — Создание первого коммита

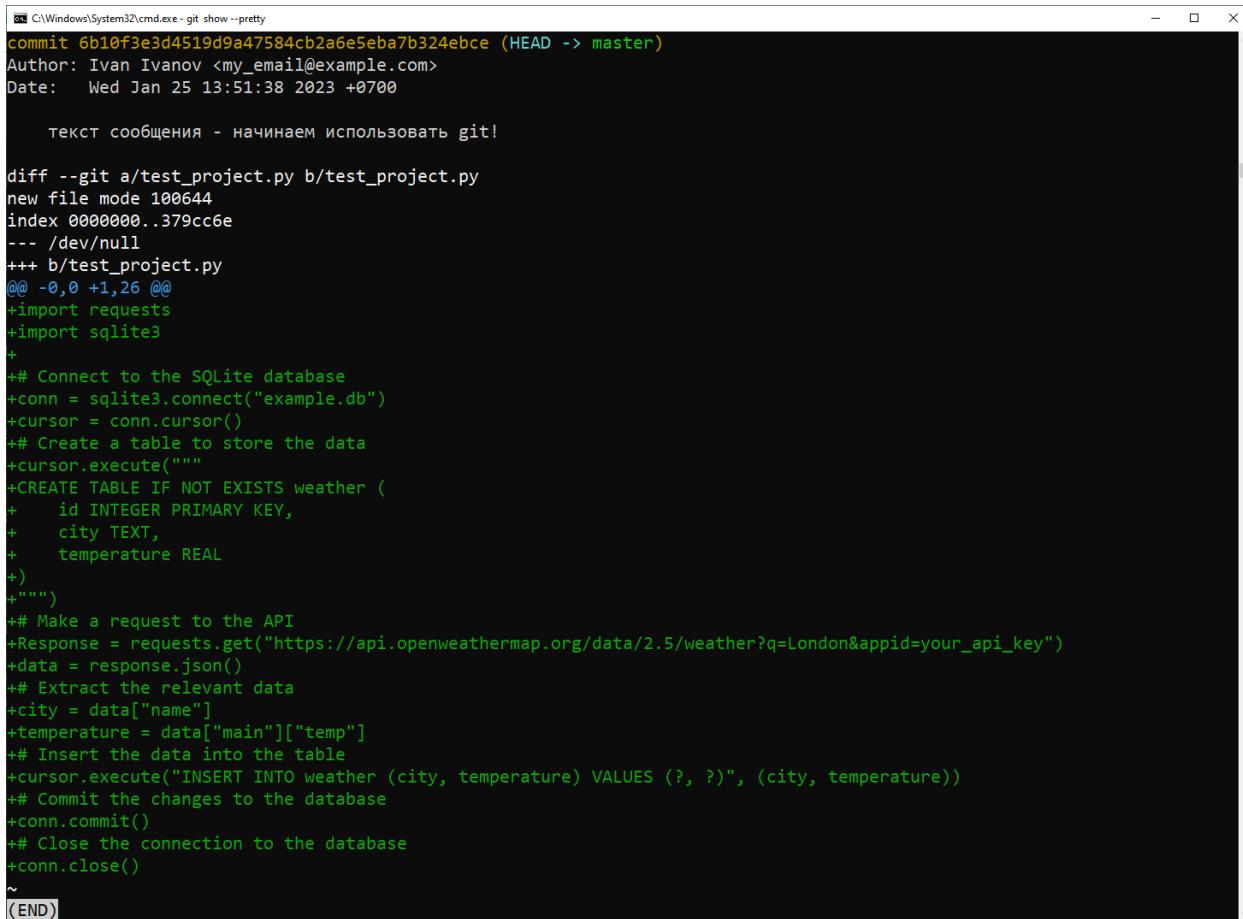
Другими словами, до того, как мы добавили файл (`git add`) и создали коммит (`git commit`), нашей системе Git нечего было отслеживать, поскольку в нем не было ни одного коммита. Теперь немного остановимся на самой команде `«git commit»`.

После команды `git commit` мы использовали параметр `-m` и в кавычках " " написали сообщение для коммита. Это очень полезная фишка для комментирования изменений кода. Например, мы добавили в код новую функцию или исправили ошибку в работе программы и хотим зафиксировать это улучшение, кратко сформулировав смысл вносимых в код изменений.

Сейчас в нашем проекте `«test_project.py»` нет проверок корректности данных, как и проверки наличия связи с удаленным сервером погоды (*OpenWeatherMap*). Когда мы добавим проверку корректности данных, нужно будет создать коммит и в качестве комментария указать: `«Добавлена проверка кода ответа от сервера OpenWeatherMap. Если код ответа: 200, данные корректны, в другом случае возникает обработка исключительной ситуации»`.

Конечно, помимо сообщения при создании коммита рекомендуется использовать комментарии в коде программы и вести документацию.

Для отслеживания всех изменений Git удобно использовать утилиту `git-show` — это утилита командной строки, которая используется для просмотра подробных данных об объектах Git (рисунок 11).



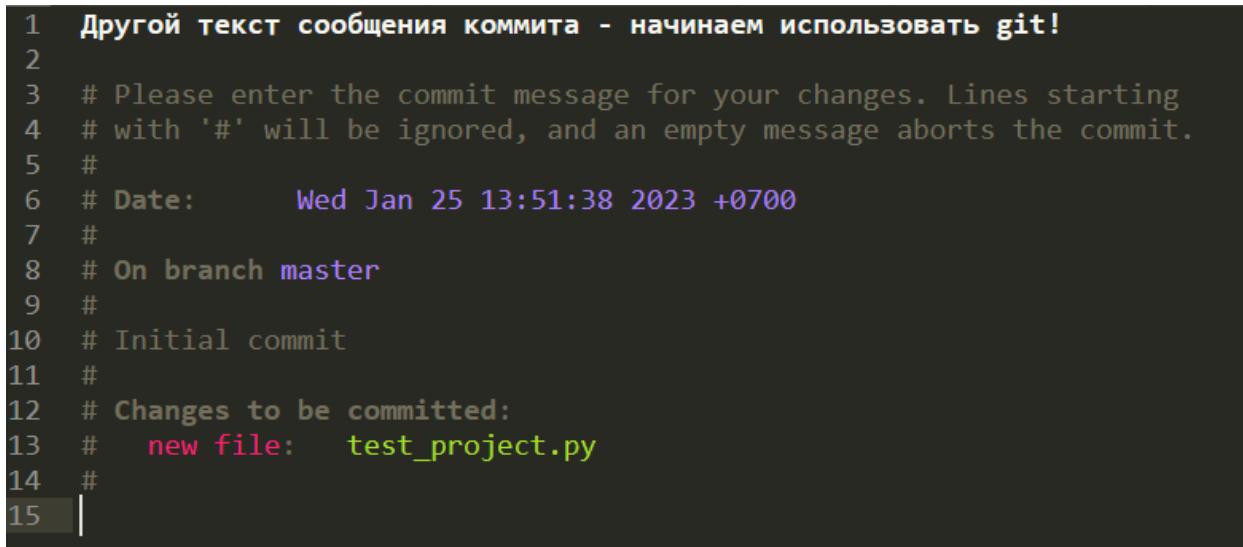
```
C:\Windows\System32\cmd.exe - git show --pretty
commit 6b10f3e3d4519d9a47584cb2a6e5eba7b324ebce (HEAD -> master)
Author: Ivan Ivanov <my_email@example.com>
Date:   Wed Jan 25 13:51:38 2023 +0700

    текст сообщения - начинаем использовать git!

diff --git a/test_project.py b/test_project.py
new file mode 100644
index 000000..379cc6e
--- /dev/null
+++ b/test_project.py
@@ -0,0 +1,26 @@
+import requests
+import sqlite3
+
+## Connect to the SQLite database
+conn = sqlite3.connect("example.db")
+cursor = conn.cursor()
## Create a table to store the data
+cursor.execute("""
+CREATE TABLE IF NOT EXISTS weather (
+    id INTEGER PRIMARY KEY,
+    city TEXT,
+    temperature REAL
+)
+""") 
## Make a request to the API
+Response = requests.get("https://api.openweathermap.org/data/2.5/weather?q=London&appid=your_api_key")
+data = response.json()
## Extract the relevant data
+city = data["name"]
+temperature = data["main"]["temp"]
## Insert the data into the table
+cursor.execute("INSERT INTO weather (city, temperature) VALUES (?, ?)", (city, temperature))
## Commit the changes to the database
+conn.commit()
## Close the connection to the database
+conn.close()
~
(END)
```

Рисунок 11 – Отображение подробной информации вызовом команды git show --pretty

**Что делать, если ошибочно внесли неполное или некорректное описание коммита?** В этом случае не стоит паниковать — всегда можно изменить содержание коммита. Чтобы изменить последний коммит, используйте команду: git commit -amend. После вызова команды откроется текстовый редактор, в котором можно изменить текст сообщения (рисунок 12).



```
1 Другой текст сообщения коммита - начинаем использовать git!
2
3 # Please enter the commit message for your changes. Lines starting
4 # with '#' will be ignored, and an empty message aborts the commit.
5 #
6 # Date:      Wed Jan 25 13:51:38 2023 +0700
7 #
8 # On branch master
9 #
10 # Initial commit
11 #
12 # Changes to be committed:
13 #   new file: test_project.py
14 #
15 |
```

Рисунок 12 – Окно редактора текста, в котором можно изменить последний коммит

После того, как вы внесете изменения в текст коммита, нужно сохранить изменения в этом файле. После этого в консоли отобразится «новое сообщение» измененного коммита (рисунок 13).

```
D:\my_project>git commit --amend
[master 4e317a7] Другой текст сообщения коммита - начинаем использовать git!
  Date: Wed Jan 25 13:51:38 2023 +0700
  1 file changed, 26 insertions(+)
  create mode 100644 test_project.py

D:\my_project>
```

Рисунок 13 — Отображение изменений коммита

Попробуйте внести изменения в файл «`test_project.py`». Например, следующим образом:

(код):

```
import requests
import sqlite3

# Connect to the SQLite database
conn = sqlite3.connect("example.db")
cursor = conn.cursor()
# Create a table to store the data
cursor.execute("""
CREATE TABLE IF NOT EXISTS weather (
    id INTEGER PRIMARY KEY,
    city TEXT,
    temperature REAL
)
""")
# Make a request to the API
try:
    url = """https://api.openweathermap.org/data/2.5/weather?"""\n
    ↪"q=London&appid=your_api_key"""
    Response = requests.get()
    Response.raise_for_status()

    data = response.json()
    # Extract the relevant data
    city = data["name"]
    temperature = data["main"]["temp"]
    # Insert the data into the table
    cursor.execute("INSERT INTO weather (city, temperature) VALUES (?, ?)",
                  (city, temperature))
    # Commit the changes to the database
    conn.commit()
    # Close the connection to the database
    conn.close()
except requests.exceptions.HTTPError as err:
    print(err)
```

Мы добавили в код проверку возникновения исключительных событий. Исключения необходимы для того, чтобы сообщать программисту об ошибках, а также корректно их обрабатывать, чтобы избавиться

от экстренного завершения работы программы. Сохраните изменения в файле с кодом и вызовите команду `git status`. Посмотрите, какой будет результат.

### 3.1.9 Вот несколько шагов, которые помогут начать работу с Git:

Изучите основы: начните с основных команд Git, таких как `git init`, `git add`, `git commit`, `git branch`, `git checkout`, `git merge` и `git push`. В Интернете есть множество ресурсов: учебные пособия, видеоролики и документация, которые могут помочь понять основные концепции и команды. Лучший способ изучить Git — это использовать его. Создайте новый проект, возьмите наш тестовый проект или найдите проект с открытым исходным кодом и начните экспериментировать с командами Git. Совершайте коммиты, создавайте ветки и пробуйте разные сценарии рабочего процесса Git.

### 3.1.10 Что такое публичный и приватный репозиторий?

В Git репозиторий (или сокращенно «репо») — это набор файлов и каталогов, которые отслеживаются системой контроля версий. Репозиторий может быть как общедоступным, так и частным.

1. Общедоступный репозиторий. Общедоступный репозиторий — это репозиторий, доступный для всех, у кого есть URL-адрес репозитория. Любой может просмотреть файлы, загрузить их и сделать копию репозитория. Публичные репозитории часто используются для проектов с открытым исходным кодом, где каждый может внести свой вклад в код. Публичные репозитории также используются для обмена кодом с другими или для того, чтобы сделать код доступным для использования другими.
2. Частный репозиторий. Частный репозиторий недоступен никому, кроме людей, которым был предоставлен доступ. Эти репозитории часто используются для проектов с закрытым исходным кодом, где код не предназначен для общего доступа. Частные репозитории также используются для хранения кода, который не готов к публичному выпуску, или для конфиденциальной информации, которой не следует делиться с общественностью. Основное различие между двумя типами репозиториев — в том, что общедоступные репозитории видны и доступны всем, а частные репозитории видны и доступны только избранной группе людей. Стоит отметить, что большинство служб хостинга Git, таких как GitHub, GitLab, Bitbucket, предлагают как общедоступные, так и частные репозитории с разными ценами и ограничениями. Некоторые из них предлагают бесплатные планы для общедоступных репозиториев и взимают плату за частные.

GitHub, GitLab и Bitbucket — популярные платформы хостинга Git, которые предоставляют как общедоступные, так и частные репозитории. На этих платформах вы можете сделать свой репозиторий общедоступным или частным при его создании. Вы также можете изменить видимость репозитория позже, но изменить его обратно может быть немного сложнее, если он общедоступный. Важно помнить, что термины «общедоступные» и «частные» репозитории относятся к хостинговой платформе и пользователям, имеющим доступ к репозиторию, а не к Интернету в целом.

### 3.1.11 Зафиксируйте изменения git

Коммит — это моментальный снимок изменений, внесенных в файлы в репозитории Git. Он используется для сохранения и отслеживания хода проекта с течением времени. Вот шаги для фиксации изменений в Git:

1. Добавьте файлы в промежуточную область. Прежде чем делать фиксацию, вам необходимо добавить файлы, которые вы хотите включить в фиксацию, в промежуточную область. Вы можете сделать это, выполнив команду `git add file-name` или `git add`, чтобы добавить все измененные файлы.

2. Создайте фиксацию: после добавления файлов в область подготовки вы можете создать фиксацию. Используйте команду `git commit` для создания коммита. Это откроет текстовый редактор по умолчанию, в котором вы можете написать сообщение фиксации.
3. Фиксация с сообщением: вместо того, чтобы открывать текстовый редактор по умолчанию, вы можете использовать флаг `-m` для фиксации с сообщением, выполнив команду `git commit -m "Ваше сообщение фиксации"`.
4. Зафиксировать с определенным файлом. Вы также можете зафиксировать определенный файл, выполнив команду `git commit имя-файла -m «Ваше сообщение о фиксации»`. Важно использовать четкие и описательные сообщения фиксации, которые объясняют сделанные изменения и причину, по которой они были сделаны. Это упрощает понимание истории проекта и помогает другим членам команды понять изменения. Кроме того, вы также можете использовать команду `git commit --amend`, чтобы изменить последнюю фиксацию, или `git commit --amend -c original-commit`, чтобы изменить предыдущую фиксацию.

### 3.1.12 Немного статистики о системе git

Git — одна из наиболее широко используемых систем контроля версий в мире, и статистика ее использования отражает это. Вот некоторые показатели использования Git:

1. Популярность. Согласно опросу разработчиков Stack Overflow 2020, Git — самая популярная система контроля версий, которую используют более 88% разработчиков.
2. Проекты с открытым исходным кодом. По данным GitHub, на их платформе размещено более 100 миллионов репозиториев, и большинство из них представляют собой проекты с открытым исходным кодом, использующие Git в качестве системы контроля версий.
3. Использование в отрасли: Git используется в самых разных отраслях, включая разработку программного обеспечения, финансы, здравоохранение и госуправление.
4. Кроссплатформенность: Git можно использовать в операционных системах Windows, Mac и Linux, что делает его кроссплатформенным инструментом.
5. Интеграция: Git можно интегрировать с широким спектром инструментов и сервисов, таких как GitHub, GitLab, Bitbucket, Jenkins и другими.
6. Совместная работа: Git позволяет нескольким людям одновременно работать над одной кодовой базой и имеет встроенные инструменты для совместной работы, такие как запросы на вытягивание и проверки кода.
7. Принятие: согласно официальному веб-сайту Git, более 100 000 организаций по всему миру используют Git для контроля версий.

Эти статистические данные показывают, что Git — распространенный и универсальный инструмент, который используется большим количеством разработчиков и организаций по всему миру.



## Git как инструмент командной работы. Работа с удаленными репозиториями

---

### 4.1 Git как инструмент командной работы. Работа с удаленными репозиториями.

Современный мир технологий и инноваций требует от специалистов не только глубоких знаний и навыков в своей области, но и умения эффективно сотрудничать в команде. Командная разработка стала неотъемлемой частью современного программирования, позволяя создавать сложные и инновационные продукты, которые были бы невозможны для одного разработчика. В данном учебном пособии мы погрузимся в мир командной работы, исследуя его принципы, лучшие практики и вызовы.

Так что же такое командная работа? Почему этому уделяется так много внимания? Посмотрим плей-лист с записями видео докладов с конференции IT-шной компании, или в статьи на хабре и наверняка найдем что-нибудь про: «Работу в команде», «Управление проектом», «Эффективный менеджмент» «Навыки Soft Skill», и много тому подобных тем. Глядя со стороны на эти темы, может показаться, что это глупости и ведь всё настолько очевидно, что нет смысла повторяться и говорить о том, «как одному разработчику в команде попросить о помощи другого», или «как мне сообщить коллеге о найденном баге в его коде». Можно ведь просто позвонить ему и сообщить о своей находке, а можно самому исправить баг и позже сообщить коллеге при встрече. Наверняка каждый из нас сталкивался с подобной ситуацией. Однако, тут не всё так просто.

Работу в команде можно сравнить с системой автопилота самолета или беспилотного автомобиля. В которых за процессом управления непрерывно следят сотни или даже тысячи подсистем, информирующих бортовой компьютер о своем состоянии и показаниях датчиков. Таким образом, бортовой компьютер всегда обладает актуальной информацией и принимает лучшее решения для управления движением.

Аналогичным образом выглядит процесс управления разработкой ИТ проекта, в котором условный «бортовой компьютер» это «Руководитель проекта». Его роль - принимать эффективные решения, способствующие успешному запуску ИТ продукта и его дальнейшего развития. Как и у «бортового компьютера», у Руководителя проекта есть команда, с которой он взаимодействует, получает информацию, принимает управленческие решения. Например, расширить команду разработчиков, сформировать команду тестировщиков, разработать стратегию вывода продукта на рынок и т.д.

**Состав команды и их задачи в проекте разработки программного продукта**

---

**1. Менеджер проекта:**

- Организует работу команды, следит за сроками и бюджетом.
- Распределяет задачи между участниками и координирует их деятельность.
- Составляет план разработки и следит за его выполнением.
- Решает организационные и коммуникационные вопросы.

**2. Архитектор:**

- Определяет общую архитектуру продукта и его компонентов.
- Разрабатывает технические решения и структуры системы.
- Обеспечивает согласованность и соответствие архитектуры целям проекта.

**3. Разработчики:**

- Отвечают за разработку кода, интеграцию со сторонними сервисами и реализацию дополнительной функциональности продукта.
- Работают над конкретными модулями или компонентами.
- Проводят тестирование своего кода и участвуют в код-ревью.

**4. Тестировщики:**

- Занимаются тестированием функциональности продукта.
- Разрабатывают тестовые сценарии и проводят автоматическое и ручное тестирование.
- Выявляют и документируют ошибки и несоответствия в работе продукта.

**5. Дизайнер/Интерфейсный дизайнер:**

- Разрабатывает дизайн пользовательского интерфейса продукта.
- Создает макеты, элементы дизайна и визуальные компоненты.
- Обеспечивает удобство использования и эстетическое качество интерфейса.

**6. Технический писатель/Разработчик документации и инструкций пользователя (User guide):**

- Разрабатывает документацию для продукта: руководства пользователя, API-документацию и т.д.
- Обеспечивает четкость и понятность технической информации.
- Взаимодействует с другими членами команды для сбора и структурирования информации.

**7. Системный администратор/DevOps инженер:**

- Занимается настройкой инфраструктуры и серверов для разработки и развертывания продукта.
- Обеспечивает автоматизацию процессов сборки, тестирования и развертывания.
- Решает вопросы масштабирования и безопасности инфраструктуры.

Это классический состав команды, который охватывает основные этапы разработки программного продукта – от планирования и проектирования до разработки, тестирования и документирования. Каждый участник команды вносит вклад в успешное выполнение проекта, обеспечивая высокое качество и соблюдение сроков.

С командой и её составом мы разобрались. Казалось бы, что этого достаточно – нужно чтобы каждый выполнял свои задачи и тогда всё будет работать как часы. И тут мы с вами подходим к главному

вопросу: Как организовать и запустить рабочий процесс в команде так, чтобы каждый знал что ему нужно делать и понимал как взаимодействовать с коллегами в команде.

Конечно, мы можем «смоделировать ситуацию и мысленно представить этот рабочий процесс», как лучше поступить каждому участнику команды, выбрав на наш взгляд самый эффективный способ взаимодействия внутри команды. Каждый может это представить и будет по-своему прав.

Однако, в реальном мире всё несколько иначе. Каждый сотрудник должен уметь включаться в командную работу учитывая не только свои интересы, но также и интересы коллектива взаимодействуя в команде в которой имеются внутренние договоренности, нормы и правила. Если обратиться, к примеру «автопилота», то все системы взаимодействуют друг с другом посредством «протокола передачи данных». В протоколе прописаны «правила» и «критерии», которые должны соблюдать все устройства использующие этот протокол. Например, благодаря протоколу HTTP вы можете «серфить по интернету» и не важно, каким устройством (ноутбук, смартфон, планшет и т.д.) и какой канал связи (WiFi, Ethernet, 5G) вы используете. Вы сможете открыть любой сайт и выполнить действие. Например, оформить доставку пиццы и оплатить свой заказ. Требования к протоколам сформулированы в специальном документе - «техническом стандарте». Подробнее про стандарты и протоколы можете посмотреть в справочном материале.

#### 4.1.1 Знакомство с методом SCRUM

Для управления ИТ проектами часто можно услышать про методологию SCRUM

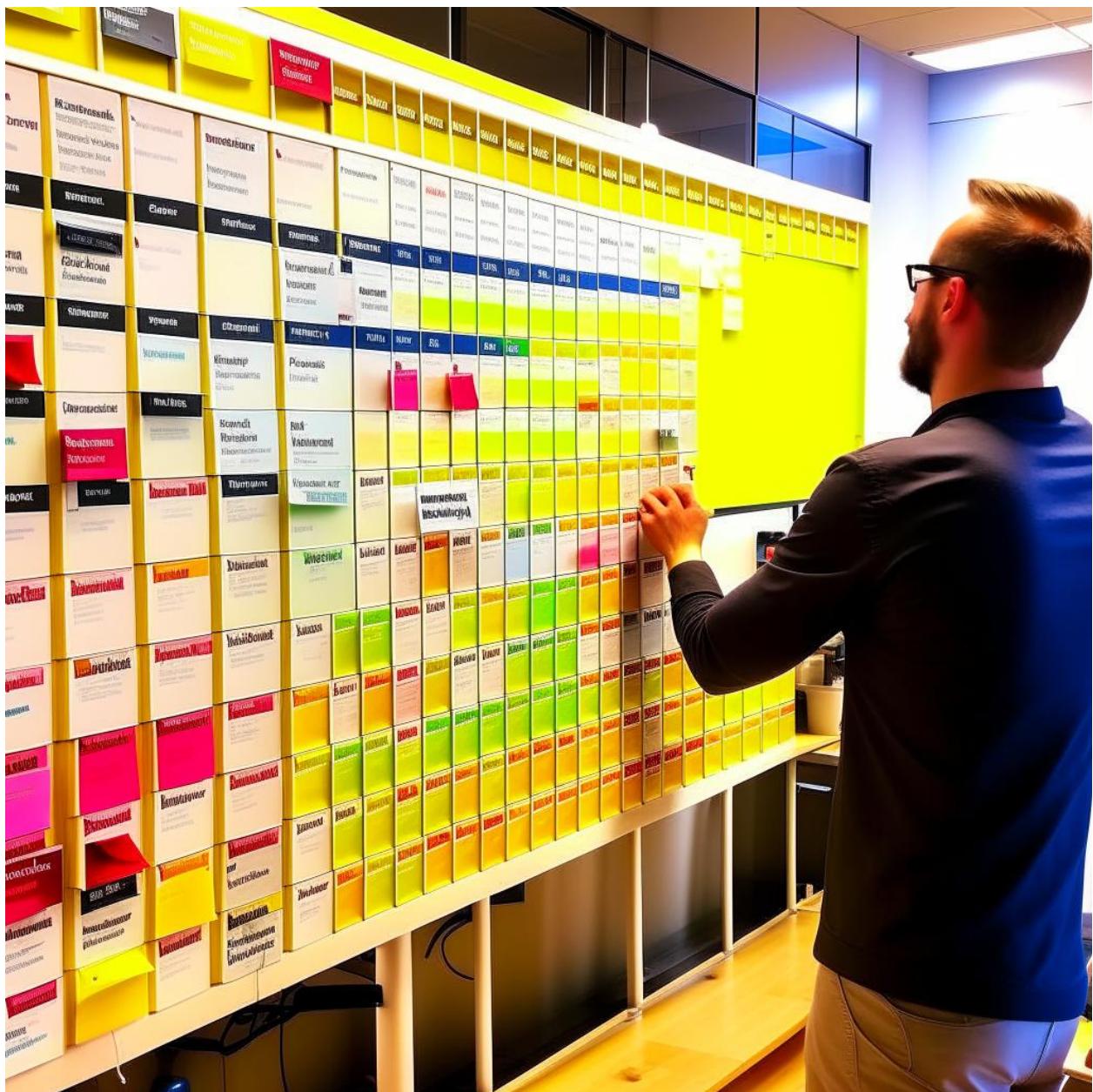
Методология SCRUM представляет собой гибкий подход к управлению проектами, ориентированный на совместную итеративную разработку программных продуктов. В ней выделяются следующие ключевые идеи:

- 1. Итеративность и поступательность:** Проект разбивается на небольшие временные интервалы, называемые спринтами, каждый продолжительностью 1-2 недели. Длительность спрента зависит от задач проекта и возможностей команды. Каждый спрент добавляет новую функциональность к продукту, или вносит серьезное изменение в существующий проект. Например, команда разрабатывает мобильную игру в которой с помощью «функциональных блоков» выстраивается стратегия игрока. Разработку нового типа «блока» с уникальным функционалом, который будет совместим с уже имеющимися блоками можно считать новой итерацией (шагом) в разработке игры. Эту задачу можно реализовать в пределах одного спрента.
- 2. Роли и обязанности:** В SCRUM выделяются три ключевые роли: Продуктовый Владелец (Product Owner), Мастер SCRUMа (Scrum Master) и Разработчики. Каждая роль выполняет определенные функции для эффективной работы команды.
- 3. Бэклог продукта:** Это список задач и требований, представляющих потребности клиентов и пользователей. Бэклог продукта непрерывно обновляется, и наиболее приоритетные задачи планируются для выполнения в ближайших спрентах. В качестве инструмента для обозначения статуса задач и заметок используют «Канбан доску». Это инструмент визуализации рабочего процесса и управления задачами. Она представляет собой физическую или электронную доску, разделенную на несколько колонок, которые отражают различные этапы выполнения задач. Каждая задача представлена карточкой, которая перемещается по колонкам в зависимости от ее текущего статуса. Подробнее с этим инструментом познакомимся чуть позже.
- 4. Планирование спрента:** На начало каждого спрента команда выбирает задачи из бэклога продукта для выполнения. Задачи декомпозируются на более мелкие, оцениваются временем и планируются для достижения цели спрента.
- 5. Ежедневные стендалы:** Команда проводит короткие ежедневные встречи, на которых каждый член команды отвечает на три вопроса: «Что я сделал вчера?», «Что я собираюсь сделать сегодня?» и «Есть ли какие-либо препятствия?». Это помогает поддерживать прозрачность и обмен информацией.

6. **Спринт-ревью и ретроспектива:** По завершении спринта проводится спринт-ревью, на котором команда представляет результаты продукту. Затем проводится спринт-ретроспектива, на которой анализируются успехи и уроки для улучшения процесса.
7. **Адаптация:** Одной из ключевых идей SCRUM является постоянное совершенствование. Команда адаптируется к изменениям, основанным на обратной связи, и внедряет улучшения для повышения эффективности работы.
8. **Прозрачность и открытость:** В SCRUM поддерживается прозрачность во всех аспектах работы. Задачи, прогресс, проблемы и решения доступны всей команде, что способствует коллективной ответственности за успех проекта.
9. **Управление рисками:** SCRUM позволяет быстро выявлять риски и проблемы и реагировать на них в реальном времени. Это помогает минимизировать негативные воздействия на проект.
10. **Цикличность и непрерывное улучшение:** Методология SCRUM ориентирована на непрерывное совершенствование процесса, продукта и команды. Постоянное обучение и адаптация помогают достигать лучших результатов.

Использование методологии SCRUM в ИТ стартапе способствует быстрой разработке и поставке качественных продуктов, учету изменений и потребностей клиентов, а также эффективной командной работе.

#### 4.1.2 Знакомство с канбан-доской



Основная цель Канбан-доски - обеспечить прозрачность рабочего процесса чтобы каждый участник проекта мог видеть какие задачи ожидают решения в списке очереди, и кто в текущий момент выполняет какие задачи. Таким образом этот инструмент помогает управлять ресурсами и распределять рабочую нагрузку для оптимизации различных затрат, в том числе и временных. Задачи могут быть разбиты на подзадачи, приоритеты могут быть ясно видны, а вся команда может легко оценить текущее состояние проекта.

На Канбан-доске обычно используются следующие базовые колонки:

1. **Backlog (Очередь задач):** Эта колонка содержит задачи, которые пока не приступили к выполнению. Здесь формируется список задач, из которого команда будет брать новые задания для работы.
2. **To Do (В работе):** В этой колонке находятся задачи, над которыми в данный момент работают

участники команды.

3. **In Progress (Выполняются)**: Задачи, которые находятся в процессе выполнения, перемещаются сюда после начала работы.
4. **Review (На проверке)**: После того как задача завершена, она переходит в эту колонку на проверку и ревью другими членами команды.
5. **Done (Готово)**: Когда задача полностью выполнена и прошла проверку, она перемещается в эту колонку.

Кроме базовых колонок, на доске могут быть добавлены и другие, в зависимости от потребностей команды и характера проекта. Например:

- **Blocked (Заблокировано)**: Здесь помещаются задачи, которые не могут быть продолжены из-за каких-либо проблем или зависимостей.
- **Testing (Тестирование)**: Задачи, находящиеся в этой колонке, ожидают проведения тестирования.
- **Deploy (Развертывание)**: Для задач, которые готовы к развертыванию, но еще не внедрены в продакшн.
- **Archived (Архив)**: Завершенные и проверенные задачи могут быть перемещены сюда, чтобы не засорять основную доску.

Выбор колонок зависит от особенностей проекта и его специфики. Например, разработка «чат-бота» и разработка «мобильного приложения» это 2 проекта которые отличаются как инструментами разработки, так и методами тестирования и внедрения. Следовательно, и колонки на Канбан доске будут отличаться как по количеству, так и по их структура может быть адаптирована для соответствия рабочему процессу.

### Канбан доска проекта: "Мобильное приложение"

В очереди	В работе	Выполняется	На проверке	Готово
Выполнить обзор аналогов				
Разработка дизайна				
Разработка архитектуры приложения				
Разработка модели базы данных				
Тестирование первого прототипа				
Разработка документации				

#### 4.1.3 Задача: Планирование и организация работы над разработкой чат-бота с использованием Канбан доски.

Описание задачи: Вашей задачей является разработка детального плана работы над проектом чат-бота с использованием методологии Канбан. Данный план должен включать в себя определение основных этапов разработки, колонок на Канбан доске и описание процесса перемещения задач между этими колонками. Вам также предстоит определить ответственных за каждый этап и роли в команде.

Шаги выполнения:

1. Определение этапов разработки: Разбейте весь процесс разработки чат-бота на основные этапы, такие как проектирование, разработка, тестирование, реview, развертывание и др.
2. Создание Канбан доски: Определите основные колонки на Канбан доске. Обычно это **Backlog**, **To Do**, **In Progress**, **Review**, **Testing**, **Done**.
3. Описание перемещения задач: Для каждой колонки опишите, какие задачи могут находиться в ней, и каким образом они будут перемещаться от этапа к этапу. Например, задачи могут начинать свой путь с **Backlog**, затем переходить в **To Do**, когда назначены на выполнение, далее в **In Progress** при начале работы и так далее.
4. Назначение ответственных: Определите, кто будет ответственным за каждый этап. Это может быть разработчик, тестировщик, реviewер и т.д. Распределите роли в команде.

5. Установление временных рамок: Определите ожидаемое время выполнения для каждого этапа (спринта) и задачи. Это поможет вам лучше планировать и контролировать процесс разработки и оценивать время.
6. Создание доски: Воспользуйтесь инструментом для создания Канбан доски, например, онлайн-сервисами или физической доской с карточками.
7. Запуск работы: Начните работу с разработкой чат-бота в соответствии с описанным планом. Перемещайте задачи по Канбан доске, следите за прогрессом и корректируйте план при необходимости.
8. Внесение изменений: При необходимости, вносите изменения в Канбан доску, добавляя новые колонки или меняя порядок этапов. Главное, чтобы доска отражала текущий процесс и удовлетворяла потребности команды.

**Цель задачи:** Разработать детальный план работы над проектом с использованием Канбан доски, который поможет команде эффективно организовать процесс разработки чат-бота, контролировать прогресс и достичь успешных результатов.

#### **4.1.4 Заключение**

В данном блоке мы постарались кратко познакомиться вас с основными элементами командной работы на ИТ проектом: От начальных этапов планирования и определения ролей, до финального слияния кода и представления результатов, мы рассказали о ключевых шагах и важных уроках, которые вам помогут при выполнении совместной работы. Конечно, есть и много других тем, которые не были представлены в данном модуле, однако все они являются ветвями развития представленных базовых основ. Надеемся что вам удалость:

- Ознакомиться с принципами формирования эффективной команды разработчиков.
- Понять, как распределение ролей и задач влияет на процесс работы над проектом.
- Исследовать методы планирования и организации проекта в команде.
- Узнать о инструментах и технологиях, которые облегчают командную разработку.
- Погрузиться в аспекты совместной работы, решения конфликтов и обмена знаний.

Мы надеемся, что этот рассказ о командной работе станет вдохновением для тех, кто стремится к профессиональному росту и успешной совместной разработке. Надеемся что уроки, которые были представлены в этом модуле, помогут вам на пути к развитию навыков командной работы и созданию качественных технологических продуктов.

#### **Проверочные вопросы**

1. Что такое методология Scrum и какие основные компоненты она включает?
2. Какие преимущества предоставляет методология Scrum при управлении ИТ-проектами?
3. Чем отличается Scrum Master от Product Owner в контексте Scrum?
4. Какие основные роли включает команда разработки в методологии Scrum и каковы их обязанности?
5. Что такое Sprint (итерация) в Scrum? Какие действия выполняются в рамках Sprint?
6. Что такое канбан доска и как она помогает в управлении проектом?
7. Какие принципы лежат в основе канбан методологии? Как она отличается от Scrum?

- 
8. Какое значение имеет распределение ролей в команде проекта? Какие роли обычно включаются в команду разработки?
  9. Какие ключевые навыки и характеристики важны для успешной работы в команде проекта?
  10. Какие практики способствуют эффективной командной работе в IT-проекте?
- 

#### 4.1.5 Справочная информация

В этом блоке представлена справочная информация, которая может послужить хорошим материалом, чтобы погрузиться в темы «Командная работы» и «Управление проектами». Представлен краткий анализ и сравнение различных методик управления проектами, а также некоторые подробности этапов их реализации. Также, здесь собраны краткие исторические факты и некоторые теоретические дополнения, которые помогут найти ответы, на некоторые вопросы, появившиеся при изучении основного материала.

#### 4.1.6 Роли каждого участника команды

- 1. Программист:** Программист отвечает за добавление нового функционала в игру. Например, он может работать над созданием новых типов блоков, улучшением алгоритма генерации фигур или добавлением новых игровых режимов. Программист анализирует код игры, понимает, какие изменения необходимы, и начинает их реализацию. Он также занимается тестированием своего кода, чтобы убедиться, что новый функционал работает корректно.
- 2. Тестировщик:** Тестировщик проверяет новый функционал на работоспособность и корректность. Он разрабатывает тестовые сценарии, которые покрывают различные аспекты игры с новым функционалом. Тестировщик также проводит тестирование на разных уровнях - от модульного тестирования до интеграционного тестирования. Если тесты выявляют ошибки или неполадки, тестировщик сообщает разработчику, чтобы проблемы могли быть исправлены.
- 3. Технический писатель:** Технический писатель разрабатывает документацию, которая объясняет, как использовать новый функционал игры. Это может быть руководство пользователя, описание новых игровых режимов или объяснение изменений для других разработчиков. Технический писатель обеспечивает четкость и понятность документации, чтобы пользователи или коллеги могли легко разобраться в новых возможностях игры.
- 4. Менеджер проекта:** Менеджер проекта координирует деятельность всей команды. Он следит за сроками выполнения задач, управляет ресурсами и обеспечивает коммуникацию внутри команды. Менеджер проекта также может помогать в планировании разработки нового функционала, распределять задачи между участниками и решать организационные вопросы. Важно, чтобы менеджер проекта обеспечивал эффективное взаимодействие между всеми участниками команды и следил за достижением целей проекта.

Каждый участник команды играет важную роль для достижения цели - успешной доработки игры «Тетрис». Их совместная работа, обмен знаниями и координация позволяют достичь хорошего результатов.

#### **4.1.7 Краткая история методологии SCRUM**

История появления методологии SCRUM началась в мире разработки программного обеспечения в начале 1980-х годов. В то время компания Easel Corporation, занимавшаяся разработкой графических приложений, столкнулась с необходимостью улучшить свой подход к разработке, чтобы более эффективно управлять проектами и достигать более предсказуемых результатов.

В 1986 году, Джек Сазерленд (Jeff Sutherland), занимавшийся исследованиями в области управления проектами, разработал первый прототип методологии SCRUM, который использовался внутри компании Easel. В 1993 году, Сазерленд и Кен Швабер (Ken Schwaber), также работавший в Easel, представили идеи SCRUM на конференции OOPSLA (Object-Oriented Programming, Systems, Languages & Applications).

Следующие годы были периодом экспериментов, модификаций и совершенствования методологии SCRUM. Команда разработчиков, включая Сазерленда и Швабера, проводила исследования и тестируя принципы SCRUM на различных проектах. В 2001 году, на знаменитой встрече «Snowbird», группа специалистов по разработке программного обеспечения, включая Сазерленда и Швабера, сформулировала Манифест гибкой разработки программного обеспечения (Agile Manifesto), который включал в себя принципы, заложенные в SCRUM.

С тех пор SCRUM приобрел широкое признание и стал одним из самых популярных и успешных методологических фреймворков в области разработки программного обеспечения. Его применяют не только в ИТ-индустрии, но и во многих других отраслях, где важно быстро и гибко реагировать на изменения и достигать высококачественных результатов.

PMBOK, Agile и Scrum являются тремя разными методологиями и подходами к управлению проектами. Вот как они связаны между собой:

**PMBOK (Project Management Body of Knowledge):** PMBOK - это набор стандартов и рекомендаций, разработанный Project Management Institute (PMI), который охватывает все ключевые аспекты управления проектами. Он включает в себя различные процессы, знания и практики, необходимые для эффективного управления проектами. PMBOK не привязан к конкретной методологии и может быть адаптирован к разным подходам.

**Agile (гибкая методология):** Agile - это общий термин, описывающий группу методологий управления проектами, которые акцентируются на гибкости, сотрудничестве и быстрой адаптации к изменениям. Agile подходы ориентированы на достижение результатов через инкрементальное итеративное развитие, приоритизацию задач и активное взаимодействие с заказчиком.

**Scrum:** Scrum - это конкретная методология Agile, которая представляет собой фреймворк для управления проектами. Он включает в себя роли (Product Owner, Scrum Master, Development Team), события (Sprint Planning, Daily Standup, Sprint Review, Sprint Retrospective) и артефакты (Product Backlog, Sprint Backlog, Increment), которые помогают командам разрабатывать продукт в коротких циклах (спринтах) с постоянной обратной связью и адаптацией.

Связь между ними:

PMBOK описывает широкий спектр практик управления проектами, включая традиционные методы. Agile - это общий подход, который акцентирует внимание на гибкости и реагировании на изменения, и он может быть применен к разным методологиям. Scrum - это конкретный фреймворк Agile, который предоставляет инструкции и инструменты для управления проектами с акцентом на сотрудничество и итеративное развитие. Многие компании и команды применяют элементы Agile и Scrum в сочетании с PMBOK для создания гибкой и эффективной методологии управления проектами, которая учитывает как традиционные, так и инновационные подходы.

#### 4.1.8 Управление коммуникациями

«Управление коммуникациями» в рамках РМВОК охватывает планирование, выполнение и контроль коммуникаций внутри проектной команды и с заинтересованными сторонами. Эффективный обмен информацией и коммуникации считаются ключевыми элементами успешного управления проектом, поскольку они способствуют пониманию, сотрудничеству и принятию правильных решений. Вот подробнее об этом аспекте:

#### 4.1.9 Планирование коммуникаций:

1. Определение целей коммуникаций: Определение, что именно должно быть достигнуто через коммуникации в проекте.
2. Идентификация аудитории: Определение заинтересованных сторон, которым необходимо предоставить информацию и с кем следует поддерживать обмен. Определение сообщений: Какие сообщения и какая информация будет распространяться.
3. Выбор методов и средств коммуникаций: Определение наиболее подходящих способов для достижения задач коммуникаций (например, встречи, отчеты, электронные письма и др.).
4. Разработка плана коммуникаций: Создание документированного плана, который описывает кто, что, когда и как будет коммуницировать. Распределение коммуникаций:
5. Распространение информации: Отправка сообщений, предоставление данных и обмен информацией с членами команды и стейкхолдерами.
6. Своевременность и четкость: Гарантирование того, что коммуникации происходят вовремя и ясно, исключая неоднозначности.

#### 4.1.10 Мониторинг и контроль коммуникаций:

1. Оценка эффективности: Оценка, насколько эффективными являются методы коммуникаций и как достигается задачи.
2. Регулирование и улучшение: Если процесс коммуникаций не работает эффективно, регулирование методов и средств для улучшения взаимодействия. Эффективное управление коммуникациями в проекте способствует более гладкому сотрудничеству между участниками команды, повышению осведомленности всех сторон о текущем состоянии проекта и позволяет решать проблемы и конфликты более оперативно.

Управление коммуникациями в контексте небольшого ИТ стартапа, который применяет методологию SCRUM, играет важную роль в обеспечении эффективной командной работы, передаче информации и обмене знаний между участниками проекта. Важно поддерживать своевременную и прозрачную коммуникацию, чтобы участники проекта были в курсе всех изменений, задач и достигнутых результатов.

#### 4.1.11 Процесс управления коммуникациями в стартапе, использующем SCRUM, может выглядеть следующим образом:

1. **Daily Stand-ups:** Ежедневные стендалпы (Daily Stand-ups) - это короткие встречи, на которых каждый член команды делится своими достижениями, планами на день и возникшими препятствиями. Это позволяет всей команде быть в курсе актуальной ситуации, устранять блокирующие факторы и поддерживать синхронизацию работ.
2. **Спринт-планирование:** На начало каждого спринта проводится встреча спринт-планирования, где команда определяет, какие задачи будут выполнены в текущем спринте. На этой встрече

важно провести детальное обсуждение и разъяснение задач, чтобы убедиться, что каждый член команды понимает свои обязанности.

3. **Спринт-ревью:** В конце спринта проводится спринт-ревью, на котором команда демонстрирует полученные результаты заказчику и другим заинтересованным сторонам. Это время для обратной связи и обсуждения проделанной работы.
4. **Спринт-ретроспектива:** После спринт-ревью проводится спринт-ретроспектива, где команда анализирует прошлый спринт, выявляет положительные моменты и обсуждает, что можно улучшить. Это способ повысить эффективность командной работы и процессов.
5. **Инструменты коммуникации:** Важно выбрать подходящие инструменты для коммуникации, например, Slack, Microsoft Teams или Trello. Они позволяют обмениваться информацией, документами и управлять задачами в режиме реального времени.
6. **Открытость и прозрачность:** Коммуникация в стартапе должна быть открытой и прозрачной. Это означает делиться информацией о ходе проекта, достигнутых успехах и возникших проблемах. Это помогает создать атмосферу доверия и сотрудничества.
7. **Управление изменениями:** В ходе разработки стартап может столкнуться с изменениями в требованиях или приоритетах. Эти изменения также должны быть прозрачно обсуждены и приняты в команде.

Управление коммуникациями в методологии SCRUM помогает снизить риски недопонимания, ускорить процесс принятия решений и создать единую видимость проекта для всех участников.

### Процесс управления в небольшом ИТ стартапе

Процесс управления в небольшом ИТ стартапе, который использует методологию SCRUM, ориентирован на достижение целей проекта через гибкое планирование, прозрачное выполнение задач и систематический анализ результатов. Процесс управления включает следующие ключевые этапы:

1. **Планирование проекта:** В начале проекта команда стартапа определяет общую картину целей, ожидаемые результаты и план спринтов. План спринтов включает список задач и приоритеты, которые будут выполнены в каждом спринте. Этот этап позволяет определить общий курс разработки и обеспечить общее понимание среди участников.
2. **Создание бэклога продукта:** Бэклог продукта представляет собой список задач, которые необходимо выполнить в проекте. Задачи в бэклоге приоритезируются, чтобы обеспечить максимальное добавление ценности для клиентов и пользователей.
3. **Планирование спринта:** Команда стартапа выбирает задачи из бэклога продукта для выполнения в следующем спринте. Задачи декомпозируются на меньшие подзадачи, оцениваются временем и приоритезируются в порядке их важности.
4. **Ежедневные стендапы:** Каждый день команда проводит короткую встречу, на которой каждый член команды дает ответ на три вопроса: «Что я сделал вчера?», «Что я планирую сделать сегодня?» и «Есть ли какие-либо препятствия?». Это помогает участникам быть в курсе прогресса и обнаруживать проблемы в ранней стадии.
5. **Выполнение спринта:** Команда работает над задачами в течение спринта (обычно 2-4 недели). Разработчики регулярно обмениваются информацией и обсуждают прогресс. Промежуточные результаты показываются заказчику на спринт-ревью.
6. **Спринт-ревью и ретроспектива:** По завершении спринта проводится спринт-ревью, где демонстрируются результаты заказчику. Затем проводится спринт-ретроспектива, на которой команда анализирует, что было сделано хорошо и что можно улучшить.
7. **Регулярные обзоры:** Регулярно проводятся обзоры с заказчиком, на которых обсуждаются текущие задачи, изменения в приоритетах и выставляются оценки выполнения работ.

8. **Анализ метрик:** Команда стартапа анализирует ключевые метрики проекта, такие как скорость выполнения задач, качество продукта и удовлетворенность клиентов. Эти метрики помогают принимать решения и корректировать планы.

Процесс управления в стартапе на базе SCRUM позволяет команде эффективно работать над проектом, реагировать на изменения в требованиях и быстро доставлять ценность клиентам. Гибкость и прозрачность методологии SCRUM способствуют успешной реализации проектов в условиях быстро меняющейся среды ИТ.

## Процесс управления проектом по РМВОК

Процесс управления в контексте управления проектами, особенно как описано в стандарте РМВОК (Project Management Body of Knowledge), включает в себя множество активностей и шагов, необходимых для планирования, мониторинга, контроля и закрытия проекта. Процесс управления охватывает всю жизненный цикл проекта, начиная с его инициирования и заканчивая завершением.

Процесс управления включает следующие ключевые аспекты:

1. **Инициирование:** Этот этап включает в себя определение бизнес-целей, целей проекта и оценку его осуществимости. Важно также определить заинтересованных сторон и их ожидания от проекта. В результате этого этапа создается документ, называемый Хартером проекта, который формализует основные параметры проекта.
2. **Планирование:** Планирование включает в себя разработку детального плана проекта, определение необходимых ресурсов, установление расписания, а также определение стратегий управления качеством, рисками, коммуникациями и другими аспектами. План проекта служит основой для выполнения и мониторинга.
3. **Исполнение:** На этом этапе команда проекта реализует план, выполняет запланированные работы и создает продукты или результаты, соответствующие требованиям проекта и ожиданиям заказчика.
4. **Мониторинг и контроль:** Этот этап включает постоянный мониторинг выполнения проекта по сравнению с планом. Если отклонения выявлены, то применяются корректирующие меры. Происходит оценка рисков и качества, а также обеспечивается соблюдение графика и бюджета.
5. **Закрытие:** В конце проекта проводится формальное закрытие, которое включает в себя проверку выполнения всех требований проекта, передачу результатов заказчику и завершение всех административных и финансовых аспектов.

Процесс управления обеспечивает системный и структурированный подход к управлению проектом, который позволяет своевременно и эффективно достигать поставленных целей. Кроме того, он учитывает не только технические аспекты, но и социальные, коммуникационные и организационные аспекты управления проектом.

## Процесс решения задачи в команде: Как обдумать, детерминировать и выполнить задачу

Шаг	Описание
1 Понимание «Задачи и Разделение задач на простые части»	Обсуждение задачи и разбиение ее на более мелкие управляемые части. Менеджер проекта и архитектор участвуют в этом процессе. Для создания страницы авторизации веб-приложения разбивается на подзадачи: «Создание формы входа», «Подключение к базе данных», «Проверка данных пользователя» и т.д.
2 Назначение ответственных и установка приоритетов	Каждая подзадача получает ответственного. Устанавливаются приоритеты. Разработчик А отвечает за создание формы, разработчик В - за подключение к базе данных, разработчик С - за проверку данных пользователя.
3 Обдумывание и Разработка	Каждый участник начинает работу над своей подзадачей. Анализ требований, деталей, рисков. Разработчик В анализирует структуру базы данных, выбирает технологии, обсуждает с архитектором.
4 Взаимодействие и ревью	Регулярное общение и ревью. Демонстрация результатов. Разработчик С демонстрирует реализацию проверки данных другим участникам.
5 Интеграция и тестирование	Интеграция всех частей продукта, тестирование функциональности и взаимодействия. Веб-приложение проходит интеграционное и функциональное тестирование, включая проверку страницы авторизации.
6 Финальная оценка и представление	Окончательная оценка работы, представление заказчику. После финального тестирования команда уверяется, что страница авторизации работает корректно, затем представляет результат заказчику.

## 4.2 Задача: Планирование и организация работы над разработкой чат-бота с использованием Канбан доски

Описание задачи: Вашей задачей является разработка детального плана работы над проектом чат-бота с использованием методологии Канбан. Данный план должен включать в себя определение основных этапов разработки, колонок на Канбан доске и описание процесса перемещения задач между этими колонками. Вам также предстоит определить ответственных за каждый этап и роли в команде.

Этапы решения задачи:

1. Определите этапы разработки. Разбейте весь процесс разработки чат-бота на основные этапы, такие как проектирование, разработка, тестирование, ревью, развертывание и др.
2. Создайте Канбан доску. Определите основные колонки на Канбан доске. Обычно это **Backlog**, **To Do**, **In Progress**, **Review**, **Testing**, **Done**.
3. Составьте список задач для команды и опишите их выполнения задач. Для каждой колонки опишите, статус задач и каким образом они будут перемещаться от этапа к этапу. Например, задачи могут начинать свой путь с **Backlog**, затем переходить в **To Do**, когда назначены на выполнение, далее в **In Progress** при начале работы и так далее.
4. Назначте ответственных для задач. Определите, кто будет ответственным для каждой задачи. Это может быть разработчик, тестировщик, ревьюер и т.д. Распределите роли в команде.
5. Установите временные рамки задач. Определите ожидаемое время выполнения для каждого этапа (спринта) и задачи. Это поможет вам лучше планировать и контролировать процесс разработки и оценивать время.

6. Создание доски в онлайн сервисе: Воспользуйтесь удобным инструментом для создания Канбан доски, например, онлайн-сервисами или физической доской с карточками. И перенесите на неё информацию с предыдущих этапов (3, 4 и 5).
7. Выполните несколько итераций работы с проектом. Начните работу по разработке чат-бота в соответствии с разработанным планом. Перемещайте задачи по Канбан доске, следите за прогрессом и корректируйте план при необходимости.
8. Внесете изменения в процесс работы если план работ будет скорректирован. Например появилась новая задача которая требует решения в первую очередь, или наоборот, возможно какая-то задача потеряет актуальность и её нужно убрать с Канбан доски. При необходимости, вносите изменения в Канбан доску, добавляя новые колонки или меняя порядок этапов. Главное, чтобы доска отражала текущий процесс и удовлетворяла потребности команды.

**Цель задачи:** Получить опыт в использовании инструментов командной работы, разработке плана работ над проектом с использованием Git и Канбан доски, который поможет команде эффективно организовать процесс разработки чат-бота, контролировать прогресс и достичь успешных результатов.

### 4.3 Вопросы

1. Какие основные задачи выполняет Scrum Master в процессе управления проектом? а) Разработка кода и тестирование продукта. б) Планирование проекта и установка приоритетов задач. в) Организация ежедневных совещаний и отчетность перед заказчиком. г) Устранение препятствий, поддержка команды и соблюдение принципов Scrum.
2. Какие характеристики делают канбан доску эффективным инструментом для визуализации рабочего процесса? а) Отсутствие ограничений по количеству задач. б) Четкое разграничение этапов выполнения задач. в) Отображение всех задач на одной доске без разделения. г) Использование сложных цветовых маркировок.
3. Какие принципы лежат в основе Agile методологий, таких как Scrum и канбан? а) Строгий контроль и управление процессом разработки. б) Предпочтение планам и документации перед работающим кодом. в) Разработка взаимодействующими командами и участниками продукта. г) Сосредоточение только на первоначально заданных требованиях.
4. Какие преимущества и недостатки связаны с гибкими методологиями управления проектами, например, Scrum и канбан? а) Преимущество: быстрая реакция на изменения. Недостаток: недостаточное внимание документации. б) Преимущество: высокая предсказуемость. Недостаток: сложность адаптации к изменениям. в) Преимущество: строгий контроль над каждым этапом. Недостаток: ограничение гибкости. г) Преимущество: формальное планирование. Недостаток: ограниченное взаимодействие с заказчиком.
5. Какие типы задач обычно можно увидеть на канбан доске и каким образом они отслеживаются? а) Только задачи разработки. Отслеживаются с помощью тегов. б) Задачи всех этапов разработки. Отслеживаются перемещением между столбцами. в) Задачи только по тестированию. Отслеживаются по названию. г) Задачи только по документации. Отслеживаются с помощью вложенных комментариев.
6. Что такое «daily stand-up» (ежедневное совещание) в методологии Scrum? Какие вопросы обычно обсуждаются на нем? а) Это встреча разработчиков после завершения проекта. Обсуждаются детали реализации. б) Это встреча для обсуждения долгосрочных планов развития проекта. в) Это ежедневное совещание для синхронизации команды. Обсуждаются ответы на три вопроса: «Что было сделано?», «Что будет сделано?», «Есть ли препятствия?». г) Это еженедельная встреча с заказчиком для сдачи результатов работы.
7. Какие из следующих характеристик являются ключевыми для эффективной работы в команде? а) Отсутствие коммуникации между членами команды. б) Стремление к индивидуальной работе

без взаимодействия. в) Регулярные конфликты и отсутствие обратной связи. г) Открытое общение, сотрудничество, разделение знаний и опыта.

**Ответы:**

1. в)
2. б)
3. в)
4. а)
5. б)
6. в)
7. г)

## Знакомство с платформами размещения исходного кода программ на примере GitFlick

---

### 5.1 Знакомство с платформами размещения исходного кода программ на примере Gitflick

В современном мире разработки программного обеспечения существует множество платформ, предоставляющих разработчикам возможность хранить, управлять и совместно работать над исходным кодом своих проектов. Выбор платформы для работы с кодом зависит от потребностей проекта, предпочтений команды разработчиков и ряда других факторов. Важно выбрать такую платформу, которая наилучшим образом соответствует целям вашего проекта и предоставляет необходимые инструменты для успешного сотрудничества.

Существует множество платформ для размещения и совместной работы над исходным кодом программ. Наиболее известные из них представлены в таблице:

Название платформы	Описание
GitHub	Один из самых популярных и распознаваемых ресурсов, предоставляющих возможность хранения, управления и совместной работы над кодом. Поддерживает систему контроля версий Git и предоставляет широкий набор инструментов для управления проектами.
GitLab	Сервис и самостоятельное программное обеспечение для управления исходным кодом с возможностью самостоятельной установки. Предоставляет инструменты для CI/CD, управления задачами и другими аспектами разработки.
Bitbucket	Платформа, поддерживающая Git и Mercurial, предлагает бесплатные и платные варианты. Интегрируется с другими инструментами разработки.
Azure DevOps (Visual Studio Team Services)	Набор инструментов от Microsoft для разработки ПО, включая хостинг исходного кода, CI/CD, отслеживание ошибок и другие функции.
SourceForge	Одна из старейших платформ для хостинга проектов с открытым исходным кодом. Предоставляет хостинг, систему контроля версий и инструменты для совместной работы.
Launchpad	Платформа от Canonical, предоставляющая хостинг и систему контроля версий для проектов с открытым исходным кодом. Интегрируется с инструментами Ubuntu.
Gitflick	Эта отечественная платформа обеспечивает хостинг исходного кода, систему контроля версий Git и инструменты для совместной работы над проектами, а также уделяет особое внимание российским разработчикам.

Одной из таких платформ является GitFlick, которая объединяет возможности системы контроля версий Git с удобством облачных сервисов. Близким по функциональности с популярной платформой GitHub. Как и на GitHub, команды совместно разрабатывают проекты, а обширный набор инструментов обеспечивает эффективное взаимодействие и отслеживание изменений. Давайте рассмотрим подробнее, как GitFlick и аналогичные платформы упрощают жизнь команд разработчиков, позволяют следить за изменениями в проектах, коммуницировать в команде и создавать инновационные продукты.

Рассмотрим основные рекомендации при работе над проектом в команде

### 5.1.1 Знакомство с отечественным сервисом GitFlic



В начале 2010-х годов, когда глобальные платформы для разработки и управления кодом уже были популярны, российское сообщество разработчиков осознало, что национальная аудитория также нуждается в удобном и адаптированном решении. Местные особенности, языковые и культурные нюансы, а также желание обеспечить более высокую безопасность данных для российских компаний и команд стали ключевыми факторами для создания собственного сервиса.

GitFlic — это сервис хранения и обмена кодом, созданный командой разработчиков из России. Он предоставляет пользователям возможность сохранять свой код в облаке, обмениваться им с другими пользователями и получать доступ к нему без необходимости оплаты. GitFlic позиционирует себя как российский аналог популярного западного хранилища кода GitHub, но при этом обладает рядом уникальных функций и возможностей.

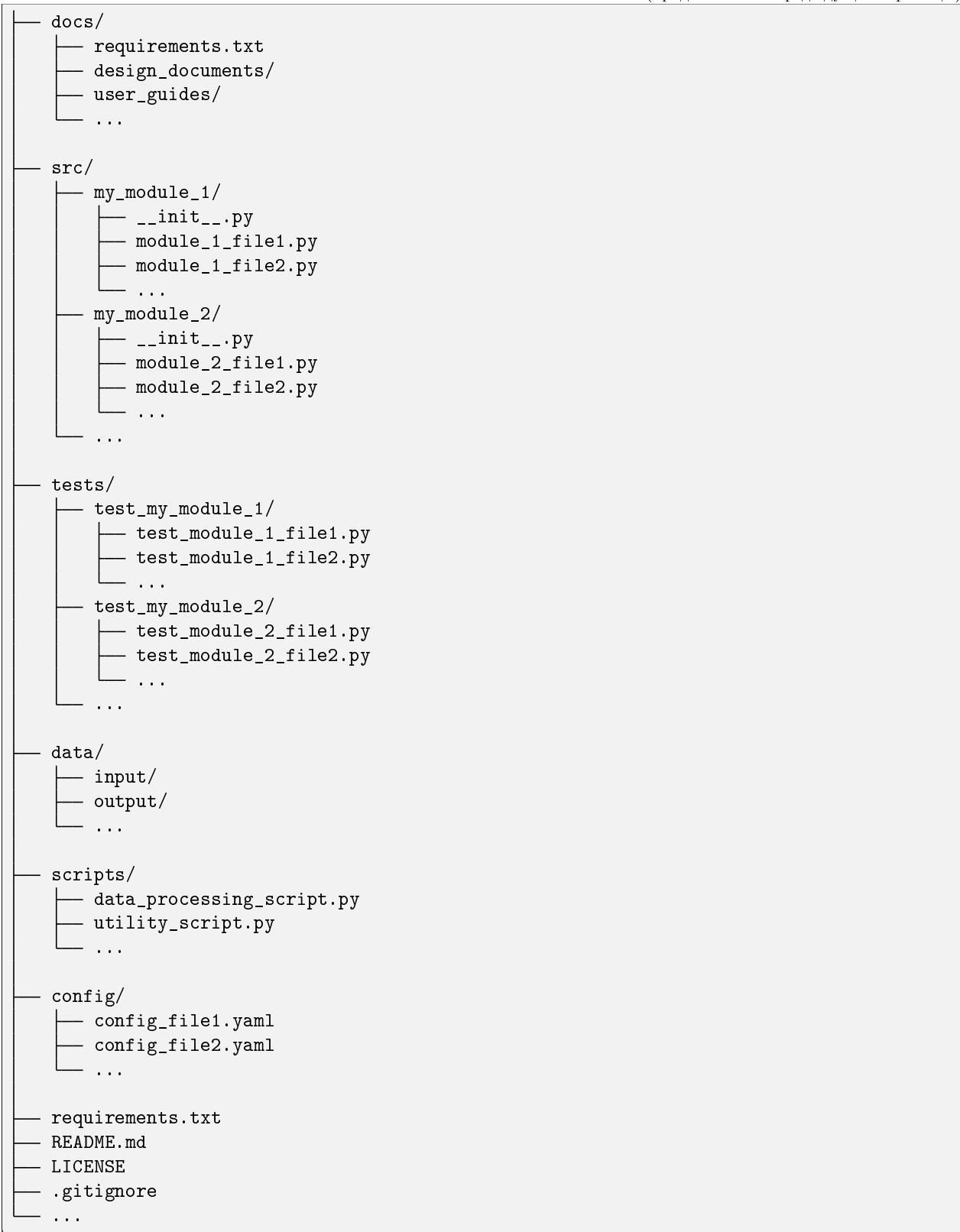
Одной из главных особенностей GitFlic является его безопасность. Сервис использует шифрование SSL/TLS для защиты передаваемых данных и имеет систему авторизации пользователей, которая обеспечивает высокий уровень безопасности. Кроме того, GitFlic предоставляет возможность создавать закрытые ветки кода, что позволяет разработчикам работать над проектами совместно и делиться своими наработками с другими участниками сообщества. Одним из преимуществ сервиса GitFlic является наличие документации и инструкций на русском языке, что облегчает знакомство с платформой.

Перед тем как приступить к работе с репозиторием и браться за выполнение задач команде стоит разработать и обсудить структуру проекта. Как будут располагаться различные файлы и компоненты проекта. Ниже представлена рекомендуемая структура директории для командной разработки проекта на Python. Предложенный вариант позволяет наядно оценить масштаб задач и структурировать различные элементы. Также это позволяет решить многие вопросы. Куда загружать данные? Куда записать промежуточные результаты? Где хранить тесты и документацию, где искать скрипты для автоматизации рутинных задач. Предложенный вариант является не более чем рекомендацией для размышления. Ведь структура проекта зависит от его тематики и предпочтений команды.

```
my_project/
|
```

(continues on next page)

(продолжение с предыдущей страницы)



Объяснение структуры:

- **docs/**: Здесь хранятся документация, связанная с проектом, включая требования, дизайн-документы, руководства пользователя и др.
- **src/**: Это место для всех исходных файлов вашего проекта. Рекомендуется организовать код в модули и пакеты.
- **tests/**: Директория для тестовых файлов. Каждый модуль следует сопровождать соответствующими тестами.
- **data/**: Здесь хранятся входные и выходные данные, используемые или генерируемые вашим приложением.
- **scripts/**: Если вам нужно включить дополнительные сценарии обработки данных или утилиты, это место для них.
- **config/**: Содержит конфигурационные файлы, которые управляют поведением приложения.
- **requirements.txt**: Список зависимостей для вашего проекта.
- **README.md**: Описание проекта, его назначение, инструкции по установке, использованию и вкладу в разработку.
- **LICENSE**: Файл с лицензией вашего проекта.
- **.gitignore**: Файл, указывающий Git'у, какие файлы и директории нужно игнорировать при коммите.

Эта структура помогает организовать код и ресурсы вашего проекта так, чтобы разработка и совместная работа были более структурированными и эффективными.

### 5.1.2 Типовой процесс совместной работы с кодом

Мы познакомились с платформами для работы с кодом, обсудили что нужно учитывать при разработке структуры проекта, пора перейти к работе с Git. Рассмотрим сценарий командной работы трех разработчиков, которые используют git для работы с платформой, например GitFlic. Они работают над проектом и каждый разработчик вносит изменения по своей задаче. Посмотрим, какие команды git используют разработчики и в какой последовательности. Они сначала создадут форк исходного проекта и будут выполнять коммиты по мере завершения работы над задачами. Каждый в свою версию проекта, а затем после внесения своих доработок отправят pull request в исходный репозиторий чтоб объединить свои части кода. Опишем каждое действие разработчиков используя команды Git при работе с удаленным репозиторием.

Аналогично можно использовать любую другую платформу для работы с кодом поддерживающую git, например, GitHub.

#### 1. Инициализация проекта:

- Разработчик 1: Создает пустой репозиторий на GitFlic (назовем его «myproject»).

```
git init
git remote add origin https://gitflic.ru/yourusername/myproject.git
```

- Разработчик 2 и Разработчик 3: Форкают репозиторий «myproject» и получают копии в своих аккаунтах на GitFlic.

#### 2. Клонирование репозиториев:

- Разработчик 1: Клонирует репозиторий «myproject» на свой локальный компьютер.

```
git clone https://gitflic.ru/yourusername/myproject.git
```

- Разработчик 2 и Разработчик 3: Клонируют свои форки репозитория «myproject» на свои локальные компьютеры.

```
git clone https://gitflic.ru/developer2/myproject.git  
git clone https://gitflic.ru/developer3/myproject.git
```

### 3. Разработка задач:

- Разработчик 1: Работает над первой задачей и вносит изменения в свой локальный репозиторий.

```
git add .  
git commit -m "Added feature 1"
```

- Разработчик 2: Работает над второй задачей и вносит изменения в свой локальный репозиторий.

```
git add .  
git commit -m "Fixed bug in feature 2"
```

- Разработчик 3: Работает над третьей задачей и вносит изменения в свой локальный репозиторий.

```
git add .  
git commit -m "Implemented enhancement 3"
```

### 4. Обновление форков:

- Разработчик 2 и Разработчик 3: Регулярно синхронизируют свои форки с оригинальным репозиторием «myproject».

```
git pull upstream master
```

### 5. Отправка изменений на сервер:

- Разработчик 1: Отправляет свои изменения на GitFlic.

```
git push origin master
```

- Разработчик 2 и Разработчик 3: Также отправляют свои изменения на свои форки на GitFlic.

### 6. Pull Request:

- Разработчик 2 и Разработчик 3: Создают Pull Request из своих форков в оригинальный репозиторий «myproject».
- Разработчик 1: Получает уведомления о созданных Pull Request'ах и проводит ревью изменений.

### 7. Ревью и слияние изменений:

- Разработчик 1: Оставляет комментарии и проводит ревью кода в Pull Request'ах Разработчика 2 и Разработчика 3.
- Разработчик 2 и Разработчик 3: Вносят изменения по комментариям и обсуждают с Разработчиком 1.
- Разработчик 1: Сливает (мерджит) Pull Request'ы Разработчика 2 и Разработчика 3 в основную ветку проекта.

Таким образом, каждый разработчик работает над своей задачей, использует Git для управления версиями кода и взаимодействия с командой, и создает Pull Request'ы для объединения изменений в основной репозиторий.

В общем виде это выглядит весьма понятно, однако перед началом работы команде следует договориться о том как они будут называть и оформлять ветки, как часто делать коммиты и что писать в комментариях чтобы другие участники могли самостоятельно разобраться в чужом коде без привлечения автора. Эти и многие вопросы, были исторически решены и сформированы в так называемый «Best Practices» состоящий в виде списка рекомендаций, которого придерживаются большинство разработчиков. Конечно в каждой команде он может быть свой хотя по большей части по смыслу они совпадают.

Список «Best Practices» для эффективной работы команды разработчиков использующих Git представлен в таблице:

Best Practice	Description
Использование стандартов именования веток, коммитов, тегов и других элементов.	Договоритесь о стандартах именования веток, коммитов, тегов и других элементов. Допустим, у нас есть команда разработчиков, которая работает над проектом по созданию онлайн-магазина. Для того чтобы следовать совету о использовании однообразных стилей именования, команда договорилась о следующих правилах: Для каждой новой функциональности или задачи создается отдельная ветка. Название ветки состоит из краткого описания задачи, разделенного дефисами. Например, feature-add-checkout-button для ветки, в которой добавляется кнопка оформления заказа. Каждый коммит содержит осмысленное и краткое сообщение, начинающееся с глагола в настоящем времени. Сообщение описывает, что именно было сделано в этом коммите. Например, «Добавляет функцию корзины для товаров». Для обозначения версий релизов используется семантическое версионирование. Теги имеют формат X.Y.Z, где X - мажорная версия, Y - минорная версия, и Z - патч-версия. Например, 1.0.0 для первого релиза. Это улучшает читаемость и понимание истории проекта. Таким образом, разработчики могут легко разобраться в назначении и состоянии веток, коммитов и тегов, что способствует пониманию работы над проектом. Также, способствует лучшей навигации по коду проекта. Можно быстро найти нужные ветки или коммиты, так как структура именования следует определенному формату.
Маленькие коммиты	Каждый коммит должен решать конкретную задачу. Это значит что в коде не должно быть недописанных кусков кода, которые нельзя проверить или оценить их качество. Если вы напишете функцию, класс или модифицируете что-то из того что было ранее и ваша вкладка готова к стадии тестирования, значит пора отправлять коммит. Это облегчает ревью кода и понимание изменений. Отправлять одним коммитом изменения касающиеся различных частей кода это плохая практика. Такого стоит избегать. Просто возьмите себе за правило: как только вы решили что задача выполнена, перед тем как браться за что-то другое отправьте коммит. Зафиксируйте результат. Это улучшает читаемость и понимание истории проекта.
Ветвление и слияние функций	Используйте ветки для каждой новой функциональности или задачи. Это позволяет изолировать изменения и упрощает слияние. Когда команда разработчиков работает над проектом, который включает множество функциональных задач или новых возможностей, хороший практикой является создание отдельных веток для каждой функциональности или задачи. Это позволяет изолировать изменения, связанные с конкретной задачей, от остального кода, что облегчает тестирование, ревью и интеграцию. Ветки также предотвращают вмешательство изменений в процесс разработки других членов команды и упрощают слияние изменений обратно в основную ветку. Когда задача завершена и готова для интеграции, ветка может быть слита с основной веткой с минимальными конфликтами.
Регулярное обновление новой ветки	Важно держать основную ветку (например, main или master) актуальной с последними изменениями. Используйте merge (слияния) или rebase (перебазирование).
Внедрение рецензий	Практика обязательного ревью кода перед вливанием изменений в основную ветку улучшает качество кода. Практика обязательного ревью кода означает, что перед тем как изменения вливаются в основную ветку (например, main или master), другие члены команды анализируют и оценивают предложенные изменения. В основном эту задачу выполняют опытные разработчики, которые достаточно хорошо знакомы с текущим проектом. Они могут выявить возможные недостатки кода и дать хороший фидбек (обратную связь автору кода), с пояснениями принятого решения и рекомендациями. Это не только улучшает качество кода.
Новая глава	5. <b>Знакомство с платформами размещения исходного кода программ на примере GitHub</b>

Представленные рекомендации могут отличаться в зависимости от специфики проекта и команды. Следование предложенными рекомендациям поможет обеспечить более эффективную и согласованную работу над проектами с использованием Git. Эти практики обеспечивают более структурированный и эффективный процесс разработки, упрощают совместную работу над кодом и могут повысить качество и надежность вашего проекта.

В рамках курса мы уже упоминали популярную платформу GitHub, которая является одной из самых крупных площадок для работы с кодом и развития Open Source. Она была основана в апреле 2008 года Томом Престон-Вернером и Крисом Ван Дамом. В том же 2008 году компанией Atlassian был создан схожий по функционалу сервис Bitbucket. Ориентированный на более широкий спектр инструментов для разработчиков, включая хостинг Git-репозиториев, системы отслеживания ошибок и интеграцию с другими продуктами Atlassian, такими как Jira и Confluence.

За годы своего существования GitHub стал ключевой платформой для open source сообщества, стартапов и больших корпораций. Однако 26 октября 2018 года была завершена сделка по приобретению платформы GitHub компанией Microsoft. Это означает, что начиная с этой даты GitHub попал под влияние компании Microsoft, что неоднозначно воспринялось участниками движения Open Source.

В целях соблюдения мер безопасности и конфиденциальности данных крупные IT компании используют собственные платформы для работы с кодом (аналогичные GitHub, GitLab) расположенные на собственных вычислительных мощностях. Разумеется что доступ к этим платформам имеют только часть сотрудников компании и никакие сторонние лица не могут размещать там свои Open Source проекты. Однако в России есть своя доступная платформа для работы с кодом - [GitFlick](#).

### **5.1.3 Задание для команды разработчиков: Создание чат-бота для мессенджера Telegram с использованием GitFlick**

**Цель:** Создать чат-бота для мессенджера Telegram, который будет предоставлять информацию о ближайших хакатонах и олимпиадах для школьников и студентов, справочную информацию о команде разработчиков, а также возможность выполнять поиск справочной информации из открытых источников.

**Задачи:**

**1. Создание GitFlick проекта и репозитория:**

- Каждый разработчик должен создать учетную запись на GitFlick.
- Один из разработчиков создает новый проект на GitFlick и приглашает остальных участников команды.

**2. Разработка чат-бота:**

- Разработчики создают структуру проекта для чат-бота.
- Реализуют функцию выдачи списка ближайших хакатонов и олимпиад.
- Разрабатывают механизм предоставления справочной информации о команде разработчиков.
- Добавляют кнопки и пункты меню для удобства навигации в интерфейсе чат-бота.

**3. Интеграция с Telegram API:**

- Разработчики настраивают интеграцию с Telegram API для взаимодействия с мессенджером.

**4. Реализация поиска справочной информации:**

- Интегрировать API для поиска информации из открытых источников (например, Wikipedia или другие источники).

- Реализовать возможность поиска по заданному запросу.

**5. Тестирование и отладка:**

- Каждый участник команды тестирует свои функции и исправляет возникающие ошибки.
- Обеспечить работоспособность и стабильность чат-бота.

**6. Документация:**

- Подготовить описание функций чат-бота и инструкции по его использованию.

**Сроки:**

- Создание проекта и репозитория: [Указать срок]
- Разработка чат-бота: [Указать срок]
- Интеграция с Telegram API: [Указать срок]
- Реализация поиска справочной информации: [Указать срок]
- Тестирование и отладка: [Указать срок]
- Документация: [Указать срок]

**Результат:**

- Реализованный и работоспособный чат-бот, размещенный на GitFlick, с функциями выдачи информации о хакатонах, олимпиадах, справочной информации о команде и возможностью поиска.

**Примечание:** Для эффективной организации работы предлагаем разбиться на команды по 3-5 человек, чтобы вы могли совместно принимать решения данной задачи. Используйте методику SCRUM и канбан-доску для формирования списка задач и их распределения в команде. Это позволит вам сформировать процесс управления разработкой ИТ продукта. Также рекомендуется изучить возможности платформы GitFlick для управления задачами, ведения обсуждений и отслеживания прогресса.

#### **5.1.4 Заключение**

В данном блоке мы постарались кратко познакомиться вас с основными элементами командной работы на ИТ проектом: От начальных этапов планирования и определения ролей, до финального слияния кода и представления результатов, мы рассказали о ключевых шагах и важных уроках, которые вам помогут при выполнении совместной работы. Конечно есть и много других тем которые не были представлены в данном модуле, однако все они являются ветвями развития представленных базовых основ. Надеемся что вам удалость:

- Ознакомиться с принципами формирования эффективной команды разработчиков.
- Понять, как распределение ролей и задач влияет на процесс работы над проектом.
- Исследовать методы планирования и организации проекта в команде.
- Узнать о инструментах и технологиях, которые облегчают командную разработку.
- Погрузиться в аспекты совместной работы, решения конфликтов и обмена знаний.

Мы надеемся, что этот рассказ о командной работе станет вдохновением для тех, кто стремится к профессиональному росту и успешной совместной разработке. Надеемся что уроки, которые были представлены в этом модуле, помогут вам на пути к развитию навыков командной работы и созданию качественных технологических продуктов.

---

### 5.1.5 Справочная информация

В этом блоке представлена справочная информация которая может послужить хорошим материалом чтоб погрузиться в темы «Командная работы» и «Управление проектами». Представлен краткий анализ и сравнение различных методик управления проектами, а также некоторые подробности этапов их реализации. Также, здесь собраны краткие исторические факты и некоторые теоретические дополнения которые помогут найти ответы на некоторые вопросы появившиеся при изучении основного материала.

Итак, погрузимся в Git и его возможностей для управления кодом в командной разработке. Освоив эти навыки, вы сможете стать более продуктивным участником команды разработки и добиться более успешных результатов в совместных проектах.

Git является мощным инструментом для командной работы над проектами разработки ПО и управления версиями кода. Он позволяет разработчикам совместно работать над кодом, отслеживать изменения, управлять версиями и координировать работу в больших командах. Вот как Git позволяет улучшить работу команды:

- Ветвление и слияние (Branching and Merging):** Git позволяет создавать ветки для различных задач, функциональности или исправлений. Каждый разработчик может работать над своей собственной веткой, изолированной от основного кода. По завершении работы ветки могут быть слиты (слияние) в основную ветку. Это позволяет избежать конфликтов и упрощает одновременную работу над разными частями проекта.
- Pull Requests (Запросы на слияние):** В платформах с поддержкой PR (Pull Requests), таких как GitHub, GitLab и др., разработчики могут создавать PR для обсуждения и рецензии изменений. Это позволяет другим разработчикам ознакомиться с кодом, оставить комментарии и предложить изменения. Эффективное средство для обсуждения и улучшения кода.
- Рецензирование кода (Code Review):** Это важная часть разработки, которая помогает обнаруживать ошибки, улучшать качество кода и обмениваться знаниями в команде. Git облегчает этот процесс через комментирование изменений в PR и возможность просмотра и обсуждения кода.
- Управление конфликтами (Conflict Resolution):** При слиянии изменений может возникнуть конфликт, когда одни и те же строки кода были изменены в разных ветках. Git предоставляет инструменты для разрешения таких конфликтов, позволяя разработчикам вручную выбирать, какие изменения следует сохранить.
- История изменений и откат (History and Reverting):** Git хранит историю всех изменений в коде. Если что-то идет не так, вы можете вернуться к предыдущему состоянию проекта. Это дает команде уверенность в том, что изменения всегда могут быть отменены, если они приведут к проблемам.
- Улучшенное управление проектами:** Платформы с поддержкой Git (например, GitHub, GitFlic, Bitbucket) предоставляют инструменты для создания задач (issues), организации проектов (projects), автоматизации сборки и развертывания (CI/CD) и многие другие функции, которые помогают управлять проектами.
- Гибкость и распределенность:** Git позволяет разработчикам работать над проектами независимо, даже без постоянного доступа к сети. Это особенно важно для удаленных команд и ситуаций, когда необходимо работать вне офиса.

## **Стандарты и протоколы**

Если вам хочется более подробно разобраться в теме стандартов и протоколов, познакомиться с процессом их разработки - вы можете поискать информацию об институтах стандартизации. Вот пара примеров:

1. Институт инженеров электротехники и электроники (Institute of Electrical and Electronics Engineers, IEEE) — некоммерческая инженерная ассоциация из США, разрабатывающая широко применяемые в мире стандарты по радиоэлектронике, электротехнике и аппаратному обеспечению вычислительных систем и сетей.
2. Международная электротехническая комиссия (МЭК; англ. International Electrotechnical Commission, IEC — международная некоммерческая организация по стандартизации в области электрических, электронных и смежных технологий. Некоторые из стандартов МЭК разрабатываются совместно с Международной организацией по стандартизации.

## **Краткая история развития популярных платформ для хостинга и совместной разработки кода с использованием Git:**

1. SourceForge (1999): Одна из первых платформ для размещения и совместной разработки открытого кода. Впоследствии SourceForge приобрела большую популярность, хотя в последние годы она уступила место новым платформам.
2. Launchpad (2005): Создан Canonical для хостинга проектов с открытым исходным кодом, включая проект Ubuntu. Помимо Git, поддерживает систему контроля версий Bazaar.
3. GitHub (2008): Основан Томом Престон-Вернером и Крисом Ван Дамом. Стал одной из самых популярных платформ для хостинга Git-репозиториев, обеспечивая удобное совместное программирование, инструменты отслеживания задач и обсуждение изменений. В 2018 году GitHub был приобретен Microsoft.
4. Bitbucket (2008): Создана компанией Atlassian. Ориентирована на более широкий спектр инструментов для разработчиков, включая хостинг Git-репозиториев, системы отслеживания ошибок и интеграцию с другими продуктами Atlassian, такими как Jira и Confluence.
5. GitLab (2011): Запущен Dmitriy Zaporozhets и Валерий Сирожин. GitLab предлагает как облачное решение, так и версию для самостоятельного развертывания на сервере. Это интегрированная платформа с инструментами DevOps, системой непрерывной интеграции и развертывания (CI/CD).

## **5.2 Задание для команды разработчиков: Создание чат-бота для мессенджера Telegram с использованием GitFlick**

**Цель:** Создать чат-бота для мессенджера Telegram, который будет предоставлять информацию о ближайших хакатонах и олимпиадах для школьников и студентов, справочную информацию о команде разработчиков, а также возможность выполнять поиск справочной информации из открытых источников.

**Задачи:**

1. Создание GitFlick проекта и репозитория:

- Каждый разработчик должен создать учетную запись на GitFlick.
- Один из разработчиков создает новый проект на GitFlick и приглашает остальных участников команды.

**2. Разработка чат-бота:**

- Разработчики создают структуру проекта для чат-бота.
- Реализуют функцию выдачи списка ближайших хакатонов и олимпиад.
- Разрабатывают механизм предоставления справочной информации о команде разработчиков.
- Добавляют кнопки и пункты меню для удобства навигации в интерфейсе чат-бота.

**3. Интеграция с Telegram API:**

- Разработчики настраивают интеграцию с Telegram API для взаимодействия с мессенджером.

**4. Реализация поиска справочной информации:**

- Интегрировать API для поиска информации из открытых источников (например, Wikipedia или другие источники).
- Реализовать возможность поиска по заданному запросу.

**5. Тестирование и отладка:**

- Каждый участник команды тестирует свои функции и исправляет возникающие ошибки.
- Обеспечить работоспособность и стабильность чат-бота.

**6. Документация:**

- Подготовить описание функций чат-бота и инструкции по его использованию.

**Сроки:**

- Создание проекта и репозитория: [Указать срок]
- Разработка чат-бота: [Указать срок]
- Интеграция с Telegram API: [Указать срок]
- Реализация поиска справочной информации: [Указать срок]
- Тестирование и отладка: [Указать срок]
- Документация: [Указать срок]

**Результат:**

- Реализованный и работоспособный чат-бот, размещенный на GitFlick, с функциями выдачи информации о хакатонах, олимпиадах, справочной информации о команде и возможностью поиска.

**Примечание:** Для эффективной организации работы предлагаем разбраться на команды по 3-5 человек, чтобы вы могли совместно принимать решения данной задачи. Используйте методику SCRUM и канбан-доску для формирования списка задач и их распределения в команде. Это позволит вам сформировать процесс управления разработкой ИТ продукта. Также рекомендуется изучить возможности платформы GitFlick для управления задачами, ведения обсуждений и отслеживания прогресса.

### **5.3 Вопросы**

1. Какая практика разработки с использованием Git помогает изолировать изменения и упрощает слияние кода? а) Регулярное обновление основной ветки. б) Использование .gitignore для игнорирования файлов. в) Ветвление по функциональности. г) Автоматизация сборки и тестирования.
2. Какой инструмент помогает обнаруживать и устранять ошибки в коде до его слияния в основную ветку? а) Взаимодействие и ревью кода. б) Использование тегов для релизов. с) Разработка с помощью комментариев. д) Поддержание резервных копий кода.
3. Какой файл используется для игнорирования временных файлов, зависимостей и других ненужных элементов в системе контроля версий? а) .gitconfig б) .gitignore в) .gitcommit г) .gitbranch
4. Для чего используются теги в системе контроля версий Git? а) Для создания резервных копий кода. б) Для автоматической сборки и тестирования. в) Для обозначения версий релизов. г) Для отправки кода на удаленный сервер.
5. Какой принцип работы с Git помогает понимать историю изменений и восстанавливать контекст? а) Ветвление по функциональности. б) Взаимодействие и ревью кода. в) Регулярное обновление основной ветки. г) Использование коммитов и их комментариев.

**Ответы:**

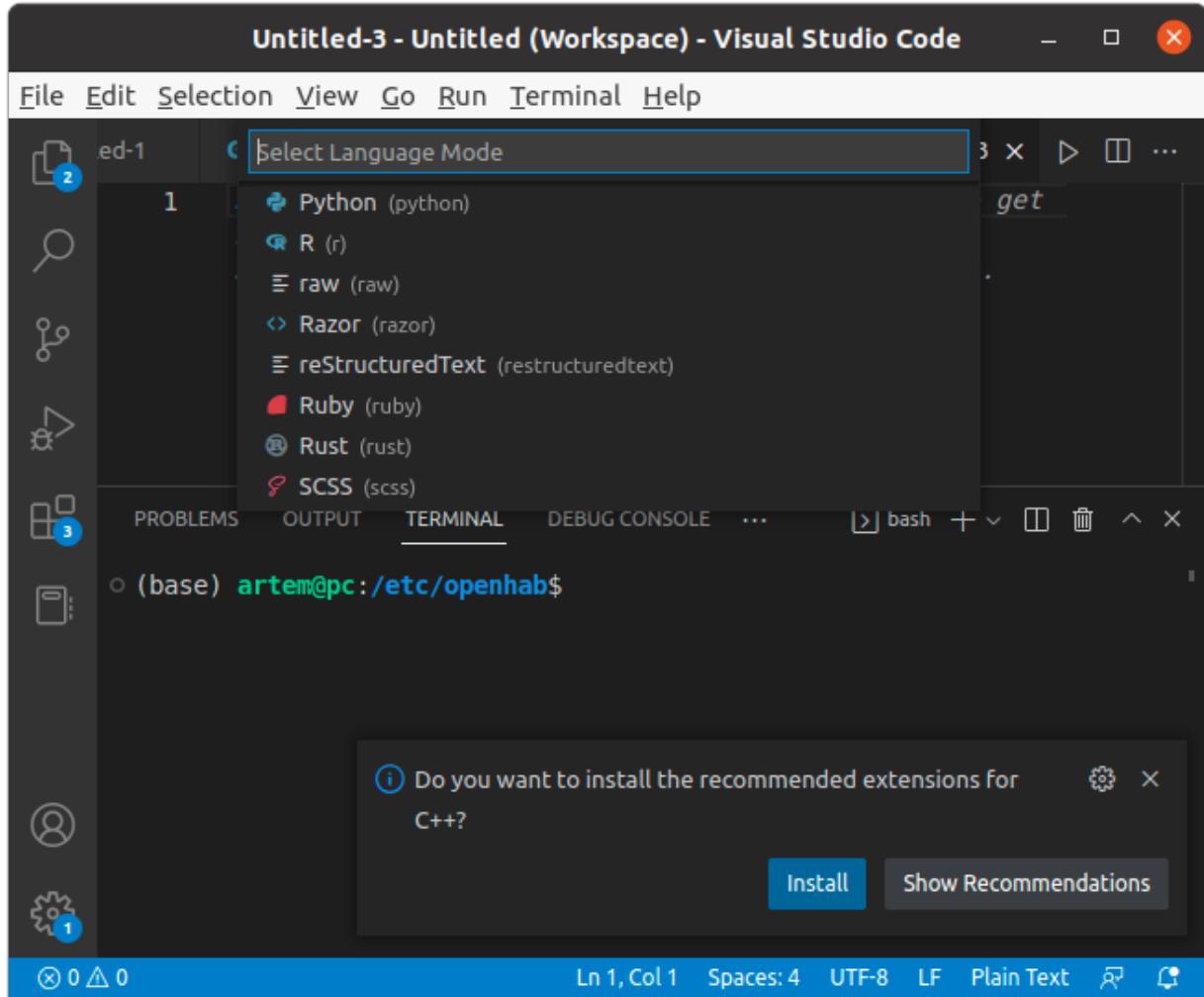
1. в)
2. а)
3. б)
4. в)
5. г)

## Среда разработки

---

### 6.1 Среда разработки

**Среда разработки** или, если дословно с английского, интегрированная среда разработки (Integrated Development Environment, IDE) — это приложение, которое позволяет разрабатывать код более эффективно, чем в текстовом редакторе, например [Kate](#) или [Notepad++](#). Среда разработки объединяет в себе несколько функций, которые обычно разработчики хотят в ней увидеть: автоматизация написания кода, подсветка синтаксиса, автоматическое дополнение кода, форматирование, рефакторинг, компиляция, сборка и отладка. Также среды разработки поддерживают основные инструменты разработчиков в виде дополнений, так, можно подключить плагины, которые позволяют пользоваться [git](#) и прогонять тесты прямо из среды разработки. Сейчас сложно найти среды разработки, которые имеют поддержку только одного языка программирования. Существуют среды разработки, специализирующиеся на одном языке программирования для серверных приложений (backend), такие как [PyCharm](#), но даже в них есть [поддержка JavaScript, HTML](#) и других инструментов для разработки приложений пользователя (frontend). Это необходимо, так как есть понятие fullstack разработки, и переключаться между средами для разработки отдельно серверной и пользовательской частей было бы неудобно. В таких средах разработки, как [VisualStudioCode](#) и [Eclipse](#), существует поддержка, наверное, всех возможных языков программирования, языков разметки, форматов сериализации данных и так далее. Поддержка устанавливается как дополнение, при этом визуальный интерфейс остается без изменений, что положительно влияет на разработку, так как мы не тратим время на привыкание к другому интерфейсу.



### 6.1.1 Подсветка синтаксиса

Подсветка синтаксиса — достаточно распространенная функция, поддерживающаяся даже в текстовых редакторах, которые нельзя отнести к IDE. Подсветка синтаксиса использует свойство человеческого глаза, который лучше (быстрее) воспринимает цвет, а не смысл написанного. Поэтому идея подсветки кода в том, чтобы разделить код на сущности, которые принадлежат к одним группам (функции к функциям, переменные к переменным) и подсветить их одним цветом.

The screenshot shows the Visual Studio Code interface. In the center, there is a code editor with the following Python code:

```
//ST_CP to pin 12 Untitled-2 • from pathlib import PurePath Untitled-3 - Untitled (Workspace) - Visual Studio Code
from pathlib import PurePath
class SmartPath(Path):
    json_counter = 0
    def __init__(self, root):
        pass
```

Below the code editor is a terminal window showing the command:

```
o (base) artem@pc:/etc/openhab$
```

A modal dialog box is displayed in the center of the screen, asking:

Do you want to install the recommended extensions for C++?

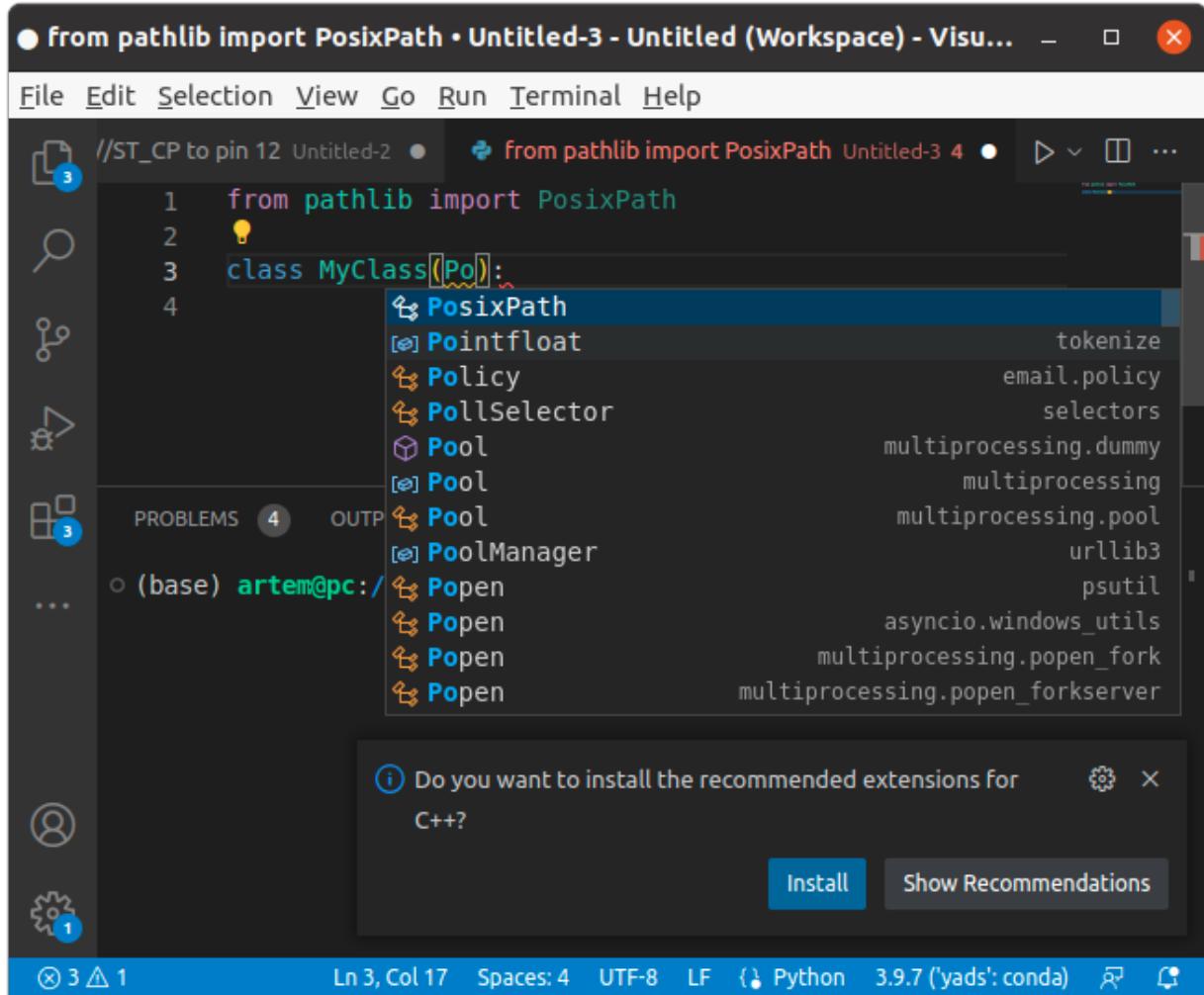
With two buttons at the bottom:

- Install
- Show Recommendations

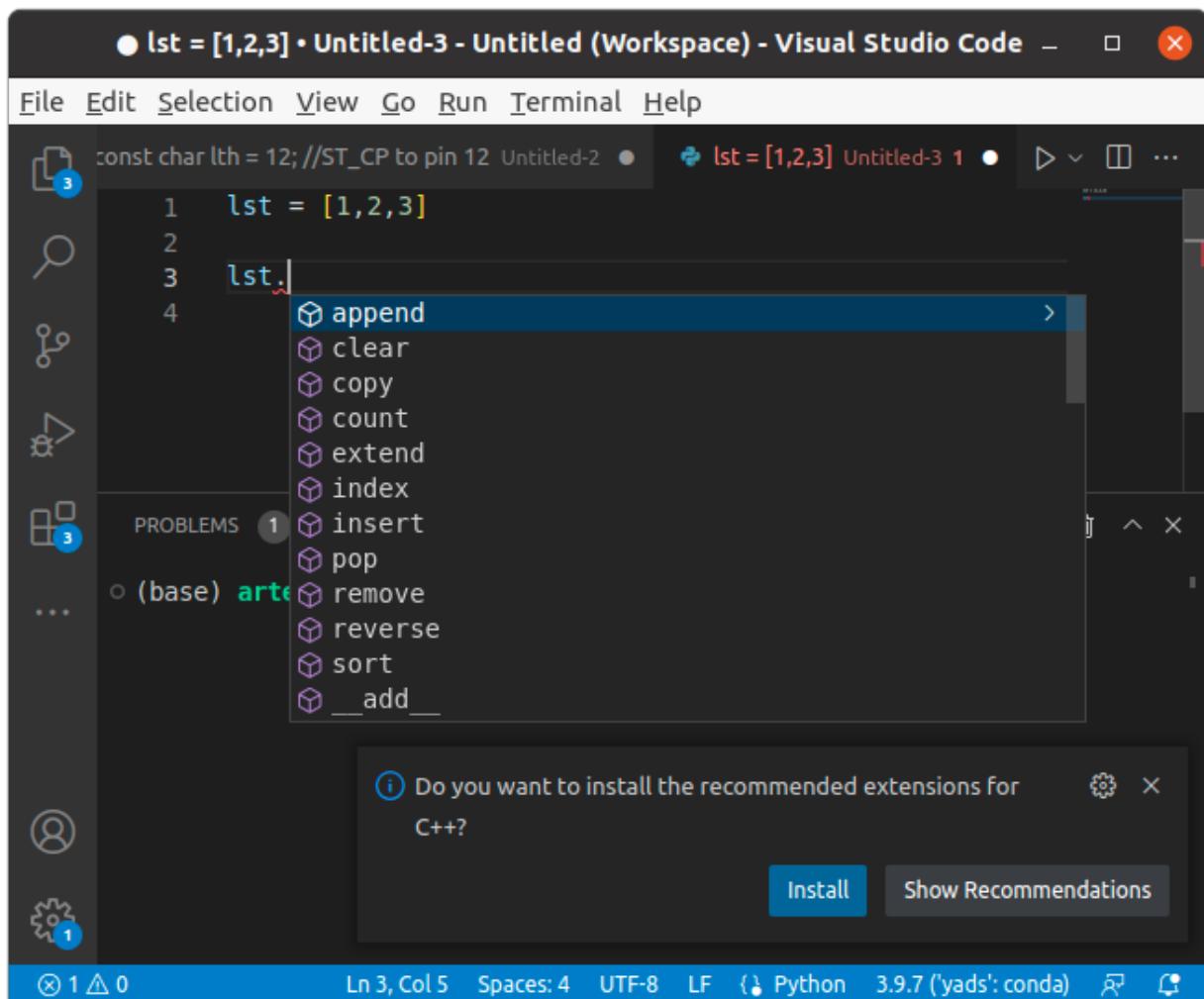
At the bottom of the interface, there are status indicators: 0△1, Ln 9, Col 5, Spaces: 4, UTF-8, LF, { Python 3.9.7 ('yads': conda), and icons for file operations.

### 6.1.2 Автодополнение кода (завершение кода)

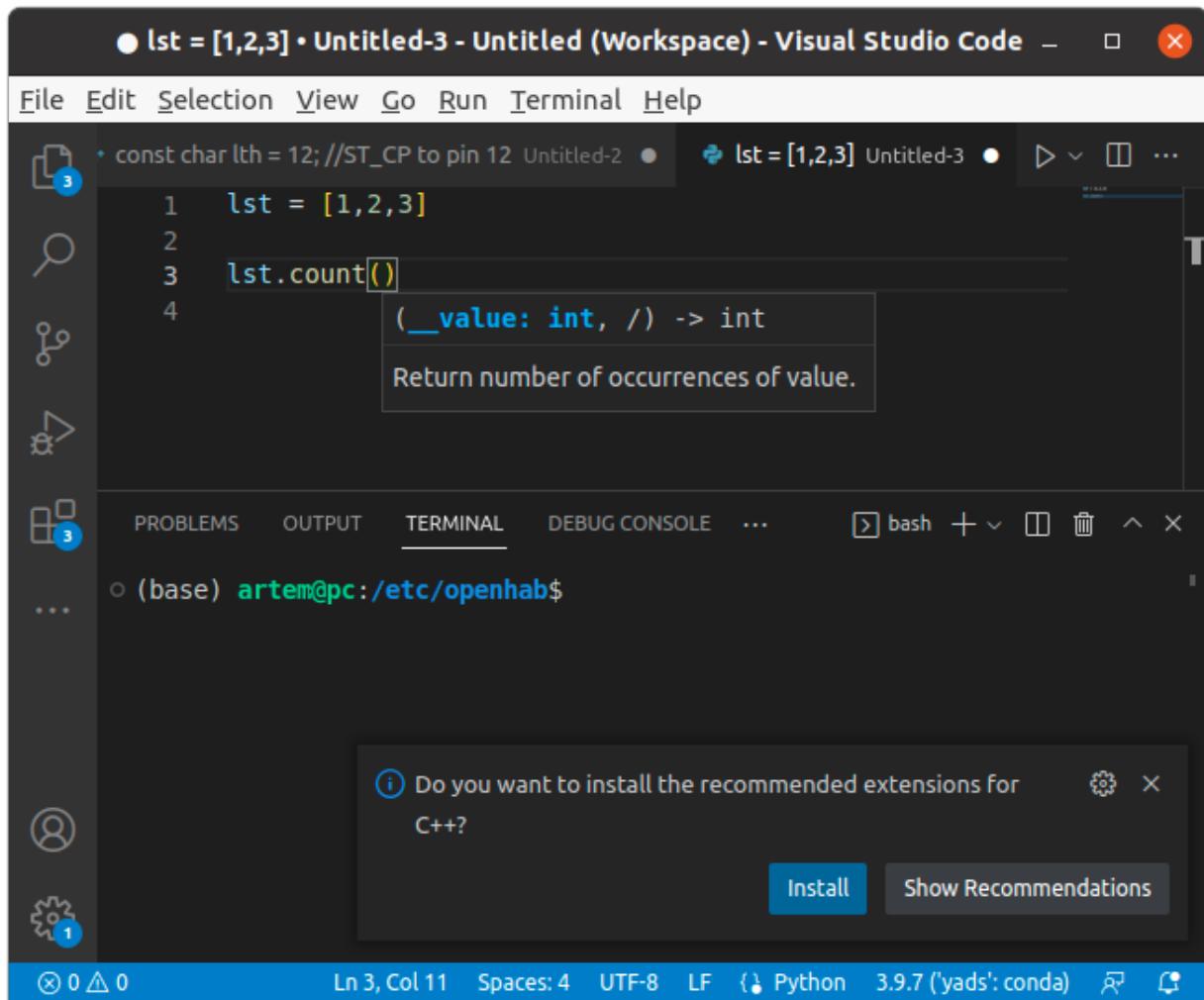
Среда разработки понимает, какие имена переменных, функций и классов доступны в данном приложении, и использует их, чтобы разработчик мог выбрать подходящий объект при наборе неполного имени:



Автодополнение бывает не особо полезно, когда мы знаем название объекта, часто написать его быстрее, чем выбрать из списка. Но если мы забыли название метода или поля класса, мы можем легко его вспомнить, если увидим название в списке, без вызова документации по классу.

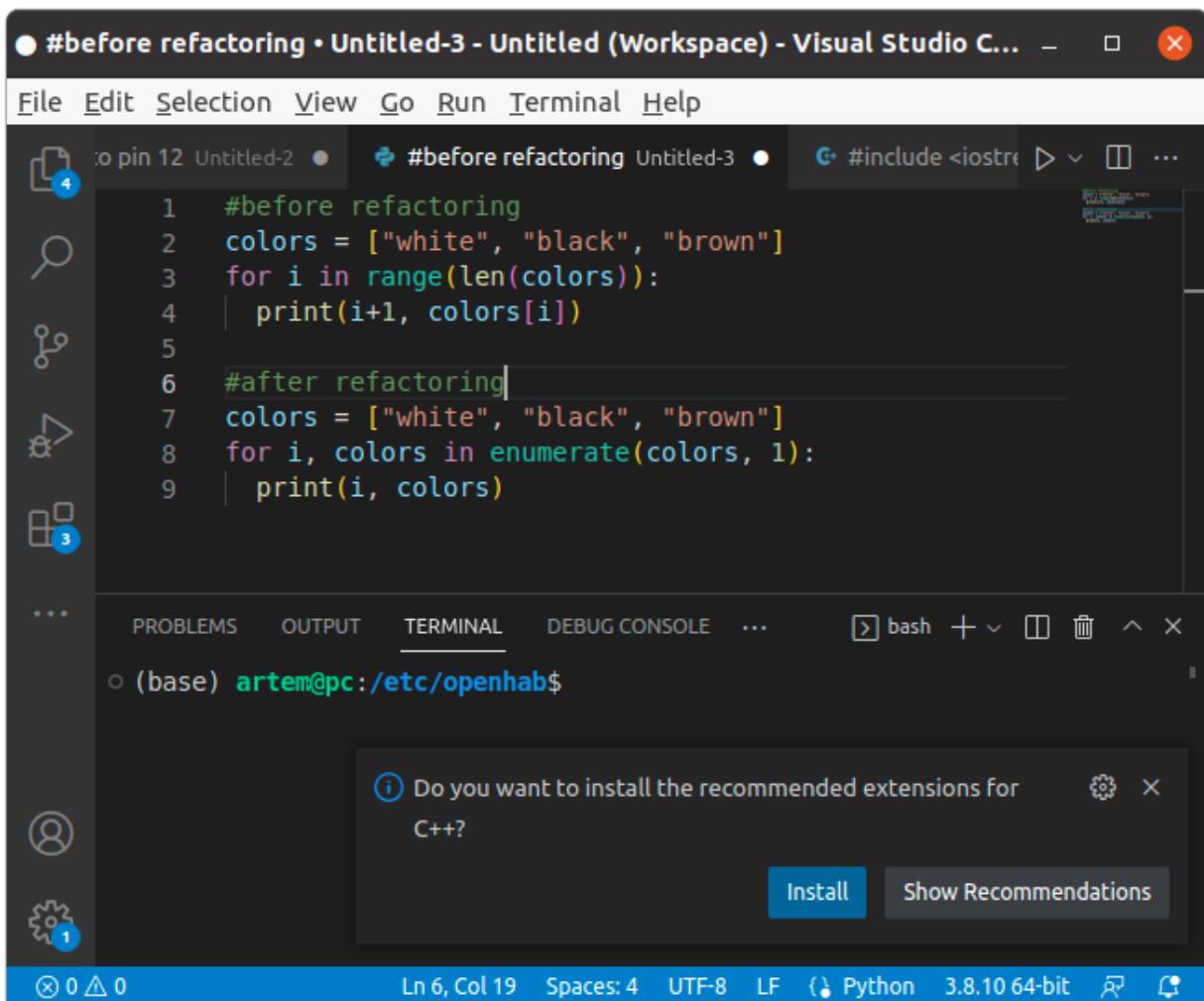


Кроме автодополнения кода существует такое понятие, как «интеллектуальное дополнение кода». Например, если мы написали название метода, среда разработки может подсказать, какие аргументы можно передать в этот метод. Так как мы ничего не писали, это уже нельзя отнести к автодополнению кода. В VSCode это реализовано в виде IntelliSense.



### 6.1.3 Рефакторинг

Рефакторинг — это изменение формы кода без изменения содержания. Мы можем объединить повторяющийся код в функцию или метод, назвать переменные более понятным образом, изменить цикл и совершить другие действия, которые улучшают читаемость кода, но не поменяют то, что он делает. Это и будет рефакторингом.



#### 6.1.4 Компиляция

Среды разработки могут выполнять все действия, чтобы запустить код или скомпилировать его в исполняемый файл. Эти действия — такие же, как и при ручном запуске или компиляции. Следует помнить, что среда разработки выполняет некоторые действия неявно, например, осуществляет выбор виртуального окружения, в котором могут быть установлены зависимости, не прописанные нигде явно. Это может приводить к тому, что код не запустится у другого разработчика без дополнительных действий. Часто начинающие просто «нажимают зеленую стрелочку» без понимания, что именно происходит после нажатия. Вы как разработчики, прошедшие этот курс, должны понимать, что запуск и сборка кода может происходить там, где не установлена та же среда разработки, что и у вас. Поэтому все инструкции по запуску, компиляции и сборки проекта должны быть отражены в соответствующих файлах, например, make-файлах. Среда разработки должна использовать эти файлы при работе. Проще говоря, **среда разработки должна упрощать разработку и не подменять ее этапы**. Сборка из командной строки становится незаменимой, когда нужно собрать множество проектов, которые должны работать в рамках одной системы. Нужно также помнить, что среды разработки гораздо разнообразнее, чем командные строки, и пользователь, скорее всего, воспользуется командной строкой вместо того, чтобы устанавливать новую среду разработки.

### **6.1.5 Настройка окружения под себя**

Вы можете приобрести какие-то привычки при написании кода. Например, использовать табуляцию для отделения блоков кода, при том что общепринятым способом делать отступы являются 4 пробела. Т.е. вы забираете код с git, правите его, и часть строк вместо отступов с пробелами получаются с табуляцией. И даже если это не вызовет ошибку при исполнении кода (в Python вызовет), это будет все равно будет неудобно, так как длина табуляции может быть различна для различных редакторов. Мы можем настроить среду разработки так, чтобы когда мы нажимаем на Tab, вместо табуляции подставлялось 4 пробела (или столько, сколько принято для данного проекта). Поэтому среда разработки может взять на себя часть функций по редактированию кода, а разработчик продолжит работать так, как привык. Это упрощенный пример.

---

## Стили оформления кода

---

### 7.1 Оформление кода

Если мы хотим создать полезную библиотеку, мы должны позаботиться об оформлении кода, чтобы его было удобно читать. Для больших проектов с активным сообществом справедливо утверждение, что код больше читают, чем пишут. И если мы хотим участвовать в таком сообществе, нужно не только производить фичи, но и хорошо их оформлять. Хорошо — это значит общепринятым способом, чтобы другой разработчик не тратил время на понимание того, что вы почему-то решили назвать переменную так, как обычно называют класс. Есть интуитивно понятные правила оформления кода, например, с использованием отступов. Но существуют не такие очевидные, например, где мы должны переносить строку, [до](#) или [после](#) знака оператора:

```
#wrong
income = (gross_wages +
           taxable_interest +
           (dividends - qualified_dividends) -
           ira_deduction -
           student_loan_interest)

#correct
income = (gross_wages
           + taxable_interest
           + (dividends - qualified_dividends)
           - ira_deduction
           - student_loan_interest)
```

Можно думать о стиле как о минимально необходимом наборе всех возможностей языка программирования по написанию и оформлению кода: даже если мы можем так написать, мы не будем — в угоду читаемости. Соблюдение стиля не всегда приводит к меньшему количеству кода. Мы можем обратиться к дзену Python и увидим, что там нет строчки про количество кода, поэтому лучше получить более читаемый, явный и простой код, чем короткий, но сложный, в котором происходит что-то неявное.

```

import this

The Zen of Python, by Tim Peters

Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one-- and preferably only one --obvious way to do it.
Although that way may not be obvious at first unless you're Dutch.
Now is better than never.
Although never is often better than *right* now.
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.
Namespaces are one honking great idea -- let's do more of those!

```

### 7.1.1 Используйте отступы

Одно из основных правил хорошего стиля программирования (code style) заключается в том, что разные уровни вложенности стоит выделять при помощи отступов, используя знаки пробела или табуляции. **Работая с кодом в рамках одного файла, стоит использовать что-то одно: или пробелы, или табуляцию.** Если вы попробуете использовать и то, и другое, набирая код программы на Python, то интерпретатор выдаст вам сообщение об ошибке. В других языках программирования такая ситуация не вызовет ошибку синтаксиса, однако читать код с отступами приятнее, он легче и быстрее воспринимается разработчиком. Еще больше информации о стиле программирования можно узнать из документа [pep8](#). В нем множество рекомендаций по оформлению кода на языке Python.

Код без отступов:

(код):

```

#include <iostream>

#define DEFINE_NAME true

int main()
{
//not indented code
char* name;
if(DEFINE_NAME)
{
name = (char*) "Name";
std::cout << "Hello, " << name << "!";
}

```

(continues on next page)

(продолжение с предыдущей страницы)

```

else
{
std::cout << "Hello World!";
}
return 0;
}

```

Код с отступами:

(код):

```

#include <iostream>

#define DEFINE_NAME true

int main ()
{
    char *name;
    if (DEFINE_NAME)
    {
        name = (char *) "Name";
        std::cout << "Hello, " << name << "!";
    }
    else
    {
        std::cout << "Hello World!";
    }
    return 0;
}

```

В Python также есть ситуации, когда отступы не будут приводить к ошибке, например, при вызове функции, название и аргументы которой не помещаются в максимальную длину строки и должны быть перенесены на следующую строку.

```

#buity
foo = long_function_name(var_one, var_two,
                         var_three, var_four)

#not_so_buity
foo = long_function_name(var_one, var_two,
                         var_three, var_four)

```

Оба варианта валидны с точки зрения интерпретатора, но первый более читаем. В разных ситуациях отступы используются по-разному. Если чувствуете, что участок кода сложен при оформлении, обратитесь к руководству по стилю.

### 7.1.2 Соблюдайте принятую длину строки

В разных языках программирования и в разных проектах может быть принята разная максимальная длина строки. Когда я работал над проектом на Java, мы использовали длину строки 120 символов. В Python принято вмешать код на каждой строке в 79 символов, а комментарии в 72 символа. Отчасти это историческое ограничение, связанное с тем, что раньше были маленькие экраны. Но даже с появлением широкоформатных экранов от этого элемента форматирования не стали отказываться, так как человеку проще воспринимать более узкие колонки текста. Во многих изданиях принято оформлять текст в виде нескольких колонок на одной странице, хотя ничто не мешает занять всю ширину страницы. Также такой подход позволяет разделить экран ноутбука на две области и быть уверенными, что код войдет по ширине в обе, и шрифт не будет слишком мелким. Не заставляйте своих коллег скролить горизонтально, это не круто.

Существуют разные способы разбить длинную строку на несколько коротких. Они будут зависеть от языка программирования и от конкретного участка кода, который нужно перенести на другую строку.

```
if super_long_condition_variable == True and another_condition_with_yet_longer_name == False:
    pass
    # do stuff here
```

Просто перенести второе условие на новую строку нельзя, будет ошибка:

```
if super_long_condition_variable == True and
another_condition_with_yet_longer_name == False:
    pass
    # do stuff here

...
    if super_long_condition_variable == True and
    ^
SyntaxError: invalid syntax
```

Можно окружить условие скобками. Скобки имеют более высокий приоритет, чем перенос, поэтому пример внизу выполнится корректно

```
if (super_long_condition_variable == True and
    another_condition_with_yet_longer_name == False):
    pass
    # do stuff here
```

Можно использовать явный способ сказать интерпретатору, что строка ниже является продолжением текущей строки:

```
if super_long_condition_variable == True and \
another_condition_with_yet_longer_name == False:
    pass
    # do stuff here
```

В математических выражениях удобнее использовать скобки:

```
income = (gross_wages
          + taxable_interest
          + (dividends - qualified_dividends)
```

(continues on next page)

(продолжение с предыдущей страницы)

```
- ira_deduction
- student_loan_interest)
```

Так же, как и при вызове функции.

```
long_function_name(var_one, var_two, var_three, var_four,
                   var_six, var_seven, var_eight)
```

### 7.1.3 Используйте фигурные скобки правильно

В C++ и в других языках, которые используют фигурные скобки для выделение блоков кода, мы можем не писать фигурные скобки у условного оператора или оператора цикла.

```
int age;
//get age from user
if(age >= 18)
    allow_18plus_content();
else
    disable_18plus_content();
```

Но лучше всегда их использовать, даже если вы полностью уверены, что в тело условия больше не будет добавлено ни одной строчки. Опять же, использование единого оформления позволит не отвлекаться и читать код быстрее.

```
int age;
//get age from user
if(age >= 18)
{
    allow_18plus_content();
}
else
{
    disable_18plus_content();
}
```

Есть также альтернативное оформление, которое, на мой взгляд, выглядит более приятно при условии использования отступов. Уточните, какой из этих способов принят в команде, к которой вы подключаетесь. Но скобки должны быть всегда.

```
int age;
//get age from user
if(age >= 18) {
    allow_18plus_content();
} else {
    disable_18plus_content();
}
```

### 7.1.4 Именуйте объекты правильно

Среди начинающих программистов и своих студентов я часто встречаю именование переменных либо одним ничего не значащим символом, либо транслитом.

```
def srednee(znacheniya):
    return sum(znacheniya) / len(znacheniya)
```

Как первое, так и второе — плохо. Именование с использованием транслита не позволит другому разработчику понять, что это за функция и что она делает. Еще одна частая проблема, которая встречается при именовании объектов, — сохранение контекста, который использовался при разработке. Например, мы работаем с датчиком, который предоставляет температуру (`temperature`) и давление (`pressure`). Мы хотим как-то обработать данные, например, найти среднее значение. Мы начинаем разработку и выбираем использовать температуру, чтобы проверять нашу работу. Мы правильно пишем название функции, а аргументом передаем список снятых значений температуры. Вроде бы все нормально.

```
def average(temperature):
    return sum(temperature) / len(temperature)

vals = measure_temperature()
average_temperature = average(vals)
```

Мы протестируем функцию и довольны ее работой, и теперь готовы применить ее к показаниям давления. И все... работает. Мы счастливы и коммитим свою работу. А разработчик, который будет потом разбираться в нашем коде, долго не сможет понять, почему температура ведет себя как давление.

```
vals = measure_pressure()
average_pressure = average(vals)
```

Используйте несколько слов там, где это нужно. Например, мы могли просто назвать функцию `pressure()`. И что это значит, «давление»? Измерить, прочитать из файла, забрать из базы данных, что я делаю с давлением? Тот вариант, который использовали мы, `measure_pressure()`, сразу даст понять, что это функция измерения давления, и скорее всего она работает с датчиком, а не получает давление каким-то другим способом. Здесь можно прокомментировать, что непонятно, что эта функция дает ряд значений, а не одно. Мы можем согласиться с этим и добавить эту информацию к названию функции, или сделать это через обязательные аргументы. Существуют два основных способа именования переменных и функций: через подчеркивания (`measure_pressure`) и CamelCase (`measurePressure`). На мой взгляд, через подчеркивание удобнее, так как мы используем название классов с заглавными буквами (`PressureSensor`), и при беглом чтении разница в одной заглавной (для классов) или строчной (для переменных и функций) менее заметна, чем при использовании подчеркиваний. При выборе имени для переменной используйте существительные, а для функций — глагол.

### 7.1.5 Придерживаться определенного подхода

Этот пункт не относится напрямую к стилю кода и не является строгим требованием. Код состоит из определенных конструкций, и неплохая идея — пользоваться одним подходом для написания одной и той же функциональности. Например, если нам нужно открыть файл, мы можем сделать это несколькими способами:

```
with open(filename) as f:
    #do something
```

или

```
f = open(filename)
#do something
f.close()
```

Это простой пример, и есть соглашение о том, что конструкция с `with` лучше. Но в других ситуациях, когда нет устоявшегося способа написать участок кода, вам придется приходить к этому внутри команды. То же самое касается и подхотов к программированию. Например, Python позволяет использовать основные парадигмы программирования: объектно-ориентированную, процедурную, функциональную и императивную. Мы можем смешивать их в рамках одной программы, и это не будет ошибкой. Есть решения, где использование другого подхода оправдано: все так делают, и решение более красивое и явное, чем при использовании другого подхода. Однако использование другого подхода только ради того, чтобы его использовать, не будет хорошей идеей и ухудшит читаемость кода.

### 7.1.6 Избегайте вложенности (Flat is better than nested)

Этот пункт взять из дзена Python, и ему можно легко дать количественную оценку — уровни вложенности можно посчитать. Возьмем код ниже:

(код):

```
#nested
def foo(a, b, c):
    if a > 5:
        if b > 10:
            if c != 6:
                return a + b + c
    return None
```

И уберем вложенность:

(код):

```
def foo(a, b, c):
    if a <= 5:
        return None
    if b <= 10:
        return None
    if c != 6:
        return None
    return a + b + c
```

Задание: попробуйте прогнать код с вложенностью через autoreper8 и проверьте, что произойдет. Как думаете, почему так?

### 7.1.7 Не перегружайте код (*Sparse is better than dense*)

Хотя мы должны вмещать код в 8 строк, мы можем написать очень насыщенный код. Код ниже заставит разработчика остановиться и потратить какое-то время, чтобы разобраться в нем, если ему нужно что-то поправить или реализовать подобный функционал у себя.

(код):

```
capacity = [(j * 8, 256**j-1) for j in (1 << i for i in range(4))]
print("\n".join("%i bits can store number up to %i" % bc for bc in capacity))
```

```
8 bits can store number up to 255
16 bits can store number up to 65535
32 bits can store number up to 4294967295
64 bits can store number up to 18446744073709551615
```

Написав больше кода, мы можем уменьшить время, которое разработчик тратит на изучение этого участка.

(код):

```
capacity = list()
for i in range(4):
    j = 1 << i
    bits = j * 8
    max_val = 256**j - 1
    capacity.append((bits , max_val))

for bits, max_val in capacity:
    print("%i bits can store number up to %i" % (bits, max_val))
```

### 7.1.8 Выводы

Придерживаться стиля нужно для того, чтобы писать код понятно. Хороший стиль помогает быстро сориентироваться в коде и найти тот участок, за которым вы пришли в этот модуль. Часть работы по оформлению кода можно делегировать автоматическим средствам, таким как `autoreper8`. Про другую часть придется помнить и писать код с учетом этих знаний. Стиль — это не про то, чтобы писать меньше кода. Иногда (редко) вам будет казаться, что придерживаться стиля контринтуитивно, а получивший код некрасив. Но если так пишет большинство разработчиков — придерживайтесь рекомендаций, со временем вы привыкнете. Вы не сможете начать хорошо оформлять код без практики: пишите, а самое главное — читайте чужой код, чтобы понять, как пишут ваши коллеги.

## 7.2 Задачи

### 7.2.1 Задача 1

Уберите вложенность из кода ниже без изменения условий:

```
#nested
def foo(a, b, c):
    if a > 5:
```

(continues on next page)

(продолжение с предыдущей страницы)

```

if b > 10:
    if c != 6:
        return a + b + c
return None

```

## 7.2.2 Задача 2

Улучшите читаемость кода ниже. Для решения примите, что нам нужно импортировать функции `one_function`, `another_function` из `my_module`. Обратитесь к этой статье, если вам нужна помощь в оформлении.

```

import math, pandas as pd, numpy
from my_module import *

```

### Ответ

Хорошим стилем считается использование импортов каждой сущности отдельно.

```

import math
import pandas as pd
import numpy
from my_module import (one_function,
                       another_function)

```

## 7.2.3 Задача 3

Улучшите читаемость кода ниже, убрав лишние пробелы и добавив нужные там, где это необходимо.

```

def multiply_numbers(a,b, factor= 2):
    """Multiply two numbers with factor."""
    return a*b

x =10
y =5

result = multiply_numbers(x,y)

if result>10 :
    print("Result is greater than 10.")

```

### Ответ

Мы должны поправить несколько моментов. \* Вокруг арифметических, логических операторов есть пробелы, например, (`a * b`, `x = 10` и `y = 5`), что упрощает идентификацию выполняемой операции. Вокруг оператора присваивания пробелы также ставятся, кроме случаев использования при объявлении функции. \* Перед двоеточием в условном операторе пробела нет, что улучшает связность кода. \* После запятой, которая разделяет аргументы функции, мы ставим пробел.

```

def multiply_numbers(a, b, factor=2):
    """Multiply two numbers."""

```

(continues on next page)

(продолжение с предыдущей страницы)

```

    return a * b

x = 10
y = 5

result = multiply_numbers(x, y)

if result > 10:
    print("Result is greater than 10.")

```

#### 7.2.4 Задача 4

Улучшите именование переменных, функций и классов. Используйте CamelCase для классов и разделение подчеркиванием для функций.

##### Фрагмент 1

```

# Bad style
def calcrect1(l, w):
    """Calculate the perimeter of a rectangle."""
    return (l + w) * 2

def calcrect2(l, w):
    """Calculate the area of a rectangle."""
    return l * w

class my_class:
    def __init__(self, n):
        self.n = n

    def pn(self):
        print(f"My name is {self.n}.")

```

##### Ответ 1

Мы должны поправить имена переменных, чтобы они не состояли из одной буквы. Имя функции сокращено и не дает понимания того, что функция делает: `calcrect1` не дает понимания, что именно будет рассчитано.

```

def calculate_area_of_rectangle(length, width):
    """Calculate the area of a rectangle."""
    return length * width

def calculate_perimeter_of_rectangle(length, width):
    """Calculate the perimeter of a rectangle."""
    return (length + width) * 2

class MyClass:
    def __init__(self, name):
        self.name = name

```

(continues on next page)

(продолжение с предыдущей страницы)

```
def print_name(self):
    print(f"My name is {self.name}.")
```

**Фрагмент 2**

Для примера ниже предполагается, что база данных хранит данные о покупателе и его карточку.

```
# Bad style
maxAttempts = 5

def circleArea(r):
    """Calculate the area of a circle."""
    return 3.14 * r ** 2

class customer_db:
    def __init__(self):
        self.cust = []
        self.cars = {}

    def add_cust(self, c):
        self.cust.append(c)

    def get_cust_count(self):
        return len(self.cust)

    def add_cust_car(self, c, cc):
        self.cars[c] = cc
```

**Ответ 2**

Часто бывает, что сокращение начинает читаться как другое слово. Так в примере карточка (card) стала машиной (car).

```
MAX_RETRY_ATTEMPTS = 5

def calculate_circle_area(radius):
    """Calculate the area of a circle."""
    return 3.14 * radius ** 2

class CustomerDatabase:
    def __init__(self):
        self.customers = []
        self.cards = {}

    def add_customer(self, customer):
        self.customers.append(customer)

    def get_customer_count(self):
        return len(self.customers)

    def add_customer_card(self, customer, card):
        self.cards[customer] = card
```

**Фрагмент 3**

Бывает, что мы называем объект, используя какие-то свои ассоциации или что-то еще, что делает название понятным для нас, но будет сбивать с толку других разработчиков. Поправьте код ниже чтобы, каждый мог понять, что мы создали класс Кольцо. Поправьте название методов.

```
# Bad style (objects named improperly)
class Donut:
    def __init__(self, material, size):
        self.material = material
        self.size = size

    def get_ingredient(self):
        return self.material

    def get_circle(self):
        return self.size

new_donut = Donut("gold", "medium")
print(new_donut.get_ingredient()) # Output: gold
print(new_donut.get_circle())
```

**Ответ**

```
class Ring:
    def __init__(self, material, size):
        self.material = material
        self.size = size

    def get_material(self):
        return self.material

    def get_size(self):
        return self.size

new_ring = Ring("gold", "medium")
print(new_ring.get_material()) # Output: gold
print(new_ring.get_size())
```

**7.2.5 Задача 5**

Напишите комментарии к функции, чтобы дать понимание о ее работе. Попробуйте описать функцию, которая рассчитывает прибыль по инвестициям.

**Фрагмент 1**

```
def calculate_interest(principal, rate, time):
    n = 1
    final_amount = principal * (1 + rate / n) ** (n * time)
    return final_amount

# Example usage
```

(continues on next page)

(продолжение с предыдущей страницы)

```
principal_amount = 1000.0
annual_interest_rate = 0.05
investment_time = 5
final_amount = calculate_interest(principal_amount, annual_interest_rate, investment_
    ↪time)
print(f"The final amount after {investment_time} years is: ${final_amount:.2f}")
```

**Ответ**

```
# Less obvious code that benefits from comments

def calculate_interest(principal, rate, time):
    """Calculate compound interest for the given parameters.

    Args:
        principal (float): The principal amount (initial investment).
        rate (float): The annual interest rate in decimal form (e.g., 0.05 for 5%).
        time (int): The time period in years.

    Returns:
        float: The final amount after compounding the interest.
    """

    # Formula for compound interest: A = P(1 + r/n)^(nt)
    # A: Final amount after compounding the interest
    # P: Principal amount
    # r: Annual interest rate
    # t: Time period in years

    # In this implementation, we'll assume interest is compounded annually (n=1)
    n = 1

    # Calculate the final amount after compounding interest
    final_amount = principal * (1 + rate / n) ** (n * time)

    return final_amount

# Example usage
principal_amount = 1000.0
annual_interest_rate = 0.05 # 5% interest rate
investment_time = 5 # 5 years
final_amount = calculate_interest(principal_amount, annual_interest_rate, investment_
    ↪time)
print(f"The final amount after {investment_time} years is: ${final_amount:.2f}")
```

**Фрагмент 2**

Часто бывает, что мы реализуем логику, которая специфична для конкретной ситуации и может не встречаться за ее пределами. Например, поведение функции `sum` для чисел очевидно, она вернет суммы переданных в нее аргументов-чисел. Но давайте представим подсчет очков в настольной игре-ходилке, где количество очко зависит от тех клеток, которые прошла фишка игрока. Допустим, клетки с номером, кратным трем, содержат сундуки, которые увеличивают вклад на этой клетке в два раза. А на клетках с номером, кратным двум, находятся разбойники, которые уменьшают вклад на этой клетке

в два раза.

```
def calculate_player_total(data):
    total = 0

    for item in data:
        if item % 3 == 0:
            total += item * 2
        elif item % 2 == 0:
            total += item / 2
        else:
            total += item

    return total
```

**Ответ**

```
def calculate_player_total(data):
    """Process the given list of integers, that presents cells ID.

    Contribution of the cell to the total is based on divisibility of cell ID.

    Args:
        data (list): A list of integers.

    Returns:
        int: The total value after processing the list.
    """
    total = 0

    for item in data:
        if item % 3 == 0:
            # if cell id is divisible by 3, it is chest, double the value and add to
            # 'total'
            total += item * 2
        elif item % 2 == 0:
            # if cell id is divisible by 2, it is robber, halve the value and add to
            # 'total'
            total += item / 2
        else:
            # For other numbers, add the number to 'total'
            total += item

    return total
```

## 7.3 Вопросы

1. Мы именуем переменные и функции как `snake_case` или `CamelCase`?
2. Мы именуем классы как `snake_case` или `CamelCase`?
3. Используются ли комментарии для комментирования каждой строки кода?
4. Является ли написание `docstring` для функций и классов, объясняющие их назначение, хорошей практикой?
5. Считается ли одинаковый отступ (например, использование четырех пробелов для каждого уровня) хорошим стилем даже для языков, которые используют другой способ для разделения вложенных блоков кода?
6. Какая рекомендованная длина строки в Python, если в команде не принято другое? **Ответ:** 79 или 88 символов.
7. Считается ли использование импортов в любом месте кода хорошей практикой?
8. Является ли `from module import *` хорошей практикой?
9. В Python строковую переменную можно создать, используя одинарные (" „) или двойные кавычки (» «). Как регламентировано их использование? **Ответ:** Выбираются одинарные (" „) или двойные кавычки (» «), и выбранный вид используется для строковых литералов во всех модулях проекта.
10. Правильно ли с точки зрения стиля использовать тернарные операторы для упрощения кода?
11. Верно ли утверждение, что код должен включать как можно больше комментариев, даже если они избыточны (например, констатация очевидного), так как из всех комментариев собирается документация?
12. Верно ли утверждение, что константы, которые встречаются в коде всего раз, должны быть использованы прямо в выражении без объявление дополнительного имени? **Ответ:** Нет, магические числа должны быть именованными переменными или константами с описательными именами.
13. Верно ли утверждение, что необходимо добавить пробелы в конце строки, чтобы длина строки совпадала с регламентированной, например, 79 символов?
14. Должна ли функция быть краткой, насколько это возможно, и сосредоточенной на решении одной задачи?
15. Верно ли утверждение, что аргументы функций и методов должны быть краткими, желательно однобуквенными или аббревиатурой из слов, определяющих их назначение, для экономии места?
16. Чем больше вложенность кода, тем легче его понимать, так как сразу понятен переход на уровень ниже?



Польза документации на примере поиска проекта на GitHub

---

## 8.1 Сервис GitHub

### 8.1.1 Учимся читать и использовать в своем проекте чужой код с GitHub

Перенимаем опыт других разработчиков: ищем, читаем, клонируем, внедряем...



Как только вы начинаете погружаться в атмосферу IT-сообщества, в вашем информационном поле постепенно формируется круг экспертов, которых вы встречаете на конференциях, подкастах и за

просмотром онлайн-митапа крупной компании. Большинство советов, которыми делятся опытные разработчики, можно разделить на 3 категории:

1. Читайте документацию и книги;
2. Изучайте новый язык или инструмент, используйте его в разработке своего PetProject;
3. Изучайте чужой код и участвуйте в «open source» разработке.

Касательно книг обычно все понятно, рекомендуют конкретных авторов и печатные издания. Про PetProject тоже довольно ясно: хотите понять, как работает Django? Создайте свой веб-сервис на любую тему. Например, для обработки заказов небольшой пиццерии, с админкой, меню и регистрацией клиентов. В общем, используйте все, что идет из коробки Django, чтобы разобраться, как работает этот фреймворк.

А что по поводу третьего пункта? «Изучайте чужой код...». Обычно мысль на этом обрывается.

Сегодня попытаемся восполнить этот пробел и закрасим «белое пятно». Выстроим цепочку алгоритма, которая поможет начинающим программистам сориентироваться в поиске «чужого кода» и использовании его в своем проекте. Чтобы «не научиться плохому».

1. Где и как искать код?
2. Выбрали проект — на что в первую очередь стоит обратить внимание?
3. С чего начать знакомство с чужим кодом?
4. Как использовать его в своем проекте?
5. Как опубликовать свой проект на GitHub?

### 8.1.2 Где и как искать код?

Самой популярной площадкой для поиска и размещения кода является GitHub (<https://github.com>). Если вы ранее не пользовались этим сервисом, следует начать именно с него.

GitHub — это веб-платформа для контроля версий и совместной работы, которая была запущена в апреле 2008 года. Она была основана Томом Престон-Вернером, Крисом Ванстратом и П. Дж. Хайеттом, а сейчас принадлежит компании Microsoft. Платформа была построена на основе системы контроля версий Git, созданной Линусом Торвальдсом в 2005 году. Git — мощная и эффективная система для управления исходным кодом и другими файлами. GitHub стремился сделать Git более доступным, предоставляя веб-интерфейс и набор инструментов для совместной работы и совместного использования. GitHub быстро стал популярным среди разработчиков открытого исходного кода. Одной из ключевых особенностей платформы является возможность разветвления репозитория, что позволяет пользователям создавать собственную копию проекта и вносить изменения, не затрагивая оригинал. Это упростило для разработчиков совместную работу и участие в проектах с открытым исходным кодом. За прошедшие годы GitHub добавил множество функций и инструментов, чтобы сделать совместную работу еще проще, в том числе:

- Запросы на вытягивание (Merge Request), которые позволяют пользователям отправлять изменения в проект для проверки.
- Проблемы (Issue), которые позволяют пользователям отслеживать ошибки и запросы функций.
- Вики (Wiki), которые позволяют пользователям создавать документацию и другую информацию для проекта.
- GitHubPages, который позволяет пользователям создавать и размещать веб-страницы для своих проектов.
- GitHubActions, которые позволяют пользователям создавать и развертывать свой код с помощью GitHub.

В 2018 году Microsoft приобрела GitHub за 7,5 млрд долларов. С тех пор он продолжал развивать и улучшать платформу, включая запуск GitHubDiscussions, GitHubCodespaces и GitHubActions. Сегодня GitHub — это крупнейшее хранилище исходного кода в мире с более чем 40 миллионами пользователей и более чем 200 миллионами репозиториев.

Для поиска проекта достаточно ввести в строку поиска одно или несколько ключевых слов.

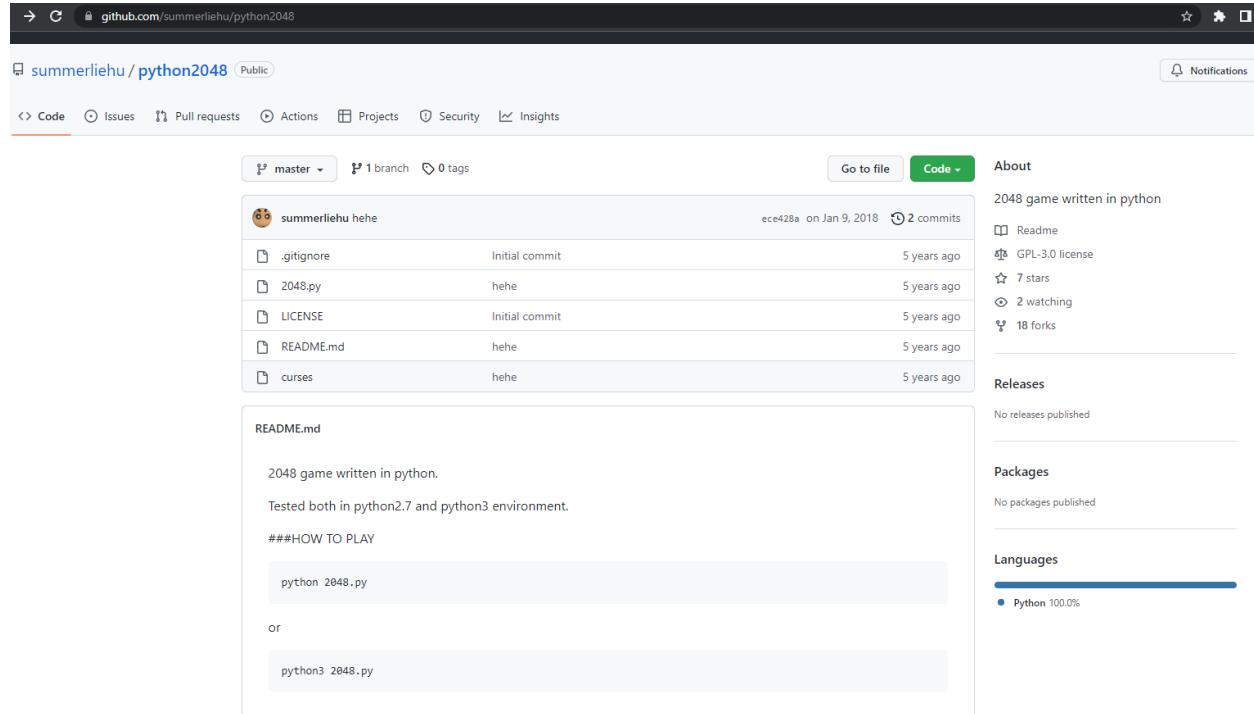


Рисунок 1 — Репозиторий с плохой структурой проекта

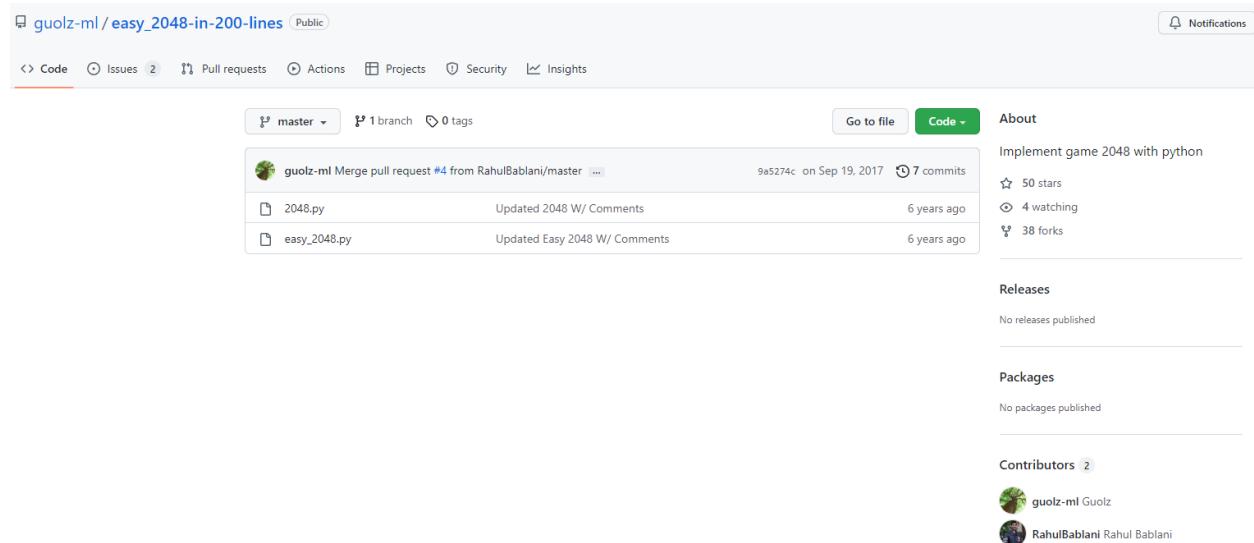


Рисунок 2 — Репозиторий без описания и инструкций

## toolchain, Выпуск 1.0

The screenshot shows a GitHub repository page for `bfontaine/term2048`. The repository has 233 commits across 3 branches and 13 tags. The commit history includes several commits from `bfontaine`, such as adding classifiers, fixing formatting, and releasing version 0.2.7. The repository also contains files like `img`, `tests`, `.gitignore`, `.pylintrc`, `.travis.yml`, `CHANGES.txt`, `CONTRIBUTING.md`, `LICENSE`, `MANIFEST.in`, `Makefile`, `README.rst`, `py26-requirements.txt`, `requirements.txt`, `setup.py`, and `tox.ini`. The repository has 800 stars, 27 watchers, and 142 forks. It is categorized under game, python, and cli. The README file describes the project as "2048 in your terminal". The repository is public.

Рисунок 3 — Репозиторий с хорошей структурой

Посмотрим описание «Readme.md»

☰ README.rst

## Install

```
pip install term2048
```

To upgrade a previous installation, use:

```
pip install -U term2048
```

To install via [snap](#):

```
sudo snap install term2048
```

## Install from source

```
git clone https://github.com/bfontaine/term2048.git && cd term2048
[sudo] python setup.py install
```

## Play

```
term2048
```

Then use arrow keys to move. VI keys (h,j,k,l) are also supported.

Use `-h` to check the list of available options, and `--rules` for the game rules. Press `<space>` at any time during the game to pause it. You can resume it later using `term2048 --resume`.

## Tests

Clone this repo, then:

```
[sudo] make deps
make check
```

Note: while `term2048` should work on Windows, tests are meant to run on UNIX-like OSes and `term2048.keypress` - related tests may fail on Windows.

Рисунок 4 – Инструкция по работе с кодом проекта

После знакомства с инструкцией приступаем к просмотру файлов проекта, чтобы выполнить «поверхностную реверсивную инженерию» объектов и понять его структуру. Этот процесс может занять некоторое время. В результате нам надо разобраться в коде и оценить его достоинства и недостатки, понять зависимости и попробовать составить структуру проекта в виде блок-схемы или псевдокода. В процессе изучения кода можно встретить различные решения, которые мы бы могли реализовать в этой задаче иначе, это вполне распространенная практика. Давайте кратко постараемся определить, каковы признаки «хорошего кода»:

- Структура проекта понятна.
- Задачи распределены по отдельным файлам.

- Предусмотрена модификация кода проекта.
- Реализована поддержка разных операционных систем (ОС). В коде есть проверка текущей ОС и сценарий корректной работы в случае отличий из-за разных версий ОС.
- Ключевые места в коде прокомментированы.
- Реализована обработка исключений.
- Минимальная зависимость от сторонних библиотек.
- Внешние библиотеки представлены в файле requirements.txt.

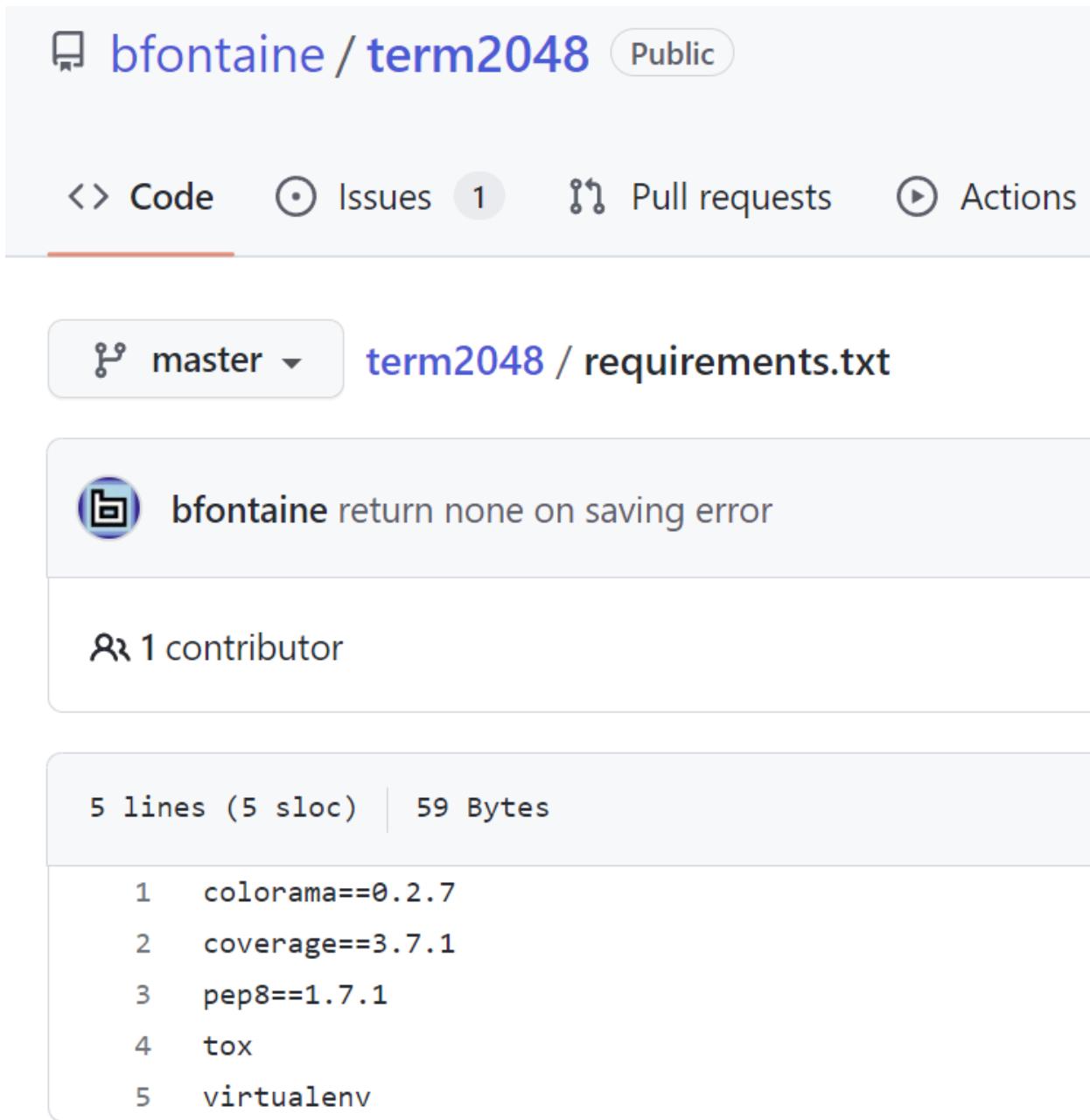


Рисунок 5 — Файл requirements.txt — список сторонних библиотек

Выбрали проект — на что в первую очередь стоит обратить внимание?

bfontaine / term2048 Public

<> Code Issues 1 Pull requests Actions Wiki Security

master term2048 / term2048 /

bfontaine release 0.2.7

..

File	Description
_init_.py	release 0.2.7
board.py	Fix pylint issues
game.py	Fix formatting to meet PEP8
keypress.py	Add docstrings
ui.py	Fix pylint issues

Рисунок 6 — Файлы проекта

#### С чего начать знакомство с чужим кодом?\*\*

Скачаем проект и проанализируем его содержимое. Скачать проект с GitHub можно двумя способами:

- используя консольное приложение git;
- скачать проект zip-архивом.

Наша цель — не только посмотреть исходный код проекта, но и модифицировать его по мере знакомства с проектом.

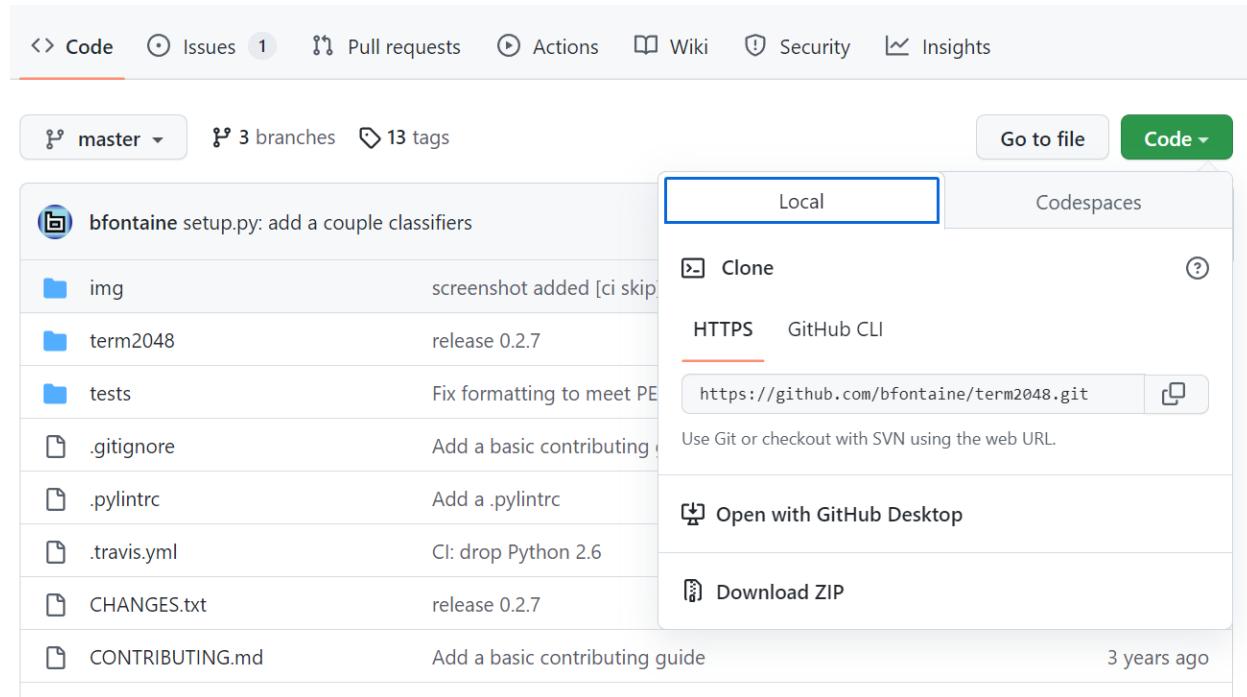


Рисунок 7 — Вкладка загрузки кода с GitHub

В таком случае использование системы контроля версий git нам сильно облегчит задачу отслеживания изменений в проекте. Внося изменения в чужой код, стоит учитывать возможность возникновения ошибок и сбоев на этапе тестирования ПО. Таким образом, мы сможем лучше разобраться в коде, откатить последние изменения и выложить свой проект на GitHub, и т.д.

Приступим к подготовке рабочего окружения. Создадим на компьютере папку, в которой будет размещаться проект. Для примера будем считать, что такой директорией является: «**project\_github**».

1. Откроем консоль в данной папке и выполним инициализацию нового проекта командой `git init`.
2. Выполним загрузку кода из репозитория, используя команду `git pull`, добавив в качестве аргумента ссылку на проект: «<https://github.com/bfontaine/term2048.git>».

```
D:\project_github>git init
Initialized empty Git repository in D:/project_github/.git/
D:\project_github>git pull https://github.com/bfontaine/term2048.git
remote: Enumerating objects: 1006, done.
Receiving objects: 92% (926/1006), 161.27 KiB | 642.00 KiB/s, done.
Receiving objects: 100% (1006/1006), 161.27 KiB | 642.00 KiB/s, done.
Resolving deltas: 100% (610/610), done.
From https://github.com/bfontaine/term2048
 * branch          HEAD      -> FETCH_HEAD

D:\project_github>
```

Рисунок 8 — Инициализация локального Git репозитория и загрузка проекта из GitHub

В качестве альтернативы можно использовать команду `git clone`. Можно сказать, что команда `git`

`clone` выполняет две команды: `git init` и `git pull`. `git init` применяется для того, чтобы создать пустой репозиторий, а `git pull` — чтобы наполнить его из другого репозитория. Поэтому, если вы новый участник проекта, и вам нужно получить код проекта, чтобы с ним познакомиться, то рекомендуется делать `git clone`:



После успешного клонирования проекта из GitHub мы можем приступить к изучению кода. В нашей папке с названием «`project_github`» появятся все файлы репозитория (см. рисунок), их можно открыть любым редактором, например, «*Sublime 3*». На что обращать внимание при работе с чужим кодом, чтобы прокачать *hard & soft skills*?

1. Постарайтесь внимательно изучить код, учитывая инструкции и комментарии автора.
2. Может быть так, что вам попался код, который вам непонятен. Например, вы не знакомы с таким объектом/функцией, или ранее вы не использовали его подобным образом. Вам стоит попробовать написать небольшой пример для его исследования и постараться ответить на вопрос: почему разработчик использовал такое решение и чем оно лучше/хуже альтернативного варианта, который известен вам.
3. Обратите внимание, можно ли сократить код, если встречаются повторения. Например, вынести часть кода в отдельные функции или методы класса, а может, и вовсе стоит вынести участок кода в отдельный модуль. Подумайте об этом.
4. Если вы видите, что в коде допущены очевидные ошибки, например: отсутствует проверка типа аргументов; в арифметических выражениях отсутствует проверка деления на ноль; или повторяется набор одинаковых действий, которые можно оформить в цикл и т.д., оставьте комментарий и сформулируйте, как сделать лучше. Но не пытайтесь сразу приступать к работе над изменением кода до того, как ознакомитесь с ним полностью. Могут выясниться различные ограничения или структурные решения, которые привели к такой реализации.

Если вы запустите команду `git log` в каталоге клонированного проекта, вы увидите следующий вывод:

```
C:\ C:\Windows\System32\cmd.exe - git log

D:\project_github_v2\term2048>git log
commit 6f2d7fce0bbb00a217a907dd72db3f95d54c6b3 (HEAD -> master, origin/master, origin/HEAD)
Author: Baptiste Fontaine <b@ptistefontaine.fr>
Date:   Sun Aug 18 20:18:51 2019 +0200

    setup.py: add a couple classifiers

commit d05ebaf6697290511bdc19dd60c7e9d9a082dfc4
Author: Baptiste Fontaine <b@ptistefontaine.fr>
Date:   Sun Aug 4 20:16:52 2019 +0200

    Add a basic contributing guide

commit 08bf948c9556a2f62c4608a5e6b844f7e3f2dfd5
Author: Baptiste Fontaine <b@ptistefontaine.fr>
Date:   Sun Aug 4 20:07:58 2019 +0200

    release 0.2.7

commit df2c096eebf15d3fda576687d52a8314019cc8a
Author: Sinhs <pgrty45879@gmail.com>
Date:   Mon Aug 5 02:02:28 2019 +0800

    solve encoding problem in windows (#36)

commit 85dd3beb8a399bffd228540025a7be53307b3ecd
Author: Baptiste Fontaine <b@ptistefontaine.fr>
Date:   Wed Jul 24 22:19:58 2019 +0200

    clean 'since <version>' from the docs
....skipping...
commit 6f2d7fce0bbb00a217a907dd72db3f95d54c6b3 (HEAD -> master, origin/master, origin/HEAD)
Author: Baptiste Fontaine <b@ptistefontaine.fr>
Date:   Sun Aug 18 20:18:51 2019 +0200

    setup.py: add a couple classifiers

commit d05ebaf6697290511bdc19dd60c7e9d9a082dfc4
Author: Baptiste Fontaine <b@ptistefontaine.fr>
Date:   Sun Aug 4 20:16:52 2019 +0200

    Add a basic contributing guide

commit 08bf948c9556a2f62c4608a5e6b844f7e3f2dfd5
Author: Baptiste Fontaine <b@ptistefontaine.fr>
Date:   Sun Aug 4 20:07:58 2019 +0200

    release 0.2.7

commit df2c096eebf15d3fda576687d52a8314019cc8a
Author: Sinhs <pgrty45879@gmail.com>
Date:   Mon Aug 5 02:02:28 2019 +0800

    solve encoding problem in windows (#36)

commit 85dd3beb8a399bffd228540025a7be53307b3ecd
Author: Baptiste Fontaine <b@ptistefontaine.fr>
Date:   Wed Jul 24 22:19:58 2019 +0200

    clean 'since <version>' from the docs

commit 8ad4ca4cb685412136c49179c66bde73f107ac42
Merge: b21c1fb 6d6bb72
Author: Baptiste Fontaine <b@ptistefontaine.fr>
```

Рисунок 9 — Просмотр изменений командой git log

```
D:\project_github_v2\term2048>git log -p -1
commit 6f2d7fce0bbbfb00a217a907dd72db3f95d54c6b3 (HEAD -> master, origin/master, origin/HEAD)
Author: Baptiste Fontaine <b@ptistefontaine.fr>
Date:   Sun Aug 18 20:18:51 2019 +0200

    setup.py: add a couple classifiers

diff --git a/setup.py b/setup.py
index 27c093d..d1f0689 100644
--- a/setup.py
+++ b/setup.py
@@ -35,8 +35,11 @@ setup(
    classifiers=[
        'Environment :: Console',
        'License :: OSI Approved :: MIT License',
+       'Development Status :: 5 - Production/Stable',
+       'Programming Language :: Python :: 2',
        'Programming Language :: Python :: 2.6',
        'Programming Language :: Python :: 2.7',
+       'Programming Language :: Python :: 3',
        'Programming Language :: Python :: 3.2',
        'Programming Language :: Python :: 3.3',
        'Programming Language :: Python :: 3.4',
@@ -44,6 +47,7 @@ setup(
        'Programming Language :: Python :: 3.6',
        'Programming Language :: Python :: 3.7',
    ],
+   keywords='2048 terminal',
    entry_points={
        'console_scripts':[
            'term2048 = term2048.ui:start_game'
D:\project_github_v2\term2048>
```

Рисунок 10 — Просмотр изменений последнего коммита

### Как использовать чужой код в своем проекте?

В процессе подробного изучения кода проекта у вас наверняка появится несколько идей, как его изменить, что-то поменять или добавить новый функционал. Например, мы можем изменить размер игрового поля, количество генерируемых чисел на поле и ввести свой рейтинг-лист с 10 лучшими результатами, и т.д. Так вы не только потренируете изучение чужого кода, но и разовьете навык внесения в него модификаций. Также после завершения этапа отладки кода и тестирования можно выгрузить код в репозиторий GitHub.

### **8.1.3 Работа с GitHub**

Первым делом надо зарегистрироваться на сервисе [github.com](https://github.com) и авторизоваться на сайте (рисунок 11).

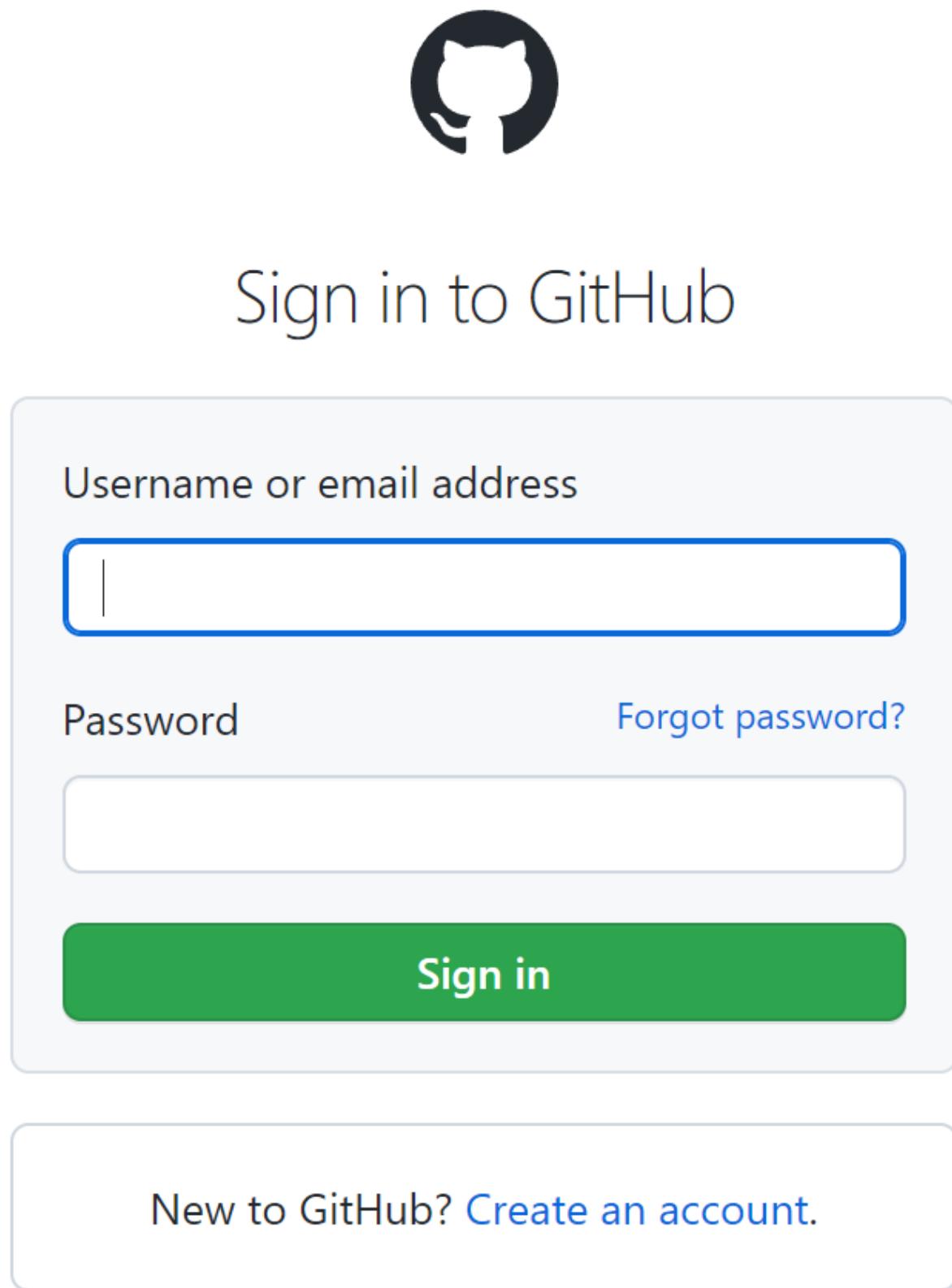


Рисунок 11 — Авторизация в Github

Затем переходим к созданию первого репозитория. В правой верхней части, рядом с иконкой вашего профиля, нужно нажать на значок + (плюс) и выбрать пункт «New repository» (рисунок 12).

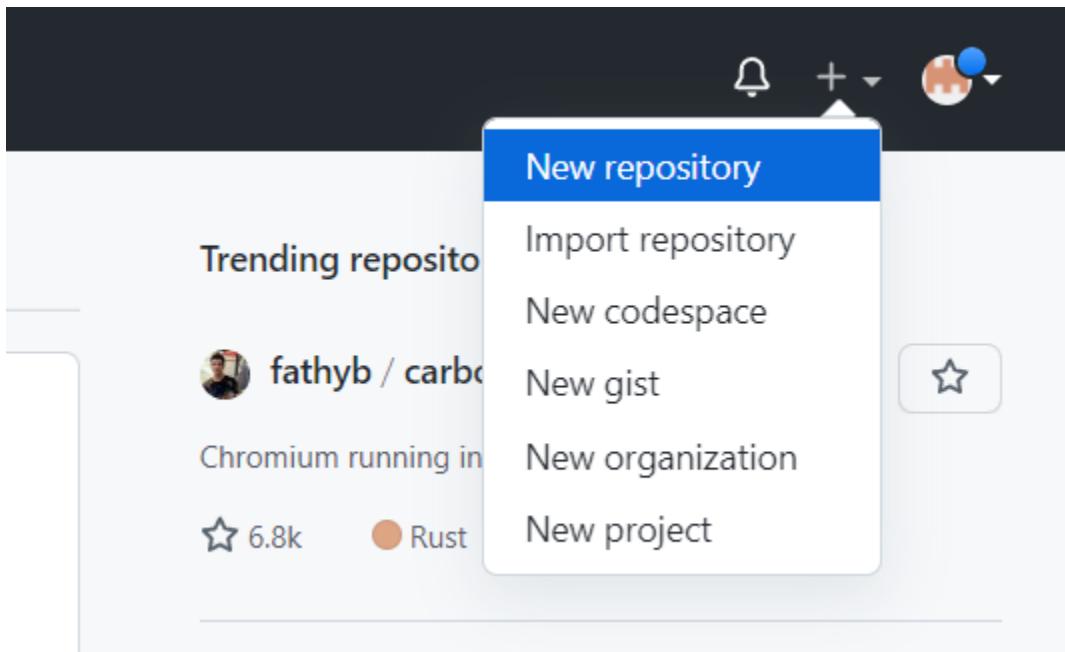


Рисунок 12 — Создание нового репозитория

Откроется страница, на которой нужно выбрать параметры и указать дополнительную информацию о проекте (рисунок 13). Введите название нового репозитория и краткое описание, отражающее главную задачу и способ ее решения, который вы планируете реализовать. Например, в нашем случае можно назвать репозиторий «*terminalGame2048*» или «*py2048CLI*». При выборе названия старайтесь использовать «говорящие слова», чтобы разработчику со стороны было интуитивно понятно, о чем ваш проект. Использование «ру» укажет, что проект на Python, а слово «game» говорит о том, что это игра. 2048 — это очень известное название игры и многих ее вариаций, а слова «terminal» и «CLI» указывают на то, что игра использует в качестве интерфейса пользователя консольное окно (терминал, командная строка). Таким образом, мы кратко в 3 словах сказали, что это игра 2048, запускается в консоли и написана на языке *Python*.

**Create a new repository**

A repository contains all project files, including the revision history. Already have a project repository elsewhere? [Import a repository.](#)

Owner \* Ваш логин  
 / Repository name \* Название нового репозитория

Great repository names are short and memorable. Need inspiration? How about [glowing-octo-eureka?](#)

Description (optional) Описание репозитория

Public Публичный репозиторий  
 Anyone on the internet can see this repository. You choose who can commit.

Private  
 You choose who can see and commit to this repository.

Initialize this repository with:  
 Skip this step if you're importing an existing repository.

Add a README file Добавить файл с инструкцией к проекту  
 This is where you can write a long description for your project. [Learn more.](#)

Add .gitignore  
 Choose which files not to track from a list of templates. [Learn more.](#)

.gitignore template: None ▾

Choose a license  
 A license tells others what they can and can't do with your code. [Learn more.](#)

License: None ▾ Выбор лицензии

ⓘ You are creating a public repository in your personal account.

Завершить создание репозитория

**Create repository**

Рисунок 13 — Страница с параметрами нового репозитория

По желанию в поле «Описание репозитория» можно ввести краткую информацию об особенностях проекта. Например, какой новый функционал вы предлагаете реализовать; в чем отличие от классической игры 2048; разработка кода выполнена в рамках конкурса, школьного/студенческого проекта или хакатона. Затем следует выбрать, какого типа будет ваш новый репозиторий — открытый или закрытый:

- **Открытый/публичный** — исходный код этого репозитория будет доступен всем. Вы даете согласие, что другие разработчики могут просматривать и скачивать ваш код. Все проекты open

source распространяются в открытых репозиториях.

- **Закрытый / приватный** — данный вариант часто используется коммерческими организациями. Приватные репозитории не отображаются в процессе поиска, и весь код будет скрыт для сторонних разработчиков, не участвующих в разработке этого проекта.

Далее вы можете добавить файл «**README**» — это документ, в котором разработчики размещают справочную информацию и инструкции по работе с проектом.

Следует помнить о том, что существует большое количество лицензий, которые регламентируют условия распространения и использования кода. В частности, это необходимо для защиты прав авторства разработчиков. Например, вы можете полностью разрешить доступ к вашему коду и дать права на его использование всем желающим в любом виде.

После нажатия на кнопку «Create repository» перед вами откроется страница с вашим только что созданным репозиторием (рисунок 14). Не стоит волноваться о настройках репозитория, их можно изменить в любое время.

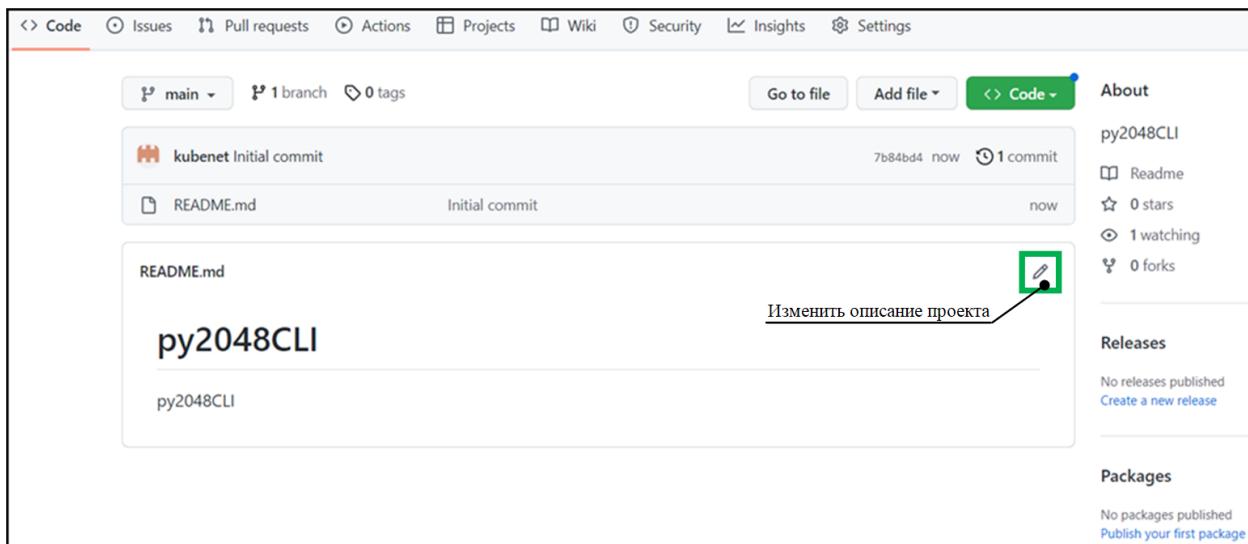


Рисунок 14 — Страница репозитория

После того как был создан репозиторий, мы можем составить описание проекта в файле «**Readme.md**». Информация из этого файла отображается на странице репозитория в поле сразу после файлов с кодом. Добавим описание нашего проекта, как показано на рисунке 15.

```
# py2048CLI
![This is an image](https://dashboard.snapcraft.io/site_media/appmedia/2017/12/term2048.png)
## Этот репозиторий используется для изучения кода проекта игры 2048.
> Ссылка на базовый репозиторий игры: https://github.com/bfontaine/term2048
В данном проекте будет предложено несколько вариантов модификаций которые отсутствуют в базовой версии, например: возможность изменения параметров игры:
- [ ] размер поля
- [ ] выбора уровня сложности
- [ ] ведение подсчета очков и т.д.
```

Рисунок 15 — Редактирование описания репозитория

Чтобы посмотреть, как будет отображаться информация с описанием на странице, нужно переключиться на вкладку «Preview» (рисунок 16).

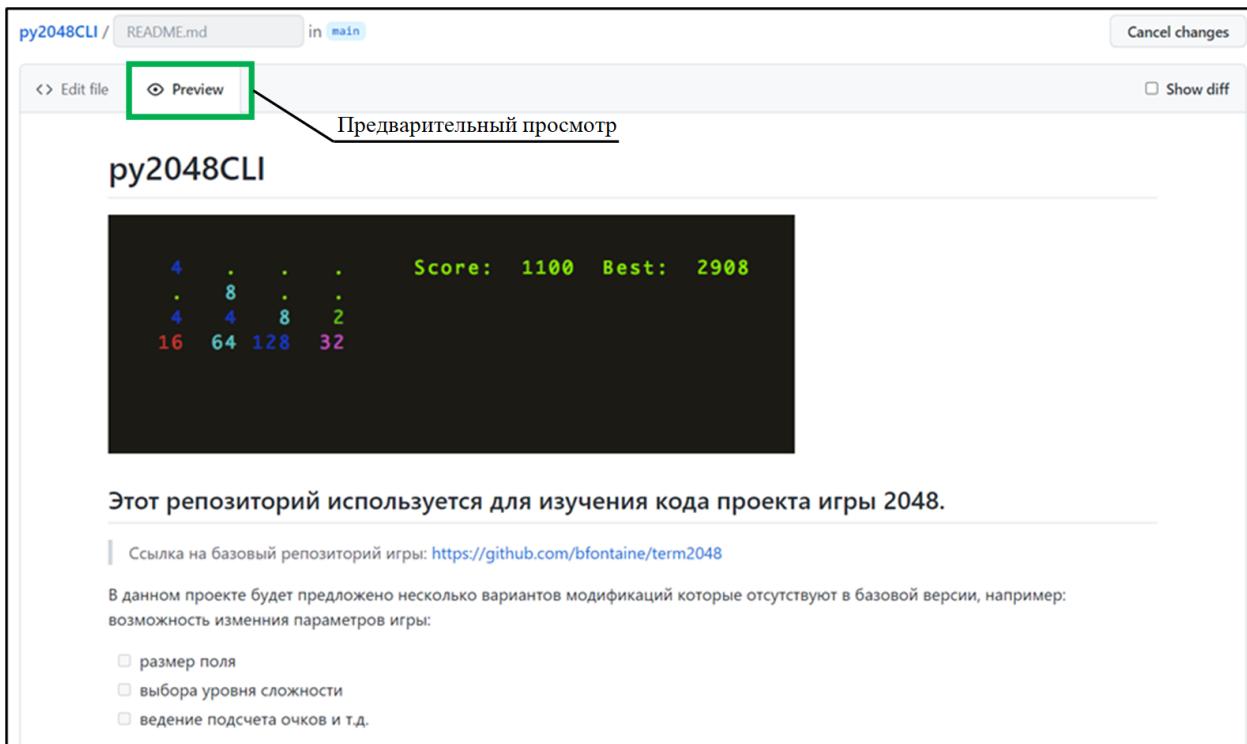


Рисунок 16 — Предварительный просмотр с использованием стилей Markdown

Когда отображение справочной информации о нашем проекте будет удовлетворять нашим интересам, нужно сохранить изменения (рисунок 17), нажав на кнопку «Commit change». Кстати, это будет зарегистрировано системой git как создание нового коммита (commit). Это значит, что файл «Readme.md» изменил свое содержание и, следовательно, нужно его обновить.

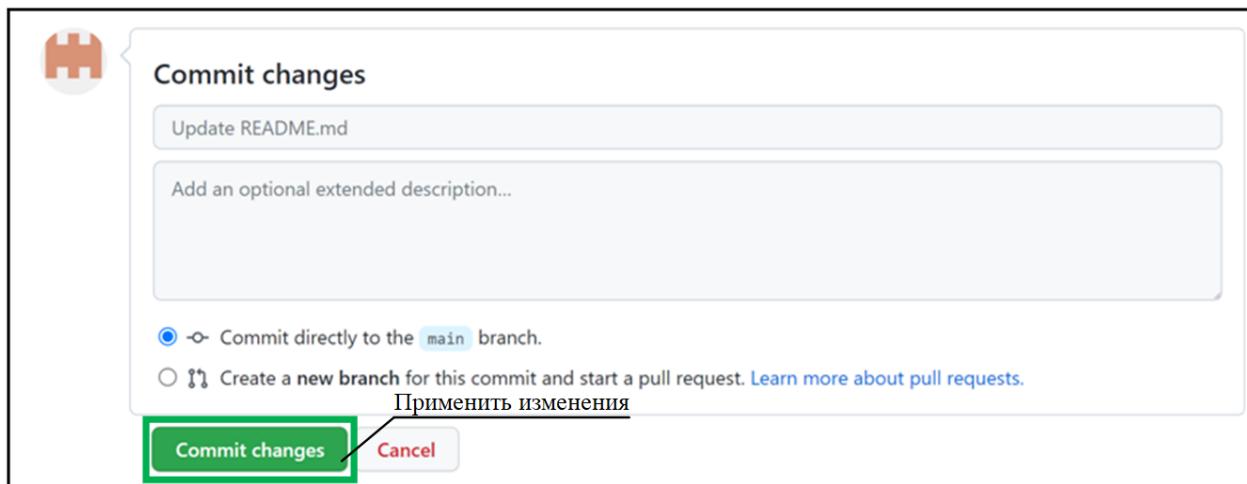


Рисунок 17 — Сохранение изменений в файле Readme

Далее нам нужно загрузить код из нашего локального репозитория (на компьютере) в удаленный репозиторий GitHub. Для этого нужно скопировать путь к удаленному репозиторию (рисунок 18).

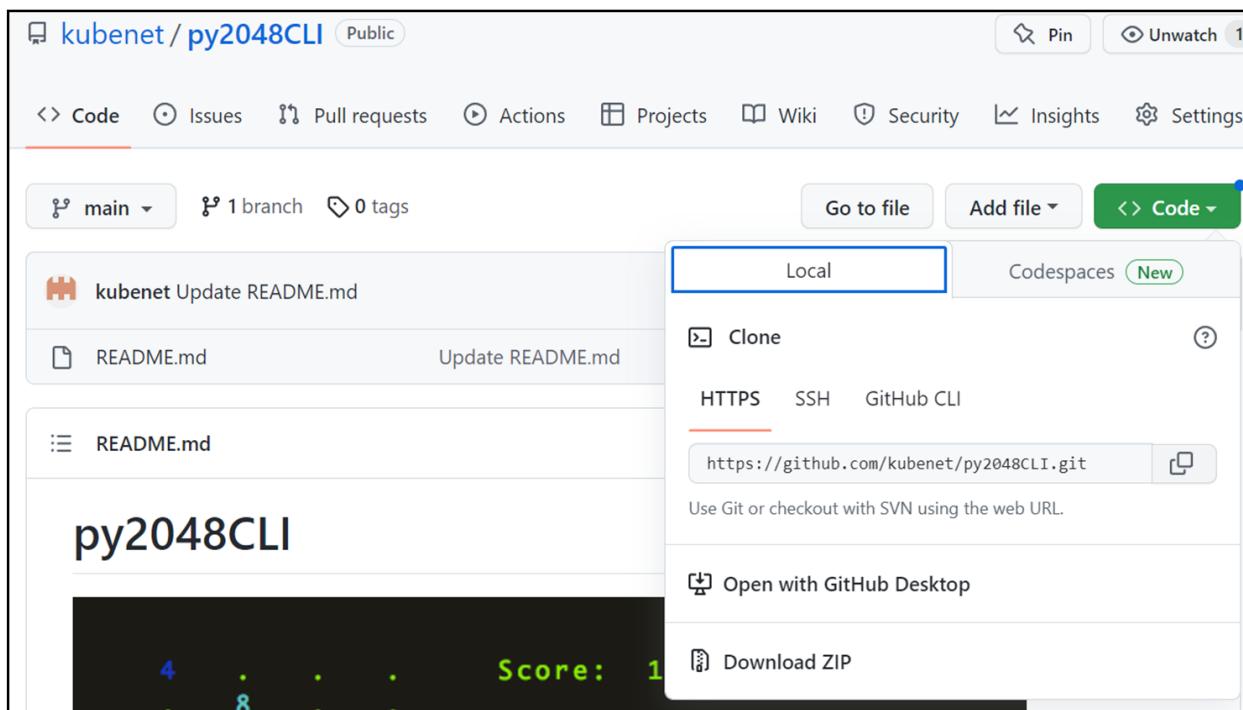
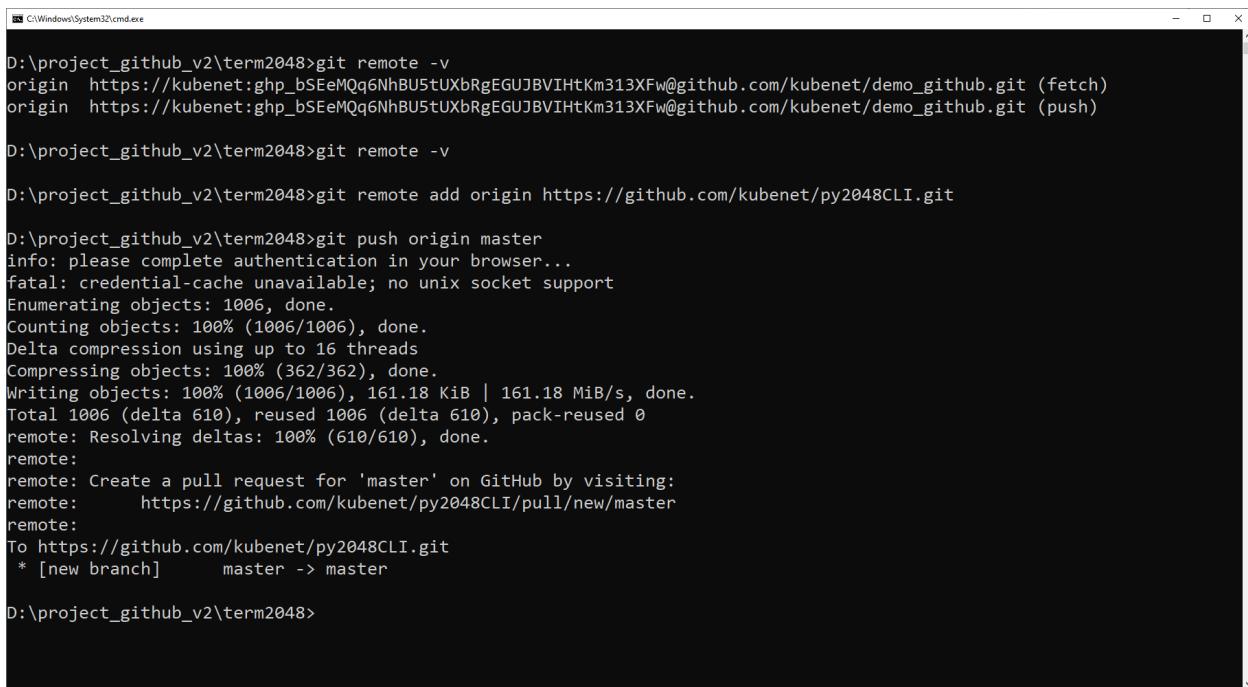


Рисунок 18 — Адрес удаленного репозитория в GitHub

Теперь перейдем в окно консоли git. Нам надо добавить ссылку на удаленный репозиторий в конфигурацию git на нашем локальном репозитории, чтобы система git понимала, куда отправлять код (сервер и репозиторий).

```
# Проверка удаленного адреса репозитория
git remote -v
# Указание на новый удаленный репозиторий
git remote add origin <<ссылка_на_удаленный_репозиторий*>
# Отправка изменений локального репозитория в удаленный, помеченный как источник
git push origin master
```

Система git потребует авторизации на сервере github. Нужно будет ввести логин и пароль, указанный при регистрации.



```

C:\Windows\System32\cmd.exe

D:\project_github_v2\term2048>git remote -v
origin  https://kubenet:ghp_bSEeMQt6NhBU5tUXbRgEGUJBVIHtKm313XFw@github.com/kubenet/demo.github.git (fetch)
origin  https://kubenet:ghp_bSEeMQt6NhBU5tUXbRgEGUJBVIHtKm313XFw@github.com/kubenet/demo.github.git (push)

D:\project_github_v2\term2048>git remote -v

D:\project_github_v2\term2048>git remote add origin https://github.com/kubenet/py2048CLI.git

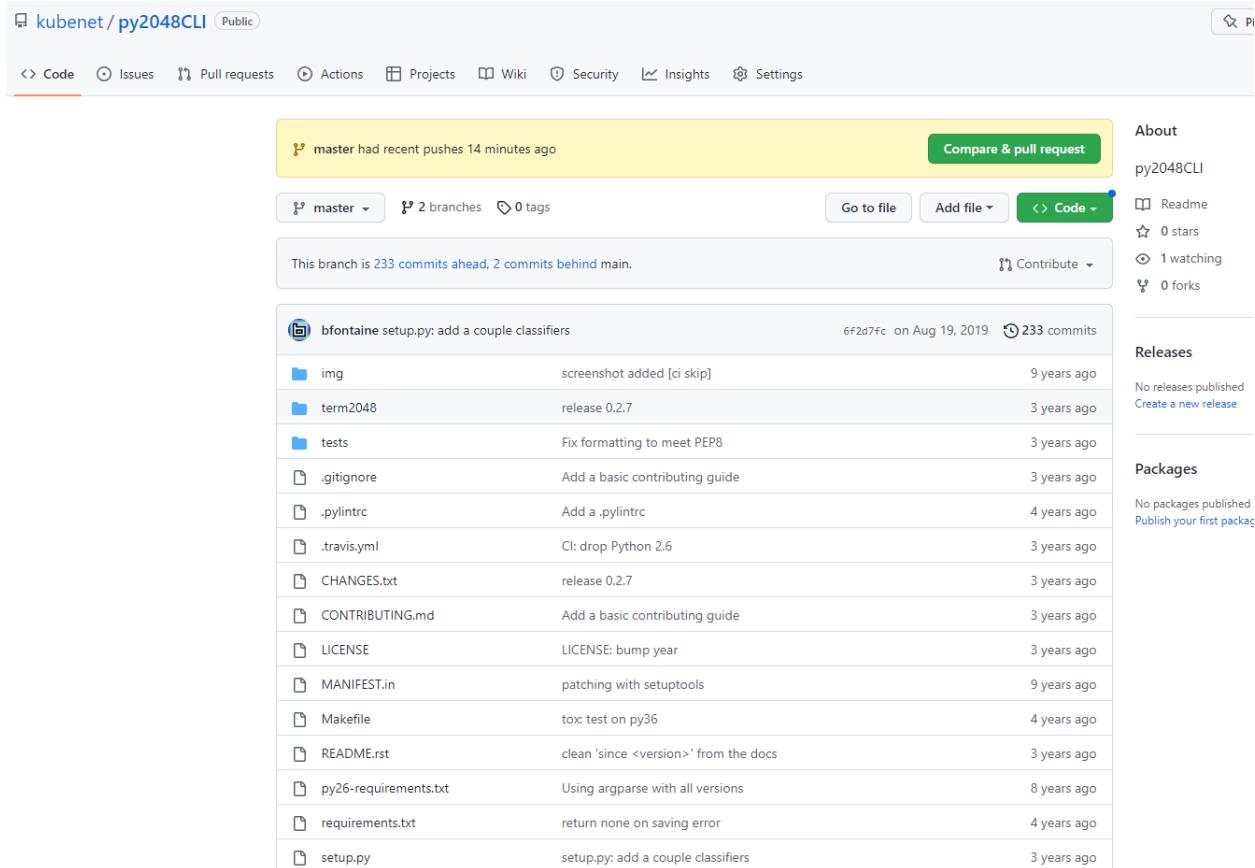
D:\project_github_v2\term2048>git push origin master
info: please complete authentication in your browser...
fatal: credential-cache unavailable; no unix socket support
Enumerating objects: 1006, done.
Counting objects: 100% (1006/1006), done.
Delta compression using up to 16 threads
Compressing objects: 100% (362/362), done.
Writing objects: 100% (1006/1006), 161.18 KiB | 161.18 MiB/s, done.
Total 1006 (delta 610), reused 1006 (delta 610), pack-reused 0
remote: Resolving deltas: 100% (610/610), done.
remote:
remote: Create a pull request for 'master' on GitHub by visiting:
remote:   https://github.com/kubenet/py2048CLI/pull/new/master
remote:
To https://github.com/kubenet/py2048CLI.git
 * [new branch]      master -> master

D:\project_github_v2\term2048>

```

Рисунок 19 — Успешная загрузка локального репозитория на GitHub

После успешной загрузки кода из локального репозитория мы увидим наши файлы в репозитории на GitHub (рисунок 20).



The screenshot shows the GitHub repository page for `kubenet / py2048CLI`. The repository is public and has 2 branches and 0 tags. The master branch is 233 commits ahead of main. The commit history lists 233 commits by `bfontaine`, with the most recent being `setup.py: add a couple classifiers` on Aug 19, 2019. The sidebar includes sections for About (py2048CLI, Readme, 0 stars, 1 watching, 0 forks), Releases (No releases published, Create a new release), and Packages (No packages published, Publish your first package).

Commit	Message	Date
<code>setup.py</code>	setup.py: add a couple classifiers	Aug 19, 2019
<code>MANIFEST.in</code>	patching with setuptools	9 years ago
<code>Makefile</code>	tox: test on py36	4 years ago
<code>README.rst</code>	clean 'since <version>' from the docs	3 years ago
<code>py26-requirements.txt</code>	Using argparse with all versions	8 years ago
<code>requirements.txt</code>	return None on saving error	4 years ago
<code>setup.py</code>	setup.py: add a couple classifiers	3 years ago
<code>CHANGELOG.md</code>	release 0.2.7	3 years ago
<code>CONTRIBUTING.md</code>	Add a basic contributing guide	3 years ago
<code>LICENSE</code>	LICENSE: bump year	3 years ago
<code>.travis.yml</code>	CI: drop Python 2.6	3 years ago
<code>.gitignore</code>	Add a .gitignore	3 years ago
<code>tests</code>	Fix formatting to meet PEP8	3 years ago
<code>term2048</code>	release 0.2.7	3 years ago
<code>img</code>	screenshot added [ci skip]	9 years ago

Рисунок 20 — Файлы проекта в удаленном репозитории GitHub

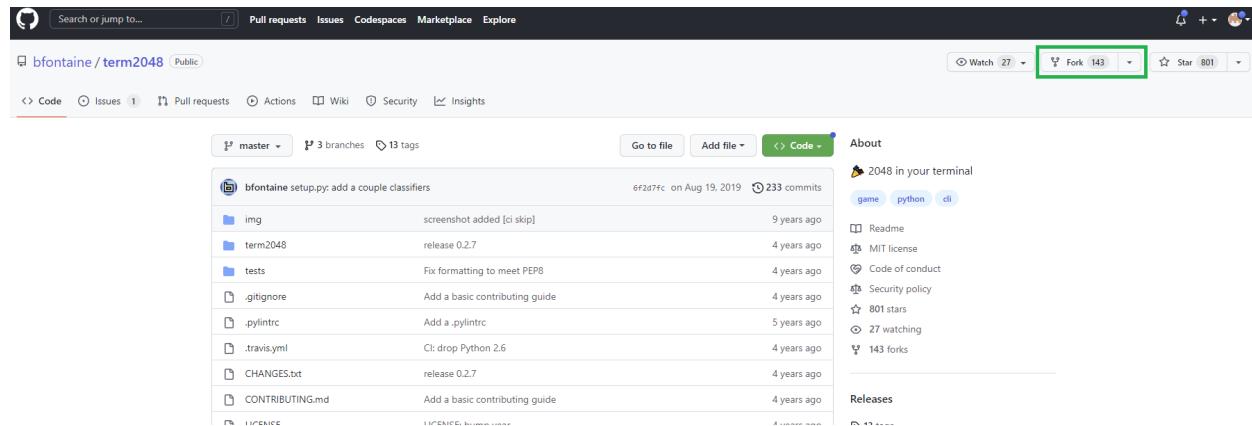
### Альтернативный вариант

Мы подробно рассмотрели вариант работы с GitHub, но есть и более простой способ создать копию чужого репозитория. Если вы хотите вносить свой вклад в уже существующие проекты, в которых у вас нет прав на внесение изменений путем их отправки (push), вы можете создать свое собственное ответвление (fork) проекта. Это означает, что GitHub создаст вашу собственную копию проекта, она будет находиться в вашем пространстве имен, и вы сможете легко менять ее путем отправки (push) изменений.

**Примечание** Исторически так сложилось, что англоязычный термин «fork» (создание ветвления проекта) имел негативный контекстный смысл. Он означал, что кто-то повел проект с открытым исходным кодом в другом, отличном от оригинала, направлении. Иногда данный термин также означал создание конкурирующего проекта с другими авторами. В контексте GitHub «fork» просто означает создание ветвления проекта в собственном пространстве имен, что позволяет вносить публичные изменения и делать свой собственный вклад в более открытом виде.

Таким образом, авторы проектов не обеспокоены тем, чтобы пользователи, которые хотели бы выступать в роли соавторов, имели право на внесение изменений путем их отправки (push). Люди просто могут создавать свои собственные ветвления (*fork*), вносить туда изменения, а затем отправлять свои внесенные изменения в оригиналный репозиторий проекта, создавая запрос на принятие изменений (*Pull Request*). Сами же запросы на принятие изменений (*Pull Request*) будут описаны далее. Запрос на принятие изменений (*Pull Request*) откроет новую ветвь с обсуждением отправляемого кода, и автор оригинального проекта, а также другие его участники, могут обсуждать предлагаемые изменения до тех пор, пока автор проекта не будет ими доволен, после чего сможет добавить предлагаемые изменения в проект.

Для того, чтобы создать ответвление проекта, зайдите на страницу проекта и нажмите кнопку «Создать ответвление» («*Fork*»), которая расположена в правом верхнем углу.



Далее

## Create a new fork

A *fork* is a copy of a repository. Forking a repository allows you to freely experiment with changes without affecting the original project. [View existing forks.](#)

Owner \* Repository name \*

 kubenet  / term2048 

By default, forks are named the same as their upstream repository. You can customize the name to distinguish it further.

Description (optional)

:tada: 2048 in your terminal

Copy the master branch only  
Contribute back to bfontaine/term2048 by adding your own branch. [Learn more.](#)

 You are creating a fork in your personal account.





## Глава 9

---

Раздел 2. Работа над проектом

---



## Стадии жизни проекта

---

### 10.1 Стадии жизни проекта

#### 10.1.1 От человека:

Одним из важных навыков является возможность делегировать часть своей работы. При этом делегировать можно как и другим людям, так и своим инструментам. Ранее мы говорили про фреймворки, которые берут на себя рутину, позволяя сосредоточиться на функциональности. Когда мы осуществляем поиск проблемы, мы также делегируем эту задачу поисковой машине (мало кто заходит далее, чем первая страница поисковой выдачи). Искусственный интеллект в разных своих проявлениях входит в нашу жизнь, и даже в такую сложную область, как разработка ПО и [создание](#) научных публикаций. Главная претензия к нему состоит в том, что инструменты, построенные с привлечением искусственного интеллекта, — это черный ящик, результат работы которого мы не можем полностью контролировать. Это так, но их можно использовать как ассистента, проводя оценку и корректировку выдачи данных от искусственного интеллекта. Нам кажется, что такие инструменты, как ChatGPT, являются одним из способов масштабировать свою производительность. Посмотрите вокруг: возможно, вы найдете в своей области и другие инструменты, которые позволят упростить рутинные операции. Но не перекладывайте на них творческую работу, именно она делает вас ценным. Данная глава была написана с помощью ChatGPT и является экспериментальной. Весь материал, приведенный ниже, прочитан и отредактирован или прокомментирован (каждый комментарий начинается так — **Комментарий:**...) человеком там, где это было нужно. Относитесь к содержимому данного занятия не только как к чему-то ценному самому по себе, но и как к демонстрации инструмента, который поможет вам коснуться темы, в которую вы бы хотели погрузиться. Вы можете проделать то же самое на эту же или другую тему, но помните, что критическое мышление вам не заменит ни одна нейросеть, кроме вашей.

### 10.1.2 Поехали

Запрос:

Стадии жизни проекта: идея, визуализация, архитектура, разработка (написание кода), отладка, документация, релиз, сборка в пакет, контейнеризация. Инструменты для достижения целей на каждом из этапов. Зачем нужно пользоваться инструментами.

#### От машины:

Этапы разработки:

- **Идея:** в этой стадии вы формируете общую концепцию проекта, определяете его основные цели и задачи.
- **Визуализация:** на этой стадии вы создаете схемы и другие визуальные материалы, которые помогают вам лучше представить проект как набор компонентов и процессов, протекающих между ними. В разработке ПО существуют общепринятые способы представления тех или иных процессов.
- **Архитектура:** здесь вы определяете структуру и компоненты проекта, их взаимодействие и зависимости.
- **Разработка:** на этой стадии вы пишете код, который реализует задачи, определенные на предыдущих этапах.
- **Отладка:** в этой стадии вы исправляете ошибки в коде, чтобы обеспечить правильную работу всего проекта.
- **Документация:** здесь вы создаете различные документы, описывающие проект и его компоненты, а также инструкции по его использованию и обслуживанию.
- **Релиз:** на этом этапе проект готов к использованию, и вы можете выпустить его для целевой аудитории.
- **Сборка в пакет:** здесь вы создаете установочный пакет или дистрибутив проекта, который может быть установлен на различные компьютеры.
- **Контейнеризация:** на этом этапе вы размещаете проект в контейнере, который позволяет запускать его на различных plataформах.

Инструменты для достижения целей на каждом из этапов:

- **Идея:** Блокноты, мозговые штурмы, доски для мозгового штурма и т.д.
- **Визуализация:** Adobe Photoshop, Adobe XD, Sketch, Figma, InVision, Balsamiq Mockups и т.д.
- **Архитектура:** UML-диаграммы, ER-диаграммы, ARIS, Microsoft Visio и т.д.
- **Разработка:** Visual Studio Code, PyCharm, IntelliJ IDEA, Eclipse, Xcode и т.д.
- **Отладка:** Инструменты для отладки, такие как PyCharm Debugger, Visual Studio Debugger, Xcode Debugger и т.д.
- **Документация:** Sphinx, Javadoc, Doxygen, Markdown и т.д.
- **Релиз:** Git, GitHub, GitLab, Bitbucket и т.д.\* (**Комментарий:** перечисленные системы являются системами контроля версий, но, как мы увидим далее по курсу, их также можно использовать как универсальный способ для дистрибуции своего кода, в отличии от, например, PyPI, который будет работать только с проектами на Python.)
- **Сборка в пакет:** InstallShield, MSI, RPM, Deb и т.д.

- **Контейнеризация:** Docker, Kubernetes, AWS Elastic Container Service и т.д.

**Комментарий:** были описаны только те пункты, которые мы указали в запросе. Например, такой важный аспект, как контроль версий, не был описан, а GitHub и GitLab отнесены к инструментам для релиза. Мы считаем, что контроль версий должен сопровождать проект на всех этапах, поэтому мы не выделили его как отдельный этап.

### 10.1.3 Идея

Запрос:

Расскажи подробнее про идею.

#### От машины:

Идея — это первый этап в жизненном цикле проекта, на котором формируется концепция проекта и определяются его цели и задачи. Идея может появиться из различных источников, таких как потребности рынка, проблемы, которые нужно решить, построение инфраструктуры вокруг новых технологий или развитие существующих идей.

Важным аспектом на этапе идеи является определение целевой аудитории, которая будет использовать продукт. Необходимо понимать, с какой проблемой они сталкиваются, какие потребности у них возникают и как проект может помочь решить их проблемы и удовлетворить потребности. **Комментарий:** на наш взгляд, понимание целевой аудитории очень важно для успеха вашей разработки. Если вы предлагаете решение проблемы X для пользователей программы Y, работающей только под Windows, но реализуете свое решение под Linux, вы потеряете большую часть своих пользователей. Это искусственный пример, как и все в этой главе, но он дает понять важность правильной оценки целевой аудитории и ваших возможностей по улучшению ее жизни.

При формировании идеи необходимо проводить анализ рынка и конкурентов, чтобы определить, какой потенциал имеет проект, насколько он уникален и востребован на рынке. Это поможет избежать ошибок в концепции проекта и определить его конкурентоспособность. **Комментарий:** уникальность очень сложно установить: ваш проект может быть немного лучше по всем параметрам, чем у конкурентов, или лучше по одному, но намного, или в него добавлено что-то новое, например, новая интеграция или функциональность, или же вообще нефункциональные свойства, такие как легкость установки. Например, для этого курса мы пробовали несколько редакторов Markdown, один, reMarkable, не запустился сходу, и мы решили не тратить время и попробовать следующий в списке, ReText. Он установился, и мы сразу начали писать текст, юзабилити нас устроило. Позже выяснилось, что экспортированные PDF-документы не такие красивые, как нам бы хотелось, и мы взяли третий редактор (осторожно, не FOSS) Turoga, и использовали только для конвертации одного из уроков. Позже рабочие документы просматривались на GitLab, а итоговая сборка сделана с использованием Sphinx. Т.е. для нас каждый из этих инструментов оказался по-своему уникальным.

Кроме того, на этом этапе необходимо определить бюджет и время, которое потребуется для реализации проекта, а также сформировать команду проекта и определять их роли. **Комментарий:** мы не указывали в запросе, что речь идет об open source, возможно, тогда ответ был бы другим. Определение времени является крайне важным, особенно для open source проектов, так как позволяет установить сроки достижения желаемых результатов и, как следствие, получить дополнительную мотивацию, когда все идет по плану. Определить бюджет сложнее, так как участники таких проектов вкладывают свое свободное от основных занятий время.

Инструменты, которые могут помочь на этом этапе, могут быть различными. Например, использование блокнотов, чтобы записать все идеи и мысли, проведение мозгового штурма, создание макетов, концептуальных чертежей и т.д. Важно выбирать инструменты, которые подходят конкретному проекту и команде и которые помогают сформировать ясное понимание концепции проекта и его потенциала.

**Комментарий:** также важно оценивать достоверность озвученных предложений с учетом доступных командам ресурсов и инструментов. В 2017 мы хотели сделать конструктор для изучения ориентации робота внутри помещения, и на каком-то этапе стало понятно, что нужен лидар — устройство, которое позволяет достаточно быстро получить информацию о препятствиях на 360 градусов от робота. Тогда не было доступных вариантов, и задачу пришлось решать за счет внешней системы видеонаблюдения. Сейчас доступные лидары существуют и могут быть включены в состав конструктора. Отсутствие лидара влияет на круг задач, которые могут быть рассмотрены с применением конструктора, что может повлиять на успех решения.

#### **10.1.4 Визуализация**

Запрос:

Расскажи подробнее про визуализацию.

**От машины:**

**Визуализация** — это процесс создания визуальных материалов, которые помогают проиллюстрировать концепцию проекта. Этот этап жизненного цикла проекта является важным, так как визуализация помогает уточнить концепцию, сделать ее более понятной для других членов команды, заказчиков и потенциальных пользователей.

Основной целью визуализации является создание прототипов и макетов, которые помогут представить, как будет выглядеть и работать готовый продукт. Это может быть макет веб-сайта или мобильного приложения, физический прототип устройства, дизайн интерфейса пользователя и т.д. Визуализация также может включать в себя создание документации, которая описывает детали проекта, такие как функциональные требования, технические спецификации, диаграммы потоков данных и т.д.

Визуализация может быть выполнена вручную или с использованием специальных программных инструментов. Существует множество инструментов для создания прототипов и макетов, таких как Sketch, Figma, Adobe XD, Axure, InVision и т.д. Они позволяют создавать интерактивные прототипы, которые можно тестировать на пользователях и улучшать на основе обратной связи.

Визуализация помогает сократить время и издержки на разработку, так как позволяет увидеть ошибки и проблемы на ранних этапах проекта. Она также помогает уточнить требования и дизайн продукта, улучшить его юзабилити и общее качество, что в итоге способствует увеличению удовлетворенности пользователей и успеху проекта.

#### **10.1.5 Архитектура**

Запрос:

Расскажи подробнее про архитектуру.

## От машины:

**Архитектура** — это процесс проектирования структуры и организации компонентов проекта, включая аппаратные, программные и сетевые компоненты, для достижения определенных целей. Она определяет взаимодействие между компонентами и обеспечивает соответствие требованиям качества проекта, таким как производительность, надежность, безопасность и масштабируемость. **Комментарий:** *позже мы будем говорить о тестах, и архитектура будет опосредованно тестируться интеграционными и системными тестами. Хорошая архитектура упрощает тестирование системы, делая его более прозрачным и легким в написании.*

В зависимости от проекта мы можем разбивать архитектуру на одну или несколько частей:

- Архитектура приложения определяет, как будут взаимодействовать различные компоненты приложения, включая клиентскую и серверную стороны, базы данных, хранилища и т.д.
- Архитектура базы данных определяет структуру и организацию данных, а также методы доступа к данным.
- Архитектура сетевого взаимодействия определяет структуру сети, включая сетевые протоколы, устройства и маршрутизацию.\* **Комментарий:** *это важно для приложений, где данные поступают из разнородных источников. Например, система, которую мы разрабатываем для научных расчетов SIMuRG, основана на микросервисах, которые могут быть расположены как на компьютерах внутри локальной сети, так и на удаленных серверах. При таком подходе нам важно, как взаимодействуют узлы системы и сможем ли мы быстро перекинуть 10 Гб данных (пропускная способность) для проведения расчетов, или сможем ли мы настроить поток данных для функционирования в режиме реального времени (наличие подходящего протокола).*
- Архитектура безопасности определяет механизмы защиты информации и данных от несанкционированного доступа и взлома.

Для достижения цели построения хорошей архитектуры необходимо использовать специальные инструменты и технологии. Например, диаграммы UML — для визуализации архитектурных концепций и отношений между компонентами. Или инструменты моделирования и прототипирования — для создания и тестирования прототипов компонентов, а также моделирования взаимодействия между компонентами. И подходящие методологии разработки, такие как Agile или DevOps.

Архитектура помогает увеличить вероятность, что проект будет соответствовать требованиям заказчика и стандартам отрасли, а также обеспечивает эффективность и производительность проекта на всех его этапах.

**Комментарий:** *к этому можно добавить, что наличие архитектуры позволяет вовремя заметить, что мы делаем в проекте что-то, что потом будет невозможно поддерживать. Таким образом, архитектура отвечает не только на вопрос, что мы делаем, но и позволяет аргументировать, чего мы делать точно не будем. В open source мы будем аргументировать это прежде всего себе, что может уберечь нас от чрезмерных «костылей» в коде, которые рано или поздно могут обрушить всю систему.*

### 10.1.6 Разработка

Запрос:

Расскажи подробнее про разработку.

## От машины:

**Разработка** — это этап создания программного продукта, который включает в себя написание кода, тестирование и интеграцию различных компонентов. Важной задачей на этом этапе является реализация концептуального дизайна, представленного на этапе визуализации и архитектуры. **Комментарий:** в русском языке дизайн ассоциируется с графическим дизайном, в английском *design* имеет более широкий смысл. То, что мы разработали на предыдущем этапе, является дизайном нашего приложения, который мы должны воплотить в коде.

На этапе разработки используются языки программирования, фреймворки, библиотеки и инструменты разработки. Разработчики пишут код, который должен соответствовать определенным стандартам качества, например, должен быть читабельным, эффективным и надежным.

Разработка может происходить по разным методологиям, таким как Agile, Waterfall или DevOps. В Agile разработка происходит в коротких итерациях, называемых спринтами, каждый из которых заканчивается работающим продуктом. В Waterfall разработка происходит последовательно, с каждым этапом, зависящим от успешного завершения предыдущего этапа. В DevOps разработка и операции объединены в цикл, чтобы обеспечить быстрое и надежное создание и внедрение продукта.

Для разработки проектов используются интегрированные среды разработки (IDE) и инструменты версионного контроля, такие как Git, SVN, Mercurial и др. IDE предоставляет программисту множество инструментов для написания кода, отладки и тестирования, а системы версионного контроля позволяют отслеживать изменения в коде и управлять версиями программного продукта.

### 10.1.7 Отладка

Запрос:

Расскажи подробнее про отладку.

## От машины:

**Отладка** — это процесс выявления, анализа и исправления ошибок в программном коде. Ошибки могут быть логическими, синтаксическими, семантическими, временными или другими, и могут привести к неправильной работе программы или даже к ее аварийному завершению.

Отладка начинается с выявления ошибки. Это может произойти в процессе тестирования продукта или приложения, а также благодаря обратной связи пользователей. На этапе отладки разработчик использует различные инструменты и технологии, чтобы определить причину ошибки и исправить ее. **Комментарий:** далее мы будем рассматривать тестирование подробнее. Написание тестов до или после создания кода не является отладкой само по себе. Существует такое понятие, как разработка через тестирование, и создание тестов можно отнести больше к разработке. Отладка помогает привести одну часть программного обеспечения (код) к другой (тестам). Да, тесты это тоже код, мы знаем. Тесты помогают выявлять ошибки, отладка нацелена на их исправление и достижение соответствия кода требованиям через тесты или через удовлетворения потребностей пользователя. Когда мы привлекаем пользователей к тестированию продукта, как правило, они взаимодействуют не с кодом, а с готовым приложением, и разработчики, опираясь на отзывы пользователей, пытаются воспроизвести ошибку и увидеть, что в коде пошло не так.

Инструменты отладки могут варьироваться в зависимости от языка программирования и платформы. В общем инструменты отладки позволяют программистам проверять переменные, отслеживать выполнение кода, устанавливать точки останова, анализировать стек вызовов и многое другое. **Комментарий:** существует профессия тестировщика программного обеспечения. И даже можно услышать «приходите в разработку (или в IT)» через тестирование. Следует помнить, что разработчики и

*тестировщики — это разные профессии, в них да же хотят разного: разработчики — чтобы все работало, а тестировщики — чтобы что-то сломалось. Они также используют и разный инструментарий: разработчики пишут код и юнит-тесты, тестировщики — нет. Тестировщики подключаются на этапе тестирования более высокого уровня.*

Один из наиболее распространенных инструментов отладки — это отладчик, инструмент, который позволяет программистам запускать код пошагово, отслеживать выполнение программы, устанавливать точки останова и проверять значения переменных во время выполнения. Также используются различные виды логирования, что позволяет программистам понимать, что происходит в коде на каждом этапе выполнения программы.

Отладка может быть достаточно трудоемкой задачей, поскольку ошибки бывают довольно сложными и неочевидными. Кроме того, исправление ошибки может привести к появлению новых ошибок или нежелательному изменению поведения программы. Поэтому важно проявлять терпение и методичность в процессе отладки.

Хорошая отладка помогает обеспечить качество программного продукта и минимизировать вероятность появления ошибок при его эксплуатации. **Комментарий:** мы можем просто исправить ошибку или сформулировать предупреждение. В процессе отладки, работая над исправлением ошибки A, мы можем понять, что другой набор данных тоже может привести к ошибке A. Также мы можем увидеть, что ситуация B не протестирована. Сейчас все работает, но неплохо было бы написать дополнительные тесты.

### 10.1.8 Документация

Запрос:

Расскажи подробнее про документацию.

#### От машины:

**Документация** — это набор документов, который описывает программный продукт и его функциональные возможности, его архитектуру, внутреннюю структуру и правила использования. Она может включать в себя как описание внешнего интерфейса программы (API), так и внутреннего функционирования.

Документация является важной частью процесса разработки программного обеспечения, поскольку она облегчает понимание программного кода, обеспечивает легкий доступ к информации о программном продукте, а также упрощает сопровождение и модификацию программного кода. **Комментарий:** документация, как и тестирование, помогает посмотреть на код, который мы пишем, с другой стороны: если мы не можем описать то, что происходит на этом участке программы, равно как если мы не можем написать для него тест, или мы можем написать документацию, но она будет состоять из множества если... то..., возможно, нам стоит пересмотреть реализацию этого участка программы.

Существует множество инструментов для создания документации, включая генераторы документации, редакторы документации и инструменты для создания диаграмм и схем. Некоторые из этих инструментов, такие как Doxygen, Javadoc и Sphinx, позволяют автоматически создавать документацию на основе комментариев в исходном коде.

Документация может содержать следующие разделы:

**Описание программы и ее функциональности:** этот раздел обычно включает описание основных функций, возможностей и особенностей программы, а также ее целевую аудиторию и предполагаемый способ использования.

**Архитектура и дизайн:** этот раздел обычно описывает внутреннюю структуру программы, ее компоненты, зависимости и отношения между ними.

**Руководство пользователя:** этот раздел описывает то, как использовать программу, ее функции и возможности, а также особенности пользовательского интерфейса и взаимодействия пользователя с программой. **Комментарий:** Мы бы еще включили туториалы как один из важных видов документации. В строгом смысле они не являются документацией, но существенно ускоряют начало использования, особенно когда речь идет о фреймворке или большой библиотеке. В отличие от руководства пользователя туториалы концентрируются на одной задаче, что позволяет пользователю быстро получить результат и решить, что проект ему полезен. Новый пользователь скорее всего пойдет сначала в туториалы, если они есть.

**Руководство по установке и настройке:** этот раздел описывает процесс установки и настройки программы, ее системные требования и зависимости.

**Руководство по разработке:** этот раздел описывает, как разрабатывать программу, какие инструменты использовать, каким правилам оформления и кодирования следовать.

Хорошо написанная и актуальная документация улучшает качество программного продукта и повышает удобство его использования. Она может также помочь уменьшить количество ошибок, связанных с неправильным использованием программы. **Комментарий:** Здесь под ошибкой подразумевается не только неправильная работа приложения или кода, но и неправильное использование. Каждый проект имеет свои ограничения, и если пользователь пытается использовать его не для того, на что он был рассчитан, он получит негативный опыт.

### 10.1.9 Релиз

Запрос:

Расскажи подробнее про релиз.

**От машины:**

**Релиз** (от англ. «release») — это процесс публикации программного продукта после его разработки и тестирования. Это конечный этап в жизненном цикле проекта, который предполагает официальный выпуск программы для широкого использования. **Комментарий:** позже на примере проекта на Python мы посмотрим, что релиз можно сделать в виде собранного пакета (*Build Distribution*) или в виде исходного кода (*Source Distribution*). Про сборку в виде пакета смотрите в следующей секции.

Процесс релиза обычно включает следующие этапы:

- Финальное тестирование: перед выпуском программного продукта должна быть проведена окончательная проверка его работоспособности и соответствия требованиям. Оно может включать автоматические и ручные тесты, проверку безопасности и производительности.
- Подготовка к выпуску: на этом этапе происходит подготовка к выпуску, включая подготовку установочных файлов, документации, лицензий и прочего.
- Распространение: программный продукт может быть распространен различными способами, включая загрузку с веб-сайта, установку из магазинов приложений, распространение по сети, установку с флэш-накопителей и т.д.
- Поддержка: после выпуска программного продукта может потребоваться его поддержка, включая предоставление технической поддержки, исправление ошибок и обновление программного кода.
- Обновление: после выпуска программного продукта может потребоваться его обновление, которое может включать исправление ошибок, улучшение функциональности и т.д.

Для успешного релиза программного продукта необходимо убедиться в его качестве, соответствии требованиям, безопасности и производительности. Для этого можно использовать различные инструменты, такие как системы контроля версий, системы автоматического тестирования, утилиты для анализа кода, системы мониторинга и т.д.

Релиз является важным этапом в жизненном цикле программного продукта, поскольку он позволяет программистам и компаниям получить обратную связь от пользователей, улучшить качество и функциональность программного продукта, а также расширить свою аудиторию и получить коммерческий успех.

**Комментарий:** *При разработке open source проекта его код доступен в открытом доступе. Мы все равно можем говорить о релизах проекта как о точках, в которых сообщество разработчиков наиболее уверено в качестве продукта. Также компании могут открыть часть своего кода по open source лицензии.*

### 10.1.10 Сборка

Запрос:

Расскажи подробнее про сборку в пакет.

#### От машины:

Сборка в пакет (package build) — это процесс создания программного пакета, который содержит скомпилированный код, необходимые библиотеки, настройки и другие файлы, необходимые для установки и запуска программы на другом компьютере или операционной системе.

Программные пакеты могут быть использованы для упрощения установки и распространения программного обеспечения. Вместо того, чтобы пользователь должен был скачать, скомпилировать и устанавливать программу вручную, он может просто скачать готовый пакет и установить его на своем компьютере. **Комментарий:** *Основная задача — сделать так, чтобы пользователь (не разработчик) смог воспользоваться проектом. Мы можем распространять код через git и собирать его с помощью Make, но это вряд ли захочет делать каждый человек. Он хочет просто нажать на кнопку и получить на своем компьютере приложение. Для этого нужно использовать встроенные в операционную систему менеджеры типа apt и упаковывать свой проект в соответствии с их требованиями.*

Процесс сборки в пакет обычно включает следующие шаги:

- Сборка программного кода: перед тем, как собрать пакет, необходимо скомпилировать программный код и создать исполняемый файл или библиотеку.
- Упаковка необходимых файлов: после того, как скомпилированный код создан, необходимо упаковать его в пакет вместе с другими необходимыми файлами, такими как настройки, файлы конфигурации, библиотеки и прочее.
- Создание метаданных: после того, как пакет сформирован, необходимо создать метаданные, такие как описание пакета, версия, автор и прочее.
- Установка и тестирование: после того, как пакет сформирован и упакован, необходимо установить его на тестовой системе и протестировать его, чтобы убедиться, что он работает корректно.
- Распространение: после того, как пакет прошел тестирование и готов к использованию, его можно распространять, например, публикую его на веб-сайте или дистрибутиве Linux.

Сборка в пакет может быть автоматизирована с помощью специальных инструментов, таких как системы управления пакетами, которые могут упростить процесс создания, тестирования и распространения пакетов. **Комментарий:** *Этот курс собирался с помощью Sphinx и небольшого скрипта на Python,*

который формировал правильный порядок файлов в `index.rst`. После этого было достаточно написать `make latexpdf` и получить весь курс в виде PDF-файла.

Преимущества использования пакетов включают упрощение установки и обновления программного обеспечения, сокращение времени и усилий, необходимых для настройки новой системы, а также более простую и надежную процедуру развертывания программного обеспечения на серверах.

### 10.1.11 Контейнеризация

Запрос:

Расскажи подробнее про сборку в контейнеризацию.

**От машины:**

**Контейнеризация** — это процесс создания, управления и развертывания приложений в изолированных контейнерах, которые содержат все необходимые для работы приложения компоненты: код, библиотеки, настройки, зависимости и т.д. **Комментарий:** *Больше похоже на запуск и использование, но нам показалось важным включить этот пункт именно в инструменты разработчика, так как это позволяет решать проблемы с окружением на компьютерах пользователей, а также легче тестировать проект. В системах, состоящих из нескольких взаимодействующих по сети узлов, контейнеры являются очень удобной альтернативой виртуальным машинам.*

Основная идея контейнеризации состоит в том, чтобы создать изолированную среду для приложения, которая не будет зависеть от других приложений или операционной системы, установленной на сервере. Каждый контейнер имеет свои ресурсы, такие как процессы, сетевые интерфейсы, файловые системы и т.д., которые не будут использоваться другими контейнерами. Контейнеры могут быть созданы с использованием различных инструментов, таких как Docker, Kubernetes, LXC и другие. Процесс контейнеризации включает следующие шаги:

- Создание образа контейнера: экземпляр контейнера создается на основе образа, который содержит все необходимые компоненты приложения.
- Конфигурирование контейнера: после того, как образ создан, необходимо настроить контейнер, например, задать настройки сети, задать переменные окружения и т.д.
- Запуск контейнера: после того, как контейнер настроен, он может быть запущен.
- Управление контейнерами: после запуска контейнерами можно управлять, например, останавливать и перезапускать.

Преимущества контейнеризации:

- Улучшенная мобильность и портативность приложений: приложения могут быть легко перенесены между различными хостами, операционными системами и облачными платформами.
- Более быстрый и простой процесс развертывания и обновления приложений: развертывание новой версии приложения может быть выполнено всего за несколько минут.
- Упрощение управления приложениями: контейнерами можно быстро и легко управлять и мониторить их, а приложения, построенные на основе контейнеров, легко масштабировать, запуская новые экземпляры контейнера.
- Уменьшение зависимостей между приложениями: каждый контейнер работает в изолированной среде, что позволяет избежать конфликтов между зависимостями приложений и упрощает управление версиями.

## Минимально полезный продукт

---

### 11.1 Работа над MVP

#### 11.1.1 Этапы разработки и проверка гипотез

Знакомим с идеей MVP и Pet-project. Проект выполняется на языке Python и должен соответствовать всем критериям и рекомендациям, представленным в предыдущих пунктах. Каждое соответствие и наличие выполненного пункта оценивается в +1 балл (если были замечания к реализации — 0,5 балла).

#### Что такое MVP и из каких этапов он состоит?

MVP (минимально жизнеспособный продукт) — это концепция разработки продукта, которая относится к прототипу или версии продукта с количеством функций, достаточным, чтобы иметь возможность протестировать идею продукта с реальными пользователями.

#### Какова цель MVP?

Цель MVP — протестировать и подтвердить идею продукта на небольшой репрезентативной выборке целевого рынка, прежде чем выделять значительные ресурсы на полномасштабную разработку. MVP предназначен для предоставления информации и отзывов о ценностном предложении продукта, его удобстве использования и его потенциале для успеха.

Запуская MVP, создатели продукта могут глубже понять свой рынок, потребности и болевые точки своих клиентов, а также жизнеспособность концепции своего продукта. Это помогает им принимать обоснованные решения о том, какие функции включить в конечный продукт, как позиционировать его на рынке и как наилучшим образом привлечь и обслужить своих целевых клиентов.

По сути, MVP — это малорискованный и недорогой способ тестирования идеи продукта, сбора отзывов и доработки продукта перед его запуском в более крупном масштабе. Таким образом, это может помочь предпринимателям и продуктовым командам повысить шансы на успех своего продукта.

**MVP обычно состоит из следующих этапов:**

1. Идея: этот этап включает в себя создание и оценку идей для продукта.
2. Исследование: этот этап включает в себя исследование целевого рынка, конкуренции и отрасли в целом, чтобы определить наилучшую концепцию продукта.
3. Проектирование: этот этап включает в себя создание базового дизайна для MVP с упором на его основные функции и взаимодействие с пользователем.
4. Разработка: этот этап включает в себя процесс создания MVP, для которого обычно требуется кодирование и тестирование, чтобы убедиться, что он работает должным образом.
5. Релиз: этот этап включает в себя выпуск MVP для выбранной группы пользователей для тестирования и получения отзывов.
6. Рефлексия и тестирование: этот этап включает в себя учет отзывов пользователей в продукте и повторение процесса проектирования и разработки, пока продукт не будет готов к полноценному запуску.

#### **11.1.2 Вот пример MVP для новой службы доставки еды:**

Идея проекта — создать сервис для службы доставки еды людям, которые ведут активный образ жизни, часто передвигаются по городу, и у них нет времени готовить самостоятельно, но они все же хотят есть здоровую и питательную пищу.

- Этап исследования: исследование показывает, что спрос на услуги по доставке здоровой пищи растет, особенно среди сотрудников среднего и крупного бизнеса. Однако большинство существующих услуг либо слишком дороги, либо предлагают ограниченный ассортимент блюд.
- Проектирование: MVP будет включать в себя простой веб-сайт и мобильное приложение, которые позволяют пользователям размещать заказы на полезные готовые блюда. Дизайн будет минималистичным и простым в использовании, ориентированным на обеспечение удобного процесса заказа здоровой пищи и информирование клиента о его статусе готовности.
- Разработка: MVP будет создан с использованием комбинации готовых инструментов и пользовательского кода. Он будет включать в себя базовую систему заказов, платежный шлюз и функцию отслеживания доставки. MVP будет масштабироваться, чтобы его можно было легко расширять в будущем.
- Релиз: MVP будет запущен на ограниченной территории, например, в небольшом городе, для избранной группы первых пользователей. Цель — протестировать продукт, собрать отзывы и подтвердить рыночный спрос.
- Рефлексия и тестирование: на основе полученных отзывов MVP будет дорабатываться и улучшаться. Это может быть добавление дополнительных опций меню, улучшение процесса доставки или улучшение процесса заказа. Цель в том, чтобы постоянно улучшать продукт и делать его более привлекательным для более широкой аудитории, пока он не будет готов к полноценному запуску.

### 11.1.3 Вот пример MVP для нового мобильного приложения:

Идея состоит в том, чтобы создать мобильное приложение, которое помогает людям управлять своими финансами, отслеживая расходы, доходы и сбережения.

- Исследование: исследование показывает, что существует растущий спрос на инструменты управления личными финансами, особенно среди молодых людей, которые только начинают управлять своими финансами. Однако большинство существующих инструментов либо слишком сложны, либо слишком ограничены.
- Дизайн: MVP будет включать в себя простое и удобное мобильное приложение, позволяющее пользователям отслеживать свои расходы, доходы и сбережения. Приложение будет простым в использовании, с чистым и интуитивно понятным интерфейсом.
- Разработка: MVP будет создан с использованием комбинации готовых инструментов и пользовательского кода. Он будет включать в себя такие функции, как отслеживание расходов, отслеживание доходов, составление бюджета и постановка целей сбережений. MVP будет масштабироваться, чтобы его можно было легко расширять в будущем.
- Запуск: MVP будет запущен для избранной группы первых пользователей. Цель — протестировать продукт, собрать отзывы и подтвердить рыночный спрос.
- Итерация: на основе полученных отзывов MVP будет дорабатываться и улучшаться. Это может включать добавление новых функций, улучшение пользовательского интерфейса или расширение возможностей финансового отслеживания и анализа.

Цель в том, чтобы постоянно улучшать продукт и делать его более привлекательным для более широкой аудитории, пока он не будет готов к полноценному запуску.

#### **Список рекомендаций, чтобы подготовить команду программистов к MVP проекту, посвященному разработке мобильного приложения для школы**

Чтобы подготовить команду программистов к разработке проекта MVP для школьного мобильного приложения, вы можете выполнить следующие шаги:

1. Определите объем проекта: четко определите цели, требования и ожидания от MVP и убедитесь, что все в команде понимают, что от них ожидается.
2. Соберите правильную команду: выберите команду опытных, талантливых и целеустремленных программистов, обладающих навыками и опытом, необходимыми для создания MVP. Убедитесь, что команда разнообразна и включает в себя людей с разным опытом и взглядами.
3. Установите четкие ожидания. Четко сообщите команде сроки, бюджет и этапы проекта и убедитесь, что все согласны с ожиданиями проекта.
4. Поощряйте открытое общение: поощряйте открытое и честное общение между членами команды и следите за тем, чтобы каждый знал свои роли и обязанности.
5. Поощряйте командную работу: поощряйте культуру сотрудничества и командной работы и убедитесь, что каждый член команды чувствует, что его ценят и поддерживают.
6. Используйте правильные инструменты: предоставьте команде необходимые инструменты, технологии и ресурсы для создания MVP, такие как программное обеспечение для управления проектами, среды разработки и инструменты тестирования.
7. Предложите обучение и поддержку: предложите обучение и поддержку, чтобы помочь команде развить свои навыки и улучшить свою работу.
8. Отмечайте успехи: отмечайте успехи и вехи на пути к успеху и убедитесь, что команда чувствует, что ее труд ценят.

Так вы можете помочь своей команде программистов более эффективно и успешно разработать проект MVP для школьного мобильного приложения.

#### **11.1.4 Как оценить качество MVP, какие показатели, критерии?**

Для оценки качества MVP можно использовать следующие показатели и критерии:

- отзывы пользователей: собираите отзывы клиентов, которые использовали MVP, и анализируйте эти отзывы, чтобы определить области для улучшения и внести соответствующие изменения;
- соответствие рынку: оцените, насколько хорошо MVP соответствует потребностям и предпочтениям целевого рынка, и при необходимости внесите корректировки;
- вовлеченность пользователей: измерьте, насколько хорошо пользователи взаимодействуют с MVP, и используйте эти данные, чтобы определить области для улучшения;
- производительность и надежность: оцените производительность и надежность MVP и определите любые технические проблемы или ошибки, которые необходимо устранить;
- простота использования: оцените, насколько легко использовать MVP, и при необходимости внесите изменения, чтобы улучшить взаимодействие с пользователем;
- масштабируемость: оцените масштабируемость MVP и определите любые ограничения, которые необходимо устранить;
- привлечение и удержание пользователей: измеряйте уровень привлечения и удержания пользователей и при необходимости вносите изменения, чтобы улучшить оба показателя;
- доход и рентабельность: оцените доход и прибыльность MVP и при необходимости внесите изменения, чтобы увеличить оба показателя.

Используя эти индикаторы и критерии, вы сможете эффективно оценить качество вашего MVP и внести улучшения, чтобы убедиться, что он соответствует потребностям вашего целевого рынка и обеспечивает отличный пользовательский опыт.

#### **Где используется MVP, в чем отличие от Pet Project?**

MVP (минимально жизнеспособный продукт) используется при разработке продукта, это термин, используемый для описания продукта с минимальным набором функций, достаточных для проверки концепции продукта и сбора отзывов клиентов.

PetProject — это личный проект, над которым человек работает для собственного удовольствия или личной выгоды, без каких-либо конкретных коммерческих или деловых целей.

Ключевое различие между ними заключается в том, что MVP сосредоточен на тестировании идеи продукта и сборе отзывов от клиентов, в то время как Pet-project ориентирован на личное удовлетворение и развитие навыков разработчика. MVP часто разрабатываются с расчетом на конкретный рынок или целевую аудиторию, а домашние проекты обычно разрабатываются по личным причинам, без учета какого-либо рынка или аудитории.

Таким образом, MVP используется при разработке продукта как способ проверки концепции продукта и сбора отзывов клиентов, а личный проект — это частный проект без конкретных коммерческих или бизнес-целей.

### 11.1.5 Задание для проверки

Выше мы познакомились с основными тезисами относительно разработки MVP. Представьте, что вам нужно разработать чат-бота для поиска информации о доступных олимпиадах с возможностью поиска и формирования списка избранных олимпиад. Ниже представлены шаги, которые вам следует выполнить самостоятельно:

1. Проведите исследование, какие чат-боты уже есть на рынке. Посмотрите, что они делают и как они работают, для решения каких задач их используют.
2. Разберитесь, для кого будет ваш чат-бот и что он будет делать. Какие задачи он будет выполнять?
3. Нарисуйте прототип того, как будет выглядеть и работать ваш чат-бот. Подумайте, как он будет общаться с клиентами (подписчиками).
4. Выберите примерный список технологий и инструментов, которые могут использоваться для создания чат-бота (библиотеки, база данных, API, сервер на котором будет развернут бот и т.д.).
5. Оцените длительность периода разработки: сколько примерно времени и ресурсов вам понадобится для создания чат-бота. Составьте план работы.
6. Подумайте, какие проблемы могут возникнуть в процессе разработки и как их решить.
7. Если у вас есть команда, определите, кто что будет делать и как вы будете общаться и контролировать процесс разработки. Например, можно использовать сервис Trello или его аналог, в котором можно перенести задачи на карточки и обозначить сроки и исполнителей.
8. Определите, как вы будете проверять, что чат-бот работает правильно. Разработайте несколько тестов, которые будут проверять работу основных сервисов (пинг API сервиса, проверка подключения к СУБД, статус приложения/сервера).
9. Начните разрабатывать чат-бот, следя за плану, который вы составил ранее. Попробуйте разработать простого бота с парой функций и оцените его работу.
10. Когда чат-бот будет готов, проверьте, что он работает так, как задумано, и что пользователи могут им пользоваться.

Вот и все! Таким образом, вы можете получить первый опыт разработки MVP чат-бота. Удачи!



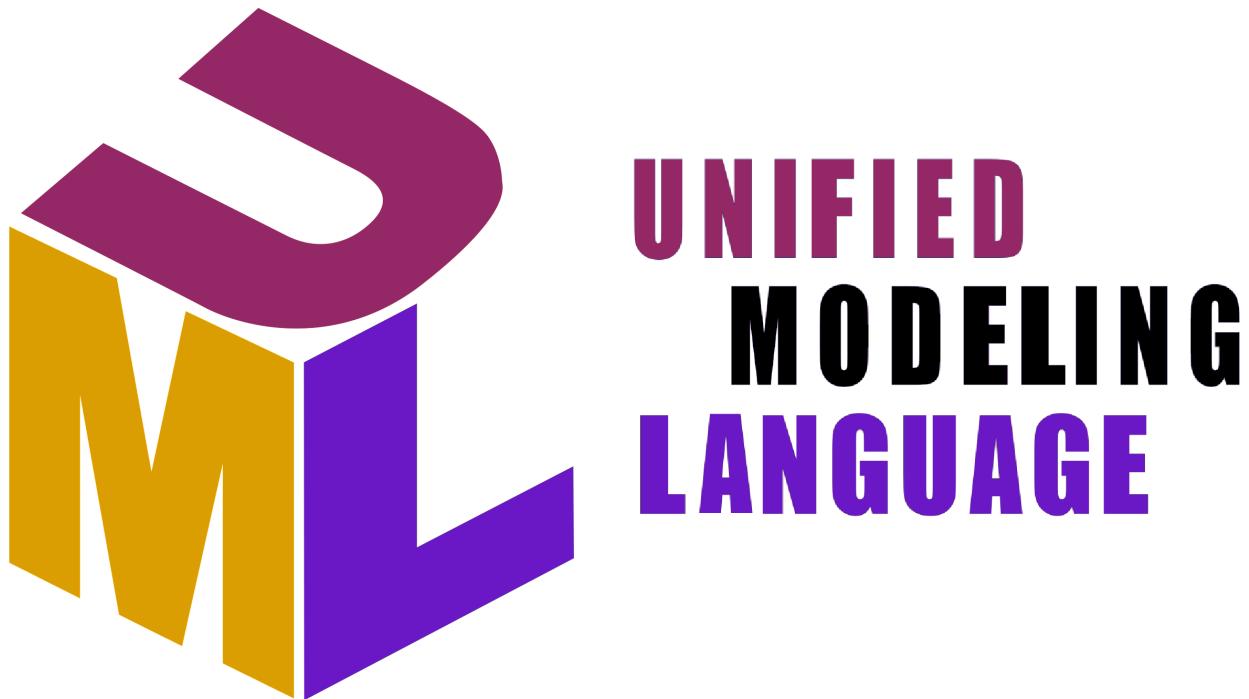
---

UML диаграммы

---

## 12.1 Факультатив. Стандартные графические схемы при разработке ПО

### 12.1.1 Возможности использования UML (Unified Modeling Language)



Рассматриваются основные схемы для визуализации процессов, которые реализует код. Основы работы с UML диаграммами (Use-case diagram, sequence diagram, component diagram,

class diagram).

На сегодняшний день разработка программного продукта предполагает использование множества различных высокоуровневых компонентов, таких как сторонние библиотеки, программные модули, фреймворки, ОС и т.д. В результате, с одной стороны, процесс разработки ускоряется за счет использования готовых компонентов, однако, с другой стороны, усложняется прозрачность такого кода. Сложнее отследить внутренние зависимости, появляются зависимости от сторонних разработчиков (модуля, фреймворка). Со временем поддерживать такой проект по ряду причин становится сложнее. Для внедрения систематизации и документирования процесса разработки продукта часто используют различные виды диаграмм в зависимости от задачи, такие как IDEF0, ER-модель и UML. Все это — методы моделирования, используемые для графического представления систем или процессов.

- IDEF0 (Integration Definition for Function Modeling) — это язык моделирования, который используется для описания функций, действий и процессов в организации. Он использует иерархический подход к моделированию системы, при этом каждый уровень представляет различный уровень детализации. IDEF0 обычно используется в правительственныех и военных учреждениях для описания и формализации бизнес-процессов. Особенность IDEF0 заключается в том, что эта методология ориентирована на соподчиненность объектов. IDEF0 была разработана для автоматизации предприятий еще в 1981 году в США.
- ER-модель (Entity-Relationship Model) — это метод моделирования, используемый для представления данных в базе данных. Он использует сущности (объекты) и их отношения друг к другу для описания данных. ER-модели обычно используются в разработке программного обеспечения для проектирования баз данных и моделей данных.
- UML (унифицированный язык моделирования) — это язык моделирования, используемый для описания программных систем. Он включает в себя набор диаграмм, которые представляют различные аспекты программной системы, такие как варианты использования, классы, компоненты и взаимодействия. UML — это широко используемый язык моделирования в разработке программного обеспечения, который поддерживается многими инструментами и платформами.

Таким образом, IDEF0 используется для моделирования функций и процессов внутри организации, ER-модель используется для моделирования данных в базе данных, а UML используется для моделирования программных систем. На сегодняшний день самым универсальным инструментом проектирования является UML. Она позволяет посмотреть на разрабатываемый программный продукт с разных сторон, с позиции архитектора ПО, архитектора баз данных, программиста (бэкенд, фронтенд) и т.д. В совокупности ознакомившись с информацией, руководитель проекта получает объективную информацию с возможностью оценить риски, этапы разработки, примерные сроки и требуемый состав команды для распределения нагрузки. Также в процессе реализации первоначальный вариант проекта может трансформироваться с учетом изменений программного продукта.

### 12.1.2 Что такое UML?

**UML (Unified Modeling Language)** — унифицированный язык моделирования, который является стандартизированной графической нотацией для моделирования и представления дизайна программных систем и других сложных систем. UML помогает определять, визуализировать и документировать структуру и поведение этих систем, облегчая понимание проекта, взаимодействие с ним и его развитие с течением времени.

**Для каких задач используют UML? Напишите пример**

UML используется в различных задачах, связанных с разработкой программного обеспечения, таких как:

1. Сбор и анализ требований: UML может быть использован для моделирования требований к системе и отражения взаимодействия между различными компонентами системы.

2. Проектирование и архитектура: UML может использоваться для моделирования высокоуровневой структуры системы, включая компоненты и отношения между ними, для разработки надежного и поддерживаемого дизайна.
3. Коммуникация: Диаграммы UML могут быть использованы для передачи информации о дизайне системы заинтересованным сторонам, таким как разработчики, менеджеры и клиенты.

Пример: Допустим, вы разрабатываете систему для интернет-магазина. Для моделирования требований к системе вы можете использовать диаграмму сценариев использования, чтобы отразить различные действия, которые пользователи могут выполнять в системе, например, просмотр товаров, добавление товаров в корзину и оформление заказа.

Для проектирования архитектуры системы можно использовать диаграмму классов для моделирования объектов системы, таких как продукты, клиенты и заказы, а также связей между ними. Вы также можете использовать диаграмму последовательности для описания взаимодействия между объектами, когда клиент размещает заказ.

Эти UML-диаграммы могут быть использованы для передачи проекта системы заинтересованным сторонам, что облегчит понимание структуры и поведения системы.

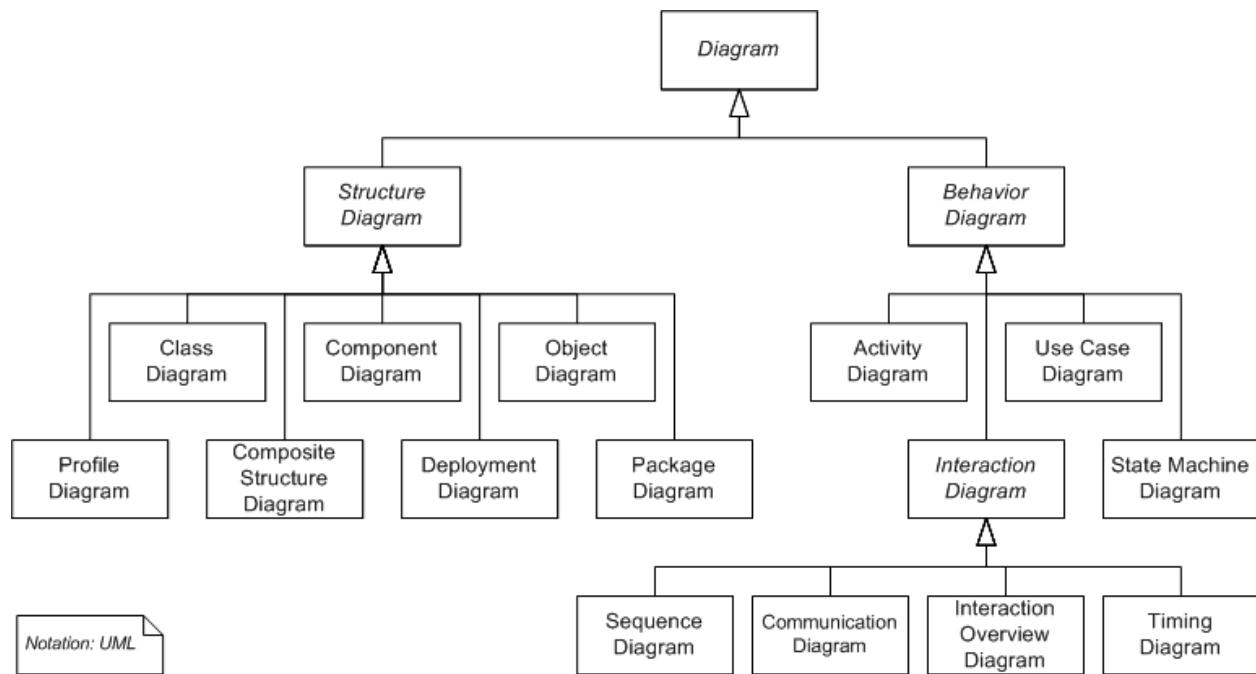
### Как UML помогает в работе над проектами? Кто их разрабатывает?

UML помогает в работе над проектами несколькими способами:

- улучшение коммуникации: UML обеспечивает общий визуальный язык для всех заинтересованных сторон, участвующих в проекте, облегчая передачу сложных концепций проектирования и понимание разрабатываемой системы;
- облегчение проектирования: UML предоставляет набор графических элементов, которые могут быть использованы для моделирования структуры и поведения системы, помогая проектировщикам исследовать различные варианты дизайна и принимать обоснованные решения о дизайне;
- захват требований: UML предоставляет диаграммы, специально разработанные для фиксации требований, такие как диаграммы вариантов использования, которые нужны для моделирования взаимодействия между системой и ее пользователями;
- улучшение совместной работы: UML облегчает совместную работу команд над проектом, предоставляя визуальное представление разрабатываемой системы, которым можно легко делиться и обсуждать его.

Разработчики, архитекторы, дизайнеры и бизнес-аналитики являются одними из заинтересованных сторон, которые могут использовать UML в проекте. Однако он не ограничивается конкретной ролью или группой людей, и любой человек, участвующий в разработке программной системы, может использовать UML для моделирования и передачи проекта системы.

### 12.1.3 Каковы основные типы диаграмм в UML?

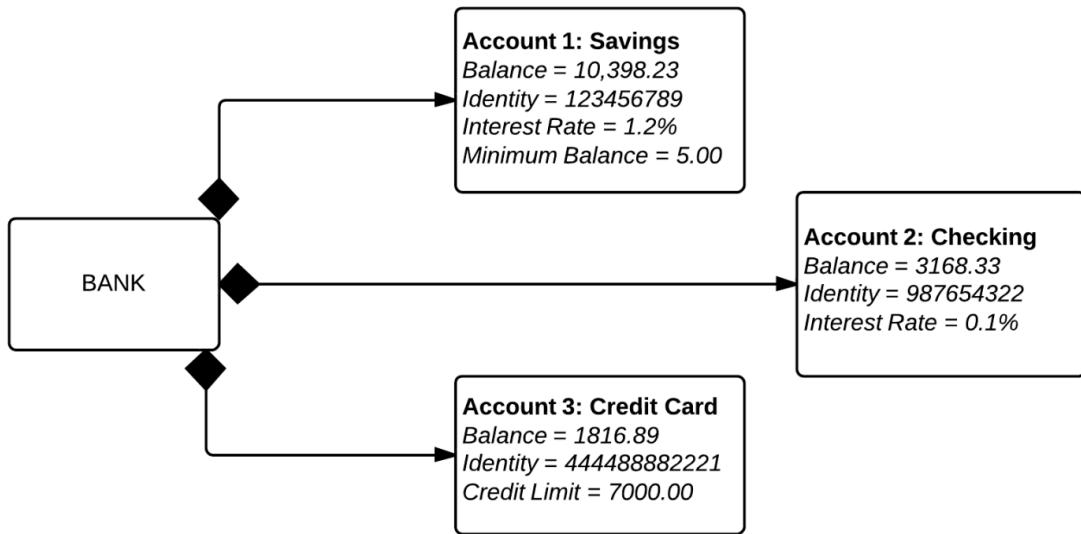


**UML** (унифицированный язык моделирования) имеет несколько типов диаграмм, которые используются для моделирования различных аспектов системы. Вот основные типы диаграмм в **UML**:

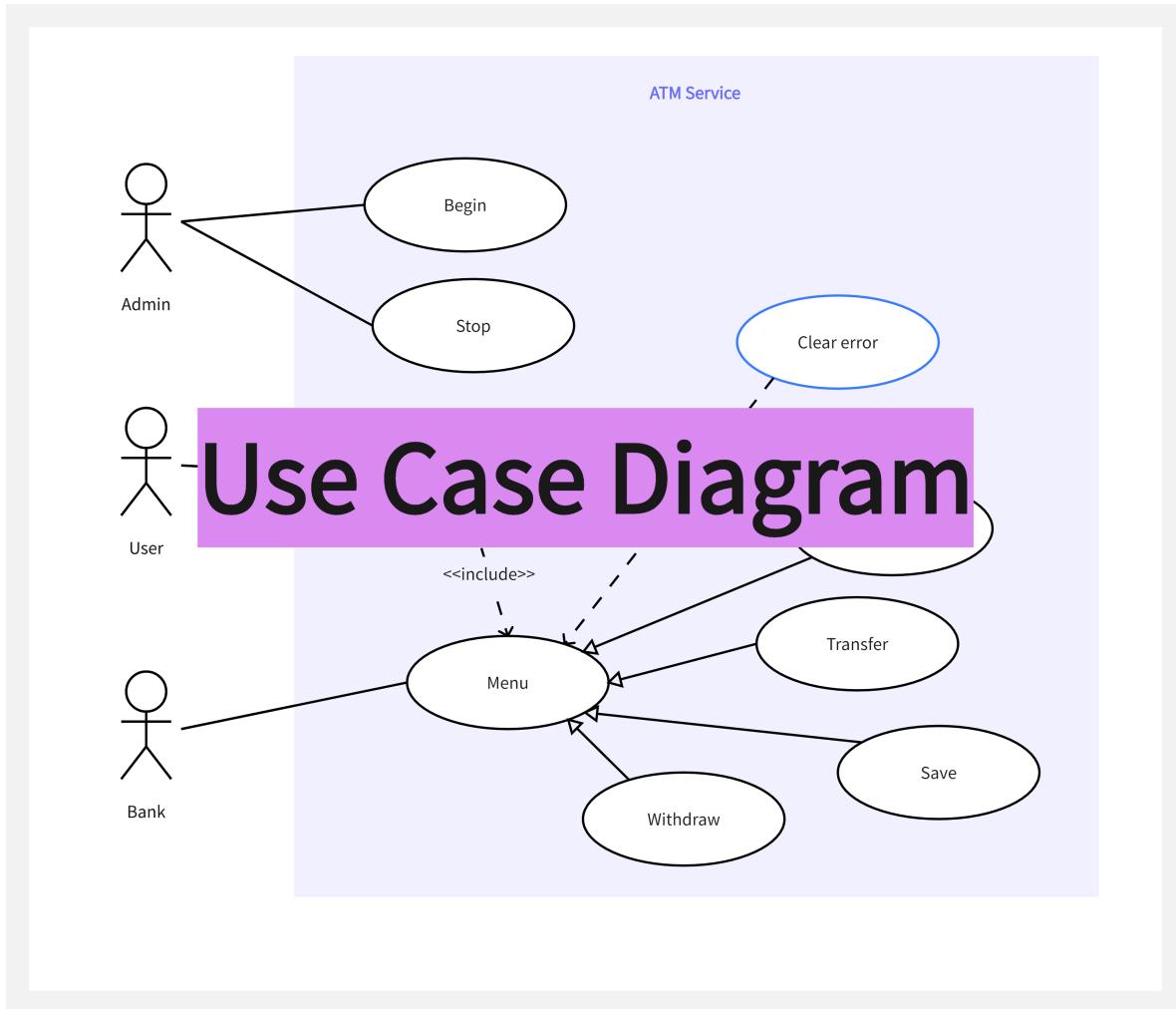
1. **Диаграмма классов (Class Diagram):** Диаграмма классов моделирует статическую структуру системы, включая классы, интерфейсы, объекты и их отношения.

<b>Книга</b>
<ul style="list-style-type: none"> <li>-название: строка</li> <li>-автор: строка</li> <li>-год_издания: целое</li> <li>-жанр: строка</li> </ul>
<ul style="list-style-type: none"> <li>+получить_название(): строка</li> <li>+получить_автора(): строка</li> <li>+получить_год_издания(): целое</li> <li>+получить_жанр(): строка</li> </ul>
<b>Библиотека</b>
<ul style="list-style-type: none"> <li>-название: строка</li> <li>-адрес: строка</li> <li>-каталог_книг: список</li> </ul>
<ul style="list-style-type: none"> <li>+добавить_книгу(): Книга</li> <li>+удалить_книгу(книга: Книга)</li> <li>+найти_книгу(название: строка)</li> <li>+показать_каталог()</li> </ul>

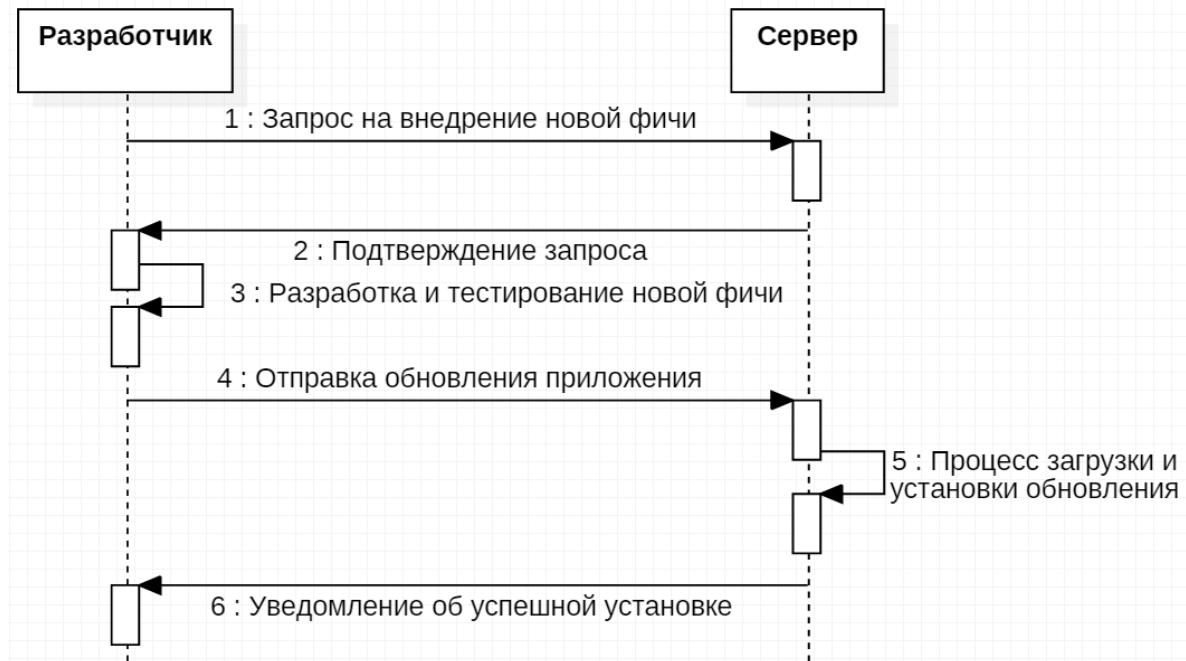
2. Диаграмма объектов (Object Diagram). Диаграмма объектов похожа на диаграмму классов, но она моделирует экземпляры классов, а не сами классы.

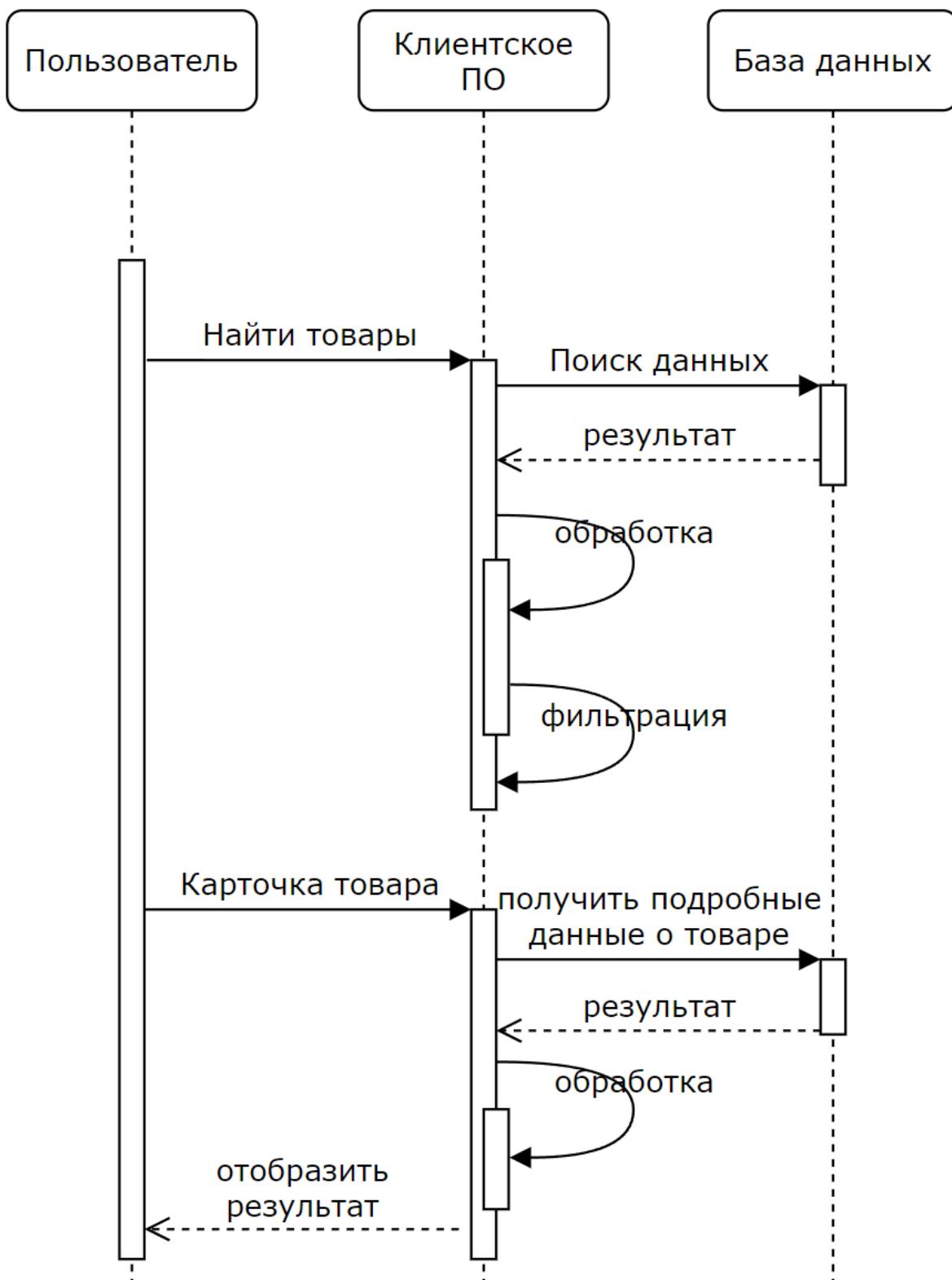


3. Диаграмма прецедентов (Use Case Diagram): Диаграмма вариантов использования моделирует взаимодействие между субъектами и системой, а также поток событий в конкретном сценарии или варианте использования.

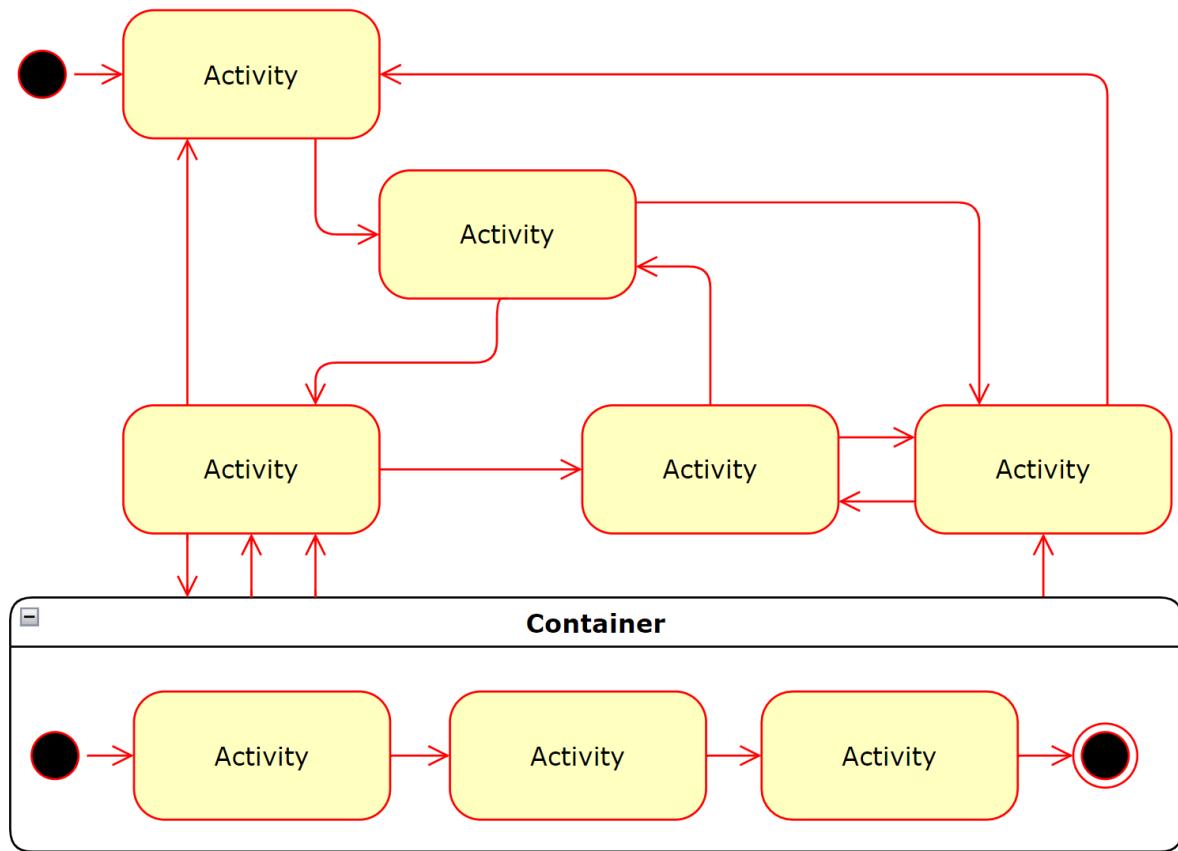


4. Диаграмма последовательности (Sequence Diagram): Диаграмма последовательности моделирует взаимодействия между объектами или компонентами в системе и порядок, в котором эти взаимодействия происходят.

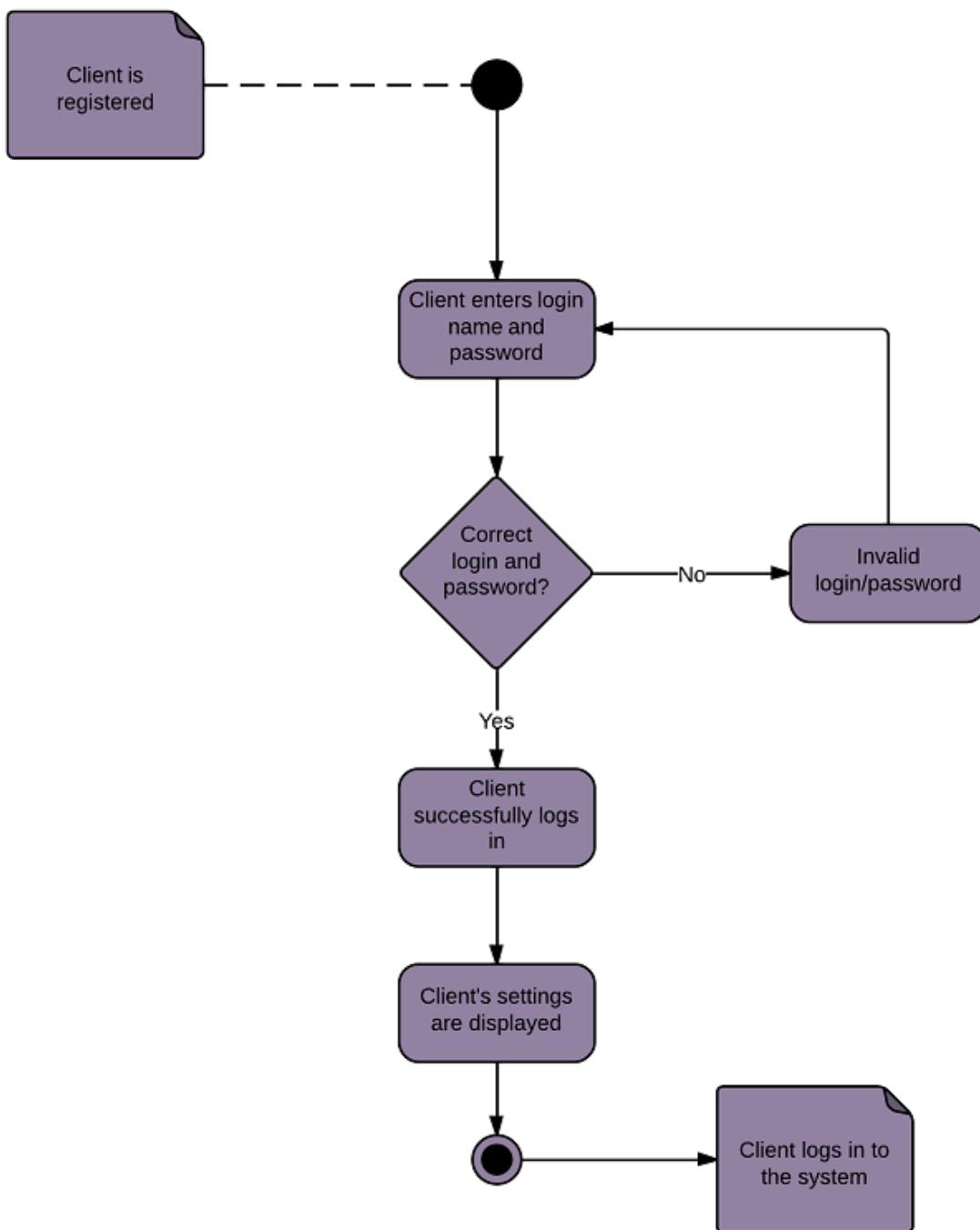




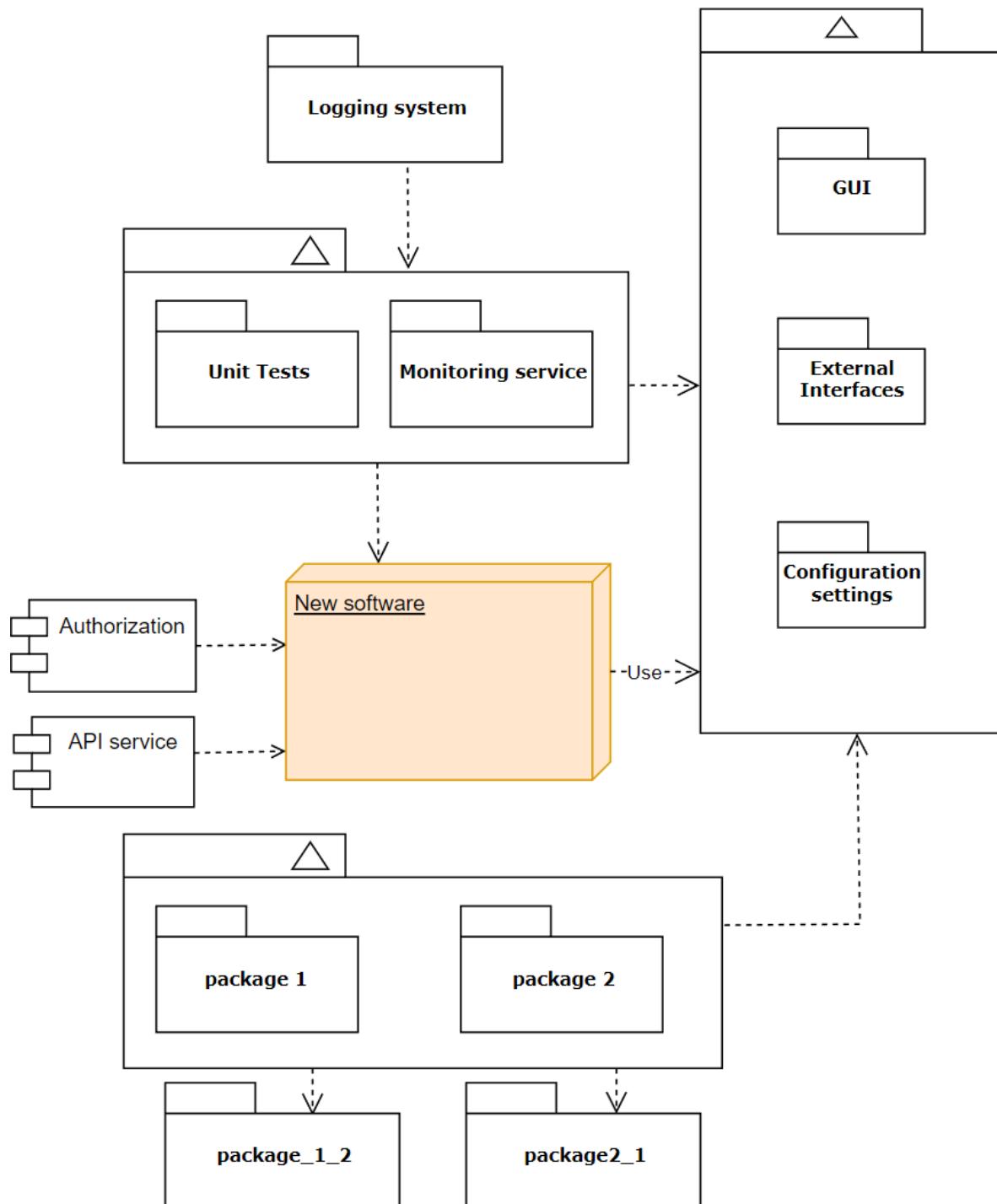
5. Диаграмма конечного автомата (State Machine Diagram): Диаграмма конечного автомата моделирует поведение объекта или системы, включая переходы между состояниями и действия, которые происходят в каждом состоянии.



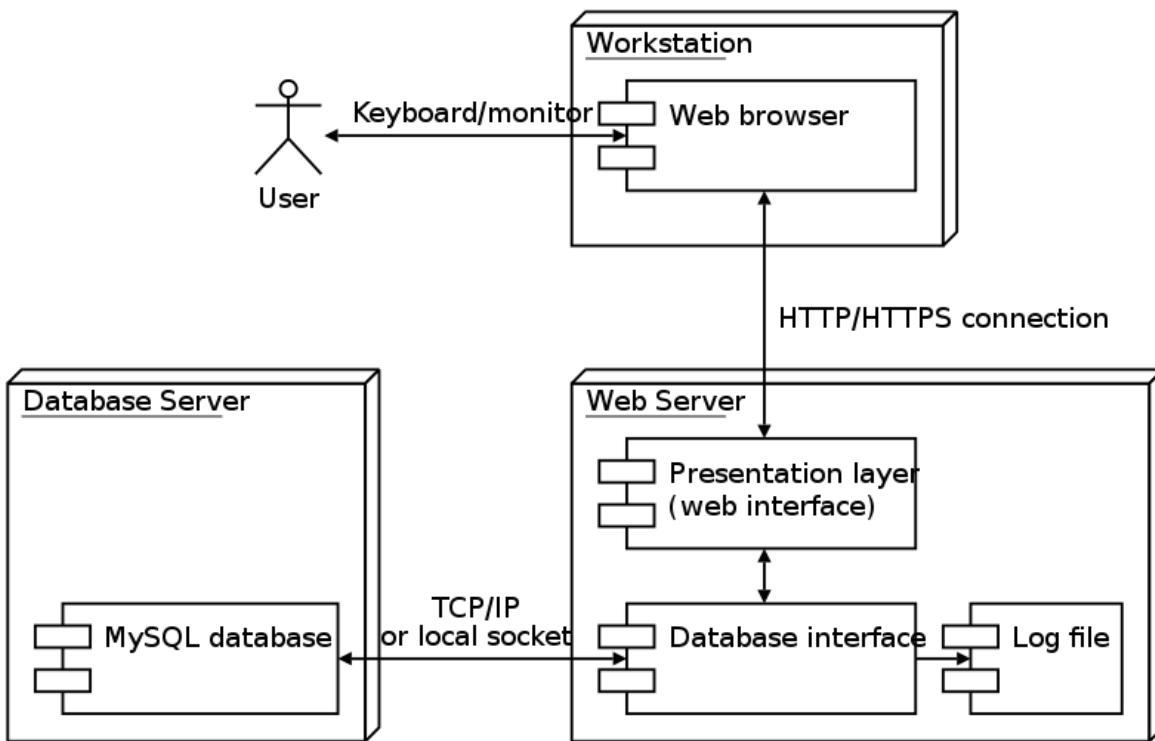
6. **Диаграмма действий (Activity Diagram):** Диаграмма действий — это диаграмма поведения, которая показывает поток действий в системе и то, как они связаны друг с другом.



7. **Диаграмма компонентов (Component Diagram):** Диаграмма компонентов моделирует статическую структуру системы, включая компоненты и отношения между ними.



8. Диаграмма развертывания (Deployment Diagram): Диаграмма развертывания моделирует физическое развертывание компонентов в системе, включая аппаратные и программные элементы и отношения между ними.



Это основные типы диаграмм в UML. В зависимости от сложности моделируемой системы могут потребоваться дополнительные диаграммы для полного представления системы.

Рассмотрим пример — диаграмму последовательности (*Sequence Diagram*). Диаграмма последовательности UML — это тип диаграммы взаимодействия, который показывает взаимодействие между объектами или компонентами в системе и порядок, в котором эти взаимодействия происходят. Диаграмма последовательности может использоваться для моделирования потока сообщений между объектами в сценарии использования.

Вот пример диаграммы последовательности для описания процесса «Загрузка мобильного приложения» с использованием маркетплейса (store) мобильных приложений:

```

User->Browse_Apps: initiate
Browse_Apps->Apps_Market: request app list
Apps_Market->Browse_Apps: return app list
Browse_Apps->User: display app list
User->Browse_Apps: select app
Browse_Apps->Search_Apps: forward app selection
Search_Apps->Apps_Market: request app information
Apps_Market->Search_Apps: return app information
Search_Apps->User: display app information
User->Search_Apps: initiate download
Search_Apps->Download_App: download request
Download_App->Apps_Market: request app download
Apps_Market->Download_App: return app download
Download_App->User: deliver_app
User->Application: install_app

```

В этом примере пользователь (*User*) инициирует процесс загрузки, просматривая доступные приложения и выбирая приложение для загрузки. Вариант использования *Browse Apps* связывается с *Apps Market* для запроса списка доступных приложений и возвращает список пользователю. Затем поль-

зователь выбирает приложение, и прикладная программа *Browse Apps* передает выбор прикладной программе *Search Apps*. Вариант использования *Search Apps* запрашивает информацию о выбранном приложении из *Apps Market* и возвращает ее пользователю. Затем пользователь инициирует загрузку приложения, и сценарий *Search Apps* передает запрос на загрузку сценарию *Download App*. Вариант использования *Download App* взаимодействует с *Apps Market* для запроса загрузки приложения и возвращает приложение пользователю. Пользователь устанавливает приложение на свое устройство.

Обратите внимание, что это лишь одно из возможных представлений сценария использования «Загрузить приложение» (*Download App*), и фактическая диаграмма последовательности может отличаться в зависимости от конкретных требований и ограничений проектируемой системы.

### **В каких случаях использование UML избыточно и можно обойтись без него?**

Использование UML может быть излишним в следующих случаях:

1. Простые системы. Для небольших и простых систем диаграммы UML могут быть излишними и пустой тратой времени. В этих случаях систему можно описать несколькими строками текста или кода, а диаграммы UML могут не иметь никакой ценности.
2. Знакомая область: если группа разработчиков знакома с областью разрабатываемой системы, им не нужны диаграммы UML для понимания требований и дизайна системы.
3. Быстрое прототипирование: при разработке быстрого прототипа основное внимание обычно уделяется максимально быстрому созданию работающей модели. В этих случаях диаграммы UML могут замедлить процесс разработки, и лучше их пропустить.
4. Гибкая разработка: методологии гибкой разработки ценят рабочий код выше документации, и диаграммы UML не могут считаться приоритетом в этой среде.
5. Как правило, использование UML является излишним, когда команду разработчиков устраивают требования и дизайн системы и когда основное внимание уделяется быстрой доставке работающего решения. Однако для более крупных и сложных систем UML-диаграммы могут обеспечить визуальное представление системы и помочь гарантировать, что все заинтересованные стороны имеют четкое представление о требованиях и дизайне.

### **В чем разница между описанием программы в виде блок-схемы и описанием программы с использованием диаграммы UML?**

Блок-схема и диаграмма UML — это инструменты, используемые для описания логики и структуры программы, но они имеют некоторые отличия:

- **Назначение:** Блок-схема в основном используется для представления потока управления в программе и для описания шагов, которые выполняются в определенном порядке. UML, с другой стороны, представляет собой язык моделирования, который можно использовать для описания различных аспектов системы, включая статическую структуру, динамическое поведение и взаимодействие между объектами или компонентами;
- **Уровень детализации:** блок-схема дает общее представление о программе и ее структуре, но может не отражать всех подробностей, необходимых для полного понимания программы. Диаграммы UML, с другой стороны, могут обеспечить более подробное представление системы, включая отношения между объектами и потоком событий в конкретном сценарии;
- **Целевая аудитория:** Блок-схемы, как правило, легче понять не техническим специалистам, тогда как UML-диаграммы в основном используют разработчики, архитекторы систем и дизайнеры UI/UX;

- Представление: блок-схемы представляют логику и структуру программы с помощью символов и фигур, соединенных стрелками. Диаграммы UML используют комбинацию различных типов диаграмм, каждая из которых представляет отдельный аспект моделируемой системы.

Таким образом, блок-схемы — это простой и интуитивно понятный способ представления потока управления в программе, в то время как диаграммы UML обеспечивают более полное представление системы и могут использоваться для описания различных аспектов системы. Выбор между блок-схемой и диаграммой UML зависит от сложности программы, требуемого уровня детализации и аудитории для описания программы.

## Преимущества и недостатки UML

### Преимущества UML

1. Стандартизация: UML — это стандартизованный язык моделирования, что означает, что все заинтересованные стороны могут эффективно общаться, используя общий язык. Это помогает уменьшить недопонимание и улучшает качество дизайна.
2. Визуализация: диаграммы UML обеспечивают визуальное представление системы, которое может быть легче понять, чем письменное описание. Это помогает заинтересованным сторонам лучше понять требования и структуру системы.
3. Улучшенная коммуникация: диаграммы UML могут помочь улучшить коммуникацию между разработчиками, дизайнерами, менеджерами и другими заинтересованными сторонами, обеспечивая общее понимание системы.
4. Раннее выявление проблем. Диаграммы UML могут помочь выявить проблемы в требованиях и дизайне системы на ранней стадии, что сэкономит время и ресурсы в долгосрочной перспективе.
5. Улучшенная документация: UML-диаграммы обеспечивают всестороннее представление системы, которое может служить источником документации для дальнейшего использования.

### Недостатки UML

1. Сложность: Для неопытного разработчика UML — сложный язык, и разработчикам, которые только начинают им пользоваться, может быть весьма сложно его изучить за короткое время.
2. Длительный жизненный цикл: Разработка диаграмм UML может занимать длительное время, что увеличит продолжительность разработки и затраты на проработку проекта. Перед разработкой моделей необходимо собрать большой объем информации, содержащей подробные данные о разрабатываемом продукте. Для многих проектов разработка полного пакета UML-диаграмм является избыточной. В случае масштабирования продукта, расширения функционала и увеличения количества используемых компонентов можно преобразовать имеющуюся техническую документацию в UML.
3. Ограниченная возможность повторного использования: диаграммы UML нельзя использовать повторно для разработки новых продуктов без внесения изменений, поскольку может потребоваться обновлять и изменять их по мере развития требований и дизайна системы.
4. Незнакомость: UML может быть малоизвестным, широко использоваться только в некоторых отраслях, что затруднит понимание или эффективное использование диаграмм заинтересованными сторонами.
5. Ограниченнная функциональность: UML — это, прежде всего, язык моделирования. Он не предоставляет прямого способа генерации кода. Это означает, что диаграммы UML могут не подходить для некоторых типов систем, таких как очень сложные системы или системы реального времени.

Итак, UML имеет как преимущества, так и недостатки, и решение об использовании UML должно основываться на конкретных потребностях проекта и целях группы разработчиков. UML может быть ценным инструментом для улучшения коммуникации и документации, но он может не подходить для всех проектов из-за его сложности и дополнительных расходов.

## Этапы разработки UML-диаграмм для чат-бота

Рассмотрим небольшой пример разработки UML-диаграмм для чат-бота. Например, нужно разработать чат-бота, который будет информировать о разных событиях и отвечать на вопросы клиентов. Перед тем, как приступить к разработке UML-диаграмм, следует выполнить следующее:

### 1. Определить требования

Начните с определения основных требований к чат-боту. Это включает функциональные и нефункциональные требования, например, какие функции бота должны поддерживаться, на каких платформах он будет работать, какая должна быть его производительность и т.д. Пример содержания определения требований для чат-бота:

#### 1.1 Функциональные требования:

- Чат-бот должен иметь возможность приветствовать пользователей и предлагать помощь.
- Чат-бот должен уметь отвечать на вопросы пользователей и предоставлять информацию по различным темам.
- Чат-бот должен поддерживать возможность регистрации и авторизации пользователей.
- Чат-бот должен предоставлять информацию о различных мероприятиях, новостях и акциях.
- Чат-бот должен уметь предлагать пользователю различные варианты ответов на вопросы и действий.

#### 1.2 Нефункциональные требования:

- Чат-бот должен быть доступен на платформах Telegram и VK.
- Время ответа чат-бота на сообщения пользователя не должно превышать 5 секунд.
- Чат-бот должен обладать высокой производительностью, чтобы обслуживать большое количество пользователей одновременно.
- Чат-бот должен быть простым в использовании и иметь интуитивно понятный интерфейс.
- Чат-бот должен обеспечивать безопасное хранение и обработку пользовательских данных.

#### 1.3 Ограничения и допущения:

- Чат-бот будет разрабатываться на языке программирования Python.
- Чат-бот будет использовать базу данных для хранения информации о пользователях и мероприятиях.
- При разработке чат-бота будет использоваться библиотека для работы с API платформ Telegram и Facebook Messenger.

#### 1.4 Запланированные улучшения:

- В будущем планируется добавить функционал платежей и возможность заказа товаров через чат-бота.
- Предполагается улучшить алгоритмы обработки естественного языка для более точного и информативного ответа на вопросы пользователей.

- В дальнейшем можно добавить функцию уведомлений для пользователей о предстоящих событиях и акциях.

Это всего лишь пример, и фактические требования могут сильно различаться в зависимости от конкретного проекта чат-бота. Важно учитывать потребности и ожидания целевой аудитории, а также технические возможности и ограничения разработки.

## 2. Изучить существующие решения

Проведите исследование рынка и познакомьтесь с существующими чат-ботами, которые решают похожую задачу, близкую к той, которую вы решаете. Посмотрите, как устроен интерфейс пользователя, какие есть варианты меню, как бот реагирует в случае отсутствия у него ответа и в каком виде выдает результат (текст, картинка, таблица, список). Попробуйте составить сценарий работы бота-аналога, чтобы учесть возможные ошибки и реакции поведения в различных ситуациях при общении с пользователем. Это поможет вам понять, какие функции можно включить в бот и какие дизайн-паттерны и архитектуры можно применить. 2.1 Исследование рынка чат-ботов:

- Проведите анализ популярных чат-ботов на рынке, работающих в схожей сфере или предоставляющих аналогичные услуги.
- Изучите их функциональность, возможности и особенности взаимодействия с пользователем.
- Оцените отзывы и оценки пользователей для выявления сильных и слабых сторон существующих решений.

### 2.2 Ознакомление с дизайн-паттернами и архитектурами:

- Изучите различные дизайн-паттерны, используемые в разработке чат-ботов, такие как State Pattern, Command Pattern и другие.
- Познакомьтесь с архитектурными подходами для разработки чат-ботов, включая Model-View-Controller (MVC), Event-Driven и другие.
- Рассмотрите преимущества и недостатки каждого подхода и выберите наиболее подходящие для вашего проекта.

### 2.3 Определение возможностей для улучшения:

- Изучите сильные стороны существующих чат-ботов и определите, какие из них можно использовать в своем проекте.
- Выявите слабые стороны и ошибки в существующих решениях, чтобы избежать их повторения в вашем чат-боте.
- Определите функции и особенности, которые вы хотите добавить в свой бот для улучшения пользовательского опыта.

### 2.4 Применение лучших практик:

- Воспользуйтесь полученными знаниями и опытом, чтобы разработать оптимальную функциональность и архитектуру для своего чат-бота.
- Учите успешные решения, которые были применены в существующих чат-ботах, и внедрите их в свой проект.
- Не ограничивайтесь прямым копированием, а адаптируйте лучшие практики к специфике вашего проекта.

Изучение существующих решений позволит вам получить ценную информацию о том, как создать успешный и функциональный чат-бот, а также избежать распространенных ошибок и проблем, которые

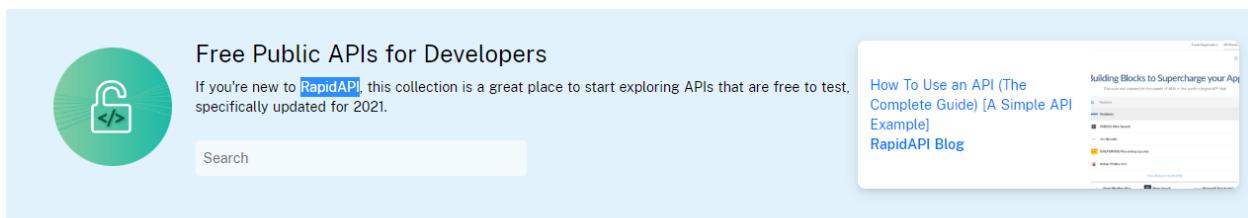
могут возникнуть в процессе разработки. Это поможет вам создать чат-бот, который будет эффективно выполнять свои функции и удовлетворять потребности пользователей.

### 3. Определить архитектуру

На основе требований и изучения существующих решений определите архитектуру вашего чат-бота. Решите, какие компоненты и функциональности будут включены в бота, как они будут взаимодействовать между собой и с внешними системами. Например, вам может потребоваться взаимодействие с внешними сервисами посредством API (Application Program Interface), для получения и обработки информации в процессе формирования ответа пользователю. Разрабатывая первый прототип (MVP/Pet project) чат-бота, рекомендуется использовать язык программирования Python. Он прост в изучении и обладает широким набором библиотек, в том числе для разработки чат-ботов. Вот несколько популярных библиотек:

- [python-telegram-bot](#)
- [pyTelegramBotAPI](#)
- [AIOGram](#)
- [TGramBot](#)
- [OrigamiBot](#)
- [pytgbot](#)
- [teleflask](#)
- [telegram-text](#)

В качестве источника данных можно выбрать бесплатный API сервис [RapidAPI](#).



**Free Public APIs for Developers**

If you're new to RapidAPI, this collection is a great place to start exploring APIs that are free to test, specifically updated for 2021.

Search

**Collections**

- Recommended APIs
- Popular APIs
- Free Public APIs for Developers
- Top AI Based APIs
- Tax APIs
- [View All Collections](#)

**Categories**

- Sports
- Finance
- Data
- Entertainment
- Travel
- [View All Categories](#)

**Collection APIs**

API Name	Description	Rating	Latency	Success Rate
Realtor	Real estate information and market analysis.	9.8	1 398 ms	100%
Bloomberg Market and Financial News	Market data and financial news from Bloomberg.	9.8	1 695 ms	100%
Alpha Vantage	The simplest and most effective way to receive stock, ETF, forex, technical indicators, and more.	9.9	702 ms	100%
IMDb	Movie and TV show database.	9.9	1 092 ms	100%
BraveNewCoin	Latest and historic cryptocurrency market data.	9.9	179 ms	100%
WeatherAPI.com	Weather API provider that provides extensive APIs that range from weather to geolocation.	9.9	163 ms	100%
Google Translate	Dynamically translate between languages.	9.9	557 ms	97%
The Cocktail DB	API to get data on alcohol & cocktails.	9.8	206 ms	100%
Yahoo Finance	This API reproduces live PUBLIC data and features of finance.yahoo.com. The response data is always live and synced with the site at all times.	9.9	1 789 ms	100%
API-NBA	All NBA Stats DATA, Games, Livescore, Standings, Players, Statistics... Documentation : <a href="https://api-nba.com/">https://api-nba.com/</a>	9.9	521 ms	100%
Live Sports Odds	Odds data API for NFL, NBA, MLB, EPL, AFL and loads more, from US, UK, EU and Aussie bookmakers. Get started for free!	9.8	148 ms	100%
API-FOOTBALL	+1 000 football leagues & cups. Livescore (15s), live & pre-match odds, events, line-ups, coaches, players, fixtures, results, stats, and more.	9.9	253 ms	100%
TheRundown	The fastest and most accurate real-time odds, scores, schedules, and stats from major sportsbooks. See our API documentation.	9.8	195 ms	100%
GeoDB Cities	Get global city, region, and country data. Filter and display results in multiple languages.	9.9	68 ms	100%
Weather	Current weather data API, and Weather forecast API - Basic access to the Weatherbit.io Weather API.	9.9	244 ms	100%
WordsAPI	Words API lets you retrieve information about English words, including definitions, synonyms, rhymes, and more.	9.9	320 ms	100%

RapidAPI содержит множество открытых источников данных, что пригодится в разработке чат-бота для большинства направлений.

RapidAPI — это онлайн-платформа, которая предоставляет доступ к тысячам бесплатных и платных API (интерфейсов программирования приложений) от различных провайдеров. API — это набор программных инструкций и методов, которые позволяют различным приложениям и сервисам обмениваться данными и функциональностью между собой.

## Порядок действий для использования API сервиса RapidAPI

1. Регистрация: Для использования RapidAPI необходимо зарегистрироваться на платформе. Регистрация бесплатна и занимает всего несколько минут.
2. Поиск API: После регистрации можно начать поиск нужных API. RapidAPI предоставляет обширный каталог бесплатных API по различным категориям, таким как социальные сети, погода, геолокация, машинное обучение, финансы и многое другое.
3. Просмотр информации о API: При выборе интересующего API можно ознакомиться с его документацией, функциональностью, ограничениями и примерами использования.

The screenshot shows the RapidAPI interface for the API-BASKETBALL service. At the top, it displays the service name, a blue icon, a PREMIUM badge, and a Verified checkmark. It also shows popularity metrics: 9.9 / 10, 398ms latency, and 100% service level. Below this, there are tabs for Endpoints, About, Tutorials, Discussions, and Pricing. The Endpoints tab is selected, showing two main sections: one for Basketball Leagues & Cups with Livescore, Odds, Bookmakers, Statistics, Standings, Historical Data, Countries, Seasons, Documentation, and another for Basketball Leagues & Cups with Livescore, Odds, Bookmakers, Statistics, Standings, Historical Data, Countries, Seasons, Documentation. Below these sections, there are links to the API documentation and widgets. On the left, a sidebar lists various endpoints such as bookmakers, odds, Standings, seasons, countries, leagues, games, and teams. On the right, a detailed view of the GET bets endpoint is shown, including its Request URL (rapidapi.com), Header Parameters (X-RapidAPI-Key and X-RapidAPI-Host), and Optional Parameters (id and search). A code snippet in Python Requests is provided for testing the endpoint.

```
[Python] Requests
import requests
url = "https://api-basketball.p.rapidapi.com/bets"
headers = {
    "X-RapidAPI-Key": "c25c61dd5mshe0edf03fc52c8cp13fa90jn77b9ca94c147",
    "X-RapidAPI-Host": "api-basketball.p.rapidapi.com"
}
response = requests.get(url, headers=headers)
print(response.json())
```

4. Получение ключа API: Для использования большинства API необходимо получить API-ключ. Обычно это бесплатный процесс, который требует регистрации на сайте провайдера API. Ключ используется для аутентификации и предотвращения злоупотребления.
5. Подключение к API: После получения ключа API можно подключиться к выбранному API через RapidAPI. Платформа предоставляет простой и удобный интерфейс для вызова методов API и обработки результатов.

The screenshot shows the RapidAPI interface for the Realtor API. At the top, there's a search bar and navigation links for API Hub, Organizations, and Apps. Below that, the Realtor API card includes its rating (9.8 / 10), latency (1 398ms), and service level (100%). The API card also features a 'Subscribe to Test' button. To the right, there are tabs for Endpoints, About, Tutorials, Discussions, and Pricing. A note below the card states: "Realtor API helps to query properties for sale, rent, sold etc... as on official site". On the left, a sidebar lists various API endpoints under categories like properties, agents, schools, and mortgage & finance tools. The main panel displays the 'GET agents/list' endpoint details, including the request URL (rapidapi.com), header parameters (X-RapidAPI-Key and X-RapidAPI-Host), and required parameters (postal\_code). To the right, a 'Code Snippets' section provides a Python Requests code example:

```

(Python) Requests
import requests

url = "https://realtor.p.rapidapi.com/agents/list"
querystring = {"postal_code": "11234", "offset": "0", "limit": "20", "types": "agent", "sort": "recent_activity_high"}

headers = {
    "X-RapidAPI-Key": "23561dd5mehe0edf03fc5c2dcp13fa90jn77b9ca94c147",
    "X-RapidAPI-Host": "realtor.p.rapidapi.com"
}

response = requests.get(url, headers=headers, params=querystring)
print(response.json())

```

6. Ограничения и лимиты: Важно обратить внимание на ограничения и лимиты использования бесплатных API. Некоторые API могут иметь ограничения на количество запросов в день или на определенный период времени.
7. Обработка результатов: После получения ответов от API можно обрабатывать и использовать данные в своих приложениях или проектах.
8. Мониторинг использования: Некоторые API предоставляют возможность мониторинга использования и статистики запросов для лучшего контроля над вашими приложениями.

Помните, что при использовании бесплатных API также могут применяться ограничения по скорости запросов и доступности, поэтому перед использованием API в коммерческих проектах или с большим объемом запросов рекомендуется ознакомиться с платными вариантами и условиями использования. RapidAPI предоставляет удобный способ объединить доступ к различным API и сделать процесс интеграции в ваши проекты более простым и эффективным.

### 3.1 Определение основных компонентов чат-бота:

- Идентифицируйте основные компоненты, необходимые для функционирования чат-бота. Например, интерфейс пользователя, обработчик сообщений, база данных и т.д.
- Определите роли и функции каждого компонента в архитектуре бота.

### 3.2 Разработка схемы взаимодействия компонентов:

- Определите взаимодействие между компонентами чат-бота. Например, как пользовательские запросы будут передаваться обработчику сообщений, какие данные будут храниться в базе данных и т.д.
- Рассмотрите возможные сценарии взаимодействия и определите наилучшие подходы для обеспечения эффективной работы бота.

### 3.3 Интеграция с внешними системами:

- Решите, какие внешние системы будут интегрированы с чат-ботом. Например, для получения новостей, данных о результатах олимпиады или рейтинга игры «Берлога».
- Определите протоколы и форматы данных для обмена информацией между ботом и внешними системами.

### 3.4 Обработка ошибок и исключительных ситуаций:

- Разработайте механизмы обработки ошибок и исключительных ситуаций. Например, как бот будет реагировать на неправильные запросы или ошибки при обращении к внешним системам.
- Предусмотрите механизмы логирования и отслеживания ошибок для облегчения отладки и улучшения производительности бота.

### 3.5 Выбор технологий и инструментов:

- Определите наиболее подходящие технологии и инструменты для реализации каждого компонента чат-бота. Например, выберите язык программирования, фреймворк для работы с базой данных и т.д.
- Обратите внимание на производительность, масштабируемость и удобство использования выбранных технологий.

Определение архитектуры чат-бота позволит вам сформировать четкий план разработки и обеспечит эффективное взаимодействие компонентов, что обеспечит успешную реализацию задачи разработки чат-бота и его функционирование в соответствии с поставленными требованиями.

## 4. Разработать диаграмму прецедентов

Создайте диаграмму прецедентов, которая описывает функциональность бота и его взаимодействие с пользователями. Определите основные сценарии использования бота и взаимодействия с ним.

### 4.1 Определение «актеров»:

- Определите актеров, которые будут взаимодействовать с чат-ботом. Например, «пользователь-клиент», «разработчик» и «тестировщик». Каждому актеру свойственен определенный (универсальный) набор прецедентов (действий). Например, чаще всего у бота есть администратор и пользователь (клиент). Администратор управляет контентом (загрузка постов, управление рассылкой, обработка запросов рекламодателей и пользователей). С другой стороны, к прецедентам клиента могут относиться следующие действия: подписаться на бота, отправить сообщение боту, выбрать интересующие темы контента с помощью фильтра в главном меню бота и т.д.
- Уточните роли каждого актера и их взаимодействие с ботом. Важно знать что несколько «актеров» могут выполнять один прецедент. Например, и администратор и клиент могут отправлять сообщения боту, выбирать действия в пунктах меню и получать информацию от бота. Однако должен быть минимум один прецедент, который будет уникальным у каждого актера. Так, например, клиент не может управлять контентом, в то время как администратор может.

### 4.2 Идентификация прецедентов:

- Идентифицируйте основные функциональности чат-бота, которые будут доступны для каждого актера.
- Определите прецеденты как действия или задачи, которые могут быть выполнены взаимодействием с ботом. Например, «Получить информацию о новостях», «Получить результаты олимпиады», «Получить рейтинг игры „Берлога“» и т.д.

### 4.3 Описание сценариев использования:

- Проведите анализ сценариев использования для каждого прецедента. Например, опишите шаги, которые пользователь-клиент будет выполнять для получения информации от бота.
- Определите возможные варианты взаимодействия с ботом и учтите различные ситуации, которые могут возникнуть в процессе общения.

### 4.4 Взаимодействие с разработчиком и тестировщиком: Помимо пользователя и администратора, нужно учитывать возможность взаимодействия с ботом разработчика/тестировщика. Это может потребо-

ваться при разработке MVP для организации эффективной работы в команде. Так вы сможете заранее «запланировать» прецеденты участников команды разработки.

- Определите прецеденты, связанные с взаимодействием разработчика и тестировщика с чат-ботом. Например, «Разработать новую функциональность», «Протестировать функциональность бота» и т.д.
- Опишите взаимодействие и обязанности каждой роли в процессе разработки и тестирования бота.

#### 4.5 Взаимодействие с внешними системами:

- Определите прецеденты, связанные с взаимодействием чат-бота с внешними системами для получения информации. Например, с базой данных для новостей, олимпиад и рейтинга игры «Берлога». Любо другими бесплатными сервисами предоставления информации посредством API.
- Уточните, какие данные будут передаваться между ботом и внешними системами и какие протоколы будут использоваться.

Разработка диаграммы прецедентов позволит вам лучше понять функциональность чат-бота и его взаимодействие с пользователями, а также обеспечит более ясное представление о том, как пользователи будут взаимодействовать с ботом и какие функции должны быть реализованы в приложении.

### 5. Разработать диаграммы классов

Составьте диаграммы классов, которые покажут структуру классов в вашем чат-боте и их взаимосвязи. Определите основные классы, их атрибуты и методы.

1. Определение основных классов:
  - Идентифицируйте основные классы, которые будут использоваться в чат-боте. Например, «ChatBot», «User», «Developer», «Tester», «News», «Olympiad», «GameRating» и т.д.
  - Уточните функциональность каждого класса и его назначение в системе.
2. Атрибуты классов:
  - Для каждого класса определите его атрибуты. Например, у класса «User» могут быть атрибуты: «Имя», «ID пользователя», «Последний запрос» и т.д.
  - Определите типы данных для каждого атрибута и их свойства (например, публичный или приватный доступ).
3. Методы классов:
  - Определите для каждого класса методы, которые будут реализовывать функциональность бота. Например, у класса «ChatBot» может быть метод «Отправить сообщение», у класса «User» — метод «Запросить информацию» и т.д.
  - Опишите, что делает каждый метод и какие параметры принимает.
4. Взаимосвязи между классами:
  - Укажите связи между классами на диаграмме. Например, связь «ассоциация» между классами «ChatBot» и «User» для обозначения взаимодействия бота с пользователем.
  - Определите типы связей (например, «агрегация», «композиция» и т.д.) и их характеристики.
5. Дополнительные аспекты:
  - Уточните, какие классы являются абстрактными или интерфейсами, если такие есть в вашем проекте.

- Обратите внимание на наследование классов и иерархию наследования, если она присутствует.

Разработка диаграмм классов позволит вам лучше понять структуру вашего чат-бота, определить его компоненты и взаимосвязи, а также обеспечит более ясное представление о том, как будут организованы данные и функциональность в вашем приложении.

## 6. Разработать диаграммы последовательностей

Создайте диаграммы последовательностей для ключевых сценариев использования бота. Эти диаграммы покажут последовательность сообщений и взаимодействие между компонентами во время выполнения сценариев.

1. Определение ключевых сценариев:

- Идентифицируйте ключевые сценарии использования вашего чат-бота. Например, «Отправка новостей», «Запрос рейтинга игры», «Ответ на вопросы пользователей» и т.д.
- Определите общий поток выполнения для каждого сценария.

2. Создание диаграммы последовательности:

- Для каждого ключевого сценария разработайте соответствующую диаграмму последовательности.
- Укажите объекты (компоненты) в диаграмме, которые участвуют в сценарии. Например, «User», «ChatBot», «News», «GameRating» и т.д.
- Определите сообщения, которые передаются между компонентами во время выполнения сценария. Например, «User» отправляет запрос на «ChatBot», «ChatBot» обрабатывает запрос и отправляет ответ «User».

3. Последовательность сообщений:

- Уточните последовательность отправки и получения сообщений между компонентами в каждом сценарии.
- Определите условия и ветвления в сценариях, если они есть.

4. Обработка и отображение данных:

- Покажите, какие данные обрабатываются во время выполнения каждого сценария и какие результаты возвращаются пользователю.

5. Дополнительные аспекты:

- Обратите внимание на взаимодействие с внешними системами, если они присутствуют в сценариях.
- Укажите возможные исключения или ошибки, которые могут возникнуть во время выполнения сценариев.

Разработка диаграмм последовательностей позволит вам визуализировать последовательность сообщений и взаимодействие между компонентами во время выполнения ключевых сценариев вашего чат-бота. Это поможет улучшить понимание работы вашего приложения и обеспечит более ясное представление о том, как будут выполняться задачи и обрабатываться данные.

## 7. Разработать диаграммы компонентов и развертывания

Создайте диаграммы компонентов и развертывания для описания архитектуры и размещения компонентов вашего бота. Эти диаграммы помогут понять, как компоненты взаимодействуют между собой и как они размещены на сервере.

### 1. Определение компонентов:

- Идентифицируйте основные компоненты вашего чат-бота, которые были определены на предыдущих этапах разработки.
- Обозначьте каждый компонент, опишите его функциональность и взаимосвязи с другими компонентами.

### 2. Создание диаграммы компонентов:

- Составьте диаграмму компонентов, которая покажет структуру вашего чат-бота и его компонентов.
- Укажите каждый компонент в диаграмме и его связи с другими компонентами.
- Определите интерфейсы, которые используются для взаимодействия между компонентами.

### 3. Размещение компонентов на сервере:

- Создайте диаграмму развертывания, которая покажет, как компоненты вашего чат-бота размещены на сервере.
- Укажите сервер и его характеристики, такие как операционная система, процессор, память и т.д.
- Определите, как каждый компонент размещен на сервере и как они взаимодействуют друг с другом.

### 4. Взаимодействие компонентов:

- Определите, как компоненты взаимодействуют друг с другом на уровне кода и какие интерфейсы используются для этого.
- Покажите последовательность сообщений и обмен данных между компонентами на диаграмме компонентов и развертывания.

### 5. Дополнительные аспекты:

- Обратите внимание на безопасность и защиту данных при размещении компонентов на сервере.
- Укажите возможные сценарии масштабирования и расширения системы при необходимости.

Разработка диаграмм компонентов и развертывания позволит вам визуализировать структуру и размещение компонентов вашего чат-бота на сервере. Это поможет более полно понять архитектуру вашего приложения и обеспечит более четкое представление о взаимодействии компонентов в системе.

## **8. Протестируйте и оптимизируйте диаграммы**

После создания диаграмм убедитесь, что они полно и точно отражают архитектуру и функциональность вашего бота. Проведите тестирование и оптимизацию диаграмм, чтобы улучшить их качество и понимание.

### 1. Проверка полноты и точности диаграмм:

- Проведите проверку каждой диаграммы на предмет полноты и точности отражения архитектуры и функциональности вашего чат-бота.
- Убедитесь, что все ключевые компоненты и функции бота отображены на диаграммах и взаимосвязи между ними корректно представлены.

### 2. Тестирование диаграмм:

- Протестируйте диаграммы с помощью коллег или экспертов в области разработки и архитектуры программного обеспечения.
- При тестировании диаграмм обратите внимание на их понятность, читаемость и соответствие спецификациям и требованиям.

### 3. Оптимизация диаграмм:

- Если необходимо, внесите корректировки в диаграммы для повышения их читаемости и понятности.
- Убедитесь, что каждая диаграмма является лаконичной и содержит только необходимую информацию, чтобы избежать избыточности и путаницы.

### 4. Обратная связь и доработка:

- Получите обратную связь от команды разработки, аналитиков и заказчика на предмет качества диаграмм и их соответствия требованиям.
- Внесите необходимые доработки и улучшения в диаграммы на основе обратной связи и результатов тестирования.

### 5. Документирование и сохранение:

- Завершив тестирование и оптимизацию, документируйте окончательные версии диаграмм и сохраните их для будущего использования и сопровождения проекта.

Протестированные и оптимизированные диаграммы обеспечат более четкое и понятное представление о структуре и функциональности вашего чат-бота. Это также поможет улучшить сотрудничество между членами команды разработки и обеспечит более эффективное планирование и управление проектом.

## **9. Итеративное развитие**

Постепенно уточняйте и дополняйте диаграммы по мере разработки бота. UML-диаграммы должны быть гибким инструментом, который помогает вам понять и описать архитектуру вашего проекта на каждом этапе разработки.

### 1. Постепенное уточнение диаграмм:

- Начните с создания основных диаграмм на ранних этапах разработки чат-бота, таких как диаграммы прецедентов, классов и последовательностей.
- Постепенно уточняйте диаграммы с учетом изменений в требованиях, дополнительных функциональностей и отклика команды разработки.

### 2. Гибкий инструмент для архитектуры проекта:

- Используйте UML диаграммы в качестве гибкого инструмента для понимания и описания архитектуры вашего проекта на различных этапах разработки.
- Уточнение и дополнение диаграмм поможет вам лучше понять структуру и взаимосвязи компонентов вашего чат-бота.

3. Итеративный подход:

- Применяйте итеративный подход к разработке и использованию диаграмм, что позволит вам улучшать их постепенно на протяжении всего процесса разработки.
- Вносите изменения в диаграммы, когда появляются новые идеи или когда требования к проекту меняются.

4. Связь с командой разработки:

- Поддерживайте постоянную связь с командой разработки и обновляйте диаграммы на основе их обратной связи и предложений.
- Диаграммы должны быть актуальными и отражать текущее состояние разработки проекта.

5. Визуализация прогресса:

- Используйте диаграммы для визуализации прогресса разработки и коммуникации с заинтересованными сторонами, такими как заказчики и руководство.
- Диаграммы позволяют лучше представить архитектуру и функциональность бота на каждом этапе разработки.

6. Документирование и сохранение:

- Документируйте каждое обновление диаграмм и сохраняйте их для последующего анализа и сопровождения проекта.
- Это поможет вам возвращаться к предыдущим версиям и отслеживать изменения в архитектуре вашего чат-бота.

Итеративная разработка UML-диаграмм позволит вам создавать более точные и актуальные представления архитектуры вашего чат-бота, а также облегчит процесс коммуникации и совместной работы в команде разработки.

### **Какие инструменты с открытым исходным кодом доступны для работы с UML?**

Существует множество инструментов с открытым исходным кодом для работы с UML, некоторые из популярных:

- StarUML: комплексный инструмент моделирования UML, который поддерживает все диаграммы UML и доступен для Windows, macOS и Linux.
- Umbrello UML Modeller: простой в использовании инструмент моделирования UML, который поддерживает все диаграммы UML и доступен для Linux.
- BOUML: быстрый и легкий инструмент моделирования UML, который поддерживает все диаграммы UML и доступен для Windows, macOS и Linux.
- ArgoUML: простой и удобный в использовании инструмент моделирования UML, который поддерживает большинство диаграмм UML и доступен для Windows, macOS и Linux.
- Visual Paradigm: комплексный инструмент моделирования UML, который поддерживает все диаграммы UML и доступен для Windows, macOS и Linux.

- PlantUML: простой и легкий инструмент моделирования UML, который поддерживает большинство диаграмм UML и доступен в виде подключаемого модуля для популярных IDE, таких как Eclipse и IntelliJ IDEA.

Эти инструменты предлагают ряд функций, включая поддержку всех диаграмм UML, настройку диаграмм, генерацию кода и обратный инжиниринг. Они идеально подходят для разработчиков и дизайнеров, которым необходимо создавать диаграммы UML для своих проектов.

### **Как оценить качество модели UML?**

Для оценки качества модели UML можно учитывать следующие факторы:

- Полнота: модель UML должна включать все необходимые элементы, отношения и атрибуты для точного представления моделируемой системы;
- Непротиворечивость: модель UML должна быть согласованной с точки зрения соглашений об именах, отношений и атрибутов;
- Корректность: модель UML должна точно отражать моделируемую систему реального мира, включая все ее функциональные и нефункциональные требования;
- Ясность: модель UML должна быть ясной и легкой для понимания, без двусмысленности и ненужной сложности;
- Повторное использование: модель UML должна быть разработана таким образом, чтобы можно было повторно использовать компоненты в других моделях или системах;
- Масштабируемость: модель UML должна быть масштабируемой, чтобы приспосабливаться к изменениям и росту моделируемой реальной системы;
- Валидация: модель UML должна быть проверена путем тестирования ее на реальной системе или проведения приемочного тестирования пользователями.

Оценка качества модели UML является непрерывным процессом и должна выполняться на разных этапах процесса моделирования. Регулярный просмотр и проверка могут гарантировать, что модель UML останется точной и актуальной на протяжении всего проекта.

## **12.2 Задачи**

### **12.2.1 1. Построение диаграммы классов для чат-бота Telegram:**

Задача: Разработать диаграмму классов для чат-бота Telegram, который имеет функционал отправки и приема текстовых сообщений от пользователей. Учесть основные классы, такие как «Бот», «Пользователь», «Сообщение», и связи между ними.

### 12.2.2 2. Создание диаграммы последовательности для обработки запросов от пользователя:

Задача: Разработать диаграмму последовательности, иллюстрирующую взаимодействие между пользователем и чат-ботом Telegram. Учесть различные сценарии, такие как отправка команды, получение ответа от бота и обработка ошибок.

### 12.2.3 3. Построение диаграммы состояний для управления состоянием чат-бота:

Задача: Создать диаграмму состояний, которая описывает различные состояния чат-бота Telegram, например, «Ожидание команды», «Отправка ответа», «Ошибка». Учесть переходы между состояниями и условия перехода.

### 12.2.4 4. Разработка диаграммы компонентов для чат-бота Telegram:

Задача: Построить диаграмму компонентов для чат-бота Telegram, которая отображает его структуру и взаимосвязи с другими компонентами. Учесть компоненты, такие как «Интерфейс Telegram», «Модуль обработки сообщений», «База данных», и связи между ними.

### 12.2.5 5. Построение диаграммы прецедентов для чат-бота Telegram:

Задача: Разработать диаграмму прецедентов, которая описывает функциональность чат-бота Telegram и взаимодействие с пользователями. Учесть прецеденты, такие как «Отправка сообщения», «Получение команды», «Обработка ошибок», и связи между ними.

## 12.3 Вопросы

1. Что такое UML и для чего он используется?
2. Какие основные виды диаграмм в UML вы знаете?
3. Что представляет собой диаграмма классов в UML?
4. Какие элементы включает диаграмма классов?
5. Какие основные элементы включает диаграмма последовательности?
6. Как можно представить взаимодействие между объектами на диаграмме последовательности?
7. Какие основные элементы включает диаграмма состояний?
8. Что такое начальное и конечное состояние на диаграмме состояний?
9. Как можно задать переходы между состояниями на диаграмме состояний?
10. Что такое диаграмма компонентов в UML и для чего она используется?
11. Какие элементы включает диаграмма компонентов?
12. Какие дополнительные диаграммы включает стандарт UML, кроме уже упомянутых?
13. Каким образом можно использовать UML в процессе разработки программного обеспечения?
14. Какая роль у UML в проектировании систем и коммуникации между участниками команды разработки?

15. Какие преимущества и недостатки имеет использование UML в процессе разработки программного обеспечения?

---

## Код vs. Библиотека

---

### 13.1 Код -> Библиотека

#### 13.1.1 Мотивация

Каждый, кто пишет код, рано или поздно упирается в необходимость использовать свои наработки в нескольких проектах. Мы можем просто копировать участки кода каждый раз в новый проект, но это несет в себе большие издержки, так как мы теряем единую точку «входа», т.е. возможность внести изменения в одном месте и получить их везде, где мы используем этот код. Поэтому мы хотим сделать из нашего кода библиотеку, которая может быть полезна как нам, так и сторонним разработчикам, которую мы сможем поддерживать, функции которой будут описаны, а также автоматически тестироваться, чтобы уменьшить вероятность ошибок. В этом уроке мы поговорим о том, что должно быть в проекте помимо кода, чтобы его можно было назвать библиотекой, а в следующем — рассмотрим, как упаковать наш код, чтобы его могли устанавливать себе другие разработчики.

#### 13.1.2 Определяем функциональность библиотеки

Например, мы пишем функцию, которая будет сообщать нам, сколько памяти потребляет та или иная функция. Воспользуемся [документацией](#) к библиотеке `tracemalloc`. Она позволит нам отследить выделенное место. Также можно использовать библиотеку `sys`, которая позволит определить только ту память, которую занимает непосредственно объект с данными.

```
import tracemalloc
import sys

data = list()
print(f"According to sys: {sys.getsizeof(data)}")
for i in range(10):
    tracemalloc.start()
    before = tracemalloc.get_traced_memory()
```

(continues on next page)

(продолжение с предыдущей страницы)

```
data.append(1)
after = tracemalloc.get_traced_memory()
tracemalloc.stop()
print(f"According to tracemalloc: {after[1] - before[1]}")
print(f"According to sys: {sys.getsizeof(data)}")
```

Пример вывода:

```
According to sys: 72
According to tracemalloc: 32
According to sys: 104
According to tracemalloc: 0
According to sys: 104
According to tracemalloc: 0
According to sys: 104
According to tracemalloc: 0
According to sys: 104
According to tracemalloc: 64
According to sys: 136
According to tracemalloc: 0
According to sys: 136
According to tracemalloc: 0
According to sys: 136
According to tracemalloc: 0
According to sys: 136
According to tracemalloc: 128
According to sys: 200
According to tracemalloc: 0
According to sys: 200
```

Мы видим, что даже пустой массив занимает какое-то место в памяти (для 64-разрядной системы это 72 байта), и не каждое добавление элементов вызывает выделение памяти. Оговоримся, что в зависимости от того, как заполнять данные, результат будет разным. Ниже приведены примеры кода, которые дадут несколько другой вывод при том же результирующем массиве.

Альтернативный пример:

```
import tracemalloc
import sys

for i in range(10):
    tracemalloc.start()
    before = tracemalloc.get_traced_memory()
    data = [1] * i
    after = tracemalloc.get_traced_memory()
    tracemalloc.stop()
    print(f"According to tracemalloc: {after[1] - before[1]}")
    print(f"According to sys: {sys.getsizeof(data)}")
```

Еще альтернативный пример:

```
import tracemalloc
import sys
```

(continues on next page)

(продолжение с предыдущей страницы)

```

for i in range(10):
    tracemalloc.start()
    before = tracemalloc.get_traced_memory()
    data = [1 for _ in range(i)]
    after = tracemalloc.get_traced_memory()
    tracemalloc.stop()
    print(f"According to tracemalloc: {after[1] - before[1]}")
    print(f"According to sys: {sys.getsizeof(data)}")

```

Мы не будем обсуждать разницу в выводе, она связана с особенностями реализации списков в Python и способе создания конкретного списка. Для наших задач необязательно знать точные числа, но нужно иметь оценку объема памяти, выделяемой на те или иные операции, для того чтобы разбить полный массив данных на части (`chunk`) и обрабатывать так, чтобы все умещалось в памяти. Практически всегда можно сделать оценку потребляемой памяти вручную, но такой подход требует погружения в детали задачи.

Давайте попробуем разделить код выше на функциональную часть, которая делает что-то полезное для нас, и ту, которая оценивает память:

(код):

```

import tracemalloc
import sys

def do_something_usefull(n):
    data = [1 for _ in range(n)]
    return data

def execute_and_get_memory_usage(function, n):
    tracemalloc.start()
    before = tracemalloc.get_traced_memory()
    data = function(n)
    after = tracemalloc.get_traced_memory()
    tracemalloc.stop()
    print(f"According to tracemalloc: {after[1] - before[1]}")
    print(f"According to sys: {sys.getsizeof(data)}")
    return data

for i in range(10):
    execute_and_get_memory_usage(do_something_usefull, i)

```

Теперь можно попробовать сделать `execute_and_get_memory_usage` более универсальным, так, чтобы эта функция могла работать с любой функцией, а не только с `do_something_usefull`. Так как мы не знаем, какие аргументы будут у функций, нужен способ, который позволит не ограничиваться определенным количеством аргументов. Для этого в Python используются `*args` и `**kwargs`.

(код):

```

import tracemalloc
import sys

def do_something_usefull(*args):
    data = [1 for _ in range(args[0])]

```

(continues on next page)

(продолжение с предыдущей страницы)

```

return data

def do_something_else_usefull(n, m, step=1):
    data = [[1 for _ in range(n)] for _ in range(0, m, step)]
    return data

def execute_and_get_memory_usage(function, *args, **kwargs):
    tracemalloc.start()
    before = tracemalloc.get_traced_memory()
    result = function(*args, **kwargs)
    after = tracemalloc.get_traced_memory()
    tracemalloc.stop()
    print(f"According to tracemalloc: {after[1] - before[1]}")
    print(f"According to sys: {sys.getsizeof(data)}")
    return result

print('1D example:')
execute_and_get_memory_usage(do_something_usefull, 10)

print('\n2D example:')
execute_and_get_memory_usage(do_something_else_usefull, 10, 10, step=2)

```

Пример вывода:

```

1D example:
According to tracemalloc: 320
According to sys: 200

2D example:
According to tracemalloc: 1112
According to sys: 136

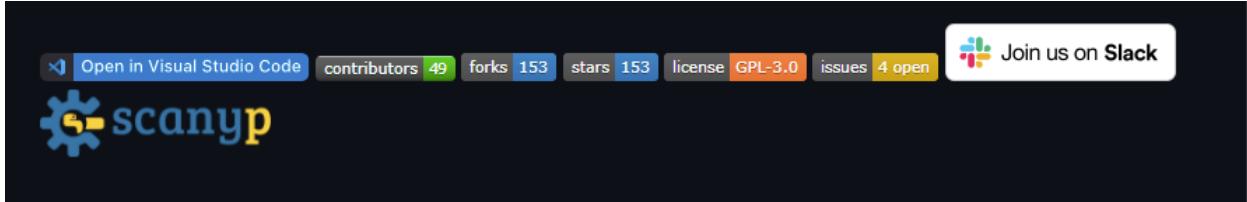
```

Мы видим, что `sys.getsizeof()` не информативен для вложенных объектов, поэтому оставим только `tracemalloc`. \*Примечание: функции типа `execute_and_get_memory_usage` в Python называются декораторами, мы не будем подробно на этом останавливаться, так как этот курс не про само программирование и изучение языка. Вы можете легко найти материалы для изучения самостоятельно, наподобие [таких](#) или [таких](#).

Мы получили инструмент для оценки памяти, которая требуется для выполнения той или иной функции с определенным набором входных данных. Теперь мы хотим использовать ее в других наших проектах. Возможно даже, она будет интересна другим разработчикам, поэтому нужно оформить все как принято при распространении кода. Здесь мы пока не будем говорить о менеджерах установки, таких как `pip` или `apt` (используется в `Ubuntu`), поговорим пока о том, что должно быть помимо кода, чтобы кодом можно было воспользоваться.

1. Первое, что должно сопровождать код, — это файл `README.md`. Мы будем использовать `Markdown` для разметки, также часто используется `reStructuredText`, тогда вы увидите `README.rst` в списке файлов проекта. Файл `README` нужен для того, чтобы понять, будет ли полезен данный проект тому, кто его читает. В `README` могут содержаться короткие примеры того, как использовать API библиотеки, и другая полезная информация, которую разработчик пожелал добавить. [Пункты](#), которые обычно включает `README`:
  - название проекта;
  - описание;

- содержание для удобной навигации;
- инструкции по установке;
- примеры использования приложения;
- упоминание участников, которые внесли вклад в проект;
- информация о лицензии (текст лицензии содержится в отдельном файле);
- бейджи;
- способы поучаствовать в проекте.



Научиться писать хороший README можно только через практику и примеры, которые вы сами считаете хорошими. [Здесь](#) собран список проектов, которые, по мнению авторов списка, являются хорошими, обратитесь к нему и составьте свое мнение о том, что должен включать этот файл.

2. Далее проект должен содержать файл **LICENSE** с текстом лицензии. По лицензиям в курсе есть [отдельный материал](#), посмотрите его и выберите подходящую.
3. Как мы уже говорили ранее, в README должны быть инструкции по установке. Установка может быть как полностью ручная, так и с той или иной степенью автоматизации. Например, в проектах на Python часто содержится файл `setup.py`, в котором прописан код для установки. Тогда инструкция по установке выглядит как одна строка:

```
python setup.py
```

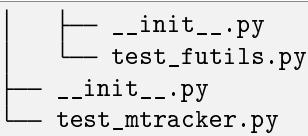
4. Любой код использует другой код, поэтому в проекте нужно прописать список используемых библиотек, **зависимостей**. Его можно вести вручную, например, в файле `requirements.txt`, который используется `pip` для установки зависимостей, так и автоматически, если мы используем другой инструмент, например, `poetry`.
5. Каталог, который содержит код проекта. Для разных языков программирования он будет называться по-разному. Для проектов на Python название каталога совпадает с названием проекта. В C++ это будут `src` и `include`.
6. Очень хорошая идея иметь тесты в проекте, это поможет понять пользователю вашей библиотеки, что код выполнился штатно, хотя бы для тестовых данных. Файлы с тестами будут иметь разную структуру для разных языков программирования.

Для Python:

```
my_project
└── mtracker
    ├── utils
    │   ├── __init__.py
    │   └── futils.py
    ├── __init__.py
    └── mtracker.py
└── test
    └── utils
```

(continues on next page)

(продолжение с предыдущей страницы)



*Примечание: \_\_init\_\_.py нужен для того, чтобы Python воспринимал каталог как пакет*

### 13.1.3 Собираем проект

Напишем файл для установки `setup.py`, для данного примера воспользуемся библиотекой `setuptools`. (код):

```

from setuptools import setup

setup(
    name='mtracker',
    version='1.0',
    description='Provides a decorator for memory usage tracking. The part of FOSS course.',
    license='MIT',
    author='Artem Vesnin',
    author_email='artemvesnin@gmail.com',
    url='https://github.com/standlab/mtracker',
    packages=['mtracker'],
    install_requires=[], # it is empty since we use standard python library
    extras_require={
        'test': [
            'pytest',
            'coverage',
        ],
    },
    python_requires='>=3',
)
  
```

Мы устанавливаем пакет `mtracker`, `install_requires` пустое, поскольку мы не используем зависимости не из стандартной библиотеки Python. `extras_require` содержат модули, которые не нужны при работе приложения, но понадобятся, если мы захотим прогнать тесты для проекта. Также указываем версию Python, под которой проект должен запуститься.

README можно посмотреть на [странице](#) проекта на GitHub.

От кода, который мы разрабатывали выше, оставим только часть с `tracemalloc`

(код):

```

import tracemalloc

def execute_and_get_memory_usage(function, *args, **kwargs):
    tracemalloc.start()
    before = tracemalloc.get_traced_memory()
    result = function(*args, **kwargs)
    after = tracemalloc.get_traced_memory()
  
```

(continues on next page)

(продолжение с предыдущей страницы)

```
tracemalloc.stop()
print(f"According to tracemalloc: {after[1] - before[1]}")
return result
```

И снабдим наш проект тестом. Здесь мы написали всего один тест, который проверит, не ломает ли наш трекер работу функций, на примере функции, которая возвращает список.

(код):

```
import pytest
from mtracker.mtracker import execute_and_get_memory_usage

def _generate_list(n):
    lst = []
    for i in range(n):
        lst.append(i)
    return lst

def test_no_change():
    n = 10
    lst1 = _generate_list(n)
    lst2 = execute_and_get_memory_usage(_generate_list, n)
    assert len(lst1) == len(lst2)
    for l1, l2 in zip(lst1, lst2):
        assert l1 == l2
```

Проверим тесты:

```
artem@pc:~/foss/mtracker$ pytest
=====
test session starts =====
platform linux -- Python 3.7.4, pytest-5.2.1, py-1.8.0, pluggy-0.13.0
rootdir: /home/artem/swdev/gitrepo/edu/toolchain_proj/mtracker
plugins: doctestplus-0.4.0, arraydiff-0.3, remotedata-0.3.2, openfiles-0.4.0
collected 1 item

test/test_mtracker.py .

[100%]

===== 1 passed in 0.04s =====
```

Устанавливаем:

```
artem@pc:~/foss/mtracker$ pip install .
Processing /home/artem/foss/mtracker
Building wheels for collected packages: mtracker
...
Successfully installed mtracker-1.0
```

Проверяем:

```
>>> from mtracker import mtracker
>>> mtracker.execute_and_get_memory_usage
<function execute_and_get_memory_usage at 0x7effef7b1cb0>
```

Проверяем установку как в README:

```
artem@pc:~$ pip install git+https://github.com/standlab/mtracker.git#egg=mtracker
Collecting mtracker from git+https://github.com/standlab/mtracker.git#egg=mtracker
...
Installing collected packages: mtracker
Successfully installed mtracker-1.0
```

### 13.1.4 Выводы

Мы снабдили написанную нами функцию необходимыми файлами, чтобы ее можно было выложить в открытый доступ и позволить другим разработчикам устанавливать ее себе и использовать ее API.

## 13.2 Задачи

### 13.2.1 Задача 1

1. Организуйте код библиотеки `useful_utils`, который содержит модули для математических операций и работы со строками, в виде библиотеки.
  - Определите модули и их функциональность: в структуре каталогов создайте отдельные файлы модулей для представления различных функций вашей библиотеки: `math_operations.py` и `string_utils.py`.
  - Функции в `math_operations.py`:
    - `add(a, b)`: принимает два аргумента `a` и `b` и возвращает их сумму.
    - `subtract(a, b)`: принимает два аргумента `a` и `b` и возвращает их разницу.
    - `multiply(a, b)`: принимает два аргумента `a` и `b` и возвращает их произведение.
    - `divide(a, b)`: принимает два аргумента `a` и `b` и возвращает результат деления `a` на `b`.
  - Функции для `string_utils.py`:
    - `concatenate(a, b)`: принимает две строки `a` и `b` и возвращает их конкатенацию.
    - `reverse(s)`: принимает строку `s` и возвращает ее реверс.
    - `substring(s, i, j)`: принимает строку `s` и возвращает подстроку.
  - Создайте файл `__init__.py` и напишите содержимое так, чтобы функции из модулей были доступны, когда кто-то импортирует вашу библиотеку, т.е. чтобы можно было выполнить импорт таким образом.

```
from useful_utils import add, subtract, multiply, divide, concatenate, reverse
```

- Протестируйте свой код: создайте отдельный каталог тестов для написания тестовых случаев для вашей библиотеки. Реализуйте тесты для каждой функции.
- Напишите `setup.py`
- Добавьте `README`: Создайте файл `README.md` с кратким описанием вашей библиотеки, инструкциями по ее установке и использованию и примерами использования каждой функции.

### Ответ

Структура проекта:

```
my_library/
| -- useful_utils/
| | -- __init__.py
| | -- math_operations.py
| | -- string_utils.py
| -- tests/
| | -- test_math_operations.py
| | -- test_string_utils.py
| -- setup.py
| -- README.md
```

Содержимое math\_operations.py:

(код):

```
def add(a, b):
    """Return the sum of two numbers."""
    return a + b

def subtract(a, b):
    """Return the difference between two numbers."""
    return a - b

def multiply(a, b):
    """Return the product of two numbers."""
    return a * b

def divide(a, b):
    """
    Return the result of dividing two numbers.

    If b is zero, raise a ValueError.
    """
    if b == 0:
        raise ValueError("Cannot divide by zero")
    return a / b
```

Содержимое string\_utils.py:

(код):

```
def concatenate(a, b):
    """Concatenate two strings and return the result."""
    return a + b

def reverse(s):
    """Return the reverse of the input string."""
    return s[::-1]

def substring(s, i, j):
    """Return the substring of the input string."""
    if j < len(s) and i < len(s)
        return s[i:j]
```

Содержимое \_\_init\_\_.py:

(код):

```
# __init__.py

# Import functions from math_operations.py
from .math_operations import add, subtract, multiply, divide

# Import functions from string_utils.py
from .string_utils import concatenate, reverse, substring
```

Содержимое setup.py:

(код):

```
from setuptools import setup, find_packages

setup(
    name='useful-utils',
    version='1.0.0',
    packages=find_packages(),
    description='Utils to make life easier',
    author='Your Name',
    author_email='your_name@example.com',
    url='https://github.com/your_username/your_repository',
    install_requires=[
        'pytest'
    ],
)
```

Содержимое тестов (может отличаться):

(код):

```
from useful_utils.math_operations import add, subtract, multiply, divide

def test_add():
    assert add(2, 3) == 5
    assert add(-1, 5) == 4

def test_subtract():
    assert subtract(5, 2) == 3
    assert subtract(10, 7) == 3

def test_multiply():
    assert multiply(3, 4) == 12
    assert multiply(0, 100) == 0

def test_divide():
    assert divide(10, 2) == 5
    assert divide(20, 4) == 5
    assert divide(7, 2) == 3.5
```

(код):

```

from useful_utils.string_utils import concatenate, reverse, substring

def test_concatenate():
    assert concatenate('Hello', 'World') == 'HelloWorld'
    assert concatenate('Python', ' is awesome') == 'Python is awesome'

def test_reverse():
    assert reverse('hello') == 'olleh'
    assert reverse('Python') == 'nohtyP'

def test_substring():
    assert substring('hello', 2, 4) == 'll'
    assert substring('hello', 2, 1) == ''
    assert substring('hello', 1, 1) == ''

```

### 13.2.2 Задача 2

Добавьте дополнительные данные в setup.py.

В этой задаче вы улучшите файл setup.py, который писали для предыдущей задачи. Также можно выполнить это задание для любого другого проекта, над которым мы работаем, или придумать новый. Это нужно, если мы будем публиковать библиотеку для других разработчиков: добавив больше данных, мы дадим дополнительную информацию о библиотеке. Дополнительная информация будет включать лицензию вашей библиотеки, подробное описание, классификаторы и ключевые слова.

- \* **Лицензия:** выберите подходящую лицензию для своей библиотеки и добавьте ее в файл setup.py. Выбирайте из MIT, BSD, Apache License 2.0 и GPL или используйте свою. Добавьте лицензию непосредственно в функцию setup().
- \* **Подробное описание:** создайте файл README.md в корневом каталоге вашего проекта и напишите подробное описание вашей библиотеки. Затем измените файл setup.py, чтобы включить это подробное описание при упаковке библиотеки. Вы можете использовать функции open() и read() для чтения содержимого файла README.md.
- \* **Классификаторы:** Классификаторы – это способ категоризировать вашу библиотеку. Вы можете добавить классификаторы, чтобы указать целевую аудиторию, статус разработки, язык программирования и многое другое. Доступные параметры см. в списке классификаторов PyPI.
- \* **Ключевые слова:** добавьте список ключевых слов, описывающих основные функции и возможности вашей библиотеки. Это поможет пользователям найти вашу библиотеку в указателях пакетов.

### Ответ

(код):

```

with open('README.md', 'r', encoding='utf-8') as f:
    long_description = f.read()

setup(
    name='useful-utils',
    version='1.0.0',

```

(continues on next page)

(продолжение с предыдущей страницы)

```

packages=find_packages(),
description='Utils to make life easier',
long_description=long_description,
long_description_content_type='text/markdown', # Specify the type of long
→description
author='Your Name',
author_email='your_name@example.com',
url='https://github.com/your_username/your_repository',
install_requires=[
    'pytest',
],
license='MIT', # Add your library's license here
classifiers=[
    # Add classifiers that describe your library
    'Development Status :: 4 - Beta',
    'Intended Audience :: Developers',
    'License :: OSI Approved :: MIT License',
    'Programming Language :: Python :: 3',
    'Programming Language :: Python :: 3.6',
    'Programming Language :: Python :: 3.7',
    'Programming Language :: Python :: 3.8',
    'Programming Language :: Python :: 3.9',
],
keywords='python library example', # Add keywords that describe your library
)

```

### 13.2.3 Задача 3

Напишите makefile, который будет автоматизировать некоторые процедуры, которые мы осуществляем с библиотекой. Определите следующие задачи (target):

- \* Задача **\*\*install\*\*** устанавливает зависимости проекта, указанные в файле requirements.txt.
- \* Задача **\*\*test\*\*** запускает тесты с помощью pytest.
- \* Задача **\*\*clean\*\*** удаляет временные файлы, созданные в процессе упаковки.

#### Ответ

(код):

```

PACKAGE_NAME := my-library
PYTHON := python3

.PHONY: install test clean

install:
    @echo "Installing dependencies..."
    $(PYTHON) -m pip install -r requirements.txt

test:
    @echo "Running tests..."

```

(continues on next page)

(продолжение с предыдущей страницы)

```
$(PYTHON) -m pytest

clean:
    @echo "Cleaning up..."
    rm -rf dist/ build/ *.egg-info
```

### 13.2.4 Задача 4

Напишите комментарии в виде docstring, которые описывают работу функций и параметры. Помните, что в некоторых случаях порядок параметров неважен, например, при сложении, поэтому можно описать их как первое число и второе число. В других случаях, например, при вычитании, порядок важен, и мы должны описать числа соответствующим образом, даже если интуитивно понятно, что из первого вычитается второе. Написать docstring нужно для функции в модуле `math_operations.py`:

```
* add(a, b): принимает два аргумента a и b и возвращает их сумму.
* subtract(a, b): принимает два аргумента a и b и возвращает их разницу.
* multiply(a, b): принимает два аргумента a и b и возвращает их произведение.
* devide(a, b): принимает два аргумента a и b и возвращает результат деления a на b.
```

Добавьте задачу `docs` в makefile для сборки документации с помощью `pdoc`. Собранная документация не должна попасть в репозиторий git, поэтому нужно модифицировать задачу `clean`.

#### Ответ

(код):

```
def add(a, b):
    """
    Return the sum of two numbers.

    Parameters:
        a (int or float): The first number.
        b (int or float): The second number.

    Returns:
        int or float: The sum of a and b.
    """
    return a + b

def subtract(a, b):
    """
    Return the difference between two numbers.

    Parameters:
        a (int or float): The minuend.
        b (int or float): The subtrahend.

    Returns:
        int or float: The result of subtracting b from a.
    """
    return a - b
```

(continues on next page)

(продолжение с предыдущей страницы)

```

def multiply(a, b):
    """
    Return the product of two numbers.

    Parameters:
        a (int or float): The first factor.
        b (int or float): The second factor.

    Returns:
        int or float: The product of a and b.
    """
    return a * b

def divide(a, b):
    """
    Return the result of dividing two numbers.

    Parameters:
        a (int or float): The dividend.
        b (int or float): The divisor.

    Returns:
        int or float: The result of dividing a by b.

    Raises:
        ValueError: If b is zero (0).
    """
    if b == 0:
        raise ValueError("Cannot divide by zero")
    return a / b

```

Содержимое makefile:

(код):

```

PACKAGE_NAME := useful-utils
PYTHON := python3
DOCS_DIR := docs

.PHONY: install test clean package docs

install:
    @echo "Installing dependencies..."
    $(PYTHON) -m pip install -r requirements.txt

test:
    @echo "Running tests..."
    $(PYTHON) -m pytest

clean:
    @echo "Cleaning up..."
    rm -rf dist/ build/ docs/ *.egg-info

```

(continues on next page)

(продолжение с предыдущей страницы)

**docs:**

```
@echo "Generating documentation..."
pdoc --html $(PACKAGE_NAME) --output-dir $(DOCS_DIR) --force
```

### 13.3 Вопросы

1. Какова цель создания библиотеки кода? а) Чтобы продемонстрировать свои навыки программирования потенциальным работодателям. б) Для организации и повторного использования часто используемого кода. в) Чтобы заменить систему контроля версий. г) Чтобы увеличить сложность вашего проекта.
2. Каков рекомендуемый способ организации библиотеки кода в Python? а) Создать один большой файл Python со всеми функциями. б) Разделить связанные функции в разные модули, а модули в пакеты. в) Использовать несколько файлов Python, при этом организация по смыслу не важна. г) Хранить все функции в одном списке или словаре.
3. Какова обычная практика документирования функций в библиотеке кода? а) Добавление комментариев перед каждой функцией. б) Написание многострочных комментариев внутри тела каждой функции. в) Предоставление отдельного файла README для документации назначения функций. г) Встраивание документации в код с помощью специальных тегов, содержимое которых определено в отдельном файле.
4. Какое из следующих утверждений об упаковке (packaging) кода в виде библиотеки в Python верно? а) Упаковка не является обязательной для библиотеки кода. б) Упаковка гарантирует, что библиотека кода работает только на компьютерах с Windows. в) Упаковка делает библиотеку кода независимой и гарантирует установку, а также работоспособность (при надлежащем покрытии тестами). г) Упаковка полезна только для крупномасштабных коммерческих проектов.
5. Каково назначение файла `setup.py` в библиотеке кода Python? а) Используется для указания имени автора и адреса электронной почты. б) Это файл конфигурации, используемый для настройки виртуальных сред. в) Он используется для определения зависимостей проекта и инструкций по установке. г) Это сценарий, используемый для автоматизации процесса тестирования библиотеки кода.
6. Как создать устанавливаемый пакет (wheel) из библиотеки кода Python? а) Путем сжатия исходных файлов библиотеки в ZIP-архив. б) С помощью команды `pip package` в основном файле библиотеки. в) Выполнив команду `python -m build` в каталоге библиотеки. г) Вручную скопировав файлы библиотеки в каталог `site-packages` Python.
7. Какова цель использования виртуальной среды при разработке библиотеки? а) Чтобы избежать установки каких-либо зависимостей для библиотеки. б) Изолировать библиотеку от других проектов Python и зависимостей, которые эти проекты несут с собой. в) Чтобы библиотеку можно было использовать только в определенной операционной системе. г) Увеличить скорость запуска тестов для библиотеки.
8. Какое из следующих утверждений о контроле версий в контексте создания библиотеки кода верно? а) Контроль версий используется для документирования изменений, внесенных в библиотеку кода. б) Контроль версий не нужен при работе с личной библиотекой кода. в) Контроль версий помогает управлять изменениями кода и документации и отслеживать их с течением времени. г) Контроль версий используется только для совместных проектов с несколькими участниками.

**Ответы:**

1. б)

2. 6)

3. 6)

4. в)

5. в)

6. в)

7. 6)

8. в)

## Основы Open Source

---

### 14.1 Основы Open Source, базовые определения и принципы устройства

#### 14.1.1 Что такое Open Source?

Открытый исходный код относится к типу лицензии, которая позволяет сделать исходный код программного обеспечения общедоступным, что позволяет любому использовать, модифицировать или распространять код. Это способствует сотрудничеству и прозрачности в разработке программного обеспечения и часто приводит к созданию более надежных и безопасных продуктов.

Программное обеспечение с открытым исходным кодом не только находится в свободном доступе, но и основано на принципах сотрудничества и сообщества. Разработчики со всего мира могут вносить свой вклад в код, исправляя ошибки и добавляя новые функции, что может привести к более быстрому внедрению инноваций и повышению безопасности.

Программное обеспечение с открытым исходным кодом также имеет тенденцию быть более прозрачным и настраиваемым, поскольку пользователи имеют доступ к базовому коду и могут вносить изменения для удовлетворения своих конкретных потребностей.

Это контрастирует с проприетарным программным обеспечением, исходный код которого обычно хранится в тайне, а пользователи могут использовать программное обеспечение только в том виде, в каком оно представляется поставщиком.

### **14.1.2 Как развивалось сообщество открытого исходного кода?**

Сообщество разработчиков программного обеспечения с открытым исходным кодом претерпело значительные изменения с момента своего создания в конце 1980-х и начале 1990-х годов. Вот некоторые из ключевых моментов, в которых сообщество открытого исходного кода изменилось за эти годы:

- Рост и расширение. Сообщество разработчиков ПО с открытым исходным кодом за последние годы значительно выросло в размерах и влиянии. Сегодня существуют миллионы проектов с открытым исходным кодом, и подавляющее большинство проектов по разработке программного обеспечения так или иначе используют компоненты с открытым исходным кодом. Это стало возможным благодаря развитию Интернета и онлайн-инструментов для совместной работы.
- Процесс взросления. Сообщество разработчиков ПО с открытым исходным кодом с годами стало более структурированным и профессиональным. Многие крупные организации теперь имеют специализированные программы с открытым исходным кодом, и есть много хорошо зарекомендовавших себя сообществ, таких как сообщество Linux, у которых есть четкие процессы для разработки и совместной работы.
- Более широкое внедрение. Модель с открытым исходным кодом была принята более широким кругом организаций и отраслей, включая правительства, университеты и коммерческие предприятия. Это привело к разработке новых проектов с открытым исходным кодом и росту существующих, а также к увеличению инвестиций в программное обеспечение с открытым исходным кодом.
- Рост коммерциализации. С годами модель с открытым исходным кодом становится все более коммерциализированной, поскольку компании стремятся монетизировать проекты с открытым исходным кодом с помощью услуг, поддержки и обучения. Это привело к созданию новых бизнес-моделей, таких как программное обеспечение с открытым исходным кодом как услуга (SaaS), и помогло обеспечить устойчивость многих проектов с открытым исходным кодом.
- Растущее разнообразие. Сообщество разработчиков открытого исходного кода стало более разнообразным, что отражает растущее разнообразие технологической отрасли в целом. Это привело к более широкому представительству недостаточно представленных групп в сфере технологий, в частности, женщин и национальных меньшинств, и способствовало инклюзивности и разнообразию в технологической отрасли.

Подводя итог, можно сказать, что сообщество открытого исходного кода значительно эволюционировало за эти годы, увеличившись в размерах и влиянии, взрослея и становясь более структуризованными, будучи принятым более широким кругом организаций и отраслей, становясь более коммерциализированными и увеличивая разнообразие.

### **14.1.3 Как открытый исходный код помогает развивать технологии и ускорять прогресс?**

Открытый исходный код помогает развивать технологии и ускорять прогресс несколькими способами:

1. Сотрудничество: открытый исходный код позволяет разработчикам со всего мира сотрудничать в проекте, объединяя свои знания и ресурсы для создания лучшего программного обеспечения.
2. Инновации: открытый исходный код поощряет инновации, позволяя разработчикам использовать существующий код, экспериментировать с новыми идеями и создавать новые технологии.
3. Сообщество: сообщество с открытым исходным кодом предоставляет разработчикам платформу для обмена знаниями, идеями и помохи друг другу в решении проблем. Это способствует развитию чувства общности и сотрудничества, которые могут способствовать техническому прогрессу.

4. Доступность: открытый исходный код делает технологию более доступной для более широкого круга людей, в том числе для тех, у кого может не быть финансовых ресурсов для покупки проприетарного программного обеспечения.
5. Качество: проекты с открытым исходным кодом часто разрабатываются большими сообществами добровольцев, которые могут быстрее выявлять и исправлять ошибки и уязвимости. Это помогает повысить качество и надежность программного обеспечения с открытым исходным кодом.
6. Рентабельность: программное обеспечение с открытым исходным кодом часто можно использовать бесплатно, что может быть экономичным вариантом для предприятий и организаций, которым необходимо развивать технологии.

В целом открытый исходный код играет важную роль в развитии технического прогресса, способствуя развитию и сотрудничеству среди разработчиков и коллективов, инновациям, формированию сообществ, доступности и качеству разработки программного обеспечения.



## Лицензии Open Source

---

### 15.1 Лицензии Open Source

Принципы работы лицензий, выбор лицензии и применение. Что можно и что нельзя делать с кодом, опубликованным под открытыми лицензиями разных типов. Примеры проектов, в которых используются те или иные лицензии.

#### 15.1.1 Самые распространенные лицензии Open Source

Лицензии с открытым исходным кодом — это юридические соглашения, которые определяют, как можно использовать, изменять и распространять исходный код программного обеспечения с открытым исходным кодом. Некоторые из самых популярных лицензий:

- **GPL** (Стандартная общественная лицензия GNU): вирусная лицензия, которая требует, чтобы производные работы также были с открытым исходным кодом и лицензировались по GPL.
- **Лицензия MIT**: разрешающая лицензия, которая позволяет использовать, изменять и распространять программное обеспечение даже в коммерческих приложениях.
- **Лицензия Apache 2.0**: разрешающая лицензия, которая позволяет использовать, изменять и распространять программное обеспечение, если пользователи включают в дистрибутив лицензию Apache и уведомление об авторских правах.
- **Лицензия BSD (Berkeley Software Distribution)**: разрешающая лицензия, которая позволяет использовать, изменять и распространять программное обеспечение, если пользователи включают в дистрибутив лицензию BSD и уведомление об авторских правах.

Выбор лицензии с открытым исходным кодом — важное решение для разработчиков программного обеспечения, поскольку оно определяет, как другие могут использовать их код, и может повлиять на рост и внедрение программного обеспечения.

---

1. **GPL (GNU General Public License)**. Одна из самых известных и популярных лицензий с открытым исходным кодом. Ее особенности:

- **Вирусный характер:** требуется, чтобы все производные работы также были с открытым исходным кодом и распространялись под лицензией GPL.
  - **Доступность исходного кода:** любой, кто получает копию программного обеспечения под лицензией GPL, имеет право на получение исходного кода.
  - **Открытость:** любой может использовать, модифицировать и распространять программное обеспечение под лицензией GPL.
  - **Бесплатное использование:** для использования программного обеспечения под лицензией GPL не требуется никаких дополнительных сборов или лицензионных платежей.
  - **Copyleft:** положение, требующее, чтобы все производные работы имели одну и ту же лицензию с открытым исходным кодом.
  - **GPL является** вирусной лицензией, поэтому все производные работы также должны лицензироваться по GPL, гарантируя, что программное обеспечение остается открытым и свободно доступным для использования и распространения любым лицом.
- 

**2. Лицензия MIT** — это разрешительная лицензия с открытым исходным кодом, которая позволяет использовать, модифицировать и распространять программное обеспечение с очень небольшими ограничениями. Некоторые из ключевых особенностей:

- Вседозволенность: лицензия MIT — одна из самых гибких разрешительных лицензий на ПО с открытым исходным кодом, допускающая широкий спектр вариантов использования, включая коммерческое.
- Отсутствие авторского лева: в отличие от некоторых других лицензий с открытым исходным кодом, таких как GPL, лицензия MIT не требует выпуска производных работ под той же лицензией.
- Простая и понятная: лицензия MIT короткая, простая и понятная, что делает ее популярным выбором среди разработчиков.
- Ограниченная ответственность: лицензия включает отказ от ответственности, защищающий лицензиара от любых убытков, которые могут возникнуть в результате использования программного обеспечения.
- Совместимость: лицензия MIT совместима со многими другими лицензиями с открытым исходным кодом, что позволяет легко комбинировать ее с другими проектами с открытым исходным кодом.

Таким образом, **лицензия MIT** — это гибкая разрешительная лицензия, которая позволяет легко интегрироваться с другими проектами с открытым исходным кодом, что делает ее популярным выбором среди разработчиков.

---

**3. Лицензия Apache** — это широко используемая лицензия с открытым исходным кодом, которая позволяет использовать, модифицировать и распространять программное обеспечение с некоторыми ограничениями. Некоторые из ключевых особенностей:

- Разрешение: лицензия Apache — это разрешающая лицензия с открытым исходным кодом, которая допускает широкий спектр вариантов использования, включая коммерческое.
- Нет авторского лева: в отличие от некоторых других лицензий с открытым исходным кодом, таких как GPL, лицензия Apache не требует выпуска производных работ под той же лицензией.
- Патенты: лицензия включает формулировку, связанную с патентами, что позволяет лицензиару предоставлять патентную лицензию пользователям программного обеспечения.

- Ограниченная ответственность: лицензия включает отказ от ответственности, защищающий лицензиара от любых убытков, которые могут возникнуть в результате использования программного обеспечения.
- Совместимость: лицензия Apache совместима со многими другими лицензиями с открытым исходным кодом, что позволяет легко комбинировать ее с другими проектами с открытым исходным кодом.

Таким образом, *лицензия Apache* — это разрешающая лицензия, которая обеспечивает баланс между защитой прав лицензиара и предоставлением пользователям возможности свободно использовать, модифицировать и распространять программное обеспечение. Она широко используется и пользуется уважением в сообществе открытого исходного кода.

---

**4. Лицензия BSD (Berkeley Software Distribution)** — это разрешительная лицензия с открытым исходным кодом, которая позволяет использовать, модифицировать и распространять программное обеспечение с очень небольшими ограничениями. Некоторые из ключевых особенностей:

- Вседозволенность: лицензия **BSD** является одной из наиболее гибких разрешительных лицензий с открытым исходным кодом, допускающей широкий спектр вариантов использования, включая коммерческое.
- Отсутствие авторского лева: в отличие от некоторых других лицензий с открытым исходным кодом, таких как **GPL**, лицензия **BSD** не требует выпуска производных работ под той же лицензией.
- Простота: лицензия **BSD** короткая, простая и понятная, что делает ее популярным выбором среди разработчиков.
- Ограниченная ответственность: лицензия включает отказ от ответственности, защищающий лицензиара от любых убытков, которые могут возникнуть в результате использования программного обеспечения.
- Совместимость: лицензия BSD совместима со многими другими лицензиями с открытым исходным кодом, что позволяет легко комбинировать ее с другими проектами с открытым исходным кодом.

Таким образом, лицензия **BSD** является разрешающей лицензией, которая обеспечивает баланс между защитой прав лицензиара и разрешением пользователям свободно использовать, модифицировать и распространять программное обеспечение. Она также широко используется и пользуется уважением в сообществе открытого исходного кода.

### 15.1.2 Какая лицензия ПО позволяет использовать исходный код для коммерческой разработки ПО?

Многие лицензии с открытым исходным кодом, в том числе лицензия MIT, лицензия Apache и лицензия BSD, позволяют использовать исходный код для разработки коммерческого программного обеспечения. Эти лицензии являются разрешительными, то есть они накладывают очень мало ограничений на использование и распространение программного обеспечения. Пока вы соблюдаете условия, указанные в лицензии, вы можете свободно использовать код в коммерческих целях, в том числе включать его в проприетарные программные продукты.

Однако важно отметить, что разные лицензии могут иметь разные условия, такие как требования к установлению авторства или выпуску производных работ по одной и той же лицензии. Всегда полезно внимательно ознакомиться с условиями лицензии, прежде чем использовать код в коммерческом контексте.

### 15.1.3 Какая лицензия ПО запрещает использование кода в коммерческих целях, для получения прибыли?

Стандартная общественная лицензия (GPL) — это лицензия, которая позволяет коммерческое использование при условии, что исходный код будет открыт. GPL способствует совместному использованию и распространению бесплатного программного обеспечения, требуя, чтобы исходный код был доступен любому, кто получит копию программного обеспечения. Это гарантирует, что код остается открытым и доступным, а также поощряет сотрудничество и инновации в сообществе открытого исходного кода. GPL является лицензией с авторским левом, что означает, что любые производные работы также должны быть выпущены под той же лицензией, и что исходный код должен быть доступен любому, кто получит копию программного обеспечения.

Важно отметить, что GPL является лишь одной из многих лицензий с открытым исходным кодом, и не все лицензии с открытым исходным кодом имеют одинаковые ограничения на коммерческое использование. Некоторые лицензии, такие как лицензия MIT и лицензия BSD, являются разрешительными и накладывают очень мало ограничений на коммерческое использование.

#### В чем отличие лицензий GPL от LGPL?

Стандартная общественная лицензия (GPL) и Стандартная общественная лицензия ограниченного применения (LGPL) являются лицензиями на программное обеспечение с открытым исходным кодом, но у них есть некоторые ключевые отличия:

- Копирайт, или авторское право: GPL является сильной лицензией с авторским правом, что означает, что любые производные работы также должны быть выпущены под той же лицензией. LGPL, с другой стороны, является более слабой лицензией с авторским правом, которая позволяет выпускать производные работы под другой лицензией, пока исходный код LGPL остается неповрежденным и открытым.
- Использование библиотек: LGPL была разработана специально для использования с библиотеками, что упрощает разработчикам использование кода под лицензией LGPL в своих собственных программных продуктах. С другой стороны, GPL больше подходит для автономных программных приложений.
- Коммерческое использование: и GPL, и LGPL допускают коммерческое использование кода, но требования к предоставлению исходного кода и выпуску производных работ под одной и той же лицензией более строгие для кода под лицензией GPL.
- Совместимость: GPL совместима с ограниченным набором других лицензий с открытым исходным кодом, в то время как LGPL более совместима с более широким спектром лицензий, включая разрешающие лицензии, такие как лицензия MIT и лицензия BSD.

Таким образом, GPL и LGPL являются лицензиями с открытым исходным кодом, но они имеют разные требования к авторскому праву и подходят для разных типов программных проектов. GPL больше подходит для автономных программных приложений, тогда как LGPL больше подходит для библиотек и других компонентов, которые могут использоваться в других программных проектах.

### 15.1.4 Какие есть варианты использования лицензии GPL?

Стандартная общественная лицензия (GPL) — это популярная лицензия с открытым исходным кодом, которая предоставляет несколько вариантов использования программного обеспечения. Вот некоторые варианты.

- Использование кода как есть: вы можете использовать код без внесения каких-либо изменений, если вы соблюдаете условия лицензии GPL, которая может включать требования об указании авторства или предоставлении доступа к исходному коду другим лицам.
- Модификация кода: вы можете модифицировать код и использовать его в своих целях, но вы должны выпустить измененный код под той же лицензией GPL и сделать его доступным для других.
- Распространение кода: вы можете распространять код как в его исходной форме, так и в виде измененной версии, но вы должны сделать исходный код доступным для всех, кто получит копию программного обеспечения.
- Коммерческое использование: вы можете использовать код в коммерческих целях, но вы должны соблюдать условия лицензии GPL, которая может включать требования о предоставлении исходного кода другим лицам или о выпуске производных работ под той же лицензией.

Важно отметить, что лицензия GPL является строгой лицензией с авторским левом, что означает, что любые производные работы должны выпускаться под той же лицензией. Это может повлиять на распространение и использование проприетарного программного обеспечения, включающего код под лицензией GPL.

Таким образом, есть несколько вариантов использования программного обеспечения под лицензией GPL, включая использование кода как есть, изменение кода, распространение кода и его использование в коммерческих целях. Однако соблюдение условий лицензии, в том числе требований авторского лева, необходимо во всех случаях.

---

Microsoft Windows является проприетарным программным обеспечением, а это означает, что оно не имеет открытого исходного кода, и его исходный код не является общедоступным. Microsoft владеет исключительными правами на код и контролирует, как его можно использовать и распространять. Microsoft предоставляет пользователям Windows лицензию на основании проприетарного лицензионного соглашения, в котором изложены условия использования программного обеспечения. Конкретное лицензионное соглашение для Windows может различаться в зависимости от версии Windows и типа пользователя, но в целом оно ограничивает использование программного обеспечения лицензованным устройством и запрещает несанкционированное копирование или распространение программного обеспечения.

### 15.1.5 Какая лицензия у ОС Android?

Операционная система Android — это программное обеспечение с открытым исходным кодом, выпущенное под лицензией Apache License 2.0. Apache License 2.0 — это разрешительная лицензия с открытым исходным кодом, которая позволяет пользователям использовать, изменять и распространять программное обеспечение, в том числе в коммерческих целях, без необходимости делать исходный код доступным для других.

В соответствии с лицензией Apache 2.0 пользователи должны включать копию лицензии и уведомление о любых изменениях, внесенных в исходный код. Лицензия также включает положения о патентной защите, которые помогают гарантировать, что пользователи программного обеспечения не будут привлечены к ответственности за нарушение патентных прав.

Таким образом, Android — это операционная система с открытым исходным кодом, выпущенная под лицензией Apache License 2.0, которая позволяет широко использовать, модифицировать и распространять программное обеспечение, обеспечивая при этом патентную защиту пользователей.

Google Chrome является проприетарным программным обеспечением, а это означает, что его исходный код не является общедоступным и контролируется Google. Google лицензирует Chrome для пользователей в соответствии с проприетарным лицензионным соглашением, в котором изложены условия использования программного обеспечения.

Однако некоторые части браузера Chrome, в том числе проект с открытым исходным кодом Chromium, выпускаются под разрешительной лицензией BSD, которая позволяет использовать, модифицировать и распространять программное обеспечение как в коммерческих, так и в некоммерческих целях. Исходный код этих частей браузера находится в свободном доступе для всех желающих.

Таким образом, браузер Google Chrome является проприетарным программным обеспечением, но некоторые его компоненты, в том числе проект с открытым исходным кодом Chromium, выпускаются под лицензией BSD, что позволяет широко использовать, модифицировать и распространять программное обеспечение.

### **15.1.6 Какую лицензию выбрать, если я хочу использовать чужой код в своем проекте?**

Выбор лицензии зависит от ваших конкретных потребностей и целей вашего проекта. Вот некоторые распространенные сценарии и типы лицензий, которые часто используются в каждом случае:

- Если вы хотите использовать чужой код как часть проприетарного проекта с закрытым исходным кодом, вам следует искать лицензию, допускающую коммерческое использование, например, лицензию Apache 2.0 или лицензию BSD.
- Если вы хотите использовать чужой код в проекте с открытым исходным кодом, но не хотите делиться своими изменениями с другими, вам следует искать разрешающую лицензию, такую как Apache License 2.0 или BSD.
- Если вы хотите использовать чужой код в проекте с открытым исходным кодом и хотите убедиться, что любые изменения, которые вы вносите в код, также являются открытым исходным кодом, вам следует искать лицензию с авторским левом, например, лицензию GPL.
- Если вы хотите использовать чужой код в проекте с открытым исходным кодом и хотите убедиться, что любые изменения, которые вы вносите в код, также являются открытым исходным кодом, но вы не хотите ограничивать использование кода в коммерческих приложениях, вам следует искать лицензию, такую как лицензия LGPL.
- Важно внимательно прочитать положения и условия любой лицензии, которую вы рассматриваете, и при необходимости обратиться за юридической консультацией, чтобы убедиться, что вы понимаете требования и обязательства лицензии и что она соответствует вашим потребностям.

Таким образом, выбор лицензии на использование чужого кода в вашем проекте зависит от ваших конкретных потребностей и целей и может включать разрешительные лицензии, такие как лицензия Apache 2.0 или лицензия BSD, или лицензии с авторским левом, такие как лицензия GPL или LGPL.

### 15.1.7 Несколько общих советов по лицензированию и защите прав разработчиков программного обеспечения

1. Выберите правильную лицензию: при выпуске программного обеспечения важно выбрать лицензию, которая соответствует вашим целям и ценностям. Рассмотрите доступные лицензии с открытым исходным кодом и подумайте, какое использование вы хотите разрешить и какие ограничения вы хотите наложить на пользователей вашего кода.
2. Прочтите и поймите лицензию: прежде чем использовать чужой код в своем проекте, обязательно прочтите и полностью поймите условия лицензии. Убедитесь, что вы соблюдаете лицензионные требования, включая любые обязательства по атрибуции или совместному использованию, и помните о любых ограничениях на использование или распространение.
3. Защитите свою интеллектуальную собственность. Если вы разработали программное обеспечение, которое вы считаете проприетарным, примите меры для защиты своей интеллектуальной собственности, например, подайте заявку на патенты, товарные знаки или авторские права. Подумайте о том, чтобы обратиться за консультацией к юристу, если у вас есть вопросы или опасения по поводу защиты ваших прав.
4. Помните о сторонних лицензиях: при использовании стороннего кода в вашем проекте помните о лицензиях используемых вами компонентов и убедитесь, что вы соблюдаете условия этих лицензий. Если вы распространяете программный продукт, будьте прозрачны в отношении используемых вами лицензий и включите необходимые уведомления и указания авторства.
5. Общайтесь со своим сообществом: если вы выпускаете программное обеспечение с открытым исходным кодом, взаимодействуйте со своим сообществом и открыто сообщайте о своих намерениях и целях. Поощряйте вклад и сотрудничество и помните о правах и интересах других участников.
6. Будьте в курсе последних событий. Лицензирование программного обеспечения и законодательство об интеллектуальной собственности могут со временем меняться, поэтому важно быть в курсе последних разработок и передового опыта в этой области. Рассмотрите возможность вступления в соответствующие организации и профессиональные сообщества и ищите ресурсы и информацию, которые помогут вам быть в курсе.

Таким образом, выбор правильной лицензии, понимание условий лицензии, защита вашей интеллектуальной собственности, соблюдение сторонних лицензий, общение с вашим сообществом и отслеживание последних событий — все это важные факторы, которые следует учитывать, когда речь идет о лицензировании и защите прав разработчиков программного обеспечения.

### 15.1.8 Интересный вопрос: кому принадлежат права на результат работы искусственного интеллекта?

Право собственности на результат работы искусственного интеллекта — сложный вопрос, и ответы на него могут быть разными в зависимости от юрисдикции и конкретных обстоятельств.

Как правило, создатель или владелец системы ИИ будет владеть правами на результат, созданный системой. Однако выходные данные также могут быть защищены авторским правом или другими формами защиты интеллектуальной собственности. В некоторых случаях результат может считаться работой по найму, а владелец системы ИИ будет считаться автором и владельцем авторских прав на результат.

Однако есть также случаи, когда результат системы ИИ не может считаться охраняемым произведением в соответствии с законом об авторском праве, например, когда он носит чисто фактический или функциональный характер.

Важно отметить, что законы и правила, касающиеся ИИ и прав интеллектуальной собственности, все еще развиваются и могут различаться в разных юрисдикциях. Если у вас есть конкретные вопросы

или опасения по поводу владения правами на результат работы ИИ, целесообразно обратиться за консультацией к юристу.

Во многих юрисдикциях законы, регулирующие ИИ и права интеллектуальной собственности, все еще находятся в процессе развития. Это может затруднить определение того, кому принадлежат права на результат работы ИИ.

В некоторых случаях может быть необходимо учитывать такие факторы, как степень человеческого творчества и авторства в создании выходных данных ИИ, или был ли результат создан системой ИИ автономно.

В некоторых случаях также может быть необходимо рассмотреть вопросы ответственности при возникновении проблем или ошибок в системе ИИ или ее выходных данных. Например, если система ИИ используется для получения юридических или медицинских рекомендаций, могут возникнуть вопросы о том, кто несет ответственность за ошибки или неточности в предоставленных рекомендациях.

В некоторых случаях можно договориться о лицензиях или других соглашениях, позволяющих использовать результаты ИИ в определенных обстоятельствах или для определенных целей.

## GitHub Pages

---

### 16.1 Github Pages



GitHub Pages — это служба веб-хостинга, предоставляемая GitHub, которая позволяет пользователям бесплатно размещать статические веб-сайты. Пользователи могут создать репозиторий на GitHub, содержащий файлы HTML, CSS и JavaScript своего веб-сайта, а затем GitHub Pages автоматически создаст и развернет веб-сайт на основе содержимого репозитория.

GitHub Pages поддерживает несколько генераторов статических сайтов, таких как Jekyll, Hugo и Gatsby, которые упрощают создание профессионально выглядящих веб-сайтов без необходимости написания всего кода HTML и CSS с нуля. Кроме того, GitHub Pages поддерживает пользовательские домены, поэтому пользователи могут использовать собственное доменное имя вместо домена `username.github.io` по умолчанию.

GitHub Pages также поддерживает управление версиями, что означает, что пользователи могут отслеживать изменения на своем веб-сайте с течением времени и в сотрудничестве с другими работать над одной и той же кодовой базой. Это делает его идеальным выбором для разработчиков, блоггеров и представителей малого бизнеса, которые хотят быстро и легко создать веб-сайт, не беспокоясь о настройке или обслуживании сервера.

**Как устроен сервис GitHub Pages?** GitHub Pages работает, размещая статические файлы на серверах GitHub и обслуживая их через Интернет. Вот общий обзор того, как это работает:

1. Создайте репозиторий GitHub. В первую очередь вам нужно создать новый репозиторий на GitHub или использовать существующий. Этот репозиторий будет содержать исходный код вашего веб-сайта, включая HTML, CSS, JavaScript и любые другие активы, которые вы хотите включить.
2. Выберите способ публикации. Вы можете выбрать один из двух способов публикации своего веб-сайта на страницах GitHub. Первый — создать отдельную ветку в вашем репозитории под названием `gh-pages`, где вы храните собранную версию своего веб-сайта. Второй способ — хранить файлы вашего веб-сайта в папке `docs` в основной ветке вашего репозитория.
3. Настройте конфигурацию вашего веб-сайта: вам необходимо создать файл конфигурации для вашего веб-сайта, который сообщает GitHub Pages, как создавать и обслуживать ваш веб-сайт. Файл конфигурации может включать такую информацию, как используемый вами генератор статических сайтов, URL-адрес вашего личного домена (если он у вас есть) и любые другие параметры, которые вы хотите настроить.
4. Создайте свой веб-сайт: с помощью предпочтаемого вами генератора статических сайтов вы можете создавать файлы HTML, CSS и JavaScript для своего веб-сайта. При необходимости вы можете добавить дополнительные функции, такие как пользовательские шрифты, изображения и другие ресурсы.
5. Отправьте внесенные изменения на GitHub: после того, как вы создали свой веб-сайт, вы можете отправить внесенные изменения в свой репозиторий GitHub. В зависимости от вашего метода публикации GitHub Pages автоматически создаст и развернет ваш веб-сайт на своих серверах.
6. Просмотрите свой веб-сайт: после того, как GitHub Pages завершит создание и развертывание вашего веб-сайта, вы сможете просмотреть его, посетив URL-адрес, предоставленный GitHub (например, <https://username.github.io/repo-name> или ваш личный домен, если у вас есть один).

В целом, GitHub Pages — это простой и удобный способ размещения статических веб-сайтов, и он может стать отличным выбором для разработчиков, блоггеров и других лиц, которые хотят быстро запустить веб-сайт, не беспокоясь о настройке или обслуживании сервера.

---

**Какова цель проекта GitHub Pages?** Цель GitHub Pages — предоставить пользователям простой и бесплатный способ размещения статических веб-сайтов непосредственно из их репозиториев GitHub. Проект предназначен для того, чтобы пользователи могли легко создавать и публиковать свои собственные веб-сайты и делиться ими, не беспокоясь о настройке сервера, настройке программного обеспечения для веб-хостинга или оплате услуг хостинга.

**GitHub Pages можно использовать для различных целей, например:**

- Личные веб-сайты и блоги. GitHub Pages можно использовать для создания личных веб-сайтов или блогов, где пользователи могут делиться информацией о себе, своих интересах или своей работе.
- Документация и учебные пособия. GitHub Pages можно использовать для размещения технической документации, учебных пособий или других типов учебных материалов.
- Веб-сайты проектов. GitHub Pages можно использовать для создания веб-сайтов для проектов с открытым исходным кодом, предоставляя информацию о проекте, его участниках и его функциях.
- Портфолио и резюме. GitHub Pages можно использовать для создания онлайн-портфолио или резюме, демонстрирующих навыки, опыт и достижения пользователя.
- Веб-сайты для малого бизнеса. GitHub Pages можно использовать для создания простых веб-сайтов для малого бизнеса, предоставляющих информацию о бизнесе, его продуктах или услугах, а также его контактную информацию.

В целом, цель сервиса GitHub Pages — упростить пользователям создание и совместное использование собственных веб-сайтов, будь то в личных, образовательных или профессиональных целях, без необходимости освоения передовых технических навыков или использования дорогостоящих услуг хостинга.

**Как программисты используют сервис GitHub Pages?** Программисты могут использовать GitHub Pages для создания и размещения статических веб-сайтов несколькими способами, в том числе:

1. Создание портфолио. Программисты могут использовать GitHub Pages для создания веб-сайта портфолио, демонстрирующего их проекты, навыки и опыт. Они могут использовать свой репозиторий GitHub для хранения исходного кода своего портфолио и использовать генератор статических сайтов, такой как Jekyll или Hugo, для создания веб-сайта.
2. Создание блога. Программисты могут использовать страницы GitHub для создания блога, где они делятся своими идеями, мнениями и знаниями с другими. Они могут использовать генератор статических сайтов, такой как Jekyll или Gatsby, для создания блога и сохранения сообщений в своем репозитории GitHub.
3. Размещение документации. Программисты могут использовать GitHub Pages для размещения технической документации по своим проектам, что упрощает поиск и использование документации другими пользователями. Они могут использовать генератор статических сайтов, такой как Sphinx или DocFX, для создания документации и сохранения ее в своем репозитории GitHub.
4. Обмен примерами кода. Программисты могут использовать GitHub Pages для обмена примерами кода или фрагментами кода, предоставляя полезный ресурс для тех, кто учится программировать. Они могут использовать генератор статических сайтов, такой как Jekyll или Gatsby, для создания веб-сайта и сохранения примеров кода в своем репозитории GitHub.
5. Создание веб-сайта проекта. Программисты могут использовать GitHub Pages для создания веб-сайта для своих проектов с открытым исходным кодом, предоставляя информацию о проекте, его участниках и его функциях. Они могут использовать генератор статических сайтов, такой как Jekyll или Gatsby, для создания веб-сайта и сохранения исходного кода в своем репозитории GitHub.
6. В целом, программисты могут использовать GitHub Pages для создания и размещения широкого спектра статических веб-сайтов, что делает его полезным инструментом для обмена информацией, продвижения своей работы и вовлечения в более широкое сообщество программистов.

---

**Как создать страницу портфолио на страницах GitHub?** Чтобы создать страницу портфолио на GitHub Pages, нужно выполнить следующие шаги:

1. Создайте новый репозиторий на GitHub: войдите в свою учетную запись GitHub и создайте новый репозиторий для страницы своего портфолио. Вы можете назвать его по аналогии с «username.github.io», где «username» — это ваше имя пользователя GitHub.
2. Создайте новую ветку или папку «docs»: в зависимости от ваших предпочтений, вы можете либо создать новую ветку с именем «gh-pages», либо создать папку «docs» в своей основной ветке, где будут храниться файлы вашего веб-сайта портфолио.
3. Выберите генератор статического сайта: вы можете использовать генератор статического сайта, такой как Jekyll, Hugo или Gatsby, для создания своего веб-сайта-портфолио. Эти генераторы позволяют создавать шаблоны для вашего веб-сайта и автоматически генерировать статические файлы HTML, CSS и JavaScript.
4. Создайте свой веб-сайт-портфолио. После того, как вы выбрали генератор статических сайтов, вы можете приступить к созданию своего веб-сайта-портфолио, настроив шаблоны и добавив свой собственный контент: ваша биография, проекты, навыки и опыт. Вы также можете добавлять изображения, видео и другие медиафайлы, чтобы сделать ваше портфолио более привлекательным.

5. Отправляйте изменения на GitHub. Создав веб-сайт портфолио, вы можете отправить изменения в репозиторий GitHub. В зависимости от вашего метода публикации, GitHub Pages автоматически создаст и развернет ваш веб-сайт на своих серверах.
6. Просмотр веб-сайта вашего портфолио: после того, как GitHub Pages завершит создание и развертывание веб-сайта вашего портфолио, вы сможете просмотреть его, посетив URL-адрес, предоставленный GitHub (например, <https://username.github.io>).

В целом, создание страницы портфолио на GitHub Pages включает в себя настройку репозитория, выбор генератора статических сайтов, создание веб-сайта и его публикацию на GitHub Pages. Используя правильные инструменты и немного творчества, вы можете создать профессионально выглядящее портфолио, демонстрирующее ваши навыки и опыт программиста.

Вот пример HTML-страницы для портфолио на GitHub Pages.

(код):

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>My Portfolio</title>
</head>
<body>
    <header>
        <h1>My Portfolio</h1>
        <nav>
            <ul>
                <li><a href="#about">About Me</a></li>
                <li><a href="#projects">Projects</a></li>
                <li><a href="#skills">Skills</a></li>
                <li><a href="#contact">Contact</a></li>
            </ul>
        </nav>
    </header>
    <main>
        <section id="about">
            <h2>About Me</h2>
            <p>Hi, I'm Ivan Ivanov, a web developer based in Moscow. I specialize in building responsive websites using HTML, CSS, and JavaScript.</p>
        </section>
        <section id="projects">
            <h2>Projects</h2>
            <ul>
                <li><a href="#">Project 1</a></li>
                <li><a href="#">Project 2</a></li>
                <li><a href="#">Project 3</a></li>
            </ul>
        </section>
        <section id="skills">
            <h2>Skills</h2>
            <ul>
                <li>HTML</li>
                <li>CSS</li>
            </ul>
        </section>
    </main>
</body>
```

(continues on next page)

(продолжение с предыдущей страницы)

```

<li>JavaScript</li>
<li>jQuery</li>
<li>Bootstrap</li>
</ul>
</section>
<section id="contact">
    <h2>Contact</h2>
    <form>
        <label for="name">Name:</label>
        <input type="text" id="name" name="name"><br>

        <label for="email">Email:</label>
        <input type="email" id="email" name="email"><br>

        <label for="message">Message:</label>
        <textarea id="message" name="message"></textarea><br>

        <button type="submit">Send</button>
    </form>
</section>
</main>
<footer>
    <p>Copyright © 2023 Ivan Ivanov</p>
</footer>
</body>
</html>

```

Этот пример HTML-страницы включает заголовок с меню навигации, основной раздел с информацией о владельце портфолио (в разделе «Обо мне»), раздел, посвященный прошлым проектам, раздел, в котором перечислены соответствующие навыки, и раздел контактов с формой для отправки сообщения. Нижний колонтикул содержит уведомление об авторских правах. Вы можете настроить этот пример кода в соответствии со своими потребностями и предпочтениями.

Вот пример HTML-страницы с CSS и JavaScript для портфолио на GitHub Pages:

(код):

```

<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>My Portfolio</title>
    <link rel="stylesheet" href="style.css">
</head>
<body>
    <header>
        <h1>My Portfolio</h1>
        <nav>
            <ul>
                <li><a href="#about">About Me</a></li>
                <li><a href="#projects">Projects</a></li>
                <li><a href="#skills">Skills</a></li>
            </ul>
        </nav>
    </header>
    <main>
        <div id="about">
            <h2>About Me</h2>
            <p>This is my portfolio website. I am a web developer with experience in ...
        </div>
        <div id="projects">
            <h2>Projects</h2>
            <ul>
                <li>Project 1</li>
                <li>Project 2</li>
                <li>Project 3</li>
            </ul>
        </div>
        <div id="skills">
            <h2>Skills</h2>
            <ul>
                <li>HTML5</li>
                <li>CSS3</li>
                <li>JavaScript</li>
            </ul>
        </div>
    </main>
</body>

```

(continues on next page)

(продолжение с предыдущей страницы)

```
<li><a href="#contact">Contact</a></li>
</ul>
</nav>
</header>
<main>
    <section id="about">
        <h2>About Me</h2>
        
        <p>Hi, I'm John Doe, a web developer based in San Francisco. I specialize in building responsive websites using HTML, CSS, and JavaScript.</p>
    </section>
    <section id="projects">
        <h2>Projects</h2>
        <div class="project">
            
            <h3>Project 1</h3>
            <p>Lorem ipsum dolor sit amet, consectetur adipiscing elit. Nulla vitae felis quis diam suscipit rhoncus. Ut dapibus purus in massa maximus gravida. Fusce ut nunc vitae nulla tincidunt volutpat sed ut mauris.</p>
            <a href="#">View Project</a>
        </div>
        <div class="project">
            
            <h3>Project 2</h3>
            <p>Lorem ipsum dolor sit amet, consectetur adipiscing elit. Nulla vitae felis quis diam suscipit rhoncus. Ut dapibus purus in massa maximus gravida. Fusce ut nunc vitae nulla tincidunt volutpat sed ut mauris.</p>
            <a href="#">View Project</a>
        </div>
        <div class="project">
            
            <h3>Project 3</h3>
            <p>Lorem ipsum dolor sit amet, consectetur adipiscing elit. Nulla vitae felis quis diam suscipit rhoncus. Ut dapibus purus in massa maximus gravida. Fusce ut nunc vitae nulla tincidunt volutpat sed ut mauris.</p>
            <a href="#">View Project</a>
        </div>
    </section>
    <section id="skills">
        <h2>Skills</h2>
        <ul>
            <li>HTML</li>
            <li>CSS</li>
            <li>JavaScript</li>
            <li>jQuery</li>
            <li>Bootstrap</li>
        </ul>
    </section>
    <section id="contact">
        <h2>Contact</h2>
        <form>
            <label for="name">Name:</label>
```

(continues on next page)

(продолжение с предыдущей страницы)

```

<input type="text" id="name" name="name"><br>

<label for="email">Email:</label>
<input type="email" id="email" name="email"><br>

<label for="message">Message:</label>
<textarea id="message" name="message"></textarea><br>

        <button type="submit">Send</button>
    </form>
</section>
</main>
<footer>
    <p>Copyright © 2023 Ivan Ivanov</p>
</footer>
</body>
</html>S

```

### 16.1.1 С какой целью используют сервис GitHub Pages?

1. Демонстрация своих проектов: разработчики и дизайнеры могут использовать GitHub Pages для демонстрации своей работы и проектов профессионально и легко доступным образом. Это может быть отличным способом привлечь потенциальных клиентов, сотрудников или работодателей.
2. Размещение личных веб-сайтов/блогов: GitHub Pages также можно использовать для размещения личных веб-сайтов или блогов, позволяя людям делиться своими мыслями, опытом и знаниями с другими.
3. Документация: разработчики могут использовать GitHub Pages для размещения документации по своим проектам, чтобы другим было проще понять их работу и внести свой вклад в нее.
4. Что касается поиска работы или интересных проектов, GitHub Pages, безусловно, также может помочь. Наличие хорошо разработанного и профессионально выглядящего веб-сайта портфолио может произвести отличное первое впечатление на потенциальных работодателей или сотрудников. Кроме того, GitHub Pages позволяют разработчикам демонстрировать свои навыки и вклад в проекты с открытым исходным кодом, что может помочь привлечь внимание потенциальных работодателей или сотрудников.

### 16.1.2 Преимущества и недостатки сервиса GitHub Pages

#### Преимущества:

1. Бесплатный хостинг: GitHub Pages предлагает бесплатный хостинг для статических веб-сайтов, что делает его доступным вариантом для частных лиц или малого бизнеса.
2. Простота настройки: создать веб-сайт GitHub Pages относительно легко и просто, особенно для тех, кто уже знаком с Git и GitHub.
3. Пользовательские домены: GitHub Pages позволяет пользователям использовать свое собственное доменное имя, что может помочь сделать их веб-сайт более профессиональным.
4. Совместная работа: GitHub Pages построен на основе Git и GitHub, что упрощает совместную работу нескольких людей над веб-сайтом.

5. Интеграция с другими сервисами: GitHub Pages можно легко интегрировать с другими сервисами, такими как Jekyll, популярный генератор статических сайтов.

**Недостатки:**

1. Ограниченнная функциональность: GitHub Pages предназначен только для размещения статических веб-сайтов, что означает, что он может не подходить для более сложных веб-сайтов или веб-приложений.
2. Ограниченный контроль: поскольку GitHub Pages является службой размещения, пользователи имеют ограниченный контроль над серверной средой и конфигурацией.
3. Ограничение хранилища: GitHub Pages имеет ограничение хранилища в 1 ГБ на учетную запись, чего может быть недостаточно для крупных веб-сайтов.
4. Кривая обучения: несмотря на то, что GitHub Pages относительно прост в настройке, для пользователей, незнакомых с Git и GitHub, все же может потребоваться кривая обучения.
5. Ограничена поддержка: поскольку GitHub Pages является бесплатным сервисом, для пользователей, сталкивающихся с проблемами или проблемами, доступна только ограниченная поддержка.

## 16.2 Задачи

Составьте страницу учебного проекта чат-бота «EducationBot» на GitPage, который позволяет получать информацию об олимпиадах, конкурсах и событиях в мире образования. Ниже представлен шаблон страницы в формате Markdown. Стили и графическое оформление выберите на свое усмотрение.

Добро пожаловать на официальную страницу проекта EducationBot! Наш чат-бот создан с целью помочь вам получать актуальную информацию о различных образовательных событиях, олимпиадах, конкурсах и мероприятиях в образовательной сфере.

### ### Как пользоваться EducationBot?

EducationBot предоставляет простой и интуитивно понятный интерфейс для получения информации. Просто напишите боту интересующий вас вопрос или выберите одну из предложенных категорий:

- Олимпиады
- Конкурсы
- Мероприятия
- Анонсы

Выберите нужную категорию и получите актуальные новости и события из образовательного мира.

### ### Основные функции:

- Получение информации об олимпиадах, конкурсах, мероприятиях и анонсах в мире образования
- Актуальные новости и события в реальном времени
- Поддержка различных категорий и фильтров для удобного поиска информации

(continues on next page)

(продолжение с предыдущей страницы)

### ### 0 проекте:

EducationBot разрабатывается как pet-project учебного проекта. Целью проекта является изучение и применение основ программирования на языке Python, работы с API и создание простого, но полезного чат-бота для образовательного сообщества.

### ### Как связаться с нами?

Если у вас есть вопросы или предложения по улучшению EducationBot, не стесняйтесь связаться с нами:

- Email: educationbot@example.com
- Telegram: @EducationBotSupport

### ### Статус проекта:

EducationBot находится в стадии разработки MVP (Minimum Viable Product). Мы стремимся создать функциональный и удобный чат-бот для пользователей и всегда открыты для обратной связи и предложений.

### ### Лицензия:

Проект EducationBot распространяется по лицензии MIT. Вы можете свободно использовать и изменять код проекта с указанием авторства.

### ### Установка и запуск:

Если вы хотите развернуть EducationBot на своем сервере, следуйте инструкциям в разделе [Installation Guide] (/docs/installation.md).

### ### Помощь в разработке:

Если вы хотите внести свой вклад в развитие проекта, ознакомьтесь с разделом [Contribution Guidelines] (/docs/contributing.md) и присоединяйтесь к нашей команде разработчиков.

Мы надеемся, что EducationBot будет полезен для всех, кто интересуется образованием и стремится быть в курсе всех актуальных событий. Благодарим вас за поддержку нашего проекта!

## 16.3 Вопросы

1. Что такое GitPages и какова его роль в системах контроля версий?
2. Как создать GitPages?
3. В чем разница между GitPages и другими популярными платформами для размещения веб-сайтов?
4. Как развернуть статический веб-сайт с использованием GitPages?
5. Можно ли иметь несколько GitPages для разных веток в одном репозитории? Если да, как это настроить?
6. Какой максимальный размер репозитория для GitPages?
7. Как настроить пользовательскую тему для вашего веб-сайта GitPages?
8. Опишите процесс обновления вашего веб-сайта GitPages после внесения изменений в репозиторий.
9. Можно ли размещать веб-сайт с динамическим контентом с использованием GitPages? Почему да или почему нет?
10. Как ограничить доступ к вашему веб-сайту GitPages и разрешить просмотр только определенным пользователям?
11. Что такое тема Jekyll и как ее использовать для настройки внешнего вида вашего веб-сайта GitPages?
12. Как включить HTTPS для вашего веб-сайта GitPages?
13. Можно ли использовать GitPages для размещения частных репозиториев? Почему да или почему нет?
14. Как включить непрерывное развертывание с помощью GitPages?
15. Опишите процесс настройки пользовательской страницы ошибки 404 для вашего веб-сайта GitPages.

Раздел 3. Инструменты разработки

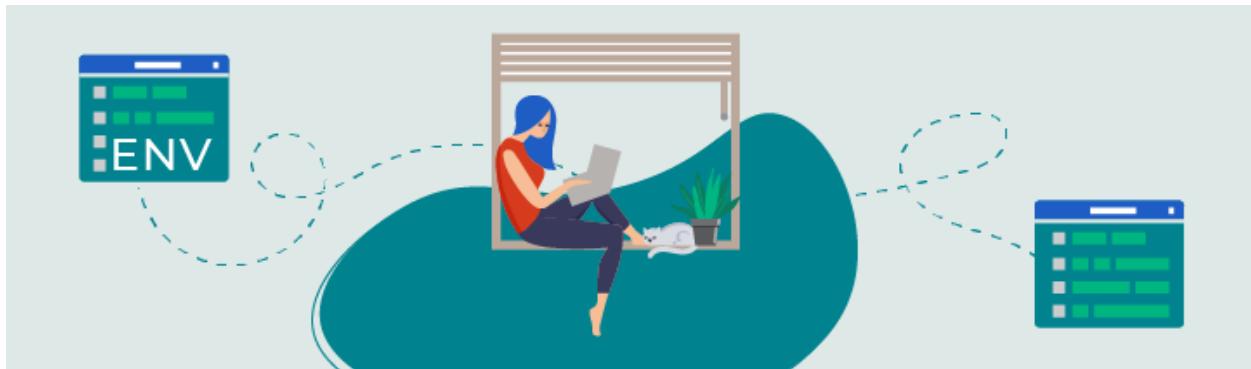
---



## Виртуальное окружение

### 18.1 Знакомство с рабочим окружением

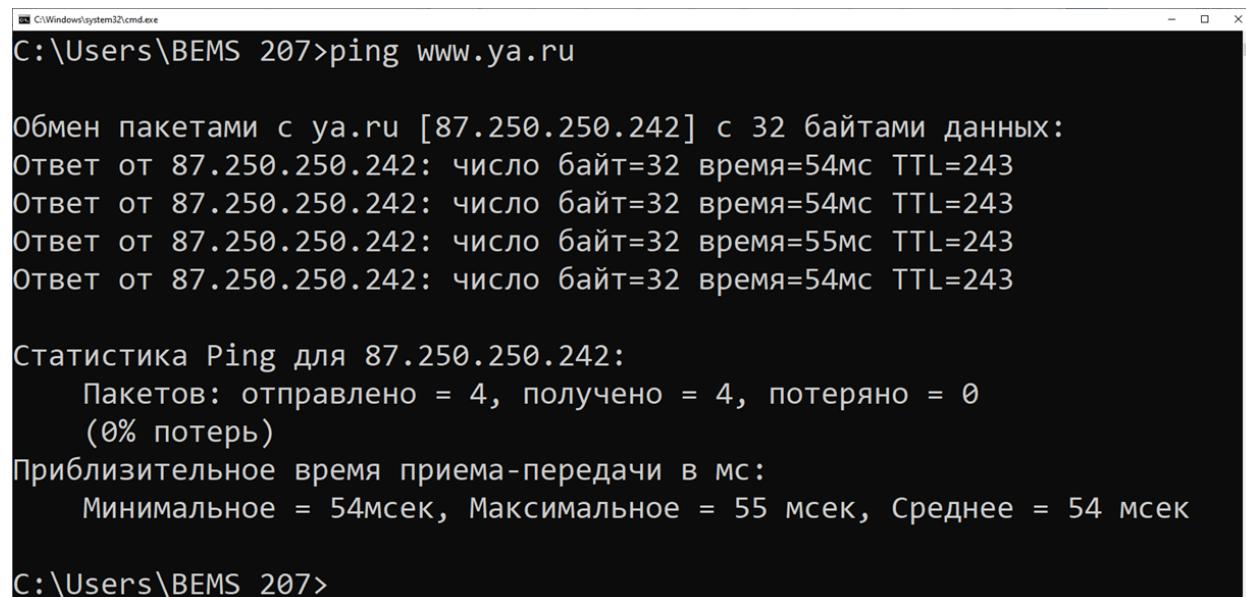
*Path — пути к системному окружению. Создание виртуального окружения *venv* для проекта.*



Знакомимся с темой рабочего окружения. Что это такое? Начнем с рабочего окружения Windows и перейдем к созданию отдельных окружений для проектов Python. Научимся переключаться между локальным и глобальным окружениями. Разберемся почему это важно, а также узнаем, как легко и просто поделиться проектом с другими разработчиками, используя виртуальное рабочее окружение, и как это помогает экономить трафик и управлять размером проекта.

После запуска операционной системы пользователю доступны различные инструменты, как системные (входящие в состав ОС), так и сторонние (установленные дополнительно). Часто в процессе работы возникает необходимость использования командной строки для вызова различных утилит, помогающих в процессе работы. Например, в случае, когда браузеру не удается открыть страницу удаленного веб-сервиса: зайти на почту или открыть новостную ленту, мы можем наблюдать следующую ошибку — «404 NotFound». Чтобы определить, на чьей стороне ошибка: вашей сети или удаленного сервера, часто используют системные утилиты, консоли для работы с сетью. Такие как PING, IPCONFIG, ARP, NETSTAT и т.д. Например, вызовем утилиту ping с параметром «www.ya.ru», чтобы проверить

наличие связи с серверами. Если в окно терминала в Windows ввести команду «explorer», откроется графическое окно.



C:\Users\BEMS 207>ping www.ya.ru

Обмен пакетами с ya.ru [87.250.250.242] с 32 байтами данных:

Ответ от 87.250.250.242: число байт=32 время=54мс TTL=243  
Ответ от 87.250.250.242: число байт=32 время=54мс TTL=243  
Ответ от 87.250.250.242: число байт=32 время=55мс TTL=243  
Ответ от 87.250.250.242: число байт=32 время=54мс TTL=243

Статистика Ping для 87.250.250.242:

Пакетов: отправлено = 4, получено = 4, потеряно = 0  
(0% потеря)

Приблизительное время приема-передачи в мс:

Минимальное = 54мсек, Максимальное = 55 мсек, Среднее = 54 мсек

C:\Users\BEMS 207>

Рисунок 1 — Терминал с результатом работы команды ping.

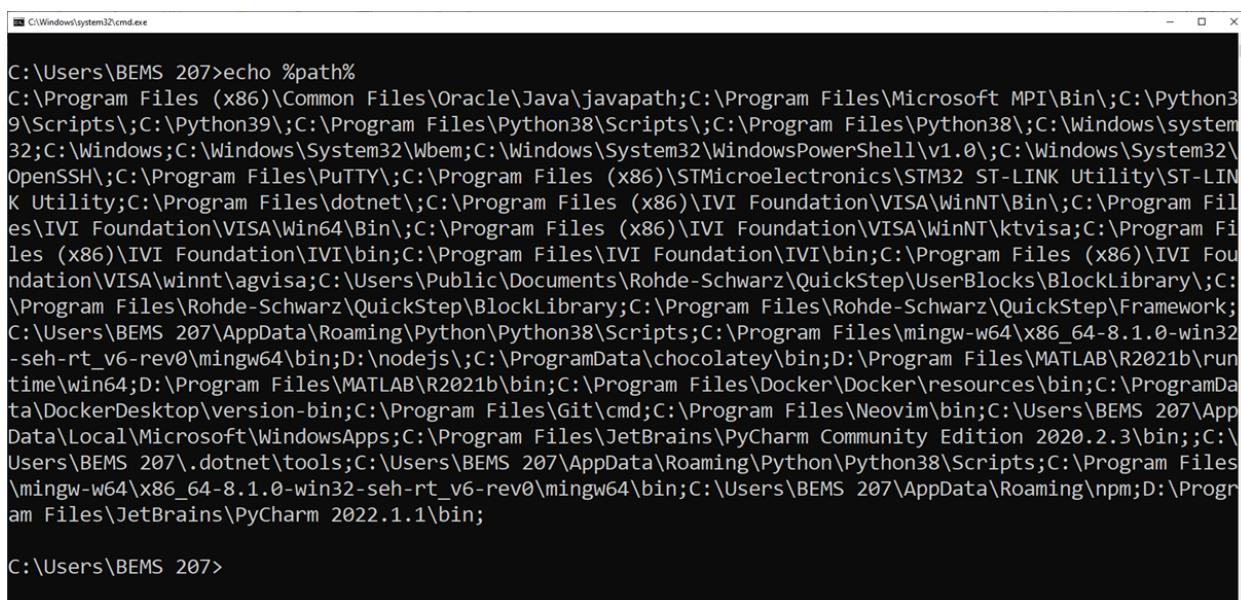
**Как это работает?** Дело в том, что операционная система (ОС) ищет введенную команду в рабочем окружении системы. В случае ОС Windows это объект PATH. В этом объекте хранятся все пути, по которым выполняется поиск исполняемых файлов. Чтобы посмотреть содержание PATH, достаточно ввести в окно командной строки:

```
echo %PATH%
```

На рисунке 2 видно, что в окне терминала отобразилось множество путей к различным программам, не входящим в состав ОС Windows. Все они разделены символом точки с запятой «;». Одним из таких программ является интерпретатор языка программирования Python.

```
C:\Python39\Scripts\; C:\Python39\;
```

Таким образом, интерпретатор команд консоли Windows «понимает», по какому пути ему обратиться в случае ввода в окно терминала python. Во время установки Python предлагает установить галочку «AddPythonToPATH» (Рисунок 3).



```
C:\Users\BEMS 207>echo %path%
C:\Program Files (x86)\Common Files\Oracle\Java\javapath;C:\Program Files\Microsoft MPI\Bin\;C:\Python39\Scripts\;C:\Python39\;C:\Program Files\Python38\Scripts\;C:\Program Files\Python38\;C:\Windows\system32\;C:\Windows;C:\Windows\System32\Wbem;C:\Windows\System32\WindowsPowerShell\v1.0\;C:\Windows\System32\OpenSSH\;C:\Program Files\PuTTY\;C:\Program Files (x86)\STMicroelectronics\STM32 ST-LINK Utility\ST-LINK Utility;C:\Program Files\dotnet\;C:\Program Files (x86)\IVI Foundation\VISA\WinNT\Bin\;C:\Program Files\IVI Foundation\VISA\Win64\Bin\;C:\Program Files (x86)\IVI Foundation\VISA\WinNT\ktvisa;C:\Program Files (x86)\IVI Foundation\IVI\bin;C:\Program Files\IVI\bin;C:\Program Files (x86)\IVI Foundation\VISA\winnt\agvisa;C:\Users\Public\Documents\Rohde-Schwarz\QuickStep\UserBlocks\BlockLibrary\;C:\Program Files\Rohde-Schwarz\QuickStep\BlockLibrary;C:\Program Files\Rohde-Schwarz\QuickStep\Framework;C:\Users\BEMS 207\AppData\Roaming\Python\Python38\Scripts;C:\Program Files\mingw-w64\x86_64-8.1.0-win32-seh-rt_v6-rev0\mingw64\bin;D:\nodejs\;C:\ProgramData\chocolatey\bin;D:\Program Files\MATLAB\R2021b\runtime\win64\;D:\Program Files\MATLAB\R2021b\bin;C:\Program Files\Docker\Docker\resources\bin;C:\ProgramData\DockerDesktop\version-bin;C:\Program Files\Git\cmd;C:\Program Files\Neovim\bin;C:\Users\BEMS 207\AppData\Local\Microsoft\WindowsApps;C:\Program Files\JetBrains\PyCharm Community Edition 2020.2.3\bin\;;C:\Users\BEMS 207\.dotnet\tools;C:\Users\BEMS 207\AppData\Roaming\Python\Python38\Scripts;C:\Program Files\mingw-w64\x86_64-8.1.0-win32-seh-rt_v6-rev0\mingw64\bin;C:\Users\BEMS 207\AppData\Roaming\npm;D:\Program Files\JetBrains\PyCharm 2022.1.1\bin;

C:\Users\BEMS 207>
```

Рисунок 2 — Терминал с результатом работы команды echo %PATH%

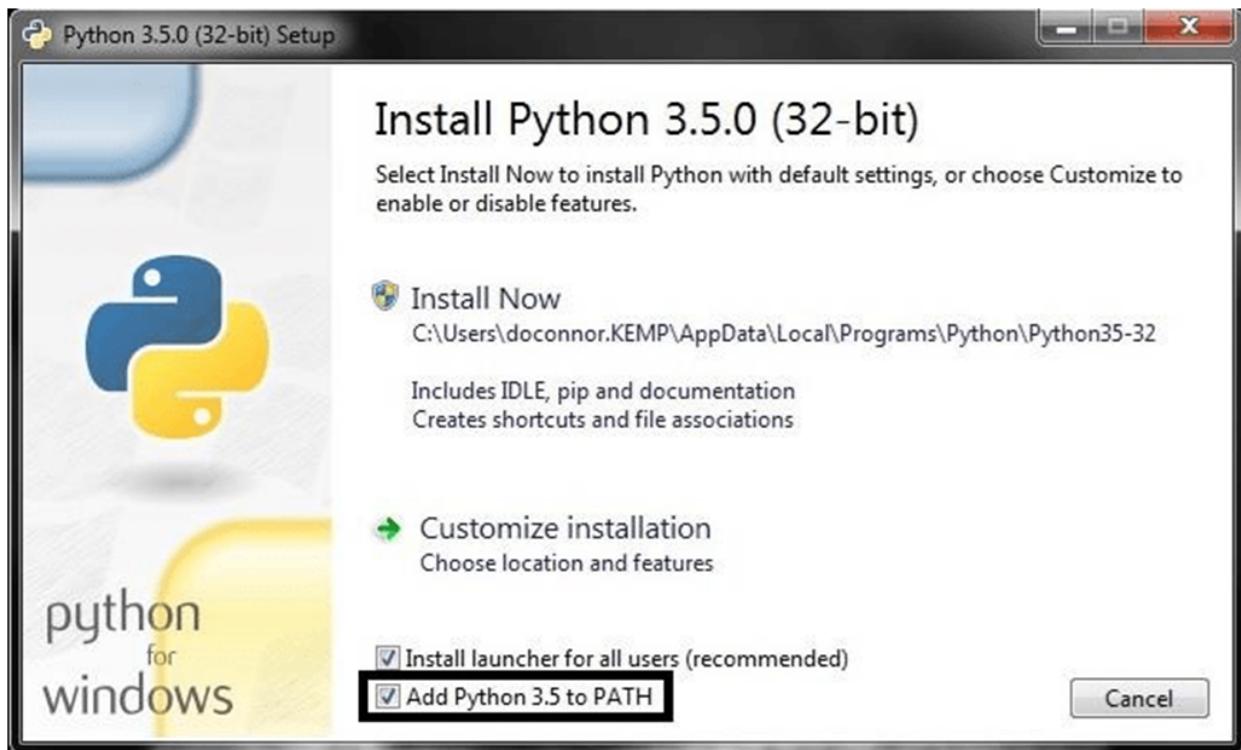


Рисунок 3 — Окно установки Python с опцией записи в системное окружение PATH

### 18.1.1 Добавить директорию в PATH

В Windows 10 можно добраться до настройки PATH следующим образом:

Мой компьютер → Свойства → Дополнительные параметры системы → Дополнительно → Переменные среды.

Или вызовом «Изменение переменных среды текущего пользователя» в результатах поиска. В оконке «Переменные среды» в блоке «Переменные среды пользователя %USERNAME%» находим строку PATH, выделяем кликом, нажимаем кнопку «Изменить...» и в появившемся оконке нажимаем «Создать» для добавления еще одного элемента.

Как посмотреть содержимое переменной окружения PATH в Linux, внести в нее временные и постоянные изменения, просматривать содержимое всех переменных окружения одновременно?

Посмотреть содержание PATH

```
echo $PATH  
/usr/local/bin:/usr/bin:/bin:/usr/local/games:/usr/games
```

Временно добавить директорию в PATH Добавить /home/andrej/.local/bin в PATH

```
export PATH="/home/andrej/.local/bin:$PATH"
```

Проверка:

```
echo $PATH  
/home/andrej/.local/bin:/usr/local/bin:/usr/bin:/bin:/usr/local/games:/usr/game
```

Добавить директорию в PATH через .bashrc Чтобы добавить директорию не постоянной основе, пропишите ее в .bashrc. Перейдите в домашнюю директорию, выполнив:

```
cd ~
```

Там, скорее всего, находится файл .bashrc. Отредактируйте его в любом текстовом редакторе:

```
vi .bashrc  
PATH="/your_folder:${PATH}"  
export PATH
```

Обратите внимание, что после PATH нет пробела, как и после знака равенства. \${PATH} означает, что вы добавляете директорию *your\_folder* к уже существующему PATH, а не переписываете его с нуля.

**Посмотреть содержимое всех переменных окружения** Чтобы изучить не только PATH, а вообще все переменные окружения, выполните:

```
env
```

```
LC_PAPER=en_US.UTF-8 XDG_SESSION_ID=2 LC_ADDRESS=en_US.UTF-8 HOSTNAME=localhost.  
↳localdomainLC_MONETARY=en_US.UTF-8 SELINUX_ROLE_REQUESTED= TERM=xterm-256colorSHELL=/  
↳bin/bashHISTSIZE=1000 SSH_CLIENT=192.168.56.1 41522 22 SELINUX_USE_CURRENT_RANGE= LC_  
↳NUMERIC=en_US.UTF-8 SSH_TTY=/dev/pts/1 USER=andrej LS_COLORS=rs=0:di=38;5;27:ln=38;5;  
↳51: mh=44;38;5;15: pi=40;38;5;11: so=38;5;13: do=38;5;5: bd=48;5;232;38;5;11: cd=48;5;232;38;  
↳5;3: or=48;5;232;38;5;9: mi=05;48;5;232;38;5;15: su=48;5;196;38;5;15: sg=48;5;11;38;5;  
↳16: ca=48;5;196;38;5;226: tw=48;5;10;38;5;16: ow=48;5;10;38;5;21: st=48;5;21;38;5;15: ex=38;  
↳5;34: *.tar=38;5;9: *.tgz=38;5;9: *.arc=38;5;9: *.arj=38;5;9: *.taz=38;5;9: *.lha=38;5;9: *.  
↳lz=38;5;9: *.lzh=38;5;9: *.lzma=38;5;9: *.tlz=38;5;9: *.txz=38;5;9: *.tzo=38;5;9: *.t7z=38;  
↳(continues on next page)
```

(продолжение с предыдущей страницы)

```

→5;9::* .zip=38;5;9::* .z=38;5;9::* .Z=38;5;9::* .dz=38;5;9::* .gz=38;5;9::* .lrz=38;5;9::* .lz=38;5;
→9::* .lzo=38;5;9::* .xz=38;5;9::* .bz2=38;5;9::* .bz=38;5;9::* .tbz=38;5;9::* .tbz2=38;5;9::* .tz=38;
→5;9::* .deb=38;5;9::* .rpm=38;5;9::* .jar=38;5;9::* .war=38;5;9::* .ear=38;5;9::* .sar=38;5;9::*;
→rar=38;5;9::* .alz=38;5;9::* .ace=38;5;9::* .zoo=38;5;9::* .cpio=38;5;9::* .7z=38;5;9::* .rz=38;5;
→9::* .cab=38;5;9::* .jpg=38;5;13::* .jpeg=38;5;13::* .gif=38;5;13::* .bmp=38;5;13::* .pbm=38;5;
→13::* .pgm=38;5;13::* .ppm=38;5;13::* .tga=38;5;13::* .xbm=38;5;13::* .xpm=38;5;13::* .tif=38;5;
→13::* .tiff=38;5;13::* .png=38;5;13::* .svg=38;5;13::* .svgz=38;5;13::* .mng=38;5;13::* .pcx=38;5;
→13::* .mov=38;5;13::* .mpg=38;5;13::* .mpeg=38;5;13::* .m2v=38;5;13::* .mkv=38;5;13::* .webm=38;5;
→13::* .ogm=38;5;13::* .mp4=38;5;13::* .m4v=38;5;13::* .mp4v=38;5;13::* .vob=38;5;13::* .qt=38;5;
→13::* .nuv=38;5;13::* .wmv=38;5;13::* .asf=38;5;13::* .rm=38;5;13::* .rmvb=38;5;13::* .flc=38;5;
→13::* .avi=38;5;13::* .fli=38;5;13::* .flv=38;5;13::* .gl=38;5;13::* .dl=38;5;13::* .xcf=38;5;13::*;
→xwd=38;5;13::* .yuv=38;5;13::* .cgm=38;5;13::* .emf=38;5;13::* .axv=38;5;13::* .anx=38;5;13::*;
→ogv=38;5;13::* .ogx=38;5;13::* .aac=38;5;45::* .au=38;5;45::* .flac=38;5;45::* .mid=38;5;45::*;
→midi=38;5;45::* .mka=38;5;45::* .mp3=38;5;45::* .mpc=38;5;45::* .ogg=38;5;45::* .ra=38;5;45::*;
→wav=38;5;45::* .axa=38;5;45::* .oga=38;5;45::* .spx=38;5;45::* .xspf=38;5;45: LC_TELEPHONE=en_
→US.UTF-8 MAIL=/var/spool/mail/andrei PATH=/usr/local/bin:/usr/bin:/usr/local/sbin:/usr/
→sbin:/home/andrei/.local/bin:/home/andrei/bin LC_IDENTIFICATION=en_US.UTF-8 PWD=/home/
→andreiLANG=en_US.UTF-8 LC_MEASUREMENT=en_US.UTF-8 SELINUX_LEVEL_REQUESTED=_u
→HISTCONTROL=ignoredupsSHLVL=1 HOME=/home/andreiLOGNAME=andrei XDG_DATA_DIRS=/home/
→andrei/.local/share/flatpak(exports/share:/var/lib/flatpak(exports/share:/usr/local/
→share:/usr/share SSH_CONNECTION=192.168.56.1 41522 192.168.56.116 22 LESSOPEN=|| /usr/
→bin/lesspipe.sh %s XDG_RUNTIME_DIR=/run/user/1000 LC_TIME=en_US.UTF-8 LC_NAME=en_US.
→UTF-8_= /usr/bin/env

```

## 18.1.2 Знакомство с virtualenv

**virtualenv** — это инструмент, используемый для создания изолированного рабочего пространства для приложения Python. Он дает некоторые преимущества: например, возможность локальной установки модулей, экспорта рабочей среды и выполнения программы Python внутри этого окружения. Виртуальное окружение (virtual environment) — это каталог, в который устанавливаются некоторые исполняемые файлы и скрипты. Среди файлов есть `python` для выполнения скриптов и `pip` для установки других модулей в окружении. По сути, виртуальное окружение имитирует полную системную установку Python и всех необходимых модулей, не вмешиваясь в работу системы, на которой будет запускаться приложение.

В 2017 году был опубликован `Pipenv`, который управляет всеми вышеперечисленными инструментами: виртуальными окружениями интерпретаторов Python, зависимостями пакетов, их активацией и блокировкой версий в `Pipfile`.

### Установка

Python 3.3+ поставляется с модулем `venv`. Для более старых версий Python можно использовать сторонний инструмент `virtualenv`. Пакеты Установите один из следующих пакетов: Python 3.3+: `python` Python 3: `python-virtualenv` Для `Pipenv`: Python 3: `python-pipenv` Все три инструмента похожи.

### Создание

Используйте `venv` или `virtualenv` для создания виртуального окружения в каталоге вашего проекта. Не забудьте исключить каталог `venv` из вашей системы контроля версий, для его восстановления достаточно копии `pip freeze`.

Он входит в состав `python` (3.3+):

```
$ python -m venv env name
```

Доступен в пакете python-virtualenv.

```
$ virtualenv envname
```

### Активация

Для активации виртуального окружения используйте один из имеющихся скриптов для вашей командной оболочки. Пример для bash:

```
$ source env_name/bin/activate  
(envname) $
```

Теперь команды python и pip будут запускаться и управлять пакетами только внутри виртуального окружения, не затрагивая систему. Для выхода из виртуального окружения выполните функцию, которую создал скрипт активации: (envname) \$ deactivate.

### Версии Python

По умолчанию виртуальные окружения создаются с использованием стандартного системного Python. Файл bin/python — это просто символическая ссылка на системный python:

```
$ ls -l envname/bin/python  
lrwxrwxrwx 1 foo foo 15 янв 29 18:48 envname/bin/python -> /usr/bin/python
```

Если вы хотите использовать другую версию Python внутри виртуального окружения, можно использовать опцию -p/--python у virtualenv:

```
$ virtualenv -p 3.8 envname  
$ ls -l envname/bin/python  
lrwxrwxrwx 1 foo foo 18 янв 29 18:48 envname/bin/python -> /usr/bin/python3.8
```

Также можно использовать руру3:

```
$ virtualenv -p ruру3 env name
```

### Использование

Все команды можно выполнять в папке проекта, и pipenv распознает текущую ситуацию, найдет виртуальное окружение в текущем каталоге и будет использовать его. В этом разделе будут показаны установка, настройка, а также проведен полный обзор всех возможностей использования утилит venv и virtualenv для Python. Но начнем мы с рассмотрения общих вопросов.

В процессе разработки приложений на языке программирования Python может возникнуть ряд проблем, которые связаны с применением различных версий библиотек. Например, для работы двух различных приложений может потребоваться одна и та же библиотека, но разных версий. Или же нужно обеспечить корректную работу приложения вне зависимости от обновления версии библиотеки, которую оно использует.

Чтобы решать подобные проблемы, разработчики придумали интересный способ: запускать каждое приложение со своим набором версий библиотек в изолированных средах. Таким образом, виртуальная среда (*virtual environment*) в языке Python позволяет управлять проектами изолированно друг от друга, то есть:

- у каждого проекта могут быть свои зависимости;
- зависимости одного проекта не влияют на зависимости другого проекта.

Создавать виртуальное окружение в Python можно при помощи различных утилит. Поговорим про `venv` и `virtualenv`.

Важно: перед началом работы у вас должен быть установлен Python 3 и pip. Если вы используете Windows, все следующие команды, кроме тех, где будет альтернативный вариант, выполняются таким же образом, но необходимо заменять `pip3` на `pip`, а `python3` на Python.

## Работа с `venv`

Чтобы создать виртуальную среду, мы используем модуль `venv`. Это удобный инструмент для управления зависимостями и изоляции проектов, который выстроен в Python, начиная с версии Python 3.3+. Поэтому модуль `venv` нельзя использовать для изоляции программ Python 2.x.

### Создание проекта

Приступим к созданию проекта. Первым делом следует создать папку проекта и перейти в нее, в Linux это можно сделать при помощи следующих команд:

```
$ mkdir my_python_project
cd my_python_project
```

Замените `my_python_project` на желаемое название проекта.

### Создание виртуальной среды

Теперь создадим в этой папке виртуальную среду:

```
$ python3 -m venv env
```

Замените `project_name_myproject` на желаемое название виртуальной среды. Если у вас возникает следующая ошибка:

```
The virtual environment was not created successfully because ensurepip is not available. On Debian/Ubuntu systems, you need to install the python3-venv package using the following command.
```

Это говорит, что в вашей системе отсутствует пакет `python3-venv`, и вам следует его установить. Команда представлена ниже:

```
apt-get install python3-venv
```

```
You may need to use sudo with that command. After installing the python3-venv package, recreate your virtual environment.
Failing command: ['/home/osboxes/my_python_project/project_name_myproject/bin/python3',
← '-Im', 'ensurepip', '--upgrade']
```

Вам необходимо установить `venv`. Например, в Ubuntu для этого требуется выполнить следующую команду:

```
$ sudo apt-get install python3-venv-y
```

После повторите команду создания виртуальной среды Python, показанную в начале этого раздела.

### Активация виртуальной среды

Осталось активировать виртуальную среду при помощи команды:

```
$ source project_name_myproject/bin/activate
```

Замените `project_name_myproject` на название виртуальной среды, созданной на прошлом шаге. Если вы используете Windows, и у вас правильно установлен Python и pip, то вы можете активировать виртуальную среду следующей командой:

```
$ project_name_myproject/Scripts/activate
```

Замените `project_name_myproject` на название виртуальной среды, созданной на прошлом шаге.

Проверка установленных пакетов Теперь проверим установленные пакеты pip внутри виртуальной среды:

```
$ pip3list
```

Вывод должен быть примерно такой:

```
Package Version
-----
pip 20.3.4
pkg-resources 0.0.0
setuptools 44.1.1
```

Как мы видим, внутри виртуальной среды у нас нет никаких сторонних установленных библиотек с pip. Можно установить какой-либо пакет, например, модуль `requests`:

```
$ pip3 install requests
```

Теперь снова проверим список установленных пакетов pip внутри виртуальной среды:

```
$ pip list
```

В выводе должен появиться наш только что установленный пакет:

```
Package Version
-----
certifi 2021.5.30
chardet 4.0.0
idna 2.10
pip 20.3.4
pkg-resources 0.0.0
requests 2.25.1
setuptools 44.1.1
urllib3 1.26.5
```

### Выход из виртуальной среды

Для выхода из виртуальной среды необходимо выполнить команду: `deactivate`. Если после выхода снова проверить список установленных пакетов, то мы получим совсем другой вывод:

```
Package Version
-----
apturl 0.5.2
bcrypt 3.1.7
blinker 1.4
```

(continues on next page)

(продолжение с предыдущей страницы)

```
Brlapi 0.8.2
certifi 2020.6.20
chardet 4.0.0
chrome-gnome-shell 0.0.0
click 7.1.2
colorama 0.4.4
command-not-found 0.3
cryptography 3.3.2
```

## Перенос установленных пакетов

Для переноса установленных пакетов в другую среду на Linux можно использовать простую команду:

```
$ python3-m pip freeze > requirements.txt
```

Она создаст файл requirements.txt, в котором будет находиться список всех установленных пакетов. Импорт пакетов в другую среду. Теперь можно перейти в другую виртуальную среду и импортировать пакеты следующей командой:

```
$ pip3 install -r requirements.txt
```

## Работа с virtualenv

Утилита virtualenv — это наиболее популярный инструмент, который используется для создания изолированных сред в Python. Он создает папку, содержащую все необходимые исполняемые файлы для использования пакетов, которые потребуются проекту Python. В отличие от модуля venv, эту утилиту можно использовать как в Python 3, так и в версии 2.7.

### Установка virtualenv и создание виртуальной среды

Чтобы установить утилиту virtualenv, используйте pip:

```
$ pip install virtualenv
```

### Создание виртуальной среды virtualenv происходит так

```
$ cd project_folder
$ virtualenv myproject
```

virtualenv myproject создаст папку в текущей директории. В ней будут находиться исполняемые файлы Python и копия библиотеки pip, которую можно использовать для установки других пакетов. Имя виртуальной среды (в данном случае myproject) может быть любым. Если имя отсутствует, файлы будут помещены в текущий каталог. Таким образом, создается копия Python в любом каталоге, в котором вы запускали команду, помещая ее в папку с именем myproject. Вы также можете использовать интерпретатор Python по выбору. Пример использования Python 2.7:

```
$ virtualenv-p /usr/bin/python2.7 myproject
```

Для версии Python 3.7:

```
$ virtualenv-p /usr/bin/python3.7 myproject
```

Проверить версию Python и pip можно с помощью следующих команд:

```
$ python--version  
$ pip--version
```

Для выхода из виртуальной среды необходимо выполнить команду:

```
$ deactivate
```

### Копирование виртуальной среды

Чтобы копировать виртуальные среды, введите:

```
$ cpvirtualenv new-project new-project-copy  
Copying new-project as new-project-copy...  
$ workon  
new-project  
new-project-copy
```

### Удаление виртуальной среды

Удалять виртуальные среды можно при помощи команды:

```
$ rmvirtualenvnew-project-copy
```

Важно — не забудьте перед удалением выйти из виртуальной среды при помощи команды:

```
$ deactivate  
$ rmvirtualenv new-project-copy  
Removing new-project-copy...  
$ workon  
new-project
```

## 18.1.3 Справочный материал

### Синтаксис

Интерфейс использования:

```
pip <command> [options]
```

install	Установка пакетов
download	Скачивание пакетов.
uninstall	Удаление пакетов.
freeze	Вывод установленных пакетов в формате файла зависимостей.
list	Вывод списка установленных пакетов.
show	Вывод информации об установленных пакетах.
check	Проверка совместимости установленных пакетов с их зависимостями.
config	Управление локальной и глобальной конфигурацией.
search	Поиск пакетов в PyPI.
wheel	Сборка wheel из зависимостей.
hash	Вычисление хешей из hashes of package archives.
completion	Вспомогательная команда для автодополнения командной строки.
debug	Вывод информации применимой для отладки.
help	Вывод справки по командам.

Общие опции:	Описание
-h, --help	Вывод справки.
-isolated	Запуск pip в изолированном режиме, с игнорированием переменных окружения и пользовательской конфигурации.
-v, --verbose	Расширенный вывод. Опция аддитивная и может использоваться до 3 раз.
-V, --version	Вывод версии.
-q, --quiet	Сокращенный вывод. Опция аддитивная и может использоваться до 3 раз (соответствует уровням логирования WARNING, ERROR и CRITICAL).
-log	Путь инкрементального вывода логов.
--proxy	Настройки прокси в форме [user:passwd@]proxy.server:port.
--retries	Максимальное количество попыток, которое должно предпринять каждое соединение (по умолчанию 5 раз).
--timeout	Установить время ожидания сокета (по умолчанию 15 секунд).
--exists-action	Действие по умолчанию, когда путь уже существует: (s)witch, (i)gnore, (w)ipe, (b)ackup, (a)bort.
--trusted-host	Пометить хост как доверенный, в случае если HTTPS хоста недействительный.
--cert	Путь к альтернативному комплекту CA.
--client-cert	Путь к сертификату клиента SSL, отдельный файл, содержащий закрытый ключ и сертификат в формате PEM.
--cache-dir	Настройка директории кэширования данных.
--no-cache-dir	Отключение кеша.
--disable-pip-version-check	Отключение периодической проверки PyPI на доступность новой версии pip.
--no-color	Подавление цветного вывода.

#### Ссылки на дополнительные ресурсы:

- <https://docs.python.org/3/library/venv.html>
- <https://www.codecamp.ru/blog/python-virtual-environments/>

## 18.2 Задачи для практики

### 18.2.1 Задача: настроить виртуальную среду с помощью `venv` для управления зависимостями проекта

- Откройте терминал или командную строку и перейдите в каталог проекта;
- Создайте новую виртуальную среду с помощью следующей команды: `python -m venv myenv`;
- Активируйте виртуальную среду с помощью следующей команды: (Windows) `myenv\Scripts\activate.bat` или (Linux/MacOS) `myenv/bin/активировать`;
- Установите зависимости проекта с помощью pip: `pip install -r requirements.txt`;
- Убедитесь, что виртуальная среда работает правильно, запустив проект или его тесты в активированной среде;
- По окончании работы над проектом деактивируйте виртуальную среду с помощью команды `deactivate`.

### 18.2.2 Задача: настроить виртуальную среду с помощью `venv` и отправить код в репозиторий Git.

- Создайте новый репозиторий Git для вашего проекта;
- Клонируйте репозиторий на свой локальный компьютер с помощью команды `git clone`;
- Перейдите в корневой каталог вашего проекта в терминале или командной строке;
- Создайте новую виртуальную среду с помощью следующей команды: `python -m venv venv`;
- Активируйте виртуальную среду с помощью следующей команды: (Windows) `venv\Scripts\activate.bat` или (Linux/MacOS) `source venv/bin/activate`;
- Установите зависимости проекта с помощью pip: `pip install -r requirements.txt`;
- Напишите свой код и зафиксируйте изменения, используя `git add .` и `git commit -m "Commit message"`;
- Отправьте изменения в репозиторий Git с помощью `git push`;
- По окончании работы над проектом деактивируйте виртуальную среду с помощью команды `deactivate`;
- Чтобы использовать проект на другом компьютере, клонируйте репозиторий и повторите шаги 4–6, чтобы настроить виртуальную среду и установить зависимости.

Используя вместе `venv` и `Git`, вы можете изолировать зависимости вашего проекта и легко отслеживать изменения кода с течением времени.

### 18.3 Вопросы:

1. Что такое виртуальное окружение (virtual environment) для Python?
2. Какова цель использования виртуальных окружений?
3. Как создать новое виртуальное окружение с помощью venv?
4. Как активировать виртуальное окружение на различных операционных системах?
5. Как деактивировать виртуальное окружение?
6. Как проверить, что виртуальное окружение активировано?
7. Какой файл используется для хранения списка зависимостей проекта?
8. Как создать файл зависимостей (например, requirements.txt) для Python-проекта?
9. Как установить зависимости проекта из файла requirements.txt?
10. Как добавить новую зависимость в виртуальное окружение?
11. Как удалить зависимость из виртуального окружения?
12. Какие другие способы можно использовать для управления зависимостями в Python-проекте?
13. Что такое глобальные и локальные зависимости?
14. Какие файлы и директории следует исключить из системы контроля версий (например, Git) при работе с виртуальными окружениями?
15. Каковы основные преимущества использования виртуальных окружений в сравнении с установкой зависимостей глобально?
16. Что произойдет, если в разных проектах используются разные версии одной и той же библиотеки?
17. Как перенести виртуальное окружение на другую систему?
18. Как удалить виртуальное окружение?
19. Как использовать виртуальное окружение в IDE (например, Visual Studio Code)?
20. Какие еще инструменты или методы вы используете для управления рабочим окружением в Python-проектах?

Эти вопросы помогут оценить знания основ настройки и работы с виртуальными окружениями (venv) в Python, что является важным навыком для эффективной разработки проектов.



## Инструменты для создания документации

---

### 19.1 Документирование

#### 19.1.1 Мотивация

Наверное, многие из вас слышали про TDD, когда тесты пишутся до написания кода, и затем пишется сам код, который пройдет все тесты. Такой подход позволяет взглянуть на код с другой стороны и, при правильном отношении, лучше структурировать его. Другой способ лучше писать код — это так называемое «Readme-driven-development», т.е. такой способ разработки, при котором мы пишем сначала документацию к коду, включая readme и документацию функций (docstring в python). Простой пример: **если у вас есть трудности с тем, чтобы описать работу функции, скорее всего, ее нужно разбить на несколько функций и задокументировать каждую по отдельности.** Наличие документации, особенно если она хорошо составлена, поможет не только другим разработчикам, но и вам самим, чтобы восстановить контекст и понять, что же вы имели в виду, когда писали этот код год назад.

## 19.1.2 Markdown

Когда мы писали свою простую библиотеку, мы делали файл README.md, содержимое которого собиралось в красивый документ самим GitHub. В этом случае мы использовали язык разметки **Markdown**, который является одним из общепринятых способов оформления документации. Рассмотрим элементы разметки. В данном документе мы использовали онлайн-редактор, но подойдет и любой другой.

### Заголовки

Чтобы сделать текст заголовком, используйте один или несколько знаков решетки. Чем их больше, тем меньше заголовок:

The screenshot shows a Markdown editor interface. At the top, it displays 'DOCUMENT NAME: try\_markdown.md', 'READING TIME: 1 MIN READ WORDS: 6', and 'CHARACTERS: 52'. Below this is a toolbar with 'MARKDOWN' and 'PREVIEW' buttons, and a 'diff' icon. The left pane shows the raw Markdown code:

```

1
2 # Заголовок
3
4 ## Заголовок поменьше
5
6 ### Заголовок еще поменьше
7

```

The right pane shows the rendered output:

**Заголовок**  
**Заголовок поменьше**  
**Заголовок еще поменьше**

Можно обозначать заголовки через подчеркивания. **Хорошим тоном** считается отделение заголовка от текста выше и ниже, а также отделение знака решетки от текста заголовка:

```
... какой-то текст до .
```

```
# Заголовок
```

```
Начало раздела...
```

### Абзацы

Чтобы в тексте начать новый абзац, разделите текст пустой строкой:

The screenshot shows a Markdown editor interface. At the top, it displays 'DOCUMENT NAME: try\_markdown.md', 'READING TIME: 1 MIN READ WORDS: 2', and 'CHARACTERS: 14'. Below this is a toolbar with 'MARKDOWN' and 'PREVIEW' buttons, and a 'diff' icon. The left pane shows the raw Markdown code:

```

1
2 абзац1
3
4 абзац2

```

The right pane shows the rendered output:

абзац1  
абзац2

**Хорошим тоном** считается писать текст без отступов.

Так делаем:

DOCUMENT NAME

try\_markdown.md

READING TIME: 1 MIN READ WORDS: 23

CHARACTERS: 152

MARKDOWN



PREVIEW

&lt;/&gt;

```

1
2 Это первый параграф, мы просто пишем какой-то
3 текст чтобы он был на двух строках.
4
5 Это второй параграф, еще немного текста чтобы
6 заполнить пространство.

```

Это первый параграф, мы просто пишем какой-то текст чтобы он был на двух строках.

Это второй параграф, еще немного текста чтобы заполнить пространство.

А так нет:

DOCUMENT NAME

try\_markdown.md

READING TIME: 1 MIN READ WORDS: 23

CHARACTERS: 152

MARKDOWN



PREVIEW

&lt;/&gt;

```

1
2 Это первый параграф, мы просто пишем какой-то
3 текст чтобы он был на двух строках.
4
5 Это второй параграф, еще немного текста чтобы
6 заполнить пространство.

```

Это первый параграф, мы просто пишем какой-то текст чтобы он был на двух строках.

Это второй параграф, еще немного текста чтобы заполнить пространство.

Хотя результат будет один и тот же, мы всегда думаем не только о тех, кто читает собранный документ, но и о тех, кто читает исходник. Различные отступы сильно затрудняют чтение.

## Форматирование текста

Текст можно делать жирным или курсивом:

DOCUMENT NAME

try\_markdown.md

READING TIME: 1 MIN READ WORDS: 6

CHARACTERS: 41

MARKDOWN



PREVIEW

&lt;/&gt;

```

1
2 Текст можно выделять **жирным** или *курсивом*

```

Текст можно выделять **жирным** или *курсивом*

Вместо звездочек можно использовать подчеркивания, но звездочки являются более универсальным способом выделения текста.

**Задание:** как сделать шрифт жирным курсивом.

## Списки

В Markdown можно делать нумерованный список или просто перечисление пунктов.

DOCUMENT NAME: try\_markdown.md

READING TIME: 1 MIN READ WORDS: 12  
CHARACTERS: 85

MARKDOWN	PREVIEW
1 2 Мы можем создать нумерованный список, со списком дел: 3 4 1. Проснуться 5 2. Поесть 6 3. Лечь спать	Мы можем создать нумерованный список, со списком дел:  1. Проснуться 2. Поесть 3. Лечь спать

Мы можем использовать разные символы для ненумерованных списков (-, \*, +), **хорошей практикой** является использование одного из них:

DOCUMENT NAME: try\_markdown.md

READING TIME: 1 MIN READ WORDS: 5  
CHARACTERS: 42

MARKDOWN	PREVIEW
1 2 Хорошая практика: 3 4 * Это 5 * Нормальный 6 * Список	Хорошая практика:  • Это • Нормальный • Список

Не смешивайте несколько разных символов:

DOCUMENT NAME

try\_markdown.md

READING TIME: 1 MIN READ WORDS: 9

CHARACTERS: 44

MARKDOWN



PREVIEW

&lt;/&gt;

```

1
2 А эта не очень:
3 * А так
4 - Лучше
5 + Не делать

```

А эта не очень:

- А так
- Лучше
- Не делать

## Ссылки

Для того чтобы вставить ссылку, пишем ее название в [] и после саму ссылку в (), название будет отображено в собранном документе:

DOCUMENT NAME

try\_markdown.md

READING TIME: 1 MIN READ WORDS: 7

CHARACTERS: 46

MARKDOWN



PREVIEW

&lt;/&gt;

```

1
2 Вот ссылка на [яндекс](https://ya.ru/), она
постоянно теряется.

```

Вот ссылка на [яндекс](https://ya.ru/), она постоянно теряется.

Можно не вставлять ссылки в самом тексте, а вести их список отдельно и использовать id ссылки там, где нужно.

DOCUMENT NAME

try\_markdown.md

READING TIME: 1 MIN READ WORDS: 7

CHARACTERS: 46

MARKDOWN



PREVIEW

&lt;/&gt;

```

1
2 Вот ссылка на [яндекс][ya_id], она постоянно
теряется.
3
4 [ya_id]: https://ya.ru/

```

Вот ссылка на [яндекс](https://ya.ru/), она постоянно теряется.

### **Вставка изображений**

Вставка изображений напоминает использование ссылок. Мы пишем текст, который будет показан вместо изображения, если оно не будет подгружено, в [], и указываем путь или ссылку на изображение. Вначале мы ставим восклицательный знак, а в круглых скобках можем указать всплывающую подсказку:

```
![opensource_icon] (/graphics/opensource.png "This opensource icon")
```



Если ошибиться в пути, будет показан текст из []:

```
![opensource_icon] (https://not_existing_site/graphics/not_exist.png)
```

**Задание:** проверьте, можно ли вести список картинок отдельно, например, в конце документа, и вставлять их в нужное место, используя id.

## Код

В случае необходимости процитировать в документации исходный код программы или его части используются специализированные форматированные блоки, в которые помещаются фрагменты исходного кода программы. Это похоже на HTML-теги. Для создания блока кода в языке Markdown необходимо обрамить его символами ``` код программы ``` . Многие программы поддерживают форматирование текста Markdown в соответствии со стилем содержимого. Если мы оформим код программы, то в режиме просмотра его синтаксис будет подсвечиваться (как в среде разработки). Для лучшего распознавания стиля языка программирования следует указывать название языка или скрипта для применения подсветки его синтаксиса. Например, для представления фрагмента программы на языке Python лучше оформить его следующим образом:

```
```python
import os

def foo():
    path = os.getcwd()
    print(f" Путь к текущей директории = {} \n\n Список файлов: \n")
    rez = sorted(os.listdir(path))
    for n, item in enumerate(rez):
        print(n+1, item)
```

```

После того как вы откроете этот документ, например, в браузере, вы увидите что код преобразился, его стиль изменился и стал больше похож на программный код.

```
import os

def foo():
    path = os.getcwd()
    print(f" Путь к текущей директории = {} \n\n Список файлов: \n")
    rez = sorted(os.listdir(path))
    for n, item in enumerate(rez):
        print(n+1, item)
```

The screenshot shows a Markdown editor interface. At the top, it displays the document name "try\_markdown.md", reading time "1 MIN READ", words "16", and characters "104". Below this, there are tabs for "MARKDOWN" and "PREVIEW". The MARKDOWN tab shows the original code block. The PREVIEW tab shows the transformed code where the code block is highlighted with syntax coloring. A callout box points to the word "import" in the preview, which is highlighted in green.

## Цитирование

В текст можно вставлять цитаты:

The screenshot shows a simple Markdown editor interface. At the top, it displays 'DOCUMENT NAME: try\_markdown.md' and 'READING TIME: 1 MIN READ WORDS: 10 CHARACTERS: 50'. Below this, there are two tabs: 'MARKDOWN' and 'PREVIEW'. In the 'MARKDOWN' tab, the code is:

```
1
2 >Code is read much more often than it is
   written
```

In the 'PREVIEW' tab, the rendered output is:

*Code is read much more often than it is  
written*

### 19.1.3 reStructuredText

Сейчас будет неожиданно. Но дальше нам будет нужен другой язык разметки — `reStructuredText`. Для тех, кто потратил время на материал выше, будет [нечтожно](#) освоить и `reStructuredText`. Причины, по которым мы описали `Markdown` и затем предложили переключиться на `reStructuredText`, две:

1. Курс написан с использованием `Markdown`.
2. Инструмент (`Sphinx`), который соберет документацию для нашего пакета и сделает из нее красивые `html`-страницы или `pdf`, работает нативно с файлами `reStructuredText`. Для работы с `Markdown` требуется подключать дополнительный плагин.

Вы можете изучить отличия чуть позже, сейчас мы возьмем open source [конвертер](#) `md` в `rst` и используем его. Для README это вполне рабочий вариант. Серьезную документацию лучше сразу писать в формате `rst`.

### 19.1.4 Sphinx

`Sphinx` — это достаточно мощный инструмент, который может собрать не только документацию к проекту и сделать из нее веб-страницы, которые мы можем потом разместить на [readthedocs](#), но и текст для мануалов `man`, которые являются стандартными для `bash`.

**Обращайте внимание на то, в каких каталогах мы выполняем те или иные операции в bash во вставках кода ниже**

Создадим пустой каталог и перейдем в него, создадим каталог для документации в этой папке и заберем себе `README.md`:

```
(mtracker) artem@pc:~/tmp$ mkdir test_the_docs
(mtracker) artem@pc:~/tmp/test_the_docs$ cd test_the_docs
(mtracker) artem@pc:~/tmp/test_the_docs$ mkdir docs
(mtracker) artem@pc:~/tmp/test_the_docs$ wget https://raw.githubusercontent.com/standlab/
  ↵mtracker/main/README.md
(mtracker) artem@pc:~/tmp/test_the_docs$ ls
docs  README.md
```

Теперь установим `sphinx` и конвертер `rst -> md`, конвертируем `README.md`:

```
(mtracker) artem@pc:~/tmp/test_the_docs$ pip install sphinx
(mtracker) artem@pc:~/tmp/test_the_docs$ pip install m2r
(mtracker) artem@pc:~/tmp/test_the_docs$ m2r README.md
(mtracker) artem@pc:~/tmp/test_the_docs$ ls
docs  README.md  README.rst
```

*Примечание: если бы у нас сразу был файл в формате `rst`, команды с `m2r` можно было бы пропустить.*

Теперь мы готовы инициализировать каталог с документацией.

```
(mtracker) artem@pc:~/tmp/test_the_docs$ cd docs
(mtracker) artem@pc:~/tmp/test_the_docs/docs$ sphinx-quickstart
(mtracker) artem@pc:~/tmp/test_the_docs/docs$ ls
build  make.bat  Makefile  source
```

На этом этапе вам будут заданы вопросы про название проекта, автора, версию и язык документации. На вопрос, разделять ли исходники и собранную документацию, отвечаем «да»:

```
> Separate source and build directories (y/n) [n]: y
```

Теперь зайдем в файл `nano ./source/index.rst` и добавим в конец строки:

```
Readme File
=====
.. include:: ../../README.rst
```

Это нужно для того, чтобы содержимое `README` было доступно в документации. *Мы пишем ../../, так как `README.rst` находится на два уровня выше, чем содержимое каталога `source`.* Теперь мы готовы собрать документацию:

```
(mtracker) artem@pc:~/tmp/test_the_docs/docs$ sphinx-build -b html source build
Running Sphinx v5.3.0
loading translations [ru]... готово
loading pickled environment... готово
building [mo]: targets for 0 po files that are out of date
building [html]: targets for 1 source files that are out of date
updating environment: 0 added, 1 changed, 0 removed
reading sources... [100%] index
looking for now-outdated files... none found
pickling environment... готово
checking consistency... готово
preparing documents... готово
writing output... [100%] index
generating indices... genindex готово
writing additional pages... search готово
copying static files... готово
copying extra files... готово
dumping search index in Russian (code: ru)... готово
dumping object inventory... готово
сборка завершена успешно.

(mtracker) artem@pc:~/tmp/test_the_docs/docs$ browse ./build/index.html
```

После этого должна открыться страница, на которой, кроме всего прочего, мы можем увидеть README.

### 19.1.5 Markdown и Sphinx

Ниже приведены шаги (в ОС Linux), чтобы настроить окружение для сборки документации Markdown с помощью Sphinx. Мы использовали этот подход при сборке данного документа. Создайте виртуальное окружение (опционально):

```
conda create -n sphinx_md python=3.10
conda activate sphinx_md
```

Установите Sphinx и поддержку Markdown:

```
pip install sphinx
pip install --upgrade myst-parser
```

Укажите, какие файлы должны войти в документацию:

```
.. toolchain documentation master file, created by
sphinx-quickstart on Wed Mar 1 10:24:16 2023.
You can adapt this file completely to your liking, but it should at least
contain the root `toctree` directive.

Welcome to toolchain's documentation!
=====

.. toctree::
:maxdepth: 2
:caption: Contents:

00_terms/main.md
01_open_source/main.md

Indices and tables
=====

* :ref:`genindex`
* :ref:`modindex`
* :ref:`search`
```

Соберите html (находясь в корневой директории проекта):

```
make html
```

В корневой директории появится папка build, где будет находиться собранная документация.

Для сборки pdf установите latexmk и поддержку кириллицы:

```
sudo apt install latexmk
sudo apt install texlive-lang-cyrillic
make latexpdf
```

### 19.1.6 Документация из кода

Здесь речь пойдет о том, как писать комментарии к коду так, чтобы из них можно было собрать документацию. В Python мы можем снабдить функции и классы многострочным комментарием, который будет распознаваться системами сборки документации как docstring, и на их основе будет формироваться документация к вашему проекту. В документацию попадут не все комментарии их кода, а только те, которые оформлены как docstring. Правильно оформленный docstring может помочь другим разработчикам, а спустя время даже вам, разобраться в работе функции.

Для того чтобы создать комментарий, нужно заключить текст в тройные кавычки. Существует несколько форматов оформления docstring, ниже приведено форматирование в **стиле Google**.

```
def my_function(arg1, arg2):
    """
    Brief description of the function.

    More detailed explanation of the function's purpose,
    arguments, and expected behavior.

    Parameters:
        arg1 (type): Description of the first parameter.
        arg2 (type): Description of the second parameter.

    Returns:
        return_type: Description of the value returned by the function.

    Raises:
        ExceptionType: Description of the exception raised, if any.
    """
    # Function implementation here
    return result
```

Мы видим, что есть несколько разделов внутри docstring.

- \* Краткое описание (в одну строку).
- \* Более подробное описание.
- \* Входные аргументы с указанием типов.
- \* Тип и описание выходного значения.
- \* Какие ошибки могут быть выброшены из функции?

Также можно оформить docstring в других форматах, например, в **стиле Sphinx**.

```
def my_function(arg1, arg2):
    """
    Brief description of the function.

    :param arg1: Description of the first argument.
    :type arg1: type
    :param arg2: Description of the second argument.
    :type arg2: type
    :return: Description of the value returned by the function.
    :rtype: return_type
    :raises ExceptionType: Description of the exception raised, if any.
    """
```

(continues on next page)

(продолжение с предыдущей страницы)

```
# Function implementation here
return result
```

Или в **стиле NumpyDoc**, который был вдохновлен документацией к библиотеке numpy:

```
def my_function(arg1, arg2):
    """
    Brief description of the function.

    Parameters
    -----
    arg1 : type
        Description of the first argument.
    arg2 : type
        Description of the second argument.

    Returns
    -----
    return_type
        Description of the value returned by the function.

    Raises
    -----
    ExceptionType
        Description of the exception raised, if any.
    """
    # Function implementation here
    return result
```

### 19.1.7 Оформление документации

Теперь попробуем попрактиковаться с документацией. Мы хотим создать следующую структуру проекта, назначение файлов станет понятно ниже. Часть файлов будет сгенерирована автоматически, другие мы будем править, оставшиеся нам нужно будет создать самим.

```
project/
└── docs/
    ├── Makefile
    ├── conf.py
    ├── index.rst
    ├── usage.rst
    └── api.rst
└── src/
    └── utils_mix.py
```

Создадим новый проект, который назовем `utils_mix`. Создадим в нем две папки: `src` для кода и `docs` для документации.

```
mkdir src
mkdir docs
```

Перейдем в папку `docs` и выполним там инициализацию с помощью `sphinx`, который мы ранее установили с помощью `pip install sphinx`.

```
cd docs
sphinx-quickstart
```

После этого утилита запросит нужную информацию.

```
Please enter values for the following settings (just press Enter to
accept a default value, if one is given in brackets).

Selected root path: .

You have two options for placing the build directory for Sphinx output.
Either, you use a directory "_build" within the root path, or you separate
"source" and "build" directories within the root path.
> Separate source and build directories (y/n) [n]: n

The project name will occur in several places in the built documentation.
> Project name: utils_mix
> Author name(s): ArtemVesnin
> Релиз проекта []: 1.0.
> Язык проекта [en]: en
```

И создаст несколько файлов, включая `Makefile` для автоматический сборки.

```
ls
_build conf.py index.rst make.bat Makefile _static _templates
```

В текущем состоянии документация будет собираться и доступна, если открыть в браузере `_build/html/index.html`, но мы пока мы не увидим документации по коду.

```
make html

Running Sphinx v7.0.1
making output directory... готово
....
....
dumping object inventory... готово
сборка завершена успешно.
```

Теперь перейдем в папку `src` и создадим там модуль `utils_mix.py`:

```
cd ../src
touch utils_mix.py
```

И добавим в этот модуль код с написанными docstring.

```
def add_numbers(num1, num2):
    """
    Add two numbers together.

    This function takes two numeric arguments, `num1` and `num2`,
    and returns their sum.

```

(continues on next page)

(продолжение с предыдущей страницы)

```

Parameters:
    num1 (int or float): The first number to be added.
    num2 (int or float): The second number to be added.

Returns:
    float: The sum of `num1` and `num2`.
    """
    return num1 + num2

def multiply_numbers(num1, num2):
    """
    Multiply two numbers.

    This function takes two numeric arguments, `num1` and `num2`,
    and returns their product.

    Parameters:
        num1 (int or float): The first number to be multiplied.
        num2 (int or float): The second number to be multiplied.

    Returns:
        float: The product of `num1` and `num2`.
        """
    return num1 * num2

def greet(name):
    """
    Greet a person.

    This function takes a string argument `name` and returns a greeting message.

    Parameters:
        name (str): The name of the person to greet.

    Returns:
        str: A greeting message with the person's name.
        """
    return f"Hello, {name}! How are you today?"

```

Мы можем структурировать нашу документацию, в нашем случае мы хотим, чтобы docstring попали в раздел API. Создадим файл api.rst и добавим в него следующее содержание.

## API Reference

---

```

.. automodule:: utils_mix
    :members:
    :undoc-members:
    :show-inheritance:

```

Таким образом мы говорим, что нужно сгенерировать документацию из модуля `utils_mix` (файл

utils\_mix.py).

```
* `:members:` чтобы включить функции и классы из модуля
* `:undoc-members:` для того чтобы включить даже то у чего нет docstring (опционально)
* `:show-inheritance:` для того чтобы показывать информацию о наследовании классов другим.
```

И изменим index.rst, чтобы api.rst стал включен в общую структуру. Добавим строку api в раздел

```
.. toctree::
```

```
.. toctree::
   :maxdepth: 2
   :caption: Contents:

   api
```

Последнее изменение, которое нам нужно сделать, это добавить в conf.py несколько строчек кода так, чтобы система сборки находила модули автоматически:

```
import os
import sys

sys.path.insert(0, os.path.abspath('../src'))
```

И включить поддержку сборки документации из docstring:

```
extensions = ['sphinx.ext.autodoc']
```

Запускаем сборку еще раз и получаем

**utils\_mix API Reference**

**Navigation**

Contents:

[API Reference](#)

**Quick search**

Go

**utils\_mix.add\_numbers(*num1, num2*)**

Add two numbers together.

This function takes two numeric arguments, *num1* and *num2*, and returns their sum.

**Parameters:**

- num1* (int or float): The first number to be added.
- num2* (int or float): The second number to be added.

**Returns:**

- float*: The sum of *num1* and *num2*.

**utils\_mix.greet(*name*)**

Greet a person.

This function takes a string argument *name* and returns a greeting message.

**Parameters:**

- name* (str): The name of the person to greet.

**Returns:**

- str*: A greeting message with the person's name.

Мы также можем включать документы, которые написаны нами (а не сгенерированы автоматически), например, примеры по использованию кода. Создадим файл `usage.rst` и добавим в него следующее содержимое.

```
Usage Guide
=====
Introduction
-----
Welcome to the usage guide for `utils_mix`. This guide provides examples and explanations on how to use the functions defined in the `utils_mix` Python module.

Importing the Module
-----
To start using `utils_mix`, you need to import it into your Python script or interactive session:

.. code-block:: python

    import utils_mix
```

(continues on next page)

(продолжение с предыдущей страницы)

## Add and Multiply Numbers

---

The `utils\_mix` provides two simple functions for basic arithmetic operations:

- :func:`utils\_mix.add\_numbers`: Add two numbers together.
- :func:`utils\_mix.multiply\_numbers`: Multiply two numbers.

Example usages:

```
.. code-block:: python

    # Import the module
    from utils_mix import add_numbers, multiply_numbers

    # Add two numbers
    result1 = add_numbers(1, 2)
    print("Result of addition:", result1)  # Output: 3

    # Multiply two numbers
    result2 = multiply_numbers(5, 6)
    print("Result of multiplication:", result2)  # Output: 30
```

## Greet a Person

---

The `utils\_mix` also provides a simple function to greet a person:

- :func:`utils\_mix.greet`: Greet a person with a custom message.

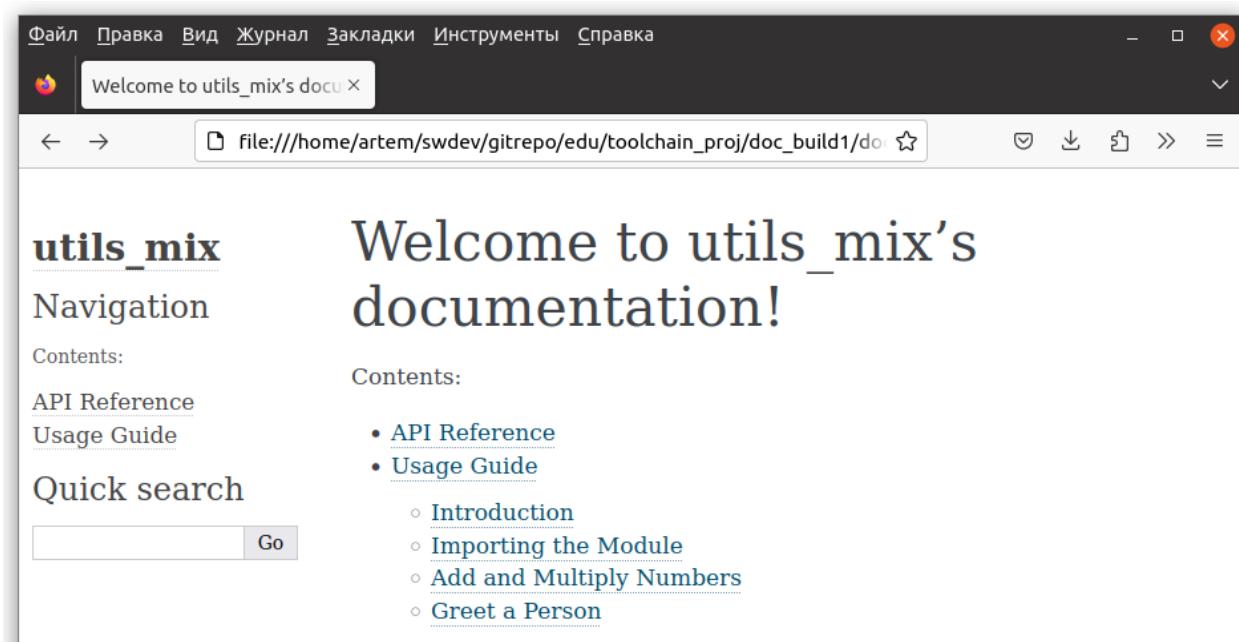
Example usage:

```
.. code-block:: python

    # Import the module
    from utils_mix import greet

    # Greet a person
    greeting = greet("Alice")
    print(greeting)  # Output: Hello, Alice! How are you today?
```

Включим его в документацию так же, как мы делали с `api.rst`, сделаем сборку `make html` и увидим обновленную документацию:



## 19.2 Задачи

### 19.2.1 Задача 1

Есть два модуля. Функции одного рассчитывают хэши и делают поиск дубликатов по ним. Функции второго заключаются в поиске дубликатов файлов. Напишите документацию к ним и сделайте автоматическую сборку при помощи Sphinx.

(код):

```
#content of duplicate_finder.py
import hashlib

def calculate_hash(text):
    return hashlib.sha256(text.encode()).hexdigest()

def find_duplicates(lines):
    duplicates = []
    for idx, line in enumerate(lines):
        line_hash = calculate_hash(line)
        if line_hash in duplicates:
            duplicates[line_hash].append(idx)
        else:
            duplicates[line_hash] = [idx]
    return duplicates
```

(код):

```
#content of file_processor.py
import os
```

(continues on next page)

(продолжение с предыдущей страницы)

```

from duplicate_hash import find_duplicates

def process_files(directory):
    duplicates_found = {}
    for filename in os.listdir(directory):
        file_path = os.path.join(directory, filename)
        if os.path.isfile(file_path):
            with open(file_path, 'r') as file:
                lines = file.readlines()
                file_duplicates = find_duplicates(lines)
                for line_hash, line_numbers in file_duplicates.items():
                    if line_hash in duplicates_found:
                        duplicates_found[line_hash].append((filename, line_numbers))
                    else:
                        duplicates_found[line_hash] = [(filename, line_numbers)]
    return duplicates_found

if __name__ == "__main__":
    # Example usage
    directory_path = "example_files" # Replace with your target directory path
    duplicates = process_files(directory_path)
    for line_hash, occurrences in duplicates.items():
        print(f"Hash: {line_hash}")
        for filename, line_numbers in occurrences:
            print(f" - File: {filename}, Line Numbers: {line_numbers}")
    print()

```

## Ответ

Так как написание документации — творческий процесс и, в отличие от кода, нельзя сказать, что документация работает правильно, в качестве критериев для оценивания этого задания можно взять два:

- \* после прочтения документации становится понятно, как использовать функцию;
- \* после прочтения документации складывается представление о внутренней реализации [функции](#).

### 19.2.2 Задача 2

Напишите генератор для тестовых данных для функции поиска дубликатов, используя документацию к функциям. Предположим, что ее написали заранее.

(код):

```

def generate_unique_words(num_words, max_len):
    """
    Generate a list of unique words.

    This function generates a list of unique words with the specified
    number of words (num_words) and maximum word length (max_len).
    It uses a set to ensure uniqueness.

```

(continues on next page)

(продолжение с предыдущей страницы)

*Parameters:*

*num\_words (int): The number of unique words to generate.*  
*max\_len (int): The maximum length of each word.*

*Returns:*

*list: A list of unique words.*

*Example:*

```
>>> unique_words = generate_unique_words(num_words=20, max_len=8)
>>> print(unique_words)
['ghijklmn', 'opqrst', 'abcd', 'uvwxyz', 'ef', 'yz', 'ij', 'klm', 'gh',
'nopq', 'rs', 'tuuw', 'jkl', 'defg', 'cde', 'ijk', 'lmn', 'mn', 'hi',
'pqrs']
"""
pass
```

```
def generate_unique_lines(words, num_lines, max_words):
    """
    Generate a list of unique lines.
```

*Parameters:*

*words (list): A list of unique words.*  
*num\_lines (int): The number of unique lines to generate.*  
*max\_words (int): The maximum number of words in each line.*

*Returns:*

*list: A list of unique lines.*

*Example:*

```
>>> unique_words = ['ghijklmn', 'opqrst', 'abcd', 'uvwxyz', 'ef', 'yz',
                     'ij', 'klm', 'gh', 'nopq', 'rs', 'tuuw', 'jkl',
                     'defg', 'cde', 'ijk', 'lmn', 'mn', 'hi', 'pqrs']
>>> unique_lines = generate_unique_lines(words=unique_words, num_lines=10, max_
words=6)
>>> print(unique_lines)
['pqrs ij ghi pqrs pqrs klm nopq', 'opqrst pqrs pqrs ij pqrs',
'klmnopqrs pqrs jkl ij ghi rs pqrs', 'jkl ij jkl defg pqrs ijk lmn',
'defg cde ghi opqrst pqrs klm pqrs', 'cde ghi klm ij', 'jkl jkl ijk klm',
'lmn pqrs opqrst', 'ijklmn ghi', 'rs']
"""
pass
```

```
def generate_duplicates(lines, num_dup):
    """
    Generate duplicates and their positions.
```

*Duplicates are placed in the end of input list. Duplicate positions should include atleast 2 values for every duplicate: original line position and position of duplicates originated from this line).*

*Parameters:*

*lines (list): A list of unique lines.*

(continues on next page)

(продолжение с предыдущей страницы)

```
num_dup (int): The number of duplicates to generate.
```

*Returns:*

```
list: A list of unique lines extended with duplicates.
```

```
dict: A dictionary containing positions of duplicates in the extended list.
```

*Example:*

```
>>> unique_lines = ['pqrs ij ghi pqrs pqrs klm nopq',
                    'opqrst pqrs pqrs ij pqrs',
                    'klmnopqrs pqrs jkl ij ghi rs pqrs',
                    'jkl ij jkl defg pqrs ijk lmn',
                    'defg cde ghi opqrst pqrs klm pqrs',
                    'cde ghi klm ij', 'jkl jkl ijk klm',
                    'lmn pqrs opqrst', 'ijklmn ghi', 'rs']

>>> num_duplicates = 3
>>> extended_lines, duplicates_positions = \
        generate_duplicates(lines=unique_lines, num_dup=num_duplicates)
>>> print("Extended Lines:")
>>> print(extended_lines)
['pqrs ij ghi pqrs pqrs klm nopq', 'opqrst pqrs pqrs ij pqrs',
 'klmnopqrs pqrs jkl ij ghi rs pqrs', 'jkl ij jkl defg pqrs ijk lmn',
 'defg cde ghi opqrst pqrs klm pqrs', 'cde ghi klm ij', 'jkl jkl ijk klm',
 'lmn pqrs opqrst', 'ijklmn ghi', 'rs', 'klmnopqrs pqrs jkl ij ghi rs pqrs',
 'cde ghi klm ij', 'ijklmn ghi']
>>> print("\nPositions of Duplicates:")
>>> print(duplicates_positions)
{'klmnopqrs pqrs jkl ij ghi rs pqrs': [2, 10],
 'cde ghi klm ij': [5, 11],
 'ijklmn ghi': [9, 12]}
.....
pass
```

**Ответ**

(код):

```
import random

def generate_unique_words(num_words, max_len):

    words = set()
    while len(words) < num_words:
        word_len = random.randint(1, max_len)
        word = ''.join(random.choice('abcdefghijklmnopqrstuvwxyz') for _ in range(word_
len))
        words.add(word)
    return list(words)

def generate_unique_lines(words, num_lines, max_words):
    lines = []
    while len(lines) < num_lines:
        line_words_count = random.randint(1, max_words)
        line_words = random.sample(words, line_words_count)
```

(continues on next page)

(продолжение с предыдущей страницы)

```

line = ' '.join(line_words)
if line not in lines:
    lines.append(line)
return lines

def generate_duplicates(lines, num_dup):
    duplicates = random.sample(lines, num_dup)
    duplicates_positions = {}
    for idx, line in enumerate(lines):
        if line in duplicates:
            if line in duplicates_positions:
                duplicates_positions[line].append(idx)
            else:
                duplicates_positions[line] = [idx]

    # Extend the lines with duplicates
    extended_lines = lines + duplicates

    # Add duplicate positions for each duplicated line
    for line in duplicates_positions:
        occurrences = duplicates_positions[line]
        while len(occurrences) < 2:
            occurrences.append(random.choice(occurrences))

    return extended_lines, duplicates_positions

```

### 19.2.3 Задача 3

Работайте в командах. Опишите и реализуйте функции для тестирования функций поиска дубликатов с помощью генератора. Сначала сделайте описание того, что вы хотите получить и какие условия (случаи) нужно реализовать, чтобы считать, что функции работают правильно. Поменяйтесь описанием с другой командой.

Реализуйте описанное другой командой в коде. Передайте свою реализацию команде, которая делала описание, и обсудите с ней, правильно ли реализовано задуманное (ошибки в реализации могут быть как по причине неполной документации, так и по причине неправильного прочтения документации).

Возьмите реализацию у команды, которой передавали свое описание, и также обсудите правильность реализации, но уже с позиции тех, кто предоставлял документацию.

## 19.3 Вопросы

1. Какова основная цель строки документации (docstring) в Python? а) Написать подробные пояснения к каждой строке кода. б) Предоставить обзор всей кодовой базы. в) Служить руководством пользователя для конечных пользователей программного обеспечения. г) Документировать назначение, использование и параметры функций, классов или модулей.
2. Какой из следующих форматов разметки обычно используется для строк документации (docstring) в Python? а) Markdown б) HTML в) ReStructuredText г) Latex

3. Как обычно называется файл документации, содержащий обзор проекта и инструкции по его использованию? а) PROJECT.md б) DOCS.txt в) README.md г) ОБЗОР.первый
4. Какая команда используется в Sphinx для создания HTML-документации из исходных файлов с текстом документацией в форматах Markdown или ReStructuredText? а) `make html` б) `sphinx-build html` в) `build html` г) `generate html`
5. Как получить доступ к строкам документации (docstring) в Python? а) Путем запуска скрипта Python с определенным аргументом командной строки. б) Вызовом функции `docstrings()` для модуля или класса. в) Используя атрибут `__doc__` функции, класса или модуля. г) Импортируя модуль `docstrings`.
6. Что должно быть включено в строку документации (docstring) для функции? а) Детали реализации функции. б) Описание назначения, параметров, сценариев использования функции. в) Личное мнение и комментарии о полезности функции. д) Информация о лицензировании и сведения об авторских правах.
7. В какой раздел файла README вы обычно включаете инструкции по установке данного пакета/библиотеки? а) Overview б) Usage в) Installation г) Contributing
8. Какова основная цель файла README в проекте Python? а) Предоставить обзор интерпретатора Python. б) Предоставить список зависимостей, необходимых для запуска проекта. в) Документировать назначение и использование функций в проекте. г) Предоставить введение и основную информацию о проекте.

**Ответы:**

1. г)
2. в)
3. в)
4. а)
5. в)
6. б)
7. в)
8. г)



## Основы тестирования

---

### 20.1 Тестирование

Тестирование — это способ убедиться, что код делает то, что мы ожидаем от него. Существует ручное и автоматическое тестирование. При ручном тестировании человек выполняет шаги из инструкции и фиксирует поведение приложения или кода. При автоматическом тесты выполняются специальным инструментом, и собирается статистика пройденных или проваленных тестов. Зачем нужно ручное тестирование, когда автоматические тесты выглядят гораздо удобней — один раз написал и пользуйся? Ручные тесты с привлечением человека незаменимы, когда речь идет об апробации интерфейса. И в любых других задачах, когда сложно сформулировать критерии правильной работы. Здесь речь шла о готовом приложении, когда речь идет о коде, автоматическое тестирование незаменимо. Без автоматического тестирования невозможно было обеспечить CI / CD (continuous integration/continuous delivery). CI/CD позволяют иметь оттестированную версию приложения в любой момент времени.

#### 20.1.1 Мотивация

Существует несколько аргументов, что тесты — это не пустая трата времени:

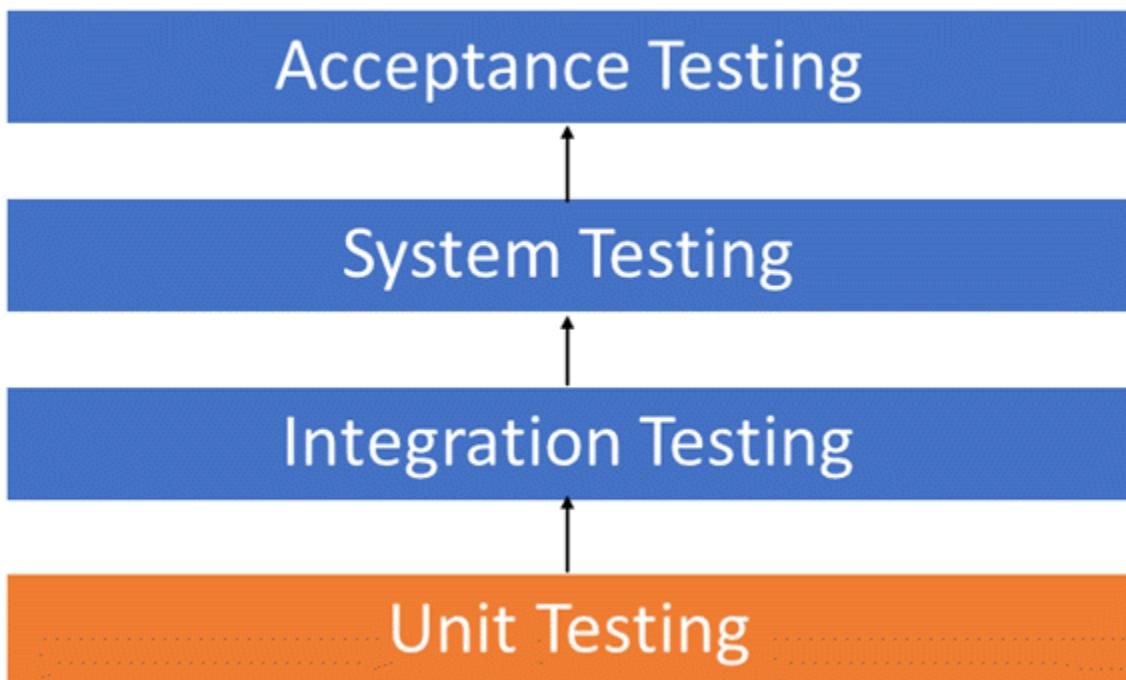
1. Первая очевидная функция — это уменьшение вероятности возникновения ошибок в коде. Много мелких ошибок могут привести к серьезной проблеме, и тесты призваны предотвратить появление таких ошибок в коде. Существуют системы, которые будут автоматически проводить тестирование при загрузке кода в репозиторий, и это просто не даст разработчику залить не полностью рабочий код, но это возможно только при наличии автоматизированных тестов.
2. Этот пункт скорее психологический — у разработчика появляется инструмент, который подскажет ему, если он ошибся. Разработчик тратит меньше внимания на возможность возникновения ошибок, ведь у него есть система тестов, и соответственно больше внимания уделяет написанию кода, более смело экспериментирует, ведь тесты подскажут ему, если он сломал другой участок программы. Таким образом, тесты способствуют более быстрой работе и появлению более интересных решений.
3. Тесты можно рассматривать как своего рода документацию: то, как мы можем использовать функционал нашего приложения.

### 20.1.2 Виды тестов

Выделяют несколько видов тестирования.

#### Модульное тестирование

Модульное или unit-тестирование. Как видно из названия, мы тестируем отдельные модули, или «единицы» (участки) кода. Модульные тесты наиболее близки к контексту конкретных функций или других логически обособленных участков кода. Отсюда вытекает одно из основных свойств модульных тестов — изолированность: если мы тестируем функцию `foo`, то там не должно быть участков, связанных с функцией `bar`. Модульные тесты позволяют проверить отдельные части кода, прежде чем тестиировать их совместно.

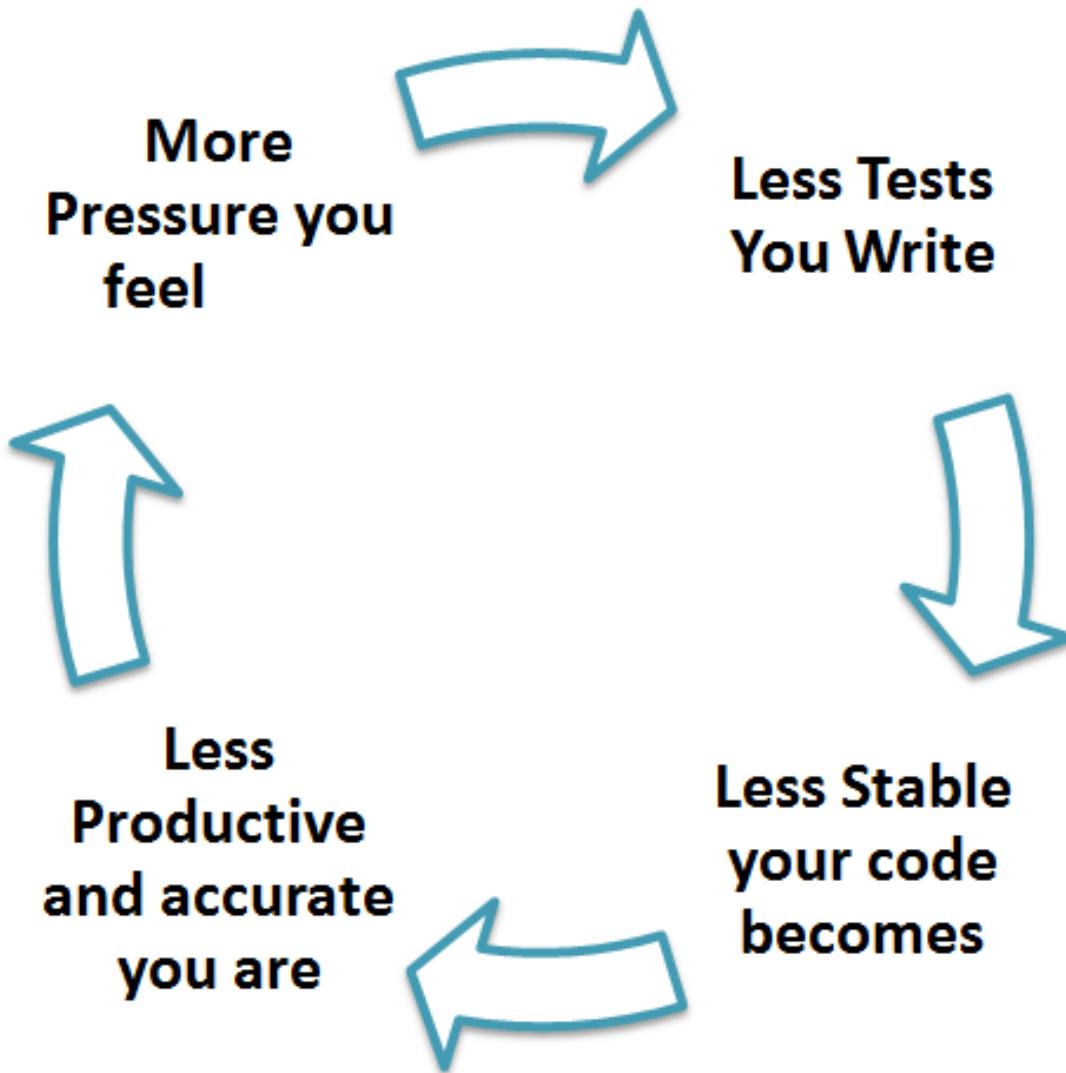


Модульное тестирование:

- может помочь при разработке: понимая, что каждый участок кода нужно будет протестировать отдельно, разработчик с большей вероятностью не будет создавать большие функции со смешанной логикой, а разобьет ту же функциональность на несколько маленьких функций, каждую из которых будет понятно, как протестировать. Это, в конечном счете, положительно влияет на то, как выглядит API библиотеки.
- заставляет задуматься о входных данных, которые подаются на вход коду, и лучше погрузиться в контекст решаемой задачи.
- позволяет протестировать часть кода, если в данный момент на другой частью работает другой человек, и она не проходит все тесты.
- покрытие тестами всех возможных вариантов зависит от разработчика.
- позволяет реализовывать подход «пиши код и тестируй одновременно». Так как модульные тесты атомарны по природе, мы можем выделять новые ситуации непосредственно во время написания кода и облачать их в тест.

- позволяет другим разработчикам убедиться, что у них все настроено правильно и можно продолжать разработку. Если тесты не проходят, значит, нужно остановиться и донастроить рабочее окружение.

Преимущества наличия модульных тестов в коде могут быть неочевидны сначала, но проблемы, которые копятся из-за отсутствия или недостаточного количества тестов, постепенно будут копиться, и со временем мы перестанем писать новую функциональность, а будем только разбирать баги.



## Пример

Рассмотрим функции, которые генерируют последовательность N первых чисел целиком и отдельно четные и нечетные.

```
#content of series.py in my_math package
def get_series(n):
    return list(range(n))

def get_even_series(n):
    series = get_series(n)
    even_series = [i * 2 for i in series]
    return even_series

def get_odd_series(n):
    series = get_series(n)
    odd_series = [i * 2 + 1 for i in series]
    return odd_series
```

Мы видим, что функции `get_even_series()` и `get_odd_series()` используют другую функцию `get_series()`. Это нормально и хорошо, когда есть возможность переиспользовать код в другом месте. Мы выделяем его в отдельный блок. Но все три функции должны быть протестированы без учета внутреннего устройства функции.

```
#content of test_series.py
import pytest

from my_math.series import (get_series,
                            get_even_series,
                            get_odd_series)

class TestSeries:
    def test_series(self):
        assert get_series(5) == [0, 1, 2, 3, 4]

    def test_even_series(self):
        assert get_even_series(5) == [0, 2, 4, 6, 8]

    def test_odd_series(self):
        assert get_odd_series(5) == [1, 3, 5, 7, 9]

    # test include the same code as in the function implementation
    def test_even_series_bad(self):
        n = 5
        even_series = [i * 2 for i in get_series(n)]
        assert get_even_series(n) == even_series

    # test include several functions that can be tested separately
    def test_all_series_bad(self):
        n = 5
        assert get_series(n) == [0, 1, 2, 3, 4]
        assert get_even_series(n) == [0, 2, 4, 6, 8]
        assert get_odd_series(n) == [1, 3, 5, 7, 9]
```

(continues on next page)

(продолжение с предыдущей страницы)

```
# might be a good test
def test_series_full(self):
    series = get_even_series(5) + get_odd_series(5)
    series.sort()
    assert get_series(10) == series
```

Давайте разберемся, что мы здесь видим. Тесты `test_series`, `test_even_series`, `test_odd_series` тестируют отдельно функциональность трех разных функций, поэтому это нормальные юнит-тесты. Мы можем написать генератор данных вместо использования фиксированных списков `[0, 1, 2, 3, 4]` и таким образом увеличить покрытие тестами. Мы вернемся к этому чуть позже, сейчас посмотрим на другие тесты. Тест `test_even_series_bad` не является хорошим тестом, так как мы используем ту же логику, что и при реализации функции, и если этот код будет немного сложнее, чем генерация списка чисел, то мы можем унести ошибку в тест, и он будет пройден, хотя функция работает не так, как закладывалось. Например, если нужно было генерировать числа, начиная с нуля, а мы генерируем с 1 `range(n) -> range(1, n)`. Тогда `test_series_should_fail_but_passed()` ниже не вызовет ошибок, так как мы просто скопировали код из функции. Тест `test_series` не пройдет, так как здесь данные сгенерированы независимо, и так мы поймем, что сделали ошибку:

```
#content of series.py in my_math package
def get_series(n):
    return list(range(1, n))                                # we made mistake

#content of test_series.py
import pytest

from my_math.series import (get_series)

class TestSeries:
    def test_series(self):
        assert get_series(5) == [0, 1, 2, 3, 4]

    def test_series_should_fail_but_passed(self):
        assert get_series(5) == list(range(1, n))      # and propagate it here
```

Посмотрим на тест `test_series_full`. Он включает в себя вызов нескольких функций, это может быть хорошей идеей, так как мы проверяем полноту данных. Такой тест не должен подменять собой другие тесты, где мы тестируем отдельные функции, но может дополнять их. Тест не настолько объемный, чтобы считаться **интеграционным тестом**, хотя он и объединяет несколько отдельных участков кода и говорит, как они должны работать совместно.

## Параметризация тестов

В тестах выше мы использовали заранее определенные списки, которые были захардкожены прямо в тесте. Мы можем расширить функциональность тестов, используя параметризацию тестов. Добавим в `test_series.py` тесты, которые принимают параметр:

```
import pytest

from my_math.series import (get_series,
                            get_even_series,
```

(continues on next page)

(продолжение с предыдущей страницы)

```
get_odd_series)

class TestSeries:
    def test_series_parametrized(self, max_number):
        i = 0
        series = list()
        while i < max_number:
            series.append(i)
            i = i + 1
        assert get_series(max_number) == series
```

И если мы попытаемся выполнить тесты, получим ошибку:

```
def test_series_parametrized(self, max_number):
E     fixture 'max_number' not found
```

Мы видим новый термин `fixture`. В самом деле, непонятно, откуда pytest должен узнать, что подставить вместо `max_number` при автоматическом тестировании. Если добавить `max_number` значение по умолчанию, то ошибки не будет, но это не то, что мы хотим сделать. Значения `max_number` должны быть каким-то образом определены, и в pytest для этого используются `fixture`, которые хранят данные, используемые в тестах. Мы определяем `fixture` в файле `conftest.py`, и они будут доступны всем тестами. Эти данные определяются в отдельном файле, так как вызов функции отличается от вызовов тестов, и определение входных данных для тестов можно отнести к конфигурированию тестов `conftest -> configure test`. Добавим в `conftest` следующие строки:

```
import pytest

@pytest.fixture
def max_number():
    return 5
```

Теперь проходят все тесты. Посмотрим, как можно сделать вызов теста с разными параметрами. Заменим с:

```
def pytest_generate_tests(metafunc):
    if "max_number" in metafunc.fixturenames:
        # end can be retrieved from command line parameters
        end = 10
        metafunc.parametrize("max_number", range(end))
```

Имя параметра, который мы хотим менять, — `max_number`, а `range(end)` задает диапазон значений. Здесь на первый взгляд происходит небольшая магия, но это то, как работает pytest с параметризованными тестами, и такие конструкции мы просто берем из документации. Функция `pytest_generate_tests` вызовется для каждого теста, далее мы посмотрим, что среди параметров-`fixture` есть тот, который нам нужен, и с помощью `metafunc.parametrize()` сделаем из одного теста несколько, подставляя вместо одного значения диапазон значений `range(end)`. Значение `end` на текущий момент задано жестко, но может управляться при запуске тестов.

**Модульное тестирование позволяет понять, как части кода работают сами по себе**

### 20.1.3 Интеграционное тестирование

Чтобы сделать интеграционное тестирование, нужно написать такие тесты, когда различные блоки, модули или компоненты программного приложения тестируются вместе. Интеграционное тестирование нужно для того, чтобы проверить ситуации, которые могут ускользнуть при юнит-тестировании. Основное отличие интеграционного теста от юнит-теста в том, что инициализируются все компоненты системы. Для юнит-теста мы можем определить тестовые данные непосредственно в teste или через `fixture`. Для интеграционного теста мы положим такие же данные в базу данных и затем возьмем оттуда. Для интеграционного тестирования могут применяться как те же инструменты, что и для модульного тестирования, например, `pytest`, так и специальные инструменты, которые протестируют приложение с внешней стороны. Например, подадут тестовый запрос так, как бы это делал пользователь, что можно сделать с помощью `Selenium`. Подробнее про тестирование можно прочитать [здесь](#) или [здесь](#).

**Интеграционное тестирование позволяет понять, как части кода и целые модули работают в контексте всей системы**

### 20.1.4 Больше тестов

Существует также системное тестирование, на этапе которого проверяется не только правильная функциональность частей системы, но и такие аспекты, как:

- работа под нагрузкой.
- удобство использования.
- обеспечение совместимости с предыдущими версиями кода. Это не совсем то же самое, что прохождение юнит-тестов, здесь мы тестируем, что новая версия программы работает с данными так же, как старые. Даже с учетом того, что юнит-тесты не принято менять в процессе добавления новой функциональности.
- восстановление работы системы при авариях.
- миграционное тестирование, при котором мы тестируем «переезд» на другие инструменты.

В этой главе мы не будем подробно останавливаться на этих тестах, так как это делается не только силами разработчиков, для целей данного курса мы подробно остановились на модульном тестировании.

### 20.1.5 Разработка через тестирование

TDD — test-driven development, разработка через тестирование. В выше мы сначала писали функциональность и затем придумывали тесты. В подходе TDD мы сначала пишем тесты и затем делаем так, чтобы разработанная нами функциональность проходила все тесты. Для разработки через тестирование требуется создать автоматизированные модульные тесты, как мы делали выше с помощью `pytest`, которые определяют требования к коду непосредственно перед написанием самого кода. Тест содержит проверки условий, которые могут либо выполняться, либо нет. Такие условия начинаются с ключевого слова `assert` в примерах выше. Когда они выполняются, говорят, что тест пройден. Прохождение теста подтверждает поведение, которое программист предполагает правильным.

Цикл разработки через тестирование (на основе книги Кента Бека «Разработка через тестирование: на примере»):

- **Добавление теста.** Добавление каждой новой функциональности начинается с написания теста. Только что написанный тест не будет проходить проверку, потому что код, который он тестирует, еще не написан (да, именно так). Если этот тест прошел проверку до написания кода, то данная функциональность уже реализована, либо же тест работает неправильно. Для написания теста

разработчик должен четко понимать требования к новой функциональности — это и отличает TDD-подход от остальных, разработчик фокусируется на требованиях до написания кода.

- **Запускаем все тесты, убеждаемся, что они не прошли.** На этом этапе мы проверяем сами тесты. Если написанный тест проходит всегда — значит, он бесполезен. Это увеличивает уверенность, хотя не гарантирует, что тест действительно тестирует то, что нам нужно.
- **Пишем код.** На этом этапе пишем код так, чтобы он проходил тесты. Этот код не обязан быть идеальным. Код может быть некрасивым, это мы поправим на следующих этапах. Главное, чтобы код был предназначен для прохождения этого теста. Не следует добавлять лишнюю функциональность, для которой не написан тест.
- **Запускаем тесты, убеждаемся, что они проходят.** Если тесты прошли — программист может быть уверен, что код удовлетворяет всем требованиям, определенным на этапе добавления теста, но не более. Если нет — переписываем код и повторно прогоняем тесты до тех пор, пока тесты не будут выполнены.
- **Рефакторинг.** Теперь настало время привести код в порядок. Это процесс изменения внутренней структуры программы, не затрагивающий ее внешнее поведение и имеющий целью облегчить понимание ее работы, устраниТЬ дублирование кода, облегчить внесение изменений, в общем, улучшить поддерживаемость кода.
- **Повторить цикл.** Повторяя описанный цикл и реализуем все новую и новую функциональность. Если вы используете сторонние библиотеки, не следует делать небольшие изменения, которые тестируют только функциональность библиотеки, а не ваш код, который использует ее. Исключением может быть случай, когда у вас есть подозрения, что сторонняя библиотека содержит ошибки.

Разработка через тестирование позволяет сделать код чище и яснее, потому что мы пишем только те фрагменты, которые необходимы для прохождения тестирования. Также разработка тестов до написания программы позволяет писать код, который более пригоден для тестирования, что не скажешь о создании тестов после написания кода, когда бывает тяжело отделить одну функциональность от другой. TDD способствует тому, что тестами будет покрыта вся функциональность.

## Пример

Для демонстрации разработки тестирования мы будем писать класс «дробь» (`Fraction`). Назовем наш пакет `fraction`. В каталоге `fraction` создаем модуль `fraction.py`. В этом файле мы будем реализовывать функциональность нашего класса. Далее в корневой директории проекта создаем каталог `tests` и в нем файл `test_fraction.py`. Не забудьте добавить `__init__.py` в каталоги `tests` и `fraction`, чтобы они воспринимались как пакеты. Пропишем в начале файла `import pytest`. Мы будем использовать библиотеку `pytest` для написания и выполнения тестов. Если на вашем рабочем окружении не установлен `pytest`, установите его:

```
pip install pytest.
```

Структура файлов должна выглядеть так:

```
my_project
├── fraction
│   ├── __init__.py
│   └── fraction.py
└── tests
    ├── __init__.py
    └── test_fraction.py
```

В файле `test_fraction.py` мы будем писать тесты. Создадим класс `TestFraction`, в котором будем все писать. Для начала напишем код, который будет проверять корректность создания дроби. Для этого надо подумать, что наш класс будет принимать на вход. Пусть это будет числитель, `numerator`, и знаменатель, `denominator`. Все функции, которые что-либо тестируют в `pytest`, должны начинаться со слова `test`. Создадим функцию `test_fraction_creation()`, которая будет тестировать создание экземпляров `Fraction`. Давайте возьмем такие пары числитель-знаменатель:  $(5, 2)$ ,  $(6, 8)$ ,  $(-5, 2)$ ,  $(2, -5)$ ,  $(0, 1)$ ,  $(1, 0)$ ,  $(1, 1)$ . Во время подготовки данных помним про опасность дробей, которые в знаменателе содержат ноль. В результате у нас должен получиться такой код:

```
import pytest
from fraction.fraction import Fraction

class TestFraction:
    def test_fraction_creation(self):
        Fraction(5, 2)
        Fraction(6, 8)
        Fraction(-5, 2)
        Fraction(2, -5)
        Fraction(0, 1)
        Fraction(1, 0)
        Fraction(1, 1)
```

Теперь можем запустить тесты:

```
cd my_project
pytest
```

Мы запустили тесты! В отчете нам пишут, что:

```
ImportError: cannot import name 'Fraction' from 'fraction'.
```

Наша программа не может найти класс `Fraction`. Для этого в файле `fraction/init.py` пропишем следующее:

```
from .fraction import Fraction
```

А в `fraction/fraction.py` создадим заготовку кода:

```
class Fraction:
    def __init__(self, numerator, denominator):
        self.numerator = numerator
        self.denominator = denominator
```

Запускаем тесты — теперь у нас все работает:

```
=====
 test session starts =====
platform linux -- Python 3.7.4, pytest-5.2.1, py-1.8.0, pluggy-0.13.0
rootdir: /home/artem/swdev/gitrepo/edu/toolchain/23_testing/code/fraction
plugins: doctestplus-0.4.0, arraydiff-0.3, remotedata-0.3.2, openfiles-0.4.0
collected 1 item

tests/test_fraction.py . [100%]

=====
 1 passed in 0.02s =====
```

## 20.2 Задачи:

### 20.2.1 Задача 1

Напишите класс `Fraction`, который будет принимать числитель и знаменатель для создания экземпляра класса. Класс должен иметь методы для приведения дроби к простому виду  $4/6 \rightarrow 2/3$ , а также метод для определения дроби, наиболее близкой к иррациональному числу (напомним, что иррациональное число нельзя представить в виде дроби). При поиске ближайшей к иррациональному числу дроби ограничьте знаменатель, чтобы он не превышал 100. Напишите тесты к этим функциям.

**Ответ**

(код):

```
class Fraction:
    def __init__(self, numerator, denominator):
        self.numerator = numerator
        self.denominator = denominator

    def simplify(self):
        gcd = self._compute_gcd(self.numerator, self.denominator)
        self.numerator //= gcd
        self.denominator //= gcd

    def _compute_gcd(self, a, b):
        while b:
            a, b = b, a % b
        return a

    def closest_ratio_to_irrational(self, irrational):
        closest_difference = float("inf")
        closest_numerator = 0
        closest_denominator = 0

        for denominator in range(1, 101):
            numerator = round(denominator * irrational)
            difference = abs(irrational - numerator / denominator)

            if difference < closest_difference:
                closest_difference = difference
                closest_numerator = numerator
                closest_denominator = denominator

        return closest_numerator, closest_denominator
```

Тесты

(код):

```
# test_fraction.py

from fraction import Fraction

def test_simplify():
```

(continues on next page)

(продолжение с предыдущей страницы)

```

fraction = Fraction(6, 9)
fraction.simplify()
assert fraction.numerator == 2
assert fraction.denominator == 3

def test_compute_gcd():
    fraction = Fraction(15, 25)
    gcd = fraction._compute_gcd(15, 25)
    assert gcd == 5

def test_closest_ratio_to_irrational():
    fraction = Fraction(7, 20)

    # Test with an irrational number of 0.3
    irrational_number = 0.3
    numerator, closest_denominator = fraction.closest_ratio_to_irrational(irrational_
    ↵number)
    assert numerator == 3
    assert closest_denominator == 10

    # Test with an irrational number of 0.8
    irrational_number = 0.8
    numerator, closest_denominator = fraction.closest_ratio_to_irrational(irrational_
    ↵number)
    assert numerator == 4
    assert closest_denominator == 5

    # Test with an irrational number of 0.5
    irrational_number = 0.5
    numerator, closest_denominator = fraction.closest_ratio_to_irrational(irrational_
    ↵number)
    assert numerator == 1
    assert closest_denominator == 2

    # Test with an irrational number of 0.123456789
    irrational_number = 0.123456789
    numerator, closest_denominator = fraction.closest_ratio_to_irrational(irrational_
    ↵number)
    assert numerator == 10
    assert closest_denominator == 81

```

## 20.2.2 Задача 2

Напишите тесты для функций из модуля `string_functions.py`, которые проводят операции со строками.

(код):

```
def reverse_string(text):
    return text[::-1]
```

(continues on next page)

(продолжение с предыдущей страницы)

```

def is_palindrome(text):
    return text == text[::-1]

def count_vowels(text):
    vowels = "aeiouAEIOU"
    return sum(1 for char in text if char in vowels)

def remove_whitespace(text):
    return "".join(text.split())

def remove_extra_spaces(text):
    words = text.split()
    return " ".join(words)

def text_to_lines(text, line_width):
    words = text.split()
    lines = []
    current_words = []
    for word in words:
        if not current_words:
            current_words.append(word)
        elif len(" ".join(current_words)) + len(word) + 1 <= line_width:
            current_words.append(word)
        else:
            lines.append(" ".join(current_words))
            current_words = [word]

    if current_words:
        lines.append(" ".join(current_words))

    return lines

```

**Ответ**

(код):

```

from string_functions import (
    reverse_string,
    is_palindrome,
    count_vowels,
    remove_whitespace,
    remove_extra_spaces
)

def test_capitalize_string():
    result = capitalize_string("hello world")
    assert result == "Hello world"

def test_reverse_string():
    result = reverse_string("hello")
    assert result == "olleh"

```

(continues on next page)

(продолжение с предыдущей страницы)

```

def test_is_palindrome():
    assert is_palindrome("radar")
    assert not is_palindrome("python")

def test_count_vowels():
    assert count_vowels("hello") == 2
    assert count_vowels("Python is awesome") == 6

def test_remove_whitespace():
    result = remove_whitespace(" remove whitespace ")
    assert result == "removewhitespace"

def test_swap_case():
    result = swap_case("Hello, World!")
    assert result == "hELLO, wORLD!"

def test_remove_extra_spaces():
    result = remove_extra_spaces("    This    is    a    test    sentence.    ")
    assert result == "This is a test sentence."

def test_text_to_lines():
    # Test with a short input text and line width
    text = "Hello World!"
    line_width = 8
    result = text_to_lines(text, line_width)
    assert result == ["Hello", "World!"]

    # Test with a long input text and line width
    text = "Lorem ipsum dolor sit amet, consectetur adipiscing elit. Sed nonummy."
    line_width = 20
    result = text_to_lines(text, line_width)
    assert result == [
        "Lorem ipsum dolor",
        "sit amet,",
        "consectetur",
        "adipiscing elit. Sed",
        "nonummy."
    ]
    # Test with a word exceeding the line width
    text = "Supercalifragilisticexpialidocious"
    line_width = 10
    result = text_to_lines(text, line_width)
    assert result == ["Supercalifragilisticexpialidocious"]

    # Test with an empty input text
    text = ""
    line_width = 15
    result = text_to_lines(text, line_width)
    assert result == []

    # Test with a very small line width
    text = "This is a test"

```

(continues on next page)

(продолжение с предыдущей страницы)

```
line_width = 1
result = text_to_lines(text, line_width)
assert result == ["This", "is", "a", "test"]
```

### 20.2.3 Задача 3

Реализуйте функцию `parse_csv` (парсер данных в формате csv). csv (comma-separated-value) — табличный формат, когда значения каждой колонки разделяются запятой. Для реализации можно пользоваться библиотекой `csv` или написать все вручную. Основная цель этого задания — реализовать все случаи, которые покрыты тестами.

(код):

```
def test_parse_csv_empty_string():
    csv_string = ""
    result = parse_csv(csv_string)
    assert result == []

def test_parse_csv_single_row():
    csv_string = "Name,Age,City\nJohn,30,New York"
    result = parse_csv(csv_string)
    expected = [{"Name": "John", "Age": "30", "City": "New York"}]
    assert result == expected

def test_parse_csv_multiple_rows():
    csv_string = "Name,Age,City\nJohn,30,New York\nAlice,25,Chicago"
    result = parse_csv(csv_string)
    expected = [
        {"Name": "John", "Age": "30", "City": "New York"},
        {"Name": "Alice", "Age": "25", "City": "Chicago"},
    ]
    assert result == expected

def test_parse_csv_missing_columns():
    csv_string = "Name,Age,City\nJohn,30"
    result = parse_csv(csv_string)
    expected = [{"Name": "John", "Age": "30", "City": ""}]
    assert result == expected

def test_parse_csv_trailing_comma():
    csv_string = "Name,Age,City\nJohn,30,New York,"
    result = parse_csv(csv_string)
    expected = [{"Name": "John", "Age": "30", "City": "New York"}]
    assert result == expected

def test_parse_csv_extra_whitespace():
    csv_string = " Name , Age , City \n John, 30 , New York"
    result = parse_csv(csv_string)
    expected = [{"Name": "John", "Age": "30", "City": "New York"}]
    assert result == expected
```

(continues on next page)

(продолжение с предыдущей страницы)

```

def test_parse_csv_empty_fields():
    csv_string = "Name,Age,City\nJohn,,New York\nJohn,25,Alice,30,"
    result = parse_csv(csv_string)
    expected = [
        {"Name": "John", "Age": "", "City": "New York"}, 
        {"Name": "", "Age": "25", "City": ""}, 
        {"Name": "Alice", "Age": "30", "City": ""}, 
    ]
    assert result == expected

def test_parse_csv_with_quotes():
    csv_string = 'Name,Age,City\n"John",30,"New York"\n"Alice",25,"Chicago"'
    result = parse_csv(csv_string)
    expected = [
        {"Name": "John", "Age": "30", "City": "New York"}, 
        {"Name": "Alice", "Age": "25", "City": "Chicago"}, 
    ]
    assert result == expected

```

**Ответ**

(код):

```

import csv

def parse_csv(csv_string):
    data = []
    lines = csv_string.strip().split("\n")
    if not lines:
        return data

    reader = csv.reader(lines)
    header = next(reader)

    for row in reader:
        row_dict = {}
        for col_name, value in zip(header, row):
            row_dict[col_name] = value.strip()
        data.append(row_dict)

    return data

```

**20.2.4 Задача 4**

Используйте класс калькулятор.

(код):

/home/artem/swdev/gitrepo/edu/toolchain

```

class Calculator:
    def add(self, a, b):

```

(continues on next page)

(продолжение с предыдущей страницы)

```

    return a + b

def subtract(self, a, b):
    return a - b

def multiply(self, a, b):
    return a * b

def divide(self, a, b):
    if b == 0:
        raise ValueError("Cannot divide by zero.")
    return a / b

```

Напишите тесты, которые используют `@pytest.fixture` вместо того, чтобы создавать экземпляр класса внутри каждого теста.

**Ответ**

(код):

```

# test_calculator.py

import pytest
from calculator import Calculator

# Define the fixture to set up an instance of the Calculator class for testing
@pytest.fixture
def calculator_instance():
    return Calculator()

def test_add(calculator_instance):
    result = calculator_instance.add(3, 4)
    assert result == 7

def test_subtract(calculator_instance):
    result = calculator_instance.subtract(10, 5)
    assert result == 5

def test_multiply(calculator_instance):
    result = calculator_instance.multiply(2, 3)
    assert result == 6

def test_divide(calculator_instance):
    result = calculator_instance.divide(10, 2)
    assert result == 5

def test_divide_by_zero(calculator_instance):
    with pytest.raises(ValueError):
        calculator_instance.divide(10, 0)

def test_divide_float(calculator_instance):
    result = calculator_instance.divide(5, 0.1)
    assert result == pytest.approx(50)

```

(continues on next page)

(продолжение с предыдущей страницы)

```
if __name__ == '__main__':
    pytest.main()
```

## 20.3 Вопросы

1. Какой тип тестирования направлен на проверку правильности отдельных функций или методов? а) Модульное тестирование б) Интеграционное тестирование в) Системное тестирование г) Регрессионное тестирование
2. Какая библиотека тестирования Python предоставляет удобный способ для написания тестовых случаев, fixture и других необходимых для тестирования вещей? а) unittest б) pytest в) nose2 г) doctest
3. Какова цель контрольного примера (теста) в тестировании программного обеспечения? а) Выявить и исправить синтаксические ошибки в коде б) Проверить функциональность всего приложения в) Задокументировать требования к программному обеспечению г) Определить конкретные входные данные, действия и ожидаемые результаты для тестирования конкретной случая или функции
4. Какова роль `assertion` в тестировании кода (python)? а) Для обработки исключений и ошибок в коде б) Для генерации случайных тестовых данных в) Чтобы проверить, соответствует ли фактический результат ожидаемому результату г) Чтобы измерить покрытие кода
5. Что измеряет покрытие кода при тестировании программного обеспечения? а) Время, необходимое для выполнения тестового примера б) Процент кода, проверенного коллегами в) Количество выполненных тестовых случаев г) Процент кода, который был протестирован тестовыми примерами
6. Каков основной принцип разработки через тестирование (TDD)? а) Написание тестов после написания кода, чтобы проверить его правильность б) Написание тестов перед написанием кода, чтобы определить желаемое поведение в) Написание тестов только для критических частей кода г) Написание тестов в конце цикла разработки (после реализации всех модулей)
7. Каковы преимущества автоматизации тестирования в тестировании программного обеспечения? а) Сокращение количества тестовых случаев, необходимых для полного покрытия кода б) Полное устранение необходимости в ручном тестировании в) Обеспечение того, чтобы все дефекты были идентифицированы и исправлены г) Повышение скорости и эффективности процессов тестирования
8. В чем заключается тестировании методом черного ящика для разработки тестов? а) Тестовые случаи выводятся из внутренней структуры кода б) Тестовые случаи основаны на знании реализации кода в) Тестовые примеры получены из спецификаций и требований к продукту без знания реализации кода г) Тестовые случаи предназначены для проверки производительности и масштабируемости приложения

### Ответы

1. а)
2. б)
3. г)
4. в)
5. г)

- 6. б)
- 7. г)
- 8. в)

## Логгирование работы приложения

---

### 21.1 Логирование. Что это и в чем его польза?

Что значит логирование? Логирование (или журнал событий) является важным инструментом при разработке программного обеспечения. Логирование нужно нам для того, чтобы записать шаги и результат выполнения различных участков приложения для дальнейшего использования с целью анализа или поиска ошибок.

#### 21.1.1 Мотивация

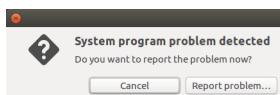
При использовании логирования мы можем не гадать, что именно произошло в нашем приложении, а опереться на конкретную информацию. При этом нужно помнить, что переизбыток информации также вреден, как и ее недостаток: в куче разнородных сообщений, часть из которых являются бесполезными, а часть избыточными, ошибку найти еще можно, а вот причины, которые к ней привели, уже гораздо сложнее. Речь идет о записи специального текстового файла (лога) с особым образом структурированной информацией о работе программы. Каждое сообщение может сопровождаться метками, самая простая из которых — метка времени, для удобного поиска и группировки.

#### 21.1.2 Зачем нужно логирование?

Инструменты для логирования можно рассматривать как отдельную систему при разработке ПО, при этом она не менее важна, чем, например, система контроля версий. При этом системами контроля версий принято пользоваться почти всегда, а системе логирования не уделяется достаточного внимания, так как ее поддержка в коде приложения (все это дополнительные строчки кода) не несет дополнительного функционала, если все идет по плану. Однако логи очень полезны. Если все пошло не так, как должно, и если их нет, единственный способ найти ошибку — работать с пользователем и воспроизводить проблему.

Существует несколько сценариев, в которых ведение логов необходимо:

- Отладка** Логи помогают разработчикам понимать и диагностировать проблемы в их ПО, предоставляя подробную информацию о том, что происходит внутри приложения. Особенно это полезно, когда приложение работает с данными от пользователей, когда невозможно предусмотреть все возможные ситуации заранее. Эта информация может быть использована для выявления и исправления ошибок. Всем знакомо подобное окно, когда происходит ошибка, которую не предусмотрел разработчик. В таком окне обычно есть кнопка «Сообщить о проблеме», если ее нажать, разработчику уйдет сообщение об ошибке и часть логов.



*Примечание об ошибках, появление которых предусмотрено, можно сообщить текстом и предложить варианты исправления. Это можно также отправить в логи*

Посмотреть когда-либо отправленные системой сообщения об ошибках на Ubuntu можно, выполнив команду в терминале:

```
xdg-open https://errors.ubuntu.com/user/`sudo cat /var/lib/whoopsie/whoopsie-id`
```

- Мониторинг** Этот сценарий предполагает использование логов для выявления процессов, которые не связаны с ошибками, но могут означать другую проблему. Например, если нагрузка на сервер (число одновременных подключений) выросла в 10 раз за короткий промежуток времени. Возможно, происходит какая-то атака. Так логи могут использоваться для мониторинга работоспособности, а также для выявления потенциальных проблем. Логи также можно использовать для отслеживания состояния важных процессов, например, миграции базы данных или обновления системы.
- Proof of operation** В некоторых случаях ведение журнала логов просто необходимо для демонстрации работоспособности приложения и требуется в соответствии с нормативными актами или отраслевыми стандартами. Логи могут служить доказательством соответствия требованиям и использоваться для демонстрации того, что ПО работает согласно установленным политикам и процедурам.
- Аудит** Логи могут использоваться в целях аудита, чтобы помочь убедиться, что система используется правильно, а также для обнаружения и предотвращения несанкционированного доступа или неправильного использования. В данном случае мы можем хранить время входа в систему, локацию, откуда произведен вход, и, например, выяснить, не пользуется ли часть сотрудников ключами других.

### 21.1.3 Уровни логирования

Заниматься логированием необходимо с самого начала разработки, чтобы во время последующей эксплуатации мы могли собирать наиболее полную информацию о работе ПО. Даже при хорошо продуманном логировании в файлах-логах копится очень много информации. Чтобы систематизировать важную информацию и упростить ее последующее использование, существуют уровни логирования. Уровни логирования предусмотрены не только в качестве меток для поиска и выборки событий, но и используются самой библиотекой логирования для того, чтобы можно было оставить вывод только логов определенного уровня. Всего выделяют 5 основных:

- **Trace** Уровень *Trace* используется для регистрации самых мелких событий, которые позволяют отследить каждый шаг алгоритма или другую подобную информацию. Как правило, это не требуется, поскольку алгоритмы обычно хорошо известны и покрываются тестами. Но если вам нужен очень подробный отчет по каждой строке кода, то вам сюда. Не забудьте запастись достаточным дисковым пространством для файлов-логов.

- **Debug** Уровень *Debug* используется для регистрации подробной информации, которая в основном предназначена для разработчиков. Сообщения на этом уровне менее подробны, чем на *Trace*, но не настолько, чтобы быть полезны кому-то кроме разработчиков. Отладочные сообщения обычно используются на этапе разработки и тестирования, чтобы помочь понять, что происходит внутри системы. При сборке проекта для пользователей этот уровень отключают, и события уровня *Debug* не собираются на машинах пользователей.
- **Info** Уровень *Info* используется для регистрации информации о нормальных, ожидаемых событиях. Информационные сообщения могут дать представление о поведении системы и использоваться в целях мониторинга. Например, так можно собирать количество подключений от пользователя за определенный промежуток времени, например, 1 час.
- **Warning** Уровень *Warning* используется для регистрации необычных или неожиданных событий, но необязательно ошибок. Предупреждения могут указывать на потенциальные проблемы, которые необходимо устранить и которые уже могут привести к более серьезным последствиям. Например, падение скорости передачи данных ниже уровня, когда это начинает влиять на работу приложения.
- **Error** Уровень *Error* используется для регистрации событий, которые представляют собой сбой или непредвиденную ситуацию. Сообщения об ошибках могут указывать на проблему, которую необходимо устранить, например, падение канала связи так, что данные пользователя вообще не будут передаваться.
- **Fatal** Уровень *Fatal* используется для регистрации наиболее серьезных событий, таких как полный сбой системы или критическая проблема безопасности. Критические сообщения указывают на то, что требуется немедленное внимание для предотвращения или устранения серьезной проблемы. К таким событиям можно отнести получение файла с данными, у которого не сходится контрольная сумма, это может привести не только к неудобству пользователей, но и к потерям со стороны компании-разработчика.

Благодаря разным уровням ведения журнала программное обеспечение для ведения журнала предоставляет способ фильтровать регистрируемые события, что облегчает фокусировку на наиболее важных событиях.

#### 21.1.4 Основы грамотного логирования

Чтобы получить файлы логирования, которые будут удобными в последующей работе, следует грамотно подойти к процессу их создания:

- Логировать все важные события, будь то запуск приложения, остановка или возобновление транзакций, успешная регистрация, открытие на чтение файла, ошибки.
- Исключить из описания события часто повторяющиеся слова, что существенно упростит поиск нужной информации в будущем.
- Добавлять к логам теги: переходя по ним, вы сможете находить данные, даже если они находятся в разных категориях. Теги могут относиться как бизнес-логике (ПРОДАЖА, РЕГИСТРАЦИЯ), так и внутренним процессам в коде (ПОДКЛЮЧЕНИЕ\_К\_БД, ЧТЕНИЕ\_ФАЙЛА), и любые другие осмысленные варианты.
- Использовать только нужную информацию: лишние сведения будут только засорять логи и делать работу с ними медленнее даже при автоматической обработке событий.
- Использовать типы и уровни логирования осознанно. Это упрощает использование журнала логов для последующей аналитики и визуализации.

Следуя этим основам грамотного ведения журнала программного обеспечения, можно гарантировать, что данные журнала собираются так, чтобы облегчать отладку и мониторинг вашего ПО.

### 21.1.5 Библиотеки логирования для языка программирования Python

Существует несколько популярных библиотек ведения журнала для Python:

#### logging

Встроенный модуль ведения журнала в стандартной библиотеке Python — это гибкая и мощная библиотека ведения логов, которая предоставляет множество функциональных возможностей «из коробки». (код):

```
import logging
import sys
import time
from logging.handlers import TimedRotatingFileHandler

FORMATTER_STRING = "%(asctime)s - %(name)s - %(levelname)s - %(message)s"
FORMATTER = logging.Formatter(FORMATTER_STRING)
LOG_FILE = "/tmp/my_app.log" # use fancy libs to make proper temp file

def get_logger(logger_name):
    logger = logging.getLogger(logger_name)
    logger.setLevel(logging.INFO)

    console_handler = logging.StreamHandler(sys.stdout)
    console_handler.setFormatter(FORMATTER)
    logger.addHandler(console_handler)

    file_handler = TimedRotatingFileHandler(LOG_FILE, when='midnight')
    #file_handler.setFormatter(FORMATTER)
    logger.addHandler(file_handler)

    return logger

if __name__ == "__main__":
    logger = get_logger("my_app_logger")
    logger.info("Start logging")
    logger.debug("Some debug message")
    while True:
        try:
            time.sleep(1)
            logger.info("Keep logging")
        except KeyboardInterrupt:
            logger.fatal("User get bored")
            break
```

Давайте поймем, что здесь происходит. Мы создаем логгер и выводим логи на уровне INFO в файл и в терминал. В файле логи записываются без форматирования. Выводим сообщение о критической ошибке, когда пользователь нажмет Ctrl-C и прервет выполнение программы. Обратите внимание, что сообщения уровня DEBUG не выводятся, так как мы установили уровень для вывода не ниже INFO logger.setLevel(logging.INFO). Ниже описаны объекты, которые мы используем в коде:

- FORMATTER — определяет, как будет выглядеть строка события в логах. В него мы передаем шаблон, который хотим реализовать.

- LOG\_FILE — определяет, в какой файл мы будем писать логи. По умолчанию мы выводим в стандартный вывод stderr, и если хотим видеть логи на экране и сохранять в файл, должны явно это указать.
- get\_logger() — создает объект класса Logger, который мы будем использовать для логирования.
- console\_handler — обработчик вывода в терминал, по умолчанию используется stderr, поэтому явно говорим, что хотим видеть сообщения в stdout.
- file\_handler — обработчик вывода логов в файл, обратите внимание, что мы закомментировали форматирование, поэтому сообщения будут записаны в стандартной форме.
- logger.info() — используем для вывода логов на уровне INFO. Другие уровни имеют соответствующие функции для вывода.

Запустим приложение и увидим вывод:

```
artem@pc:~$ python ./logging_sample.py
2023-02-14 10:05:40,074 - my_app_logger - INFO - Start logging
2023-02-14 10:05:41,075 - my_app_logger - INFO - Keep logging
2023-02-14 10:05:42,076 - my_app_logger - INFO - Keep logging
^C2023-02-14 10:05:42,392 - my_app_logger - CRITICAL - User get bored
```

В файле мы видим те же самые сообщения, но не снабженные никакой дополнительной информацией:

```
artem@pc:~$ cat /tmp/my_app.log
Start logging
Keep logging
Keep logging
User get bored
```

## structlog

Это библиотека расширяет встроенный модуль Python для ведения логов с дополнительными функциями, такими как возможность прикреплять дополнительные данные к сообщениям или манипулировать сообщениями до их отправки.

Рассмотрим слегка модифицированный пример с сайта structlog:

(код):

```
import logging
import sys
import structlog

FORMATTER_STRING = "%(asctime)s - %(name)s - %(levelname)s - %(message)s"
FORMATTER = logging.Formatter(FORMATTER_STRING)

def get_logger(logger_name):
    logger = logging.getLogger(logger_name)
    logger.setLevel(logging.INFO)
    console_handler = logging.StreamHandler(sys.stdout)
    console_handler.setFormatter(FORMATTER)
    logger.addHandler(console_handler)
    return logger
```

(continues on next page)

(продолжение с предыдущей страницы)

```

def replace_user(_, __, event_dict):
    user = event_dict.get("user")
    if user:
        # we can access data base here for user_token
        # for now keep it fake
        user_token = "some_string_that_we_can_learn_username_from"
        event_dict["user"] = user_token
    return event_dict

def censor_password(_, __, event_dict):
    pw = event_dict.get("password")
    if pw:
        event_dict["password"] = "*CENSORED*"
    return event_dict

log = structlog.wrap_logger(
    get_logger("my_app_logger"),
    processors=[
        censor_password,
        replace_user,
        structlog.processors.JSONRenderer(indent=1, sort_keys=True),
    ],
)
log.warning("something", password="secret")
log.warning("something", user="Ivan")

```

Здесь мы используем `get_logger()` как раньше. Убрали вывод в файл, так как он не важен для демонстрации. Сделали обертку для логгера и определили, что должно происходить с данными, если они передаются для логирования. Мы видим, что здесь нам необязательно определять строку форматирования для всех данных, данные передаются по ключам, и это дает нам возможность получить читаемый вывод и сделать что-нибудь со значениями до вывода. В примере ниже мы хотим заменить имена на что-то, что позволит нам восстановить эти имена, например, если мы держим соответствие имен и некоторых токенов (если требования к личным данным не позволяют нам публиковать имена людей). Пароли мы хотим скрыть полностью. Для манипуляции с данными в логгер от `structlog` мы можем определить несколько *обработчиков* (*processors*), в которых описано то, что мы хотим сделать.

- `censor_password` — функция-обработчик, которая скроет пароль, если он присутствует в данных, заменив на `*CENSORED*`.
- `replace_user` — функция-обработчик, которая заменит имя пользователя.

```

2023-02-14 10:53:42,214 - my_app_logger - WARNING - {
  "event": "something",
  "password": "*CENSORED*"
}
2023-02-14 10:53:42,214 - my_app_logger - WARNING - {
  "event": "something",
  "user": "some_string_that_we_can_learn_username_from"
}

```

## loguru

Эта библиотека сделает ваш журнал более простым в использовании. Loguru предоставляет простой и интуитивно понятный интерфейс для протоколирования сообщений и поддерживает расширенные функции, такие как фильтрация журналов, цветной вывод и обработка исключений.

Пример из статьи позволяет понять возможность loguru:

(код):

```
import sys
from loguru import logger

logger.remove(0)
logger.add(sys.stderr, backtrace=True, diagnose=True)

def func(a, b):
    return (a / b) + func(a, b-1)

def nested(c):
    try:
        func(5, c)
    except ZeroDivisionError:
        logger.exception("Division by zero error!")

nested(1)
```

При запуске мы получим такой вывод:

```
File "/home/artem/edu/kd_foss/logging/code/loguru_sample.py", line 8, in func
    return (a / b) + func(a, b-1)
           ^ 1
           |  1
           |  |
           |  5
           |      <function func at 0x7fce125fb760>
           |
           5

File "/home/artem/edu/kd_foss/logging/code/loguru_sample.py", line 8, in func
    return (a / b) + func(a, b-1)
           ^ 0
           |  0
           |  |
           |  5
           |      <function func at 0x7fce125fb760>
           |
           5

ZeroDivisionError: division by zero
(kd_foss) artem@pc:~/edu/kd_foss/logging/code$
```

## logbook

Это библиотека ведения логов, которая разработана таким образом, чтобы быть быстрой и эффективной, и предоставляет такие функции, как ведение логов с учетом контекста, гибкое форматирование журнала и поддержка различных уровней ведения журнала.

Как видно, все библиотеки имеют почти одинаковый функционал. Лучшая библиотека ведения журнала для вашего проекта будет зависеть от ваших конкретных потребностей.

### 21.1.6 Централизация логов

Централизация логов относится к практике сбора данных журнала из нескольких источников и хранения их в центральном хранилище. Это облегчает управление всей информацией и ее отслеживание, так что вы можете использовать ее для решения проблем и убедиться, что все работает правильно. Наличие всех логов в одном месте также повышает безопасность и помогает уберечь информацию от потери. И это также может ускорить и упростить поиск и устранение проблем, потому что вам не нужно просматривать информацию из множества разных мест.

Существуют различные способы централизации логов, но цель всегда состоит в том, чтобы иметь всю информацию в одном месте, к которому легко получить доступ и которым можно пользоваться.

Существует несколько способов централизации логов, включая использование централизованных систем ведения журнала, таких как Elastic Stack, Logstash или Graylog, или с помощью облачных служб ведения журнала, таких как Google Stackdriver. Наилучший подход к централизации данных журнала будет зависеть от конкретных потребностей.

Мы советуем обратить внимание на стек ELK для централизации логов.

### 21.1.7 Вывод

Подведем итог: ведение логов является важнейшим компонентом разработки и эксплуатации программного обеспечения, которое дает ценную информацию. Логирование дает возможность быстро находить проблемы и их источники, устранять их, выявлять конфликты в конфигурационных файлах и следить за безопасностью.

## 21.2 Задача

### 21.2.1 Задача 1

Используйте код ниже и снабдите его логами уровней SUCCES, DEBUG, INFO. Используйте loguru. Сделайте так, чтобы для вызова функций с разными аргументами сообщения отличались, т.е. были информативными с точки зрения того, что по ним можно восстановить детали работы приложения. Сообщения вида Sum calculated не являются информативными, так как каждый вызов функции приведет к одному и тому же сообщению, по которым можно восстановить только время вызова, другая информация не сохраняется, что будет неудобно при анализе логов.

```
def calculate_sum(a, b):
    return a + b

def main():
    num1 = 10
    num2 = 20
```

(continues on next page)

(продолжение с предыдущей страницы)

```

result = calculate_sum(num1, num2)

if __name__ == "__main__":
    main()

```

**Ответ**

(код):

```

import loguru

# Configure loguru to write logs to a file named 'app.log' with a maximum size of 1 MB
# per file
loguru.logger.add("app.log", rotation="1 MB")

def calculate_sum(a, b):
    result = a + b
    loguru.logger.debug(f"Sum calculated: {a} + {b} = {result}")
    return result

def main():
    num1 = 10
    num2 = 20

    # Log the beginning of the calculation
    loguru.logger.info(f"Starting the calculation of {num1} + {num2}")

    result = calculate_sum(num1, num2)

    # Log the result
    loguru.logger.success(f"Calculation successful! The sum of {num1} and {num2} is:
                           {result}")

if __name__ == "__main__":
    main()

```

**21.2.2 Задача 2**

Используйте код ниже и снабдите его логами уровней DEBUG, INFO, WARNING и ERROR. Не всегда появление `exception` в программе должно приводить к легированию уровня ERROR. Мы можем использовать другие уровни в зависимости от того, как обрабатываются исключения. Подумайте, какие еще проверки можно внести, например, мы не проверяем, что `data` в `process_data` является строкой, хотя тело функции это подразумевает.

```

def load_data(file_path):
    try:
        with open(file_path, 'r') as file:
            data = file.read()
            return data
    except FileNotFoundError:

```

(continues on next page)

(продолжение с предыдущей страницы)

```

        return None
    except Exception as e:
        return None

def process_data(data):
    if not data:
        return

    # Perform some data processing here
    processed_data = data.upper()
    return processed_data

def save_data(file_path, data):
    try:
        with open(file_path, 'w') as file:
            file.write(data)
    except Exception as e:
        pass

def main():

    file_path = "data.txt"
    data = load_data(file_path)
    if data:
        processed_data = process_data(data)
        if processed_data:
            save_data("processed_data.txt", processed_data)

if __name__ == "__main__":
    main()

```

**Ответ**

(код):

```

import loguru

loguru.logger.add("app.log", rotation="1 MB")

def load_data(file_path):
    try:
        with open(file_path, 'r') as file:
            data = file.read()
            loguru.logger.debug(f"Data loaded from {file_path}")
            return data
    except FileNotFoundError:
        loguru.logger.warning(f"File not found: {file_path}")
        return None
    except Exception as e:
        loguru.logger.error(f"Error loading data from {file_path}: {e}")
        return None

def process_data(data):
    if not data:

```

(continues on next page)

(продолжение с предыдущей страницы)

```

    loguru.logger.error("No data to process!")
    return

processed_data = data.upper()
data_len = len(data)
log_size = min(data_len / 2, 20)
loguru.logger.debug(f"Data {data[:log_size]} ... {data[log_size:]} processed"
˓→successfully")
return processed_data

def save_data(file_path, data):
    try:
        with open(file_path, 'w') as file:
            file.write(data)
            loguru.logger.info(f"Data saved to {file_path}")
    except Exception as e:
        loguru.logger.error(f"Error saving data to {file_path}: {e}")

def main():
    loguru.logger.info("Starting the program")

    file_path = "data.txt"
    data = load_data(file_path)
    if data:
        processed_data = process_data(data)
        if processed_data:
            save_data("processed_data.txt", processed_data)

    loguru.logger.info("Program execution completed")

if __name__ == "__main__":
    main()

```

### 21.2.3 Задача 3

Модифицируйте код ниже так, чтобы была информация о пользователе, который сделал запрос, а также hash переданных данных. Напишите логирование для полученного кода.

```

from flask import Flask, request, jsonify

app = Flask(__name__)

def process_request(request_data):
    try:
        # Simulate some request processing
        result = request_data.get('data', 0) * 2
        return result
    except Exception as e:
        return None

@app.route('/api/data', methods=['POST'])
def handle_request():

```

(continues on next page)

(продолжение с предыдущей страницы)

```

try:
    request_data = request.get_json()

    if not request_data or 'data' not in request_data:
        return jsonify({'error': 'Invalid request data'}), 400

    result = process_request(request_data)
    if result is not None:
        return jsonify({'result': result}), 200
    else:
        return jsonify({'error': 'Error processing request'}), 500

except Exception as e:
    return jsonify({'error': 'Internal server error'}), 500

if __name__ == "__main__":
    app.run(debug=True)

```

**Ответ**

(код):

```

import loguru
import hashlib
from flask import Flask, request, jsonify

# Configure loguru to write logs to a file named 'web_app.log' with a maximum size of 1MB per file
loguru.logger.add("web_app.log", rotation="1 MB")

app = Flask(__name__)

def compute_data_hash(data):
    data_str = str(data).encode('utf-8')
    return hashlib.sha256(data_str).hexdigest()

def process_request(request_data, user_id, data_hash):
    try:
        # Simulate some request processing
        result = request_data.get('data', 0) * 2
        loguru.logger.debug(f"User ID: {user_id} | Data Hash: {data_hash} | Request processed successfully. Result: {result}")
        return result
    except Exception as e:
        loguru.logger.error(f"User ID: {user_id} | Data Hash: {data_hash} | Error processing request: {e}")
        return None

@app.route('/api/data', methods=['POST'])
def handle_request():
    try:
        request_data = request.get_json()
        user_id = request_data.get('user_id', None)

```

(continues on next page)

(продолжение с предыдущей страницы)

```

loguru.logger.info(f"User ID: {user_id} | Received a request")
loguru.logger.debug(f"User ID: {user_id} | Request data: {request_data}")

if not request_data or 'data' not in request_data or not user_id:
    loguru.logger.warning(f"User ID: {user_id} | Data Hash: N/A | Invalid request data received")
    return jsonify({'error': 'Invalid request data'}), 400

data_hash = compute_data_hash(request_data)
result = process_request(request_data, user_id, data_hash)

if result is not None:
    loguru.logger.success(f"User ID: {user_id} | Data Hash: {data_hash} | Request processed successfully")
    return jsonify({'result': result}), 200
else:
    loguru.logger.error(f"User ID: {user_id} | Data Hash: {data_hash} | Error occurred while processing the request")
    return jsonify({'error': 'Error processing request'}), 500

except Exception as e:
    loguru.logger.critical(f"User ID: N/A | Data Hash: N/A | Unhandled exception: {e}")
return jsonify({'error': 'Internal server error'}), 500

if __name__ == "__main__":
    loguru.logger.info("Starting the web application")
    app.run(debug=True)

```

## 21.3 Вопросы

- Что такое ведение журнала событий (логов) в контексте разработки программного обеспечения?  
а) Процесс отслеживания действий пользователей на веб-сайте. б) Метод измерения производительности кода. в) Метод записи и хранения событий и сообщений во время выполнения программы. г) Способ оптимизации использования памяти в приложениях.
- Какая библиотека входит в стандартную поставку Python и используется для ведения журнала?  
а) logbook б) logging в) loguru г) structlog
- Каково назначение уровней логирования? а) Определять различные цвета для сообщений журнала. б) Группировать сообщения журнала на основе их содержания. в) Установить размер шрифта сообщений журнала. г) Классифицировать сообщения журнала по их серьезности.
- Какой уровень журнала используется для получения общей информации о потоке выполнения программы?  
а) DEBUG б) INFO в) WARNING г) ERROR
- Как вы можете настроить логгер библиотеки loguru для событий в файл с именем «app.log»?  
а) loguru.logger.write("app.log") б) loguru.logger.create("app.log") в) loguru.logger.add("app.log") г) loguru.logger.output("app.log")
- Как в библиотеке loguru можно установить ротацию (rotation) для лог-файлов?  
а) loguru.logger.rotation = "ежедневно" б) loguru.logger.rotate("ежедневно") в) loguru.logger.add\_rotation("ежедневно") г) loguru.logger.add("app.log", rotate="daily")

7. Какова цель форматирования журнала в системе регистрации? а) Для сжатия файлов журналов для повышения эффективности хранения. б) Чтобы обеспечить согласованную структуру для регистрации сообщений. в) Для шифрования сообщений журнала из соображений безопасности. г) Чтобы удалить ненужные записи журнала.
8. В контексте ведения журнала, в чем разница между уровнями журнала «INFO» и «DEBUG»?  
а) «DEBUG» используется для получения подробной низкоуровневой информации, а «INFO» используется для получения общей информации о потоке выполнения программы. б) «INFO» используется для получения подробной низкоуровневой информации, а «DEBUG» используется для получения общей информации о потоке выполнения программы. в) И «INFO», и «DEBUG» используются для получения общей информации о потоке выполнения программы, но «DEBUG» менее серьезен, чем «INFO». г) И «INFO», и «DEBUG» используются для получения подробной низкоуровневой информации, но «INFO» менее серьезен, чем «DEBUG».

**Ответы:**

1. в)
2. б)
3. г)
4. б)
5. в)
6. г)
7. б)
8. а)

## Сборка проекта в pip пакет

---

### 22.1 Менеджеры пакетов

Мы уже умеем размещать свой код на платформах типа GitHub, для того чтобы контролировать версию кода и иметь возможность взять код себе на машину локально, отправить свои правки, получать красивую визуализацию коммитов, а также получать обратную связь от других разработчиков в виде `issue`, `fork`, `merge request` и так далее. Попробуем теперь воспользоваться кодом не как программисты-разработчики, а как программисты-пользователи. Да, каждый раз когда мы пишем:

```
import lib_name  
lib_name.do_something()
```

Мы являемся пользователем функции `do_something()` из библиотеки `lib_name`. Как же нам получить `do_something()` там, где мы пишем код, и воспользоваться ей? Рассмотрим сначала простые, но неправильные способы, и потом рассмотрим, как сделать это правильно.

#### 22.1.1 Ctrl-C/Ctrl-V

Первый наивный способ — это просто скопировать функцию себе и вызвать там, где нужно (не делайте так).

Почему это плохо:

- теряется поддержка. Если разработчики найдут проблему в коде и устранит его, мы не получим эти изменения, если не находимся в режиме мониторинга версий этого пакета;
- тянем кучу другого кода. Скорее всего, у вас даже не получится так просто скопировать отдельную функцию, так как она использует другой код из этого пакета;
- теряется контроль зависимостей. Если для работы функций необходима другая библиотека определенной версии, это никак не отображается в коде, `import matplotlib` подключит ту библиотеку, которая установлена в окружении. Установка `pip install matplotlib` также ничего не даст — будет установлена последняя версия библиотеки `matplotlib`. Правильная версия библиотеки

должна располагаться в правильно структурированном репозитории, и чтобы ее найти, нужно совершить дополнительное действие;

- нет возможность запустить тесты. Как правило, в библиотеках содержатся тесты, которые можно запустить и проверить, все ли корректно работает именно на этой машине. Просто копируя функцию, мы не подтягиваем автоматически тесты.

### 22.1.2 Клонируем и указываем путь через sys

Мы можем сделать клон репозитория:

```
git clone url
```

и установить все необходимые библиотеки, используемые в проекте, в том случае, если таковые имеются. Для этого в репозитории помимо кода можно найти файл `requirements.txt`, который предназначен для перечисления названий пакетов (библиотек) с указанием их версий. Таким образом, вы сможете использовать точные версии всех библиотек, которыми пользовался разработчик. После того, как вы клонировали репозиторий, в котором присутствует файл `requirements.txt`, вам следует установить перечисленные в нем библиотеки. Это можно выполнить, используя менеджер пакетов `pip`, командой:

```
pip install -r requirements.txt
```

Если нам повезет, там будут указаны версии библиотек, для которых разработчик тестировал работоспособность этого пакета. Также мы можем прогнать тесты и убедиться, что все работает как надо. После этого мы можем указать путь до библиотеки прямо в коде (актуально для Python 3.5+):

```
from importlib.util import (spec_from_file_location, module_from_spec)
import sys

spec = spec_from_file_location("module.name", "/path/to/my_package/my_module.py")
foo = module_from_spec(spec)
sys.modules["module.name"] = foo
spec.loader.exec_module(foo)
foo.MyClass()
```

или так:

```
import sys
sys.path.append("/path/to/my_package")
import my_module
my_module.MyClass()
```

Уже лучше, но по-прежнему мы должны помнить, что нужно проверять новые версии, процесс установки полностью ручной, а то, что мы написали в коде, выглядит как костыль. Мы также можем добавить путь к пакету в `PYTHONPATH`, используя `bash`:

```
export PYTHONPATH='/path/to/my_package'
```

Проверить, что переменная установилась, можно, вызвав команду `echo $PYTHONPATH`. После этого можно запустить интерпретатор Python и проверить, что все работает, и что путь уже добавлен в `sys.path`

### 22.1.3 Ставим через pip, используя git репозиторий

Мы добрались до более распространенных способов управления зависимостями. Здесь мы уже используем pip и публично доступный репозиторий на GitHub для установки. Примечание: мы его использовали в предыдущем пункте, но чтобы поставить зависимости устанавливаемой библиотеки, а не саму библиотеку.

Прежде чем мы перейдем к тому способу, которым мы привыкли пользоваться:

```
pip install lib_name
```

Проговорим еще раз, что существует такой способ. Если вы пропустили предыдущий урок, вернитесь и посмотрите, что должно содержаться в проекте для того, чтобы другие разработчики могли им воспользоваться. Не для всех языков программирования есть аналог pip, а понимание того, что в репозитории может содержаться вся необходимая информация для того, чтобы установить/собрать библиотеку, поможет легче найти решение проблемы установки для других языков.

#### Немного про pip

Для инфраструктуры python менеджер пакетов pip является стандартом де facto, хотя появился с текущим именем не так [давно](#). Концепция менеджеров пакетов удобна для программиста, поэтому некоторые языки программирования и фреймворки реализуют ее для управления зависимостями:

- JavaScript: npm
- Ruby: gem
- .NET: NuGet

Для C++ существует проекты, которые направлены на решение этой проблемы:

- bpt
- cpm
- conan
- poac
- pacm
- spack
- buckaroo
- hunter
- vcpkg

В Python также существуют альтернативы pip, об одной из них мы поговорим ниже.

Что же делает менеджер пакетов? Первый очевидный ответ: он кладет (а перед этим еще и скачивает их сам из сети) файлы библиотеки в правильное место: там, где его сможет найти язык программирования, чтобы импортировать.

Второе свойство менеджера пакетов: он хранит историю версий библиотеки так, чтобы мы могли установить нужную нам. *Примечание: не путайте версию библиотеки и версию в плане ссылки на конкретный коммит в git; не каждый коммит ведет к обновлению версии библиотеки в менеджере проектов.* Кроме того, менеджер проектов знает, какие зависимости нужны именно этой версии библиотеки, таким образом, у нас есть дерево зависимостей.

Третье: менеджер проектов дает возможность прогнать тесты при установке так, чтобы мы были уверены, что сделали все возможное, чтобы получить работающую библиотеку. При этом никто не гарантирует, что тесты полностью покрывают все возможные ситуации, но часть из них — точно, и всегда неплохая идея проверить хотя бы их.

### 22.1.4 Делаем сборку библиотеки для публикации. `pip`

Когда нам нужна функциональность какой-либо библиотеки в Python, скорее всего, мы ставим ее через `pip`:

```
pip install lib_name
```

Когда мы это делаем, `pip` просматривает публичный репозиторий пакетов Python Package Index (PyPI). Мы также можем [хостить](#) зеркало PyPI, если нам это для чего-то нужно. Если `lib_name` находится в PyPI, будет выполнена попытка установки. В PyPI расположен сам пакет, но в чем же отличие от установки через связку `pip+git`? Библиотека может распространяться в виде исходных текстовых файлов **Source Distribution (sdist)** или собранных файлов **Built Distribution (bdist)**. Мы будем использовать `sdist` и `bdist` далее по тексту.

Процесс создания `sdist` проще, чем `bdist`, и на самом деле, мы уже это проделывали. Для этого вида нужно, чтобы весь исходный код, а также другие необходимые файлы были собраны в одном месте. Среди файлов должен быть `setup.py`, в котором содержатся инструкции для `setuputils`, как правильно собрать пакет. Мы можем создать дистрибутив `sdist`, запустив:

```
python setup.py sdist
```

*Напоминание: ранее мы использовали `python setup.py install` для установки.* По умолчанию дистрибутив будет сгенерирован в виде тар-архива `()`. Другие варианты сжатия можно указать при сборке.

```
python setup.py sdist --formats=zip,gztar,bztar,ztar,tar
```

После установки `sdist` файл `setup.py` запускается на хосте, обеспечивая правильную установку этого пакета. Таким образом, мы при `sdist` получаем исходные тексты программ, которые собираются на нашей системе.

Второй тип, `bdist`, немного сложнее для разработчика, так как он собирает пакет до того, как опубликовать его. Это убирает необходимость собирать пакет на стороне пользователя, так как он уже получит собранные библиотеки (`.pyc`, `.so`, `.dll`) и может использовать их сразу. На данный момент существуют два основных формата распространения для `bdist` — `eggs` и `wheels`, более современным считается `wheels`.

```
python setup.py bdist_wheel
```

#### Преимущества `bdist (wheels)`:

- меньший объем пакета и более быстрая установка пакетов Python или пакетов с расширениями на языке C;
- при сборке из исходных файлов (`sdist`) требуется выполнение произвольного кода для сборки (то, что записано в `setup.py`), что может быть не всегда приемлемо с точки зрения безопасности;
- по умолчанию `pip` пытается поставить `bdist`.

Помните о том, что `bdist` является предпочтительным для распространения своего пакета, публикуйте также `sdist`, так как такое распространение позволяет собирать проект под системы, для которых не предоставлен `bdist`.

## Настраиваем окружение

Возьмем проект `mtracker` с предыдущего занятия. Переключимся на ветку `pypi_ready`, чтобы не ломать основной код (и, конечно, еще раз потренируемся с `git`):

```
git clone https://github.com/standlab/mtracker.git
git checkout -b pypi_ready
```

Добавим несколько нововведений. Ранее мы не использовали никаких дополнительных библиотек, `install_requires = []`. Сейчас мы хотим добавить библиотеку `matplotlib` в зависимости к проекту. Мы будем использовать виртуальное окружение с помощью `pipenv` (о виртуальных окружениях есть отдельный урок). `pipenv` комбинирует функциональность виртуального окружения, которое можно получить с помощью `venv` или `virtualenv` с менеджером пакетов `pip`. *Настраивать виртуальное окружение всегда полезно, так как это изолирует все специфичные зависимости. У нас на практике был случай, когда установка библиотеки напрочь сломала среду разработки Spyder (стандартная среда поставки Anaconda).*

```
pip3 install pipenv
```

Переходим в папку проекта `cd ./mtracker`, если не сделали это ранее, и создаем окружение.

```
pipenv install
```

или

```
python3 -m pipenv install
```

После этого появятся два новых файла (проверьте через `git status`): `Pipfile` и `Pipfile.lock`. Содержимое `Pipfile` без дополнительных зависимостей выглядит так:

```
artem@pc:~$ cat Pipfile
[[source]]
url = "https://pypi.org/simple"
verify_ssl = true
name = "pypi"

[packages]

[dev-packages]

[requires]
python_version = "3.10"
```

Пустой `Pipfile.lock` содержит примерно ту же информацию. Теперь активируем виртуальное окружение и выполним установку `matplotlib`:

```
pipenv shell
pipenv install matplotlib
```

И еще раз взглянем на `Pipfile`:

```
artem@pc:~$ cat Pipfile
[[source]]
```

(continues on next page)

(продолжение с предыдущей страницы)

```

url = "https://pypi.org/simple"
verify_ssl = true
name = "pypi"

[packages]
matplotlib = "*"

[dev-packages]

[requires]
python_version = "3.10"

```

Содержимое `Pipfile.lock` теперь выглядит громоздко, но мы также видим, что там теперь есть информация `matplotlib`, после выполнения `cat Pipfile.lock`. И содержится вся необходимая информация по зависимостям и их версиям.

Теперь мы можем прописать `install_requires` вручную или загрузить из файла:

```

import json
from os import path

here = path.abspath(path.dirname(__file__))

with open(path.join(here, 'README.rst'), encoding='utf-8') as f:
    long_description = f.read()

def read_dependencies(fname):
    filepath = path.join(here, fname)
    with open(filepath) as piplock:
        content = json.load(piplock)
    return [dependency for dependency in content.get('default')]

```

Также мы можем выполнить перед сборкой другие операции, которые могут проверить аннотацию типов, форматирование кода, валидность кода, собрать документацию и выполнить тесты. Аннотация типов необязательна для Python, но улучшает читаемость кода. Рассмотрим некоторые из операций, остальные будем изучать на отдельных уроках. Начнем с форматирования кода. В Python общепринятым является оформление согласно `PEP8`.

*Примечание: предложение по улучшению языка Python `PEP` (Python Enhancement Proposals) можно считать руководством по хорошим практикам и подходом к решению определенных задач, до которых дошло сообщество Python-разработчиков. `PEP` пронумерованы, и `PEP8` описывает стиль кода на языке Python.*

## Немного о форматировании по `PEP8`

Здесь мы не будем подробно останавливаться на всех рекомендациях из `PEP8`, но посмотрим на `autopep8`. Этот инструмент поможет нам скорректировать форматирование, например, уберет лишние пробелы или правильно расположит код.

```
(base) artem@pc:~/tmp$ cat sample.py
if foo == 'blah': do_bla_thing()
do_one(); do_two(); do_three()
```

(continues on next page)

(продолжение с предыдущей страницы)

```
(base) artem@pc:~/tmp$ autorep8 ./sample.py
if foo == 'blah':
    do_blaah_thing()
do_one()
do_two()
do_three()
```

Не все, что написано в PEP8, может быть поправлено автоматически, например, `autorep8` не поправит неинформативные комментарии.

```
# line with not usefull comment
x = x + 1          # Increment x

# line with usefull comment
x = x + 1          # Compensate for border
```

```
(base) artem@pc:~/tmp$ cat sample.py
x = x + 1          # Increment x
(base) artem@pc:~/tmp$ autorep8 ./sample.py
x = x + 1          # Increment x
(base) artem@pc:~/tmp$
```

## Возвращаемся в проект

Установим `autorep8`, но так как это пакет для сборки, а не зависимость самого кода, сделаем это с флагом `-d`:

```
pipenv install -d autorep8
```

Посмотрим файл `Pipfile` и увидим, что в секции `[dev-packages]` появилась новая строка:

```
[dev-packages]
autorep8 = "*"
```

Также поставим `pytest`:

```
pipenv install -d pytest
```

Остальные аспекты пока оставим для дальнейших занятий.

Теперь мы готовы написать Make-файл для сборки своего пакета. Мы использовали `pipenv shell`, чтобы активировать окружение и работать в нем, в `makefile` мы пропишем явно, что команды выполняются в нашем окружении, т.е. вместо `pytest` мы напишем `pipenv run pytest`, и так далее. В `makefile` мы определяем так называемые `target` (задачи), которые будут выполняться, когда мы напишем `make SPECIFY_TARGET`. Например, для активации окружения и установки зависимостей мы определяем задачу `dev`, и когда мы выполним `make dev`, то выполнится все, что относиться к ней, в нашем случае `pipenv install --dev`. Чтобы определить зависимость от другой задачи, которая должна выполниться заранее, мы указываем ее имя после двоеточия.

```
help:
    @echo "Make project with following instructions"
```

(continues on next page)

(продолжение с предыдущей страницы)

```
@cat Makefile

dev:
    pipenv install --dev

test: dev
    pipenv run pytest --doctest-modules --junitxml=junit/test-results.xml

build: clean
    pipenv install wheel
    pipenv run python setup.py sdist bdist_wheel

clean:
    @rm -rf .pytest_cache/.mypy_cache/junit/build/dist/
    @find . -not -path './.venv*' -path '*/__pycache__*' -delete
    @find . -not -path './.venv*' -path '*/*.egg-info*' -delete
```

Указав `test: dev`, мы сделали опцию `test` зависимой от опции `dev`, и поэтому выполнится сначала она, а затем уже `test`.

```
artem@pc:~$ pipenv shell
artem@pc:~$ make test
artem@pc:~$ pip install -e .

(base) artem@pc:~$ pytest --doctest-modules --junitxml=junit/test-results.xml
===== test session starts =====
platform linux -- Python 3.10.9, pytest-7.2.1, pluggy-1.0.0
rootdir: /home/artem/swdev/gitrepo/edu/toolchain_proj/mtracker
collected 1 item

test/test_mtracker.py . [100%]

- generated xml file: /home/artem/swdev/gitrepo/edu/toolchain_proj/mtracker/junit/test-
  ↵results.xml -
===== 1 passed in 0.06s =====
```

Мы видим, что тесты проходят, и мы готовы собрать наш проект. Сделаем для начала очистку и посмотрим, какие файлы добавились при сборке.

```
artem@pc:~$ make clean
artem@pc:~$ ls
LICENSE mtracker Pipfile.lock requirements.txt test
Makefile Pipfile README.md setup.py
artem@pc:~$ make build
artem@pc:~$ ls
build LICENSE mtracker Pipfile README.md setup.py
dist Makefile mtracker.egg-info Pipfile.lock requirements.txt test
artem@pc:~$ ls dist
mtracker-1.0-py3-none-any.whl
```

У нас появилось несколько новых каталогов. И наш собранный пакет расположен в `dist`. Проверим, что `mtracker` не установлен в нашем окружении, команда ниже ничего не должна вывести.

```
pip list | grep mtracker
```

И устанавливаем собранный пакет `mtracker-1.0-py3-none-any.whl`:

```
artem@pc:~$ pip install ./dist/*.whl
artem@pc:~$ pip list | grep mtracker
mtracker      1.0
```

### 22.1.5 Публикация в PyPI

Теперь, когда у нас есть библиотека в собранном виде, мы можем опубликовать ее в PyPI. Чтобы не засорять основной индекс, можно публиковать в тестовом. Для публикации своей библиотеки мы будем использовать утилиту `twine`.

```
pip install twine
```

У нас уже есть все для публикации, так как мы уже настроили сборку проекта, и у нас есть как `bdist`, так и `sdist`. Зарегистрируйтесь в тестовом индексе пакетов и выполните две команды, после чего вам будет предложено ввести логин и пароль, **не забудьте также проверить, что имя библиотеки не занято**.

```
twine check dist/*
twine upload -r testpypi dist/*
```

После этого вы (и кто угодно) можете установить библиотеку:

```
python -m pip install -i https://test.pypi.org/simple mtracker
```

Параметр `-i https://test.pypi.org/simple` нужен только при использовании тестового индекса пакетов.

## 22.2 Задачи

### 22.2.1 Задача 1

Добавьте в `makefile` задачи для автоматического форматирования кода по `pep8`, а также для публикации проекта в PyPI. Добавьте информационное сообщение для каждой задачи в `makefile`.

**Ответ**

(код):

```
PACKAGE_NAME := mtracker
PYTHON := pipenv run python

.PHONY: install test clean package docs publish format build

install:
    @echo "Setting up pipenv virtual environment..."
    pipenv install --dev
```

(continues on next page)

(продолжение с предыдущей страницы)

```

test:
    @echo "Running tests..."
    pipenv run pytest

clean:
    @echo "Cleaning up..."
    rm -rf dist/ build/ *.egg-info

package:
    @echo "Packaging the library..."
    $(PYTHON) setup.py sdist bdist_wheel

docs:
    @echo "Generating documentation..."
    pipenv run pdoc --html $(PACKAGE_NAME) --output-dir docs --force

publish: package
    @echo "Publishing the package to PyPI..."
    pipenv run twine upload dist/*

format:
    @echo "Formatting code with autopep8..."
    pipenv run autopep8 --in-place --recursive mtracker tests

```

## 22.2.2 Задача 2

Прямой вызов `setup.py` считается не самым надежным на сегодня способом. Мы написали урок с его использованием для того, чтобы показать явно все шаги. Прочтите [статью](#) и замените вызов `python setup.py` на `python -m build`.

### Ответ

(код):

```

PACKAGE_NAME := mtracker
PYTHON := pipenv run python

.PHONY: install test clean package docs publish format build

install:
    @echo "Setting up pipenv virtual environment..."
    pipenv install --dev

test:
    @echo "Running tests..."
    pipenv run pytest

clean:
    @echo "Cleaning up..."
    rm -rf dist/ build/ *.egg-info

package:

```

(continues on next page)

(продолжение с предыдущей страницы)

```

@echo "Packaging the library..."
$(PYTHON) -m build

docs:
    @echo "Generating documentation..."
    pipenv run pdoc --html $(PACKAGE_NAME) --output-dir docs --force

publish: package
    @echo "Publishing the package to PyPI..."
    pipenv run twine upload dist/*

format:
    @echo "Formatting code with autopep8..."
    pipenv run autopep8 --in-place --recursive mtracker tests

```

## 22.3 Вопросы

1. Какова цель упаковки проекта Python? а) Для увеличения сложности проекта. б) Чтобы скрыть исходный код от других. в) Чтобы распространять проект как повторно используемую библиотеку или приложение. г) Чтобы ограничить использование проекта определенными версиями Python.
2. Какой файл обычно используется для указания метаданных проекта и зависимостей для пакета Python? а) requirements.txt б) metadata.json в) package.json г) setup.py
3. Какая команда используется для создания bdist дистрибутива (wheel)? а) pip wheel б) python -m wheel в) python -m build г) python setup.py bdist\_wheel
4. Что из следующего является стандартным инструментом для создания и распространения пакетов Python? а) pyinstaller б) virtualenv в) pipenv г) setuptools
5. Где при использовании setuptools разработчик указывает метаданные пакета, зависимости и точки входа в пакет? а) функция setup() б) package\_info.json в) requirements.txt г) модуль distutils
6. Каков рекомендуемый способ указания зависимостей проекта в пакете Python при использовании setuptools? а) Непосредственно включить зависимости в основной файл проекта. б) Вставить зависимости в файл MANIFEST.in. в) Добавить зависимости в файл requirements.txt. г) Указать зависимости в файле setup.py.
7. Какова цель файла `init.py` пакета Python? а) Он содержит основной код пакета. б) Он используется для объявления зависимостей пакетов. в) Это специальный файл, который может не содержать код, для явного указания на то, что каталог является пакетом Python. г) Генерируется автоматически в процессе упаковки.
8. В чем преимущество использования виртуальной среды Python (например, virtualenv или pipenv) для создания пакетов? а) Позволяет распространять проект в виде одного исполняемого файла. б) Это гарантирует, что зависимости проекта изолированы от основного системного Python. в) Он автоматически генерирует файл requirements.txt для проекта. г) Он включает автоматическую упаковку без необходимости в файле setup.py.

### Ответы

1. в)
2. г)
3. г)

4. г)
5. а)
6. г) (также возможен вариант в) при условии, что в ответе будет указано, что зависимости нужно подтянуть в `setup.py`)
7. в)
8. б)

## Менеджеры служб и планировщики задач

---

### 23.1 Менеджеры и планировщики

В данном уроке речь пойдет о менеджерах и планировщиках задачий. Они используются для того, чтобы запустить программы, которые написаны нами по расписанию (планировщики), и контролировать запуск/перезапуск приложений-сервисов (менеджеры).

#### 23.1.1 Мотивация

Бывает множество случаев, когда нам нужно делать что-то периодически или обеспечить надежную работу приложений. Задачей, которую можно отнести к периодической, является мониторинг состояния жестких дисков, одной из наименее надежных частей компьютера, и отправка информации пользователю компьютера или системному администратору, если есть вероятность поломки. Задачей, которую можно отнести ко второму типу, является запуск приложения в виде сервиса, который должен быть доступен всегда для других приложений или пользователей, и если сервис «упал», менеджер сервисов должен попытаться перезапустить его.

Планировщик задач можно сравнить с тренером на групповой тренировке, он говорит, что делать в определенные моменты времени, но проследить за правильностью выполнения каждого участником тренировки не может. Так, он запустит задачу, но если она завершится с ошибкой, то это будет не зона ответственности планировщика, а зона ответственность разработчика.

Менеджер сервисов больше похож на индивидуального тренер, который заинтересован не только в запуске задач, но и в отслеживании их правильного выполнения. При этом менеджер сервисов не берет на себя достижение правильного поведения приложения, это по-прежнему задача разработчика, но менеджер сервисов может предоставить разработчику инструменты для того, чтобы облегчить запуск (например, только после того как мы подключимся к интернету) и поиск ошибок (например, с помощью журнала логов).

### 23.1.2 Планировщик crontab

В качестве примера планировщика мы будем рассматривать `crontab`. Crontab — это утилита-планировщик командной строки в операционных системах Linux, название означает `cron table`. Он используется для создания, управления и поддержки списка запланированных задач, известных как `cron jobs`. Задания cron — это автоматизированные задачи, которые выполняются через определенные промежутки времени или в определенное время, не требуя ручного вмешательства. Crontab позволяет пользователям планировать выполнение задач в определенные моменты времени, что делает его мощным инструментом для автоматизации повторяющихся или периодических действий в системе Linux.

Ключевые понятия `crontab`:

- cron job — это отдельная задача или команда, выполнение которой запланировано через определенные промежутки времени. Каждое задание cron определяется как отдельная строка в файле `crontab`, т.е. в таблице заданий `cron table`.
- расписание — это таблица, которая определяет, когда будет выполняться задание cron. Формат расписания состоит из пяти полей, представляющих минуты, часы, день месяца, месяц и день недели соответственно. Каждое поле использует определенный синтаксис для определения расписания. Пять звездочек в примере ниже представляют расписание для каждой минуты, каждого часа, каждого дня месяца, каждого месяца и каждого дня недели, `command_to_run` — это задача, который нужно выполнить. Таким образом, `command_to_run` будет выполняться каждую минуту без исключений.

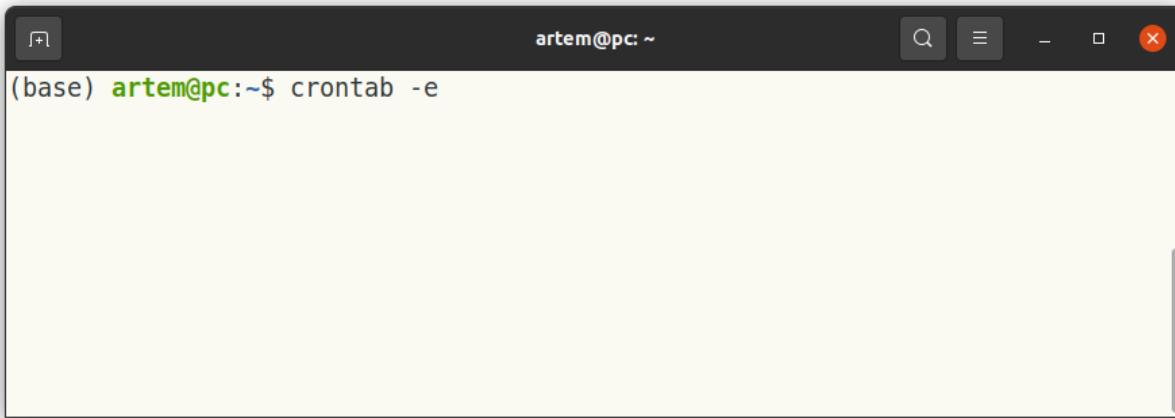
```
* * * * * command_to_run
```

- спецификаторы времени. Вместо использования звездочек для всех полей вы можете указать определенные значения или диапазоны для точной настройки расписания. Например:

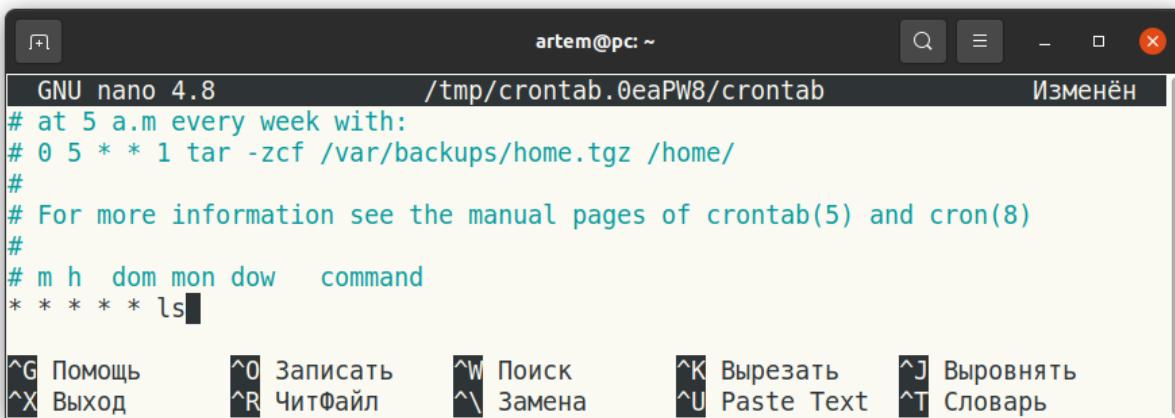
|                               |  |
|-------------------------------|--|
| <code>0 3 * * * job</code>    | запускает команду ежедневно в 3:00.                            |
| <code>*/5 * * * * job</code>  | будет запускать команду каждые пять минут.                     |
| <code>* * */5 * * job</code>  | будет запускать команду каждую минуту пятого числа каждого     |
| <code>→месяца.</code>         |  |
| <code>* * * * 1,4 job</code>  | будет запускать команду каждую минуту в понедельник и четверг. |
| <code>15 1-5 * * * job</code> | будет запускать команду в 15 минут каждого часа с 01:00 по     |
| <code>→05:00.</code>          |  |

- user-specific crontab. Каждый пользователь в системе может иметь свой собственный `crontab`, который управляется независимо. `crontab` текущего пользователя можно редактировать с помощью команды `crontab -e`, выполненной в `bash`.
- system-wide crontab. Помимо пользовательских `crontab`, существует также общесистемный `crontab`, который позволяет системным администраторам планировать задачи, влияющие на всю систему и всех пользователей.

Пользоваться `crontab` очень легко, просто наберите в командной строке Linux `crontab -e`:



После этого откроется расписание задач, которое можно будет поправить как нужно.



По умолчанию `crontab` не будет логировать вывод вашего приложения (это не касается того, что вы сами используете внутри кода, например, с помощью `loguru`). Для того чтобы записать вывод вашего приложения, можно использовать синтаксис, подобный примеру ниже.

```
* * * * * /path/to/command >> /path/to/logfile.log 2>&1
```

Если мы вспомним урок про `bash`, то не найдем ничего нового: `>> /path/to/logfile.log` перенаправляет стандартный поток `stdout` в файл, а `2>&1` перенаправляет ошибки `stderr` в `stdout`, таким образом, мы запишем и ошибки, и простые сообщения.

Типичные варианты использования `crontab`:

- Автоматическое резервное копирование: запланируйте регулярное резервное копирование важных данных, чтобы обеспечить безопасность данных и варианты восстановления. Большие программы, скорее всего, будут иметь встроенный механизм для обеспечения резервного копирования, но `crontab` — это универсальный способ, который можно использовать для любой задачи, в том числе для своих проектов.
- Обслуживание системы: автоматизируйте рутинные задачи по обслуживанию системы, такие как очистка временных файлов, оптимизация баз данных, зеркалирование данных (например, ночью, когда нет нагрузки на сеть со стороны пользователей) и так далее.
- Периодические отчеты: создавайте и отправляйте по электронной почте или в чат-боте периодически.

ческие отчеты, содержащие информацию о состоянии системы или статистику использования.

- Выполнение сценариев по расписанию: запуск пользовательских сценариев через определенные промежутки времени для различных целей.

### Практическое применение crontab

Рассмотрим скрипт, которые собирает информацию о месте на диске, которое занимает содержимое директории, и сохраняет эту информацию вместе с временем обращения к директории в файл. Настроим **crontab** так, чтобы наш скрипт запускался каждые 5 секунд. Мы будем использовать **bash** для написания скрипта, но вы можете написать скрипт на любом языке программирования.

Файл скрипта `collect_dir_space.sh` будет выглядеть следующим образом:

(код):

```
#!/bin/bash

target_directory="/tmp/monitored_directory"

log_file="/tmp/dir_space.log"

# Collect disk space occupied by the target directory
disk_space=$(du -sh "$target_directory")

current_date=$(date +"%Y-%m-%d %H:%M:%S")

# Log the disk space information with the current date
echo "$current_date - Disk Space Usage: $disk_space" >> "$log_file"
```

Теперь нужно создать директорию, которую нужно мониторить, и сделать так, чтобы скрипт можно было запускать.

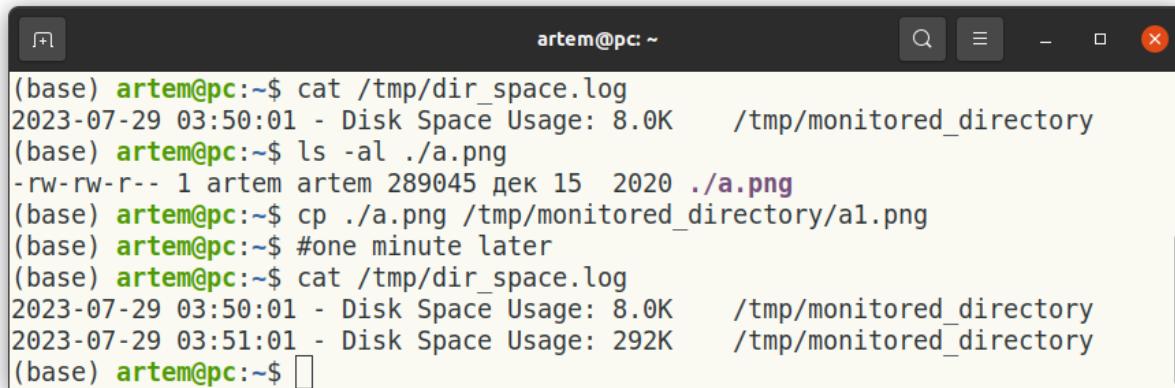
```
mkdir /tmp/monitored_directory
sudo chmod +x collect_dir_space.sh
```

И добавим следующую строку в **crontab**, для этого выполним команду `crontab -e` (см. выше на скриншоте).

```
* * * * * /path/to/collect_dir_space.sh
```



И проверим, что все работает:



```
(base) artem@pc:~$ cat /tmp/dir_space.log
2023-07-29 03:50:01 - Disk Space Usage: 8.0K      /tmp/monitored_directory
(base) artem@pc:~$ ls -al ./a.png
-rw-rw-r-- 1 artem artem 289045 дек 15 2020 ./a.png
(base) artem@pc:~$ cp ./a.png /tmp/monitored_directory/a1.png
(base) artem@pc:~$ #one minute later
(base) artem@pc:~$ cat /tmp/dir_space.log
2023-07-29 03:50:01 - Disk Space Usage: 8.0K      /tmp/monitored_directory
2023-07-29 03:51:01 - Disk Space Usage: 292K      /tmp/monitored_directory
(base) artem@pc:~$ 
```

Мы видим, что в файл `/tmp/dir_space.log` добавляется информация о дисковом пространстве, занятом файлами в директории. Между вызовами `cat /tmp/dir_space.log` мы скопировали файл `a.png`, который занимает пример 282 кБ.

### 23.1.3 Менеджер `systemctl`

Мы будем рассматривать `systemctl` — это утилита командной строки, используемая в современных дистрибутивах Linux, которые используют систему инициализации `systemd` (отвечает за загрузку системы, управление системными службами/демонами, отслеживание состояния системы и обработку отключений системы.). Хотя `systemd` и `systemctl` связаны, это не одно и тоже. Для целей освоения этого урока можно считать, что `systemctl` предоставляет интерфейс `systemd`, который уже отвечает за запуск приложений. *Примечание: правильнее было бы назвать менеджером `systemd`, но так как мы будем использовать его с помощью `systemctl`, мы позволили себе такую подмену понятий.*

Он обеспечивает контроль над системными службами, демонами и другими модулями, позволяя пользователям управлять состоянием и поведением системы. `Systemd` — это система инициализации по умолчанию во многих дистрибутивах Linux, заменяющая традиционную систему инициализации `SysV`.

Ключевым элементом при работе с `systemctl` является Unit. Unit'ы могут быть нескольких типов: службы, сокеты, таймеры, точки монтирования и другие системные ресурсы. Каждый unit определяется файлом с конфигурацией, который предоставляет информацию о том, как запускать, останавливать, включать, отключать выполняемую команду и как управляется unit. Например, unit может быть постоянно включен и перезагружаться при падении или стартовать по расписанию (похоже на то, как это делает `crontab`).

\* Ниже приведен пример конфигурации для службы/сервиса/демона, т.е. приложения, которое работает в фоне. Большинство строк можно понять интуитивно. `Description` описывает то, что делает приложение. `After` описывает зависимость, т.е. от чего зависит текущая служба, в примере ниже, мы ждем, пока не "поднимется" сеть. `ExecStart` описывает команду/программу, которая будет запущена. `Restart` описывает, когда нам нужно перезапустить службу, если она упала. [WantedBy](<https://unix.stackexchange.com/questions/506347/why-do-most-systemd-examples-contain-wantedby-multi-user-target>) и [Type](<https://superuser.com/questions/1274901/systemd-forking-vs-simple>) определяют состояние операционной системы и различные сценарии использования службы, для целей данного урока они не являются принципиальными, но вы можете изучить их самостоятельно.

```
[Unit]
Description=My Application Service
After=network.target

[Service]
Type=simple
ExecStart=python /path/to/my_script.py param1 param2
Restart=always

[Install]
WantedBy=multi-user.target
```

\* Ниже приведен пример unit timer, который можно считать заменой `crontab`, но более сложной в конфигурировании. Мы создадим сервис с типом `Type=oneshot` что говорит systemd запустить его лишь один раз, и потом создадим таймер, который будет вызывать сервис по расписанию.

```
# Service that executed once when called; file my_service.service
[Unit]
Description=My Task

[Service]
Type=oneshot
ExecStart=/path/to/your/script.sh

# Timer to call service by schedule; file my_service.timer
[Unit]
Description=Run My Task Daily

[Timer]
OnCalendar=*-*-* 03:00:00
Persistent=true

[Install]
WantedBy=timers.target
```

### Практическое применение systemctl

Еще одним интересным применением systemctl является мониторинг изменения файлов и отправка уведомлений. Ниже приведен пример сервиса, который будет логировать любую попытку изменения файла (также можно отправлять уведомления на почту или в мессенджер). Создадим файл для службы и сохраним его в файл с названием `pass_monitor.service` в домашней директории.

(код):

```
[Unit]
Description="Run script to log pass changed"

[Service]
ExecStart=/bin/bash -c "echo Changed >> /tmp/pass_change.log"
```

(continues on next page)

(продолжение с предыдущей страницы)

```
[Install]
WantedBy=multi-user.target
```

Теперь создадим path unit, который будет следить за файлом и вызывать наш сервис логирования. Создайте файл `pass_monitor.path` с содержимым.

(код):

```
[Unit]
Description="Monitor the /tmp/passwd file for changes"

[Path]
PathModified=/tmp/passwd
Unit=pass_monitor.service

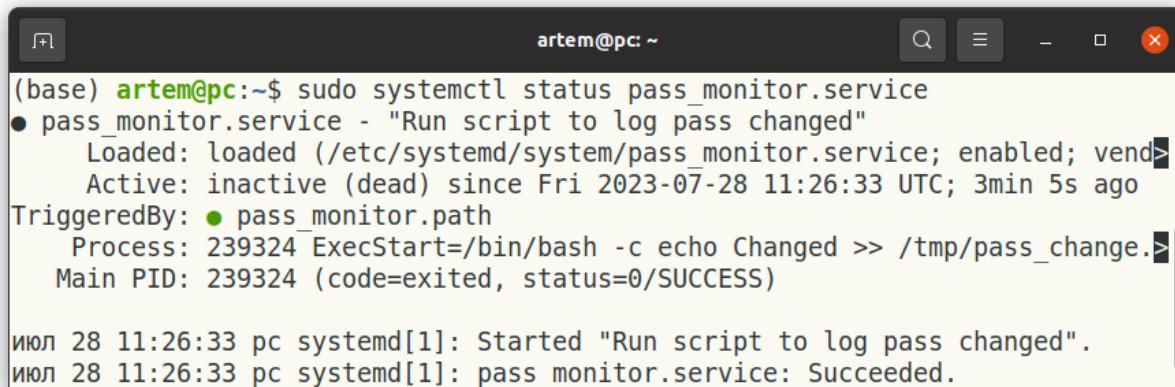
[Install]
WantedBy=multi-user.target
```

Активируем unit'ы:

```
sudo systemctl enable /home/artem/pass_monitor.service
sudo systemctl enable /home/artem/pass_monitor.path
sudo systemctl start pass_monitor.path
```

Проверим, что все работает:

```
sudo systemctl status pass_monitor.service
sudo systemctl status pass_monitor.path
```



The screenshot shows a terminal window with the title bar "artem@pc: ~". The terminal output is as follows:

```
(base) artem@pc:~$ sudo systemctl status pass_monitor.service
● pass_monitor.service - "Run script to log pass changed"
  Loaded: loaded (/etc/systemd/system/pass_monitor.service; enabled; vendor>
  Active: inactive (dead) since Fri 2023-07-28 11:26:33 UTC; 3min 5s ago
    TriggeredBy: ● pass_monitor.path
      Process: 239324 ExecStart=/bin/bash -c echo Changed >> /tmp/pass_change.>
      Main PID: 239324 (code=exited, status=0/SUCCESS)

июл 28 11:26:33 pc systemd[1]: Started "Run script to log pass changed".
июл 28 11:26:33 pc systemd[1]: pass_monitor.service: Succeeded.
```

```
(base) artem@pc:~$ sudo systemctl status pass_monitor.path
● pass_monitor.path - "Monitor the /tmp/passwd file for changes"
    Loaded: loaded (/etc/systemd/system/pass_monitor.path; enabled; vendor >
    Active: active (waiting) since Fri 2023-07-28 11:15:41 UTC; 14min ago
      Triggers: ● pass_monitor.service

июл 28 11:15:41 pc systemd[1]: Started "Monitor the /tmp/passwd file for cha>
lines 1-6/6 (END)
```

Убедимся, что все работает как задумано:

```
(base) artem@pc:~$ echo "asdassad" >> /tmp/passwd
(base) artem@pc:~$ cat /tmp/pass_change.log
Changed
(base) artem@pc:~$ echo "asdassadasdasd" >> /tmp/passwd
(base) artem@pc:~$ cat /tmp/pass_change.log
Changed
Changed
(base) artem@pc:~$ 
```

Как мы видим из примера выше, даже при использовании Unit другого типа (Path, как в примере) нам все равно нужно создать Unit service, который будет управляться Unit'ом другого типа. Мы указывали его как `Unit=pass_monitor.service`.

С помощью systemctl вы можете запускать (`start`), останавливать (`stop`), перезапускать (`restart`), перезагружать (`reload`), включать (`enable`), отключать (`disable`) и проверять состояние (`status`).

- **enable.** Команда `sudo systemctl enable my-service` используется для включения Unit. Можно рассматривать enable как функцию, которая регистрирует написанный нами Unit для systemd и включает автоматический запуск при загрузке или при выполнении условий его активации. После выполнения этой команды сервис можно запускать по имени без указания полного пути. Эта команда создает символьическую ссылку на конфигурационный файл.
- **disable.** Команда `sudo systemctl disable my-service` используется для отключения Unit'a, предотвращая его автоматический запуск при загрузке или при выполнении условий его активации. Она удаляет символьическую ссылку на Unit.
- ``. Команда `sudo systemctl start my-service` запускает работу Unit'a `my-service`. Можно считать, что эта команда используется для ручного запуска Unit'a, например, для проверки работы или после устранения ошибки, которая не дает сервису работать в штатном режиме.
- **stop.** Команда `sudo systemctl stop my-service` прекращает работу Unit'a `my-service`.
- **restart.** Команда `sudo systemctl restrat my-service` прекращает и запускает работу Unit'a `my-service` заново.

- **reload.** Команда `sudo systemctl reload my-service` используется для перезагрузки конфигурации работающего Unit'а без его остановки. Часто используется, когда службе необходимо принять изменения, внесенные в ее файл конфигурации, без полного перезапуска.
- **status.** Команда `sudo systemctl status my-service` предоставляет информацию о текущем состоянии Unit'а, например, работает служба, или она неактивна, сообщения об ошибках и время последнего запуска.

Также удобным инструментом для отслеживания работы службы, в том числе в режиме реального времени, является `journalctl`. Его можно использовать как журнал событий, связанных с работой вашего приложения. При этом это универсальный инструмент для систем Linux, который не связан с определенным языком программирования. Он обладает не таким широким функционалом, и его вывод не оптимизирован под конкретный язык, как специализированные библиотеки, его использование в некоторых случаях может быть удобнее.

```
(base) artem@pc:~$ journalctl -u pass_monitor
-- Logs begin at Mon 2021-04-26 04:37:28 UTC, end at Fri 2023-07-28 12:17:30 >
июл 28 11:05:47 pc systemd[1]: /home/artem/pass_monitor.service:8: Unknown s>
июл 28 11:05:50 pc systemd[1]: /home/artem/pass_monitor.service:8: Unknown s>
июл 28 11:05:51 pc systemd[1]: /home/artem/pass_monitor.service:8: Unknown s>
июл 28 11:15:59 pc systemd[1]: Started "Run script to log pass changed".
июл 28 11:15:59 pc echo[238683]: Password changed >> /tmp/pass_change.log
июл 28 11:15:59 pc systemd[1]: pass_monitor.service: Succeeded.
июл 28 11:17:25 pc systemd[1]: Started "Run script to log pass changed".
июл 28 11:17:25 pc echo[238725]: Password changed >> /tmp/pass_change.log
```

## 23.2 Задачи

### 23.2.1 Задача 1

Напишите расписание cron, которое позволит запускать задачу:

- в 8:30 каждый понедельник.
- в 12:00 каждый понедельник, среду и пятницу.
- в 15 минут каждого часа в 00:00, 06:00, 12:00 и 18:00.

#### Ответ

|                    |  |
|--------------------|--|
| 30 8 * * 1         | в 8:30 каждый понедельник.                             |
| 0 12 * * 1,3,5     | в 12:00 каждый понедельник, среду и пятницу.           |
| 15 0,6,12,18 * * * | в 15 минут каждого часа в 00:00, 06:00, 12:00 и 18:00. |

- каждые 10 минут в течение часа.
- в начале каждого часа с 9:00 до 17:00 с понедельника по пятницу.
- в 30 минут каждого часа с 1:00 до 5:00 и с 12:00 до 15:00.

#### Ответ

```
*/10 * * * *      каждые 10 минут в течение часа.  
0 9-17 * * 1-5    в начале каждого часа с 9:00 до 17:00 с понедельника по пятницу.  
30 1-5,12-15 * * * в 30 минут каждого часа с 1:00 до 5:00 и с 12:00 до 15:00.
```

- каждые 2 минуты.
- каждые 3 минуты с 10:00 до 18:00.
- каждые 10 минут в полночь (0:00) по понедельникам.

**Ответ**

```
*/2 * * * *      каждые 2 минуты.  
*/3 10-18 * * *  каждые 3 минуты с 10:00 до 18:00.  
*/10 0 * * 1     каждые 10 минут в полночь (0:00) по понедельникам.
```

**23.2.2 Задача 2**

Создайте скрипт и соответствующие unit'ы systemctl для следующей задачи: нужно отследить изменения в файле, и если он содержит подстроку «critical error», отправить сообщение пользователю.

**Ответ**

Скрипт для проверки `check_script.sh`.

(код):

```
#!/bin/bash  
  
file="/path/to/watched/file"  
substring="critical error"  
  
if grep -q "$substring" "$file"; then  
    echo "Found critical error in $file"  
    # Add your notification command here, e.g., sendmail or notify-send  
else  
    echo "No critical error found in $file"  
fi
```

Конфигурация для Unit service `check_critical.service`:

(код):

```
[Unit]  
Description=Check File for Critical Error  
  
[Service]  
Type=simple  
ExecStart=/path/to/your/check_script.sh
```

Конфигурация для Unit path `check_critical.path`:

(код):

```
[Unit]  
Description=Watch My File
```

(continues on next page)

(продолжение с предыдущей страницы)

```
[Path]
PathModified=/path/to/watched/file
Unit=check_critical.service

[Install]
WantedBy=multi-user.target
```

Код для активации сервиса:

```
sudo chmod +x check_script.sh
sudo systemctl enable /path/to/check_critical.service
sudo systemctl enable /path/to/check_critical.service
sudo systemctl start pass_monitor.path
```

### 23.3 Вопросы

1. Какова цель команды `systemctl start`? а) Чтобы служба (Unit) запускалась автоматически при загрузке. б) Для ручного запуска определенной службы. в) Перезагрузить конфигурацию работающей службы. г) Остановить работающую службу.
2. Какая команда перезапустит работающий модуль, выполнив остановку и запуск? а) `systemctl stop` б) `systemctl reload` в) `systemctl restart` г) `systemctl enable`
3. Что показывает команда `systemctl status`? а) Подробную информацию о конфигурации конкретной службы (Unit). б) Список всех загруженных служб и их текущий статус. в) Время последнего запуска службы. г) Журнал логов службы.
4. Какую команду следует использовать, чтобы включить службу так, чтобы она (служба) автоматически запускалась при загрузке системы? а) `systemctl start` б) `systemctl enable` в) `systemctl enable --now` г) `systemctl enable --start`
5. Какова цель команды `systemctl list-units`? а) Включить службу. б) Отобразить подробную информацию о конкретном блоке. в) Перечислить все загруженные службы вместе с их текущим статусом и описанием. г) Перезагрузить конфигурацию работающей службы.
6. Какая команда отключит устройство, предотвращая его автоматический запуск при загрузке? а) `systemctl disable` б) `systemctl stop` в) `systemctl disable --now` г) `systemctl stop --disable`
7. Для чего используется `crontab` в системах Linux? а) Управление системными демонами и службами. б) Автоматизация запуска задач в определенное время или через определенные промежутки времени. в) Управление сетевыми настройками системы. г) Установка программных пакетов.
8. Как вы редактируете расписание запуска задач `crontab` для текущего пользователя? а) `crontab -l` б) `crontab -e` в) `crontab -r` г) `crontab -n`
9. Какая запись `crontab` назначает выполнение задачи каждое воскресенье в 14:30? а) 30 2 \* \* 0  
б) 30 14 \* \* 7 в) 2 30 \* \* 7 г) 14 30 \* \* 0
10. Что означает символ \* в записи `crontab`? а) Текущий рабочий каталог. б) Корневой каталог. в) Любое значение для определенного поля. г) Предыдущий рабочий каталог.
11. Как бы вы перечислили все существующие записи `crontab` для текущего пользователя? а) `crontab -l` б) `crontab -e` в) `crontab -r` г) `crontab -n`

12. Какую команду следует использовать, чтобы удалить все записи `crontab` для текущего пользователя?  
а) `crontab -l` б) `crontab -e` в) `crontab -r` г) `crontab -n`
13. Какая строка расписания `cron` будет запускать скрипт каждые 10 минут?  
а) `*/10 * * * *` б) `10 * * * *`  
в) `0-59/10 * * * *` г) `0/10 * * * *`
14. Какова максимальная степень детализации временных интервалов, которую можно установить в `crontab`?  
а) 1 минута б) 1 секунда в) 1 час г) 1 день
15. Как указать `crontab` конкретного пользователя (`user`) для редактирования, если у вас есть необходимые разрешения?  
а) `crontab -u user` б) `crontab -l user` в) `crontab -e user` г) `crontab -r user`
16. Какая запись `crontab` запускает сценарий в 00:15 1 января и 1 июля каждого года?  
а) `15 0 1,7 * *` б) `0 15 1 1,7 *`  
в) `15 0 1 7 *` г) `0 12 1 1,7 *`

#### **Ответы**

1. 6)
2. в)
3. б)
4. б)
5. в)
6. а)
7. б)
8. б)
9. б)
10. в)
11. а)
12. в)
13. а)
14. б)
15. в)
16. а)

### 24.1 Контейнеризация на примере Docker

#### 24.1.1 Введение

С появлением персональных компьютеров, разработка программного обеспечения (ПО) стала более доступной, однако с появлением новых возможностей также возрастали и потребности. Приложения становились все сложнее, а машины – мощнее. Тем не менее, возникал вопрос о ресурсоемкости исполнения приложений и изоляции зависимостей. Первым решением стало появление и развитие виртуальных машин (Virtual machine, VM). Они позволяли создавать изолированные окружения на одной физической машине выделив ресурсы для VM. Это был большой шаг для упрощения разработки ПО и гибкому управлению ресурсов. Использование виртуальных машин позволило выполнять запуск нескольких операционных систем (ОС) на мощностях одного ПК. Это решало многие проблемы, но и создавало новые – виртуализация требовала больших ресурсов и замедляла производительность. С ростом сложности и разнообразия современных приложений, стало ясно, что использование только виртуальных машин не всегда оптимально. Они потребляют много ресурсов, так как каждая из них включает полноценную ОС и дополнительные ресурсы для виртуализации. Это может привести к избыточным затратам ресурсов и низкой плотности приложений на серверах.

Так и началось развитие контейнеризации. Вместо виртуальных машин, контейнеры используют общую ОС ПК и разделяют только необходимые ресурсы и библиотеки. Это позволяет достичь более высокой плотности приложений на серверах, экономить ресурсы и ускорять развертывание. Однако контейнеры также сталкиваются с некоторыми ограничениями, например, они могут иметь ограниченный доступ к ресурсам хоста.

С появлением виртуализации, идея контейнеризации была вполне логичной эволюцией, так и появился Docker. Контейнеры позволяют упаковать приложение и его зависимости в изолированный и легкий для перемещения образ, который может быть запущен на любом ПК с Docker. Это значит, что разработчики больше не зависят от окружения, на котором запускается приложение – оно будет работать одинаково везде потребляя на порядок меньше ресурсов, чем VM. Однако с появлением множества контейнеров возник вопрос управления ими. В этот момент появляется кластеризация – способ управления сетью из множества контейнеров. Это стало ответом на потребность в гибкости, масштабируемости и автоматизации управления развертыванием приложений.

Таким образом, процесс разработки ПО постепенно эволюционировал от огромных машин и долгих циклов разработки к легковесным контейнерам. Это открыло новые горизонты для разработчиков, позволив им создавать, тестировать и масштабировать приложения гораздо эффективнее и быстрее, а также отвечать на постоянно изменяющиеся требования технологических решений.

Во многом это привело к тому, что нас сегодняшний день Docker используют как одного из основных инструментов для организации микросервисной архитектуры, где приложение разбивается на небольшие, независимые контейнеры, каждый из которых отвечает за определенную функциональность. Таким образом, появление Docker и его развитие играет значимую роль разработке ПО и способствовало развитию современных методологий позволивших повысить эффективность разработки.

С появлением Docker в 2013 году контейнеризация стала гораздо более доступной и удобной. Docker предоставил унифицированный способ упаковки приложений и их зависимостей в контейнеры, которые могут быть развернуты на любой поддерживаемой платформе. Это позволило разработчикам и инженерам работать с одинаковой средой в разных этапах жизненного цикла приложения, от разработки до развертывания. Крупные компании, такие как Yandex, Sber, Google, Amazon, Microsoft, Netflix и другие, успешно используют Docker для разработки своих продуктов и сервисов.

#### 24.1.2 Рассмотрим пример: как Docker может помочь в разработке ПО

Представим ситуацию, вы разрабатываете программный модуль, который должен облегчить использование QR-кодов в чат-ботах. В качестве языка программирования вы выбрали Python. Спустя некоторое время ваш проект готов, вы провели тесты, и всё работает как часы и вы публикуете его в открытом репозитории, например в GitHub. Но, вскоре вы получаете сообщения от первых пользователей, которые столкнулись с проблемой: их чат-бот исправно работал на одном сервере, но неожиданно переставал функционировать на других. Оказалось, что источник проблем был некоторых различиях рабочего окружения. В одном случае проблема была в конфигурации операционной системы на VPS. В других случаях — ошибки в зависимостях некоторых модулей Python и различия в версиях используемых библиотек. Что в совокупности приводило к появлению ошибок в процессе функционирования чат-ботов использующих ваш программный модуль. Такие проблемы совместимости программных пакетов сложно отлавливать в процессе разработки.

Использование Docker может значительно упростить и решить проблему совместимости, с которой вы столкнулись при развертывании вашего программного модуля для работы с QR-кодами. Вот как вы можете это сделать. Рассмотрим вариант решения данной проблемы используя Docker:

Создадим новый Docker-образ, в котором будет все, что нужно для работы приложения — код, библиотеки, переменные окружения. Также, в нем будет запускаться ваш экземпляр чат-бота. Docker обеспечит изоляцию окружения, что позволит вам управлять зависимостями, конфигурацией операционной системы и версиями библиотек внутри контейнера. Затем можно опубликовать Docker-образ в реестре Docker Hub или на другой подобной платформе. Что позволит другим пользователям легко получить и использовать ваш образ. Пользователи могут легко развернуть ваш Docker-образ на своих серверах. Образ будет запущен в изолированном контейнере, и все зависимости будут корректно настроены, таким образом, устранив проблемы совместимости. Так, используя Docker, вы можете унифицировать окружение и избежать проблем совместимости вашего проекта с различными серверами и системами, а также облегчите развертывание и обновление вашего программного модуля.

Как можно подытожить эту историю. Если упростить, то Docker-образ похож на «компьютер», который может запустить код приложения находящийся в нем. И где бы мы не включили этот «компьютер», в офисе, дома, у заказчиков или у друзей - он всегда будет работать одинаково. Если ваше приложение использует компоненты операционной системы (ОС), вы можете её установить. Как правило, это Linux.

Примечание: В Docker отсутствует поддержка установки в Docker-образ ОС семейства Windows, и как следствие приложений под эту ОС.

## Образ и контейнер в Docker

Выше мы уже говорили об «образах». Что это такое? Хороший вопрос. То, что в терминологии Docker называется «образом», можно сравнить с чертежами, классом из ООП, или с пресс-формой для чеканки монет. В Docker «контейнер» и «образ» — это две разные вещи разными целями.

Docker образ — это шаблон, определяющий приложение и его зависимости. Это схема или снимок приложения, доступный только для чтения, который можно использовать для создания контейнеров Docker. Образ создается путем запуска файла Dockerfile, в котором указывается используемый базовый образ, любые зависимости, которые необходимо установить, а также любая пользовательская конфигурация или код. После создания образа его можно распространять и делиться с другими.

Docker контейнер — это работающий экземпляр образа. Его можно рассматривать как легкий автономный исполняемый пакет, включающий все необходимое для запуска приложения, в том числе программный код, среду выполнения, системные инструменты, библиотеки и настройки. Когда контейнер запускается, в верхнюю часть образа добавляется новый доступный для записи слой, позволяющий вносить изменения в работающее приложение.

Несколько контейнеров могут быть созданы из одного и того же образа, каждый со своим отдельным записываемым слоем. Проще говоря, образ — это статический файл, который включает в себя все необходимое для запуска приложения, а контейнер — это работающий экземпляр образа, который можно изменять и с которым можно взаимодействовать.

Давайте разберемся, в чем отличие образа и контейнера. Если сказать в двух словах, то Docker контейнер — это запущенный (активный) образ. Образ в свою очередь — это то, что представляет собой функциональную часть одной или нескольких программ, скриптов, данных и т.д., то, что будет выполняться при запуске образа в контейнере. Образы достаточно просто создаются инструментами Docker. На базовом уровне контейнер представляет собой набор инструкций, который выполняется на некоем процессоре, обрабатывая какие-то данные.

### Знакомство с Docker: контейнеры и образы:

Docker — это платформа для разработки, доставки и запуска приложений в контейнерах. Контейнеры — это изолированные среды, которые объединяют приложение и его зависимости, обеспечивая портативность однообразие в разных окружениях. Образы — это шаблоны для создания контейнеров, содержащие все необходимое для работы приложения. За счет контейнеров и образов Docker упрощает разработку, развертывание и масштабирование приложений.

Основные компоненты Docker:

1. **Docker Engine:** Это основной компонент Docker, который управляет контейнерами и образами. Docker Engine включает в себя Docker Daemon (серверная часть) и Docker CLI (командная строка). Docker Daemon управляет жизненным циклом контейнеров, созданием, запуском и остановкой, а Docker CLI позволяет разработчикам взаимодействовать с Docker Engine с помощью команд.
2. **Docker Images:** Образы Docker — это основа контейнеров. Они содержат файловую систему, зависимости и настройки приложения. Образы создаются на основе Dockerfile — текстового файла, который описывает, как создать образ. Образы могут быть сохранены в реестре, таком как Docker Hub или частном реестре, и используются для развертывания контейнеров.
3. **Docker Containers:** Контейнеры — это экземпляры образов. Каждый контейнер изолирован от других контейнеров и имеет свою собственную файловую систему, зависимости и окружение. Они используют технологии Linux-контейнеров для обеспечения изоляции, но все же разделяют ядро операционной системы хоста.
4. **Docker Compose:** Этот инструмент позволяет определить и управлять множеством связанных контейнеров в единой конфигурации. С Docker Compose вы можете описать все контейнеры, сети, тома и настройки в файле docker-compose.yml, что упрощает развертывание и управление многоконтейнерными приложениями.

5. **Docker Swarm:** Docker Swarm - это инструмент для создания и управления кластерами Docker, позволяющим распределить приложение на несколько узлов (хостов) и обеспечить высокую доступность и масштабируемость.
6. **Docker Hub:** Docker Hub - это облачный реестр образов Docker, где разработчики могут сохранять и делиться своими образами. Это упрощает распространение приложений и обмен образами между разработчиками.

Все эти компоненты взаимодействуют, чтобы предоставить разработчикам среду для создания, развертывания и управления контейнерами. Docker позволяет упаковать приложения и все их зависимости в контейнеры, что облегчает портирование и запуск приложений на разных окружениях, повышает скорость разработки и обеспечивает изоляцию между приложениями.

#### 24.1.3 Концепции Docker

Несколько контейнеров могут объединяться в сеть и совместно выполнять свою задачу. Например, Контейнер №1 содержит код приложения: Linux, Python, Flask, SQLAlchemy. Контейнер №2 содержит базу данных и выполняет обработку внешних запросов к данным, находящимся в базе данных. Контейнер №3 выполняет мониторинг и анализ данных, формирует отчетность на основе данных из базы данных в контейнере №2 и предоставляет пользователям контейнера №1. Таким образом, каждый контейнер решает отдельную задачу. Физически контейнеры могут находиться в разных помещениях, городах и даже странах. Это и есть возможность масштабирования приложений.

#### 24.1.4 Установка и использование

Установка и настройка Docker - это первый шаг в изучении контейнеризации и использовании Docker для разработки и развертывания приложений. Этот процесс обеспечивает удобный способ управления окружением и приложениями, что делает Docker популярным инструментом среди разработчиков и операционных команд.

##### Установка Докера в Windows

1. Перейдите на веб-сайт Docker и загрузите установщик Docker Desktop для Windows.
2. Запустите программу установки и следуйте инструкциям для завершения установки.
3. После установки Docker Desktop должен запуститься автоматически. Если это не так, вы можете запустить его из меню «Пуск».
4. После запуска Docker Desktop вы можете использовать его для создания и запуска контейнеров Docker.

##### Установка Докера в Linux

Точные шаги по установке Docker в Linux зависят от используемого вами дистрибутива. Вот основные шаги для Ubuntu:

1. Откройте терминал и выполните следующую команду, чтобы установить зависимости Docker:  
`sudo apt-get update  
sudo apt-get install apt-transport-https ca-certificates curl software-properties-common`
2. Добавьте ключ Docker GPG в свою систему:  
`curl -fsSL https://download.docker.com/linux/ubuntu/gpg | sudo apt-key add -`

3. Добавьте репозиторий Docker в ваши источники APT: `sudo add-apt-repository "deb [arch=amd64] https://download.docker.com/linux/ubuntu $ (lsb_release -cs) stable"`
4. Обновите индекс пакета APT и установите Docker: `sudo apt-get update sudo apt-get install docker-ce`
5. После установки Docker должен быть запущен в вашей системе. Вы можете проверить это, выполнив следующую команду: `sudo docker run hello-world`

Это загрузит тестовый образ Docker и запустит его в контейнере. Это основной процесс установки Docker в Linux. Точные шаги могут различаться в зависимости от используемого дистрибутива и версии. Здесь мы исходим из предположения, в соответствии с которым используется образ Docker, основанный на Unix-подобной ОС. Конечно, тут можно воспользоваться и образом, основанным на Windows, но использование Windows — это менее распространенная практика, работать с такими образами сложнее. В результате, если у вас есть такая возможность, рекомендуем использовать Unix. Однако для выполнения практических примеров можно использовать любую ОС, поскольку мы будем использовать только базовые команды Docker в консольном режиме.

## Быстрый Старт

Чтобы запустить контейнер Docker, вы можете выполнить следующие общие шаги:

Извлеките образ Docker. Если вы еще этого не сделали, вам потребуется извлечь образ Docker, который вы хотите запустить, из реестра Docker, такого как Docker Hub. Вы можете сделать это с помощью команды `docker pull`, за которой следует имя и тег образа. `docker pull <image-name>:<tag>` Например, чтобы получить официальный образ nginx, вы можете использовать: `docker pull nginx:latest` Запустите контейнер Docker: получив образ, вы можете запустить контейнер, выполнив команду запуска `docker`, за которой следует имя образа и любые дополнительные параметры, которые вы хотите указать. `docker run [OPTIONS] <image-name>` Например, чтобы запустить контейнер из образа nginx, вы можете использовать: `docker run -p 8080:80 nginx:latest` В результате выполнения этой команды у нас запустится контейнер с образом nginx. Следует отметить, что мы указали параметр `-p 8080:80` для предоставления доступа внешних приложений к нашему запущенному контейнеру, используя порт 8080. Чтобы в этом убедиться, откроем браузер и введем следующий адрес: (`http://localhost:8080/`). Нам откроется приветственная HTML страница nginx, сообщающая о том, что веб-сервер nginx успешно запустился. В данном случае мы указали порт 8080 для того, чтобы достучаться до 80 порта внутри нашего контейнера, это очень полезный параметр. Ведь если нам потребуется запустить несколько контейнеров, внутри которых используется стандартный порт, например, 80, то мы не сможем использовать его для отдельного обращения к каждому контейнеру. Эта команда запускает новый контейнер из образа `nginx:latest` и сопоставляет порт 80 в контейнере с портом 8080 на хост-компьютере. Параметр `-p` указывает сопоставление портов. Убедитесь, что контейнер работает, для этого вы можете использовать команду `docker ps`.

## `docker ps`

Эта команда выводит список всех запущенных контейнеров. Если вы не видите свой контейнер в списке, обязательно используйте команду `docker ps -a`, чтобы увидеть все контейнеры (включая остановленные). Чтобы остановить работающий контейнер, вы можете использовать команду `docker stop`, за которой следует идентификатор или имя контейнера. `docker stop <container-id>` Например, чтобы остановить контейнер с идентификатором `abc123`, вы можете использовать: `docker stop abc123` Это основные шаги для запуска контейнера Docker. Вы можете найти более расширенное использование Docker с дополнительными параметрами и конфигурациями в документации Docker.

**docker run**

Команда `docker run` имеет ряд опций, которые можно использовать для настройки поведения контейнера во время выполнения. Вот некоторые из наиболее часто используемых опций: `-d` или `--detach`: Запуск контейнера в фоновом режиме (режим отсоединения). `-p` или `--publish`: Публиковать порт(ы) контейнера на хост-машине. `-v` или `--volume`: Смонтировать каталог хоста в качестве тома данных в контейнере. `-e` или `--env`: Установить переменные окружения для контейнера. `--name`: Присвоить имя контейнеру. `--restart`: Задать политику перезапуска для контейнера. `-it` или `--interactive` и `--tty`: Запустить интерактивный сеанс с контейнером и выделить псевдо-TTY. `--rm`: Автоматически удалять контейнер при выходе из него. `--network`: Подключить контейнер к сети.

#### 24.1.5 Файл Dockerfile

Dockerfile – это текстовый файл, который содержит набор инструкций для автоматической сборки Docker-образа. Этот файл описывает, какие базовые образы использовать, как установить зависимости, куда и какие файлы копировать и как настроить рабочее окружение внутри контейнера. Dockerfile служит для создания образов – снимков файловой системы и настроек, необходимых для работы приложения. Он автоматизирует процесс сборки образа и обеспечивает воспроизводимость этого процесса на разных окружениях.

Допустим, вы пишете приложение для обработки QR-кода. В вашем проекте используются различные модули, сторонние библиотеки, конфигурационные файлы. Для развития проекта нужно наделить его автономность и средствами мониторинга и диагностики, а также инструментами информирования на тот случай, если потребуется ваше вмешательство. В такой ситуации вам будет удобно использовать инструменты ОС. Например, легковесный дистрибутив линукс `Alpine`. В этой задаче он будет выбран в качестве «базового образа». Базовый образ это основа (ОС, сборка Python и т.д.) для вашего проекта, который вы планируете использовать в Docker. Среди популярных официальных базовых образов можно отметить:

1. `python`,
2. `ubuntu`,
3. `alpine`.

Ознакомиться с актуальным списком образов можно на сайте [docker hub](#).

Для того чтобы собрать Docker-образ, в котором будет всё то что упоминалось выше и используют Dockerfile.

В образ контейнера, поверх базового образа, можно добавлять дополнительные слои (установка библиотек и программ, добавление внешних файлов и т.д.). Делается это в соответствии с инструкциями из Dockerfile. Например, если Dockerfile описывает образ, который планируется использовать для решения задач машинного обучения, то в нем могут быть инструкции для включения в промежуточный слой Docker-образа библиотек: NumPy, Pandas и Scikit-learn. Наконец, в Docker-образе может содержаться, поверх всех остальных, еще один тонкий слой, в котором хранятся данные, поддающиеся изменению. Это небольшой по объему слой, содержащий программу, которую планируется запускать в контейнере. Dockerfile может состоять из одной строки с базовым образом:

```
FROM ubuntu:18.04 # <-- Базовый образ Ubuntu
```

В таком случае из реестра Docker Hub будет скачан этот образ ОС `ubuntu` версии 18.04 и сформирован в виде нового образа. Для указания базового образа мы использовали команду «`FROM`». Ниже в таблице представлены основные команды Dockerfile с краткими комментариями:

**Основные команды Dockerfile:**

| Команда           | Описание  |
|-------------------|---|
| <b>FROM</b>       | Определяет базовый образ, на котором будет основан создаваемый образ.   |
| <b>RUN</b>        | Выполняет команды внутри контейнера во время сборки образа. Используется для установки программ и зависимостей.                                 |
| <b>COPY/A</b>     | Копирует файлы и папки из хостовой машины в образ. COPY просто копирует файлы, а ADD также может распаковывать архивы и загружать файлы из URL. |
| <b>WORKDIR</b>    | Устанавливает рабочую директорию внутри контейнера, где будут выполняться команды.  |
| <b>ENV</b>        | Устанавливает переменные окружения внутри контейнера.   |
| <b>EXPOSE</b>     | Определяет порты, которые контейнер будет слушать при запуске.  |
| <b>CMD</b>        | Устанавливает команду, которая будет выполнена при запуске контейнера. Если в Dockerfile есть несколько CMD, будет исполнен только последний.   |
| <b>ENTRYPOINT</b> | Определяет команду, которая будет запускаться при запуске контейнера. Может использоваться для создания исполняемых контейнеров.                |
| <b>VOLUME</b>     | Создает точку монтирования для работы с данными между хостовой машиной и контейнером.   |
| <b>ARG</b>        | Определяет переменные, которые могут быть переданы при сборке образа, но не сохраняются в окружении контейнера.                                 |

Рассмотрим пример Dockerfile.

```
# Используем базовый образ Python
FROM python:3.9-slim

# Устанавливаем рабочую директорию
WORKDIR /app

# Копируем зависимости
COPY requirements.txt .

# Устанавливаем зависимости
RUN pip install --no-cache-dir -r requirements.txt

# Копируем файлы приложения
COPY . .

# Устанавливаем команду запуска приложения
CMD ["python", "app.py"]
```

В названии файла не стоит указывать какое-либо расширение, название должно содержать только «Dockerfile». В случае проблемы, проверьте, что в настройках директории отсутствует галочка в пункте «скрывать расширения файлов» (например, .doc, .txt, .exe, .pdf).

Представленный выше Dockerfile содержит инструкции для создания Docker-образа, который будет содержать, и запускать Python-приложение. Давайте подробнее разберем каждую строку этого Dockerfile и прокомментируем их содержание:

```
# Используем базовый образ Python
FROM python:3.9-slim
```

Здесь мы выбираем базовый образ Python версии 3.9 версии (slim). Выбор версии обусловлен совместимостью с нашим приложением и требованиями к окружению.

```
# Устанавливаем рабочую директорию
WORKDIR /app
```

Устанавливаем рабочую директорию для последующих инструкций. В данном случае, мы устанавливаем /app как рабочую директорию. Все последующие операции будут выполняться в этой директории.

```
# Копируем зависимости
COPY requirements.txt .
```

Копируем файл requirements.txt из контекста сборки (текущей директории, где находится Dockerfile) в рабочую директорию /app внутри контейнера. Это позволяет установить зависимости перед запуском приложения.

```
# Устанавливаем зависимости
RUN pip install --no-cache-dir -r requirements.txt
```

Выполняем установку зависимостей, перечисленных в файле requirements.txt. Флаг --no-cache-dir используется для установки без кэша, что помогает уменьшить размер образа.

```
# Копируем файлы приложения
COPY . .
```

Копируем все файлы и директории из контекста сборки (включая все файлы Python-приложения) в рабочую директорию /app внутри контейнера.

```
# Устанавливаем команду запуска приложения
CMD ["python", "app.py"]
```

Устанавливаем команду, которая будет выполнена при запуске контейнера. В данном случае, это запуск файла app.py с помощью интерпретатора Python.

В результате, этот Dockerfile позволит создать Docker-образ, который включает в себя Python-приложение, его зависимости и команду для его запуска. Этот образ можно будет запустить как контейнер для работы приложения.

Dockerfile – это мощный инструмент для определения структуры и настроек Docker образа. Он позволяет автоматизировать процесс создания образов и обеспечивает воспроизводимость настроек на разных средах. С помощью команд в Dockerfile вы можете определить все, что необходимо для успешного развертывания вашего приложения в контейнере.

## Команды для управления контейнерами

Таблица с командами для управления Docker контейнерами

| Команда                     | Описание   | Пример   |
|-----------------------------|--|--|
| <code>docker run</code>     | Создание и запуск нового контейнера.                         | <code>docker run -it ubuntu bash</code>        |
| <code>docker ps</code>      | Просмотр списка активных контейнеров.                        | <code>docker ps</code>                         |
| <code>docker ps -a</code>   | Просмотр списка всех контейнеров (активные и остановленные). | <code>docker ps -a</code>                      |
| <code>docker start</code>   | Запуск остановленного контейнера.                            | <code>docker start my_container</code>         |
| <code>docker stop</code>    | Остановка запущенного контейнера.                            | <code>docker stop my_container</code>          |
| <code>docker restart</code> | Перезапуск контейнера.                                       | <code>docker restart my_container</code>       |
| <code>docker pause</code>   | Приостановка выполнения контейнера.                          | <code>docker pause my_container</code>         |
| <code>docker unpause</code> | Возобновление выполнения приостановленного контейнера.       | <code>docker unpause my_container</code>       |
| <code>docker exec</code>    | Выполнение команды внутри запущенного контейнера.            | <code>docker exec -it my_container bash</code> |
| <code>docker logs</code>    | Просмотр логов контейнера.                                   | <code>docker logs my_container</code>          |
| <code>docker rm</code>      | Удаление контейнера.   | <code>docker rm my_container</code>            |
| <code>docker rmi</code>     | Удаление образа.   | <code>docker rmi my_image</code>               |

### Пример использования команды `docker run` для запуска контейнера

Ниже представлен пример запуска контейнера с шуточным приложением «cowsay», которое выводит фразу, принятую в качестве аргумента:

```
docker run docker/whalesay cowsay "Hello, Docker!"
```

В данном примере команда запускает контейнер с образом `docker/whalesay` и передает приложению `cowsay` фразу «Hello, Docker!», которую оно выводит с использованием анимированного ASCII-арта. Ссылка на [Docker Hub](#) там же доступен и Dockerfile который можно взять в качестве шаблона для разработки собственной версии приложения.

## Работа с данными

Рассмотрим процесс работы с данными, в частности, тома Docker. Данные в Docker могут храниться либо временно, либо постоянно. Начнем с временных данных.

Чтобы передать файл с данными в Docker контейнер для обработки, вы можете воспользоваться механизмом монтирования томов или копирования файлов. Вот два основных способа:

- Монтирование тома (Volume Mounting):** Вы можете создать Docker том (volume), который будет связан с файловой системой вашего хоста, и затем монтировать этот том внутрь контейнера. Этот способ позволяет передавать данные между хостом и контейнером в режиме реального времени.

Пример команды для запуска контейнера с монтированием тома:

```
docker run -v /путь/к/локальной/директории:/путь/в/контейнере -it my_image
```

2. **Копирование файлов:** Вы можете скопировать файлы внутрь контейнера с помощью команды `docker cp`. При этом файлы будут скопированы внутрь контейнера, и вы сможете работать с ними внутри него.

Пример команды для копирования файла в контейнер:

```
docker cp файл.txt контейнер:/путь/в/контейнере/
```

Обратите внимание, что выбор способа зависит от того, какой уровень взаимодействия вам нужен между хостом и контейнером. Если вам нужно, чтобы контейнер мог динамически видеть и изменять данные, то монтирование тома будет более удобным. Если же данные нужно скопировать внутрь контейнера только один раз, то копирование файлов будет подходящим вариантом.

### Контейнер Docker обладает следующими характеристиками:

- В нем можно хранить различные ресурсы: скрипты, исходный код, базу данных, наборы тестов, веб-формы, модели данных и т.д. Ресурсы могут располагаться как в контейнере, так и за его пределами (через удаленный доступ в сети, либо используя общее место хранения данных — «Volume»). Его можно переносить. Контейнер Docker можно использовать на локальном ПК, на ПК, находящемся в рабочей сети, на удаленном облачном сервере (Яндекс.Облако, AWS, Google.Cloud и др.).
- Удобный и гибкий интерфейс обращения к его содержимому. Механизмы, позволяющие контейнеру взаимодействовать с внешним миром. Например, у контейнера есть порты, которые можно открывать для того, чтобы к приложению, работающему в контейнере, можно было бы обращаться из браузера или по протоколу SSH. Работать с контейнером можно и средствами командной строки.
- Образ контейнера хранится в специальном репозитории (Docker-hub). Если вам нужен некий готовый контейнер, вы можете загрузить из репозитория соответствующий образ и, используя его, этот контейнер создать.

### Особенности контейнеров

Давайте разберемся, в чем отличие образа и контейнера. Если сказать в двух словах, то Docker контейнер — это запущенный (активный) образ. Образ в свою очередь — это то, что представляет собой функциональную часть одной или нескольких программ, скриптов, данных и т.д., то, что будет выполняться при запуске образа в контейнере. Образы достаточно просто создаются инструментами Docker. На базовом уровне контейнер представляет собой набор инструкций, который выполняется на некоем процессоре, обрабатывая какие-то данные.

### Что такое контейнеры и почему они важны:

Контейнеры — это легковесные и изолированные среды, содержащие приложения и все необходимые для их работы компоненты, такие как библиотеки и зависимости. Контейнеры позволяют упаковывать приложения вместе с их средой выполнения, обеспечивая портативность и надежность. Это важно, потому что контейнеры упрощают процессы разработки, тестирования и развертывания, а также обеспечивают однообразие окружений между разработчиками, тестировщиками и операционными командами.

### Преимущества и недостатки контейнеризации:

Преимущества контейнеризации включают:

1. **Легковесность:** Контейнеры используют общую ОС и ресурсы хоста, что делает их более легкими и экономичными по сравнению с виртуальными машинами.

2. **Изоляция:** Каждый контейнер изолирован от других, что обеспечивает безопасность и избегает конфликтов между приложениями.
3. **Портативность:** Контейнеры могут работать на разных платформах и окружениях, что делает их переносимыми и совместимыми.
4. **Быстрое развертывание:** Контейнеры могут быть запущены практически мгновенно, что ускоряет процесс развертывания приложений.

Недостатки контейнеризации:

1. **Ограничения ОС:** Контейнеры зависят от хост-ОС, что может ограничить поддерживаемые платформы.
2. **Сетевая изоляция:** Иногда может возникнуть сложность в обеспечении изоляции контейнеров в сетевом окружении.

#### Сравнение с виртуализацией и традиционными методами развертывания:

Контейнеризация отличается от традиционных методов развертывания и виртуализации:

1. **Виртуализация:** Виртуальные машины (ВМ) полностью эмулируют физический компьютер, включая ОС и ресурсы. Контейнеры используют общую ОС хоста и обеспечивают легковесную изоляцию, что делает их более эффективными и быстрыми.
2. **Традиционные методы развертывания:** Раньше развертывание приложений могло быть трудоемким из-за сложности управления зависимостями и конфигурацией. Контейнеризация решает эту проблему, обеспечивая контролируемое окружение.

### 24.1.6 Практическое задание: Развертывание Python кода в Docker контейнере

**Цель:** Ознакомиться с процессом развертывания приложения на Python в Docker контейнере.

#### Задачи:

1. **Установка Docker:** Убедитесь, что на вашем компьютере установлен Docker. Если нет, следуйте инструкциям на официальном сайте Docker (<https://docs.docker.com/get-docker/>).
2. **Создание Python приложения:** Создайте простое приложение на Python. Например, это может быть «Hello, World!» приложение.
3. **Создание Dockerfile:** В папке с вашим Python кодом создайте файл с именем «Dockerfile». В Dockerfile опишите инструкции для создания образа. Например:

```
# Используем базовый образ Python
FROM python:3.9

# Копируем содержимое текущей папки в папку /app в образе
COPY . /app

# Устанавливаем зависимости
RUN pip install -r /app/requirements.txt

# Указываем рабочую директорию
WORKDIR /app

# Запускаем Python приложение
CMD ["python", "app.py"]
```

4. **Создание requirements.txt:** Если ваше приложение использует сторонние библиотеки, создайте файл «requirements.txt» и укажите их там.
5. **Сборка Docker образа:** В терминале перейдите в папку с Dockerfile и выполните команду для сборки Docker образа:

```
docker build -t my-python-app .
```

Где «my-python-app» - это имя образа, а точка означает текущую директорию.

6. **Запуск контейнера:** После успешной сборки образа, запустите контейнер командой:

```
docker run my-python-app
```

Вы увидите вывод вашего Python приложения в терминале.

7. **Оптимизация Dockerfile:** Попробуйте оптимизировать Dockerfile, чтобы уменьшить размер образа. Например, уберите лишние файлы, используйте более легковесные базовые образы и т.д.
8. **Развертывание приложения:** Попробуйте изменить ваш Python код и повторите шаги 5-6 для обновления контейнера с новым кодом.

**Замечание:** Это задание предполагает базовое понимание Docker и Python. Если вы не знакомы с Docker или Python, рекомендуется пройти соответствующие обучающие курсы.

В подавляющем большинстве контейнеров на данный момент используются минимальные образы Linux. Одним из таких примеров является проект Alpine. Подобные миниатюрные сборки ОС позволяют организовать удобную среду для мониторинга приложений и управления ими. Помимо прочего, ОС Linux обладает большим набором эффективных и не требовательных к вычислительным ресурсам инструментов для решения широкого спектра задач. Это и является основной причиной ее выбора при разработке отказоустойчивых, гибких и высоконагруженных решений. Программист, конечно, может запустить код на Python в Docker контейнере без использования образа с ОС, но в случае возникновения ошибки в процессе работы программы или сбоя в работе контейнера зафиксировать его и определить причину сбоя будет достаточно сложно, как и корректно перезапустить работу контейнера, избежав потери данных. Подробней познакомимся с тем, как применять ОС Alpine Linux в образах Docker.

#### **Практическое задание: Развёртывание Python кода в Docker контейнере с использованием образа Alpine**

**Цель:** Освоить процесс развертывания приложения на Python в Docker контейнере с использованием образа Alpine Linux.

##### **Задачи:**

1. **Установка Docker:** Убедитесь, что на вашем компьютере установлен Docker. Если нет, следуйте инструкциям на официальном сайте Docker (<https://docs.docker.com/get-docker/>).
2. **Создание Python приложения:** Напишите простое Python приложение, которое будет выводить «Hello, Docker with Alpine!».
3. **Создание Dockerfile:** В папке с вашим Python кодом, создайте файл с именем «Dockerfile». В Dockerfile опишите инструкции для создания Docker образа на основе образа Alpine. Пример Dockerfile:

```
# Используем базовый образ Alpine
FROM python:3.9-alpine

# Копирую содержимое текущей папки в папку /app в образе
COPY . /app
```

(continues on next page)

(продолжение с предыдущей страницы)

```
# Устанавливаем зависимости
RUN pip install -r /app/requirements.txt

# Указываем рабочую директорию
WORKDIR /app

# Запускаем Python приложение
CMD ["python", "app.py"]
```

4. **Создание requirements.txt:** Если ваше приложение использует сторонние библиотеки, создайте файл «requirements.txt» и укажите их там.
5. **Сборка Docker образа:** В терминале, перейдите в папку с Dockerfile и выполните команду для сборки Docker образа:

```
docker build -t my-python-app-alpine .
```

Где «my-python-app-alpine» - это имя образа, а точка означает текущую директорию.

6. **Запуск контейнера:** После успешной сборки образа, запустите контейнер командой:

```
docker run my-python-app-alpine
```

Вы увидите вывод «Hello, Docker with Alpine!» в терминале.

7. **Изменение кода:** Измените текст вывода вашего Python приложения на «Hello, Docker with Alpine! This is my Alpine containerized app.»
8. **Обновление контейнера:** Повторите шаги 5-6 для обновления контейнера с новым кодом.
9. **Оптимизация Dockerfile:** Попробуйте оптимизировать Dockerfile, чтобы уменьшить размер образа. Уберите лишние файлы и зависимости, используйте более легковесные базовые образы.

**Примечание:** Это задание предполагает базовое знание Docker и Python. Если вы не знакомы с Docker или Python, рекомендуется предварительно изучить соответствующие темы.

#### 24.1.7 Заключение

Виртуализация и контейнеризация являются ключевыми компонентами для эффективной разработки, доставки и управления программными приложениями в современных условиях. Эти инструменты существенно ускорили и улучшили процессы разработки, а также позволили быстро адаптироваться к изменяющимся требованиям в ИТ-отрасли.

Вместе с виртуализацией и контейнеризацией пришла также потребность в эффективном управлении всей инфраструктурой. Именно здесь на сцену вышли инструменты для автоматизации и управления, такие как оркестровщики и системы управления контейнерами. Оркестровщики, такие как Kubernetes, представляют собой мощное средство для автоматизированного развертывания, управления и масштабирования контейнеризированных приложений. Они позволяют организовывать вычислительные кластеры из контейнеров, управлять ими и обеспечивать доступ к сервисам и приложениям.

Также, появление инструментов для управления конфигурациями, таких как Ansible, позволило автоматизировать процессы установки, настройки и обновления программных компонентов в контейнерах и виртуальных машинах.

Следует также отметить важность DevOps-подхода в современной разработке. DevOps способствует сближению разработки и операций, позволяя создавать и поддерживать приложения более эффективно

и безболезненно. Он базируется на принципах автоматизации, непрерывной интеграции и непрерывного развертывания (CI/CD).

В итоге, эволюция разработки программного обеспечения привела к созданию более эффективных, гибких и автоматизированных инструментов и методологий. Они позволяют командам разработчиков быстро создавать, тестировать, разворачивать и масштабировать приложения в современных динамичных условиях. Все эти инновации существенно повлияли на способы работы ИТ-специалистов и дали им новые возможности для творчества, инноваций и быстрой реализации идей.

#### 24.1.8 Ссылки на дополнительные материалы по Docker

- A Beginner-Friendly Introduction to Containers, VMs and Docker <https://www.freecodecamp.org/news/a-beginner-friendly-introduction-to-containers-vms-and-docker-79a9e3e119b/>
- Книга «Полная виртуализация» <http://onreader.mdl.ru/VirtualizationComplete/content/index.html>
- Docker: основы работы <https://otus.ru/journal/docker-osnovy-raboty/>
- Docker - устройство и принципы работы <https://teletype.in/@jt255ru/OuwbmZoHuEu>
- Часть 1: основы <https://habr.com/post/438796/>
- Часть 2: термины и концепции <https://habr.com/post/439978/>
- Часть 3: файлы Dockerfile <https://habr.com/post/439980/>
- Часть 4: уменьшение размеров образов и ускорение их сборки <https://habr.com/post/440658/>
- Часть 5: команды <https://habr.com/post/440660/>
- Изучаем Docker, часть 6: работа с данными <https://habr.com/ru/companies/ruvds/articles/441574/>

## 24.2 Практическое задание 1: Развёртывание Python кода в Docker контейнере

Цель: Ознакомиться с процессом развертывания приложения на Python в Docker контейнере.

Задачи:

Установка Docker: Убедитесь, что на вашем компьютере установлен Docker. Если нет, следуйте инструкциям на официальном сайте Docker (<https://docs.docker.com/get-docker/>).

Создание Python приложения: Создайте простое приложение на Python. Например, это может быть «Hello, World!» приложение.

Создание Dockerfile: В папке с вашим Python кодом создайте файл с именем «Dockerfile». В Dockerfile опишите инструкции для создания образа. Например:

```
# Используем базовый образ Python
FROM python:3.9

# Копируем содержимое текущей папки в папку /app в образе
COPY . /app

# Устанавливаем зависимости
RUN pip install -r /app/requirements.txt
```

(continues on next page)

(продолжение с предыдущей страницы)

```
# Указываем рабочую директорию
WORKDIR /app

# Запускаем Python приложение
CMD ["python", "app.py"]
```

Создание requirements.txt: Если ваше приложение использует сторонние библиотеки, создайте файл «requirements.txt» и укажите их там.

Сборка Docker образа: В терминале перейдите в папку с Dockerfile и выполните команду для сборки Docker образа:

`docker build -t my-python-app .` Где «my-python-app» - это имя образа, а точка означает текущую директорию.

Запуск контейнера: После успешной сборки образа, запустите контейнер командой:

```
docker run my-python-app
```

Вы увидите вывод вашего Python приложения в терминале.

Оптимизация Dockerfile: Попробуйте оптимизировать Dockerfile, чтобы уменьшить размер образа. Например, уберите лишние файлы, используйте более легковесные базовые образы и т.д.

Развертывание приложения: Попробуйте изменить ваш Python код и повторите шаги 5-6 для обновления контейнера с новым кодом.

Замечание: Это задание предполагает базовое понимание Docker и Python. Если вы не знакомы с Docker или Python, рекомендуется пройти соответствующие обучающие курсы.

В подавляющем большинстве контейнеров на данный момент используются минимальные образы Linux. Одним из таких примеров является проект Alpine. Подобные миниатюрные сборки ОС позволяют организовать удобную среду для мониторинга приложений и управления ими. Помимо прочего, ОС Linux обладает большим набором эффективных и не требовательных к вычислительным ресурсам инструментов для решения широкого спектра задач. Это и является основной причиной ее выбора при разработке отказоустойчивых, гибких и высоконагруженных решений. Программист, конечно, может запустить код на Python в Docker контейнере без использования образа с ОС, но в случае возникновения ошибки в процессе работы программы или сбоя в работе контейнера зафиксировать его и определить причину сбоя будет достаточно сложно, как и корректно перезапустить работу контейнера, избежав потери данных. Подробней познакомимся с тем, как применять ОС Alpine Linux в образах Docker.

### 24.2.1 Практическое задание 2: Развёртывание Python кода в Docker контейнере с использованием образа Alpine

Цель: Освоить процесс развертывания приложения на Python в Docker контейнере с использованием образа Alpine Linux.

Задачи:

Установка Docker: Убедитесь, что на вашем компьютере установлен Docker. Если нет, следуйте инструкциям на официальном сайте Docker (<https://docs.docker.com/get-docker/>).

Создание Python приложения: Напишите простое Python приложение, которое будет выводить «Hello, Docker with Alpine!».

Создание Dockerfile: В папке с вашим Python кодом, создайте файл с именем «Dockerfile». В Dockerfile опишите инструкции для создания Docker образа на основе образа Alpine. Пример Dockerfile:

```
# Используем базовый образ Alpine
FROM python:3.9-alpine

# Копируем содержимое текущей папки в папку /app в образе
COPY . /app

# Устанавливаем зависимости
RUN pip install -r /app/requirements.txt

# Указываем рабочую директорию
WORKDIR /app

# Запускаем Python приложение
CMD ["python", "app.py"]
```

Создание requirements.txt: Если ваше приложение использует сторонние библиотеки, создайте файл «requirements.txt» и укажите их там.

Сборка Docker образа: В терминале, перейдите в папку с Dockerfile и выполните команду для сборки Docker образа:

docker build -t my-python-app-alpine . Где «my-python-app-alpine» - это имя образа, а точка означает текущую директорию.

Запуск контейнера: После успешной сборки образа, запустите контейнер командой:

```
docker run my-python-app-alpine
```

Вы увидите вывод Hello, Docker with Alpine! в терминале.

Изменение кода: Измените текст вывода вашего Python приложения на «Hello, Docker with Alpine! This is my Alpine containerized app.»

Обновление контейнера: Повторите шаги 5-6 для обновления контейнера с новым кодом.

Оптимизация Dockerfile: Попробуйте оптимизировать Dockerfile, чтобы уменьшить размер образа. Уберите лишние файлы и зависимости, используйте более легковесные базовые образы.

Примечание: Это задание предполагает базовое знание Docker и Python. Если вы не знакомы с Docker или Python, рекомендуется предварительно изучить соответствующие темы.

## 24.3 Вопросы

1. Что такое Docker?
2. В чем разница между виртуализацией и контейнеризацией?
3. Что такое Docker контейнер и какие у него преимущества?
4. Что такое Образ Docker?
5. Объясните архитектуру Docker.
6. Что такое Docker Hub?
7. Что такое Dockerfile?

8. Как создать образ Docker?
9. Как запустить образ Docker, которого нет на локальном компьютере?
10. Как запустить программу на Python внутри Docker контейнера?
11. Какие инструменты могут помочь в автоматизации процесса работы Docker?
12. Какой командой мы можем отобразить в консоли список всех запущенных контейнеров?
13. Какие компоненты составляют архитектуру Docker?
14. Какова роль Docker Image (образа) в процессе создания контейнеров?
15. Какими способами можно получить Docker Image?
16. Что такое Dockerfile, и как он используется для создания Docker Image?
17. Какие базовые команды Docker используются для управления контейнерами?
18. Как запустить контейнер на основе определенного образа?
19. Как передать порты из контейнера в хост-систему?
20. Как передать данные между контейнерами в Docker?
21. Как узнать список активных контейнеров на системе?
22. Как остановить и удалить контейнер в Docker?
23. Как работает Docker Volume, и зачем он нужен?
24. Как просматривать логи контейнеров в Docker?
25. Как присваивать имена контейнерам в Docker?
26. Какие сетевые режимы поддерживает Docker, и что они обозначают?
27. Какие есть методы для сохранения изменений в контейнере (persisting data)?
28. Что такое Docker Hub и как его использовать для обмена Docker Image с другими?



## Глава 25

---

Дополнительные материалы

---



## Работа с сетью в Docker

---

### 26.1 Концепция сети в Docker

#### 26.1.1 Познакомимся с тем, как объединять несколько контейнеров в сеть Docker

Сеть Docker в основном используется для установки связи между контейнерами Docker и внешним миром через хост-машину. Также можно сказать, что это коммуникационный канал, через который все изолированные контейнеры взаимодействуют друг с другом в различных ситуациях для выполнения необходимых действий.

Перед тем как приступить к основной части материала, закрепим некоторые базовые принципы, термины и определения относительно работы сетей, чтобы избежать противоречий и ошибочного восприятия.

#### 26.1.2 Сети: вводная часть

##### Что такое сеть?

Локальная сеть (Local Area Network, LAN) — объединение компьютеров, сосредоточенных на небольшой территории. В общем случае локальная сеть представляет собой коммуникационную систему, принадлежащую одной организации: школа, университет, больница, банк и т.д. При построении сети используют следующие компоненты:

1. Оконечные узлы: устройства, которые передают и/или принимают какие-либо данные. Это могут быть смартфоны, компьютеры, телевизоры, сервера, различные терминалы приема оплаты. Например: метро, сотовая связь, услуги ЖКХ и т.д.
2. Промежуточные устройства: это устройства, которые соединяют оконечные узлы между собой. Сюда можно отнести коммутаторы, концентраторы, модемы, маршрутизаторы, точки доступа Wi-Fi, базовые станции мобильных операторов.
3. Сетевые среды: это те среды, в которых происходит непосредственная передача данных. Сюда относятся проводные кабели (например, Ethernet), сетевая карта вашего компьютера. Если

это кабель, то передача данных осуществляется при помощи электрических сигналов. У оптоволоконных кабелей, при помощи световых импульсов, и радиоволны в случае использования беспроводной среды (WiFi, Bluetooth, LoRa и др.).

Следующее, что нам надо знать — это порт и протокол.

- Порт — это цифра. Например, 21, 22, 25, 80, 433, 8080. Это число записывается в заголовках протоколов транспортного уровня (об этом ниже). Порт указывает, для какой программы предназначен тот или иной пакет (информация, передаваемая одним приложением другому). Например, http-сервер работает через порт 80. Когда вы открываете браузер, вы отправляете запрос на веб-сервер через 80 порт, и сервер понимает, что это http-запрос и вам нужен сервер, который передаст вам страницу в формате html (ответ сервера).
- Протокол. Например, TCP/IP — это адрес узла (компьютера) с указанием порта и передаваемых данных. Чтобы передать информацию по протоколу TCP/IP, нужно указать следующие данные:

Адрес отправителя (Source address):

IP: 82.146.49.11

Port: 2049

Адрес получателя (Destination address):

IP: 195.34.32.111

Port: 53

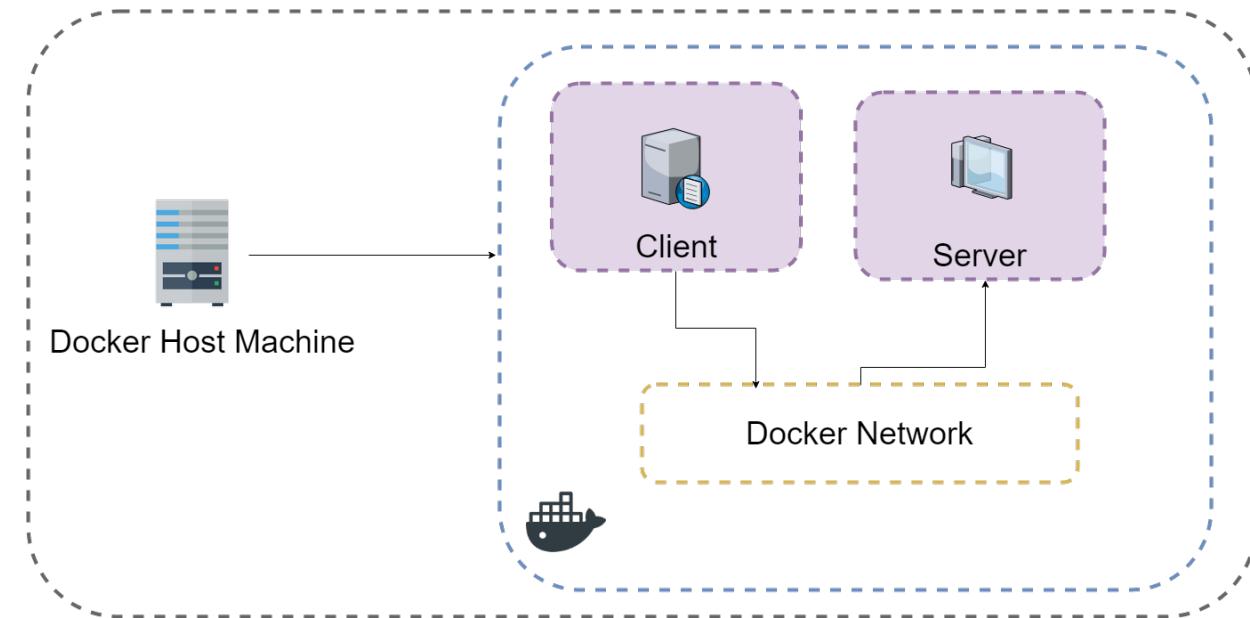
Данные пакета:

Send test message

Благодаря этим данным информация будет передана на нужный узел. По такому принципу формируется клиент-серверная передача данных.

Клиент — это модуль, предназначенный для формирования и передачи сообщений-запросов к ресурсам удаленного компьютера от разных приложений с последующим приемом результатов из сети и передачей их соответствующим приложениям.

Сервер — это модуль, который постоянно ожидает прихода из сети запросов от клиентов и, приняв запрос, пытается его обработать, как правило, с участием локальной ОС; один сервер может обслуживать запросы сразу нескольких клиентов (поочередно или одновременно).



## Объяснение работы сетей в контейнерах Docker

В зависимости от установленных версий Docker и операционной системы (ОС) на вашем компьютере, в редких случаях работа с сетью может отличаться. Рекомендуем использовать Docker последней стабильной версии, чтобы избежать возможных ошибок. Все команды, перечисленные в текущем документе, протестированы в ОС Linux дистрибутива Ubuntu с правами root (суперпользователь).

Для управления сетевыми операциями, такими как создание новой сети, подключение контейнера к сети, отключение контейнера от сети, перечисление доступных сетей, удаление сетей и т. д., мы используем следующую команду: `docker network`.

```
root@e4756e86766c:/# docker network

Usage: docker network COMMAND

Manage networks

Commands:
  connect      Connect a container to a network
  create       Create a network
  disconnect   Disconnect a container from a network
  inspect      Display detailed information on one or more networks
  ls           List networks
  prune        Remove all unused networks
  rm           Remove one or more networks

Run 'docker network COMMAND --help' for more information on a command.
root@e4756e86766c:/#
```

## Типы сетевых драйверов Docker

Чтобы получить список всех ваших сетей, запустите `docker network ls`.

| NETWORK ID   | NAME                  | DRIVER | SCOPE |
|--------------|-----------------------|--------|-------|
| bace86f87ab5 | bridge                | bridge | local |
| 4a11cd024610 | fastapi-admin_default | bridge | local |
| 126a5fe90532 | host                  | host   | local |
| 88466374fb9b | myNet01               | bridge | local |
| f50ce57f0cdd | my_network            | bridge | local |
| 5f06df5e78de | mynet                 | bridge | local |
| ec87154c4ee8 | none                  | null   | local |

Кратко познакомимся с их описанием.

1. **Bridge network:** при запуске Docker автоматически создается сеть типа мост по умолчанию. Недавно запущенные контейнеры будут автоматически подключаться к нему. Вы также може-

те создавать пользовательские настраиваемые мостовые сети. Пользовательские мостовые сети превосходят сетевые мосты по умолчанию.

2. **Host network:** удаляет сетевую изоляцию между контейнером и хостом Docker и напрямую использует сеть хоста. Если вы запускаете контейнер, который привязывается к порту 80, и вы используете хост-сеть, приложение контейнера доступно через порт 80 по IP-адресу хоста. Это означает, что вы не сможете запускать несколько веб-контейнеров на одном хосте, на одном и том же порту, так как порт теперь является общим для всех контейнеров в сети хоста.
3. **None network:** в сети такого типа контейнеры не подключены ни к одной сети и не имеют доступа к внешней сети или другим контейнерам. Итак, эта сеть используется, когда вы хотите полностью отключить сетевой стек в контейнере.
4. **Overlay network:** создает внутреннюю частную сеть, которая охватывает все узлы, участвующие в кластере swarm. Таким образом, оверлейные сети облегчают обмен данными между сервисом **Docker Swarm** и автономным контейнером или между двумя автономными контейнерами на разных демонах Docker.
5. **Macvlan network:** некоторые приложения, особенно устаревшие приложения, отслеживающие сетевой трафик, ожидают прямого подключения к физической сети. В такой ситуации вы можете использовать сетевой драйвер *Macvlan* для назначения MAC-адреса виртуальному сетевому интерфейсу каждого контейнера, что делает его физическим сетевым интерфейсом, напрямую подключенным к физической сети.

### 26.1.3 Познакомимся с сетями Bridge и Host на практике

1. Сеть типа **Bridge** Будем использовать два контейнера с ОС Alpine для объяснения этого типа сети. Для удобства в качестве имен контейнеров будем использовать короткие названия.
  - Первый контейнер назовем: **C1**.
  - Второй контейнер назовем: **C2**.

Чтобы запустить наши тестовые Docker контейнеры с образом Alpine, используем следующие команды:

```
# docker run -it -d --name c1 alpine ash
# docker run -it -d --name c2 alpine ash
```

Далее давайте выясним IP-адреса этих запущенных контейнеров. Для этого выполним следующие команды:

```
# docker exec -it c1 sh -c "ip a"
# docker exec -it c2 sh -c "ip a"
```

Результаты выполнения первой и второй команды представлены на рисунках.

```
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
        valid_lft forever preferred_lft forever
2: tunl0@NONE: <NOARP> mtu 1480 qdisc noop state DOWN qlen 1000
    link/ipip 0.0.0.0 brd 0.0.0.0
3: sit0@NONE: <NOARP> mtu 1480 qdisc noop state DOWN qlen 1000
    link/sit 0.0.0.0 brd 0.0.0.0
26: eth0@if27: <BROADCAST,MULTICAST,UP,LOWER_UP,M-DOWN> mtu 1500 qdisc noqueue state UP
    link/ether 02:42:ac:11:00:02 brd ff:ff:ff:ff:ff:ff
    inet 172.17.0.2/16 brd 172.17.255.255 scope global eth0
        valid_lft forever preferred_lft forever
```

Рисунок 1 — Отображение IP-адреса первого контейнера C1

```
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
        valid_lft forever preferred_lft forever
2: tunl0@NONE: <NOARP> mtu 1480 qdisc noop state DOWN qlen 1000
    link/ipip 0.0.0.0 brd 0.0.0.0
3: sit0@NONE: <NOARP> mtu 1480 qdisc noop state DOWN qlen 1000
    link/sit 0.0.0.0 brd 0.0.0.0
28: eth0@if29: <BROADCAST,MULTICAST,UP,LOWER_UP,M-DOWN> mtu 1500 qdisc noqueue state UP
    link/ether 02:42:ac:11:00:03 brd ff:ff:ff:ff:ff:ff
    inet 172.17.0.3/16 brd 172.17.255.255 scope global eth0
        valid_lft forever preferred_lft forever
```

Рисунок 2 — Отображение IP-адреса первого контейнера C1

Как видите, IP-адрес контейнера C1 — **172.17.0.2**, а IP-адрес C2 — **172.17.0.3**. Теперь продолжим наше знакомство с сетевыми утилитами и попытаемся проверить их связь друг с другом. Для этого будем использовать команду **ping**, чтобы убедиться в том, что они могут обращаться друг к другу. Для начала подключимся к контейнеру **C1** командой **docker attach c1** и проверим связь с контейнером **C2**. Для этого выполним команды:

```
# docker attach c1
# Ping -c 2 172.17.0.3
```

Как вы можете видеть на скриншотах, показанных выше, происходит коннекшн между контейнерами в одной сети. Мы также можем проверить сеть типа мост с помощью команды: **docker network inspect bridge**. Приведенная команда отобразит всю информацию о сети, такую как тип сети, подсеть, шлюз, имя контейнера и IP-адрес и т.д.

### Создание пользовательской Bridge сети

При запуске Docker автоматически создается сеть bridge по умолчанию. Все вновь запущенные контейнеры будут автоматически подключаться к ней. Однако вы также можете создавать пользовательские мостовые сети. Чтобы создать новый сетевой драйвер, просто запустите: **docker network create my\_net** или **docker network create --driver bridge dhruv\_net**.

Обе команды будут выполнять одинаковую работу. Если вы не укажете имя драйвера, сеть будет создана в сетевом драйвере по умолчанию, т.е. bridge.

В пользовательских сетях, таких как **dhruv\_net**, контейнеры могут не только связываться по IP-адресу, но также могут преобразовывать имя контейнера в IP-адрес. Эта возможность называется автоматическим обнаружением службы.

Чтобы убедиться, что контейнеры могут взаимодействовать друг с другом, давайте запустим три alpine контейнера, а именно, A1, A2 и A3 в сети **dhruv\_net**, которую мы создали ранее.

```
docker run -it -d --name A1 --network dhruv_net alpine ash
docker run -it -d --name A2 --network dhruv_net alpine ash
docker run -it -d --name A3 --network dhruv_net alpine ash
```

Теперь попробуйте подключиться к любому из контейнеров и пропинговать два других, используя имя контейнера.

Из приведенных выше снимков экрана доказано, что контейнеры могут общаться друг с другом.

2. Хостовая Сеть Мы запускаем контейнер, который привязывается к порту 80, используя хост-сеть, приложение контейнера доступно через порт 80 по IP-адресу хоста.

Сеть типа Host нужна только тогда, когда вы запускаете программы с очень специфической сетью.

Приложения, работающие внутри контейнера Docker, выглядят так, как будто они работают на самом хосте с точки зрения сети. Это позволяет контейнеру получить больший доступ к сети, чем он может получить.

Здесь мы использовали команду `netstat -ntlp` для отображения порта прослушивания на сервере.

Мы рассмотрели только основы сетевых концепций Docker. Для более подробной информации предлагаем вам заглянуть в руководство по работе с сетями Docker, которое прилагается ниже.  
<https://docs.docker.com/v17.09/engine/userguide/networking/#exposing-and-publishing-ports>

## Реестр Docker

### 27.1 Реестр Docker

В предыдущем разделе мы уже познакомились с технологией Docker и узнали, что такое Docker контейнер, Docker образ, как связаны эти сущности и как мы можем их использовать. Теперь пришло время познакомиться с таким понятием, как Docker Registry (реестр Docker).

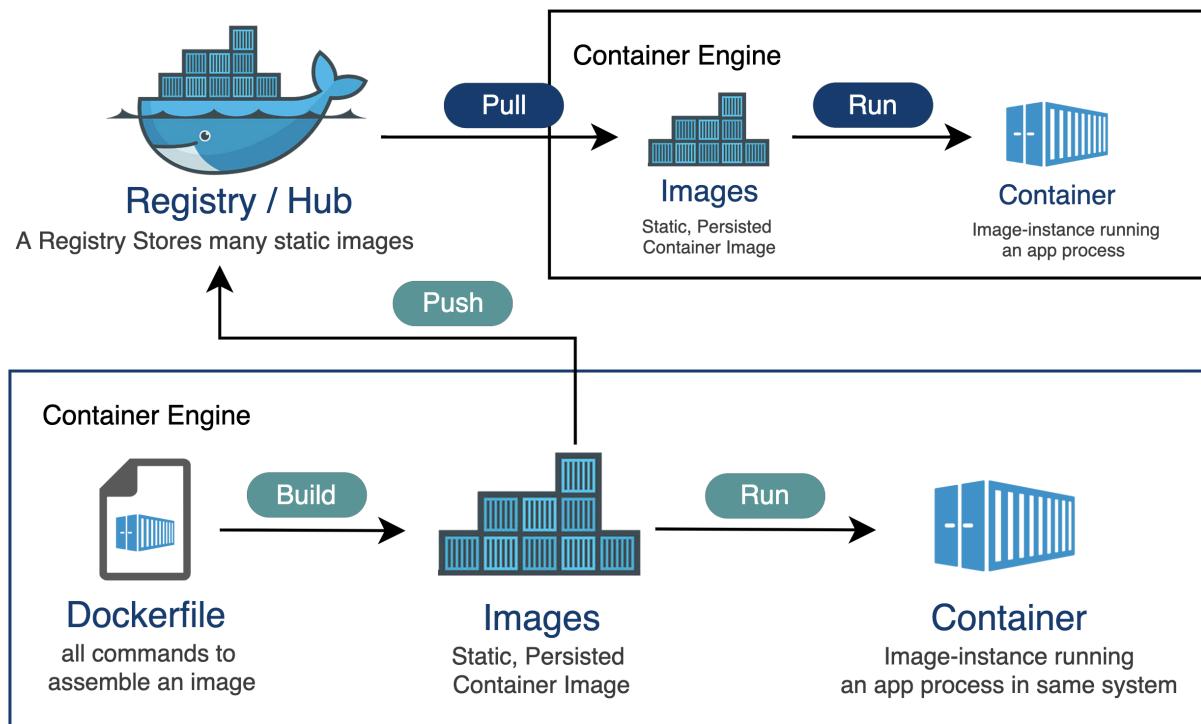


В этом разделе мы познакомимся с сервисом «**Docker Hub**», узнаем о его возможностях, как использовать его для поиска Docker образов, созданных сторонними разработчиками, опубликовавшими свои проекты в виде образа. Рассмотрим все этапы работы с сервисом «**Docker Hub**». Узнаем как скачать и загрузить образ в «**Docker Hub**». Создадим там собственный репозиторий, который позволит делиться созданными нами Docker образами.

Вам требуется базовое понимание Docker, знакомство с окружением Windows и Linux. В этом руководстве будет использоваться инструмент Docker Desktop для выполнения текущих упражнений, т.е. для создания и передачи образа в Docker Hub.

Хотя образы Docker легко создаются, и разработчикам удобно работать с простыми и портативными образами Docker, быстро обнаружилось, что управлять тысячами образов Docker довольно трудно. Решить эту проблему позволяет реестр Docker. Реестр Docker — это стандартный способ хранения и распространения образов Docker. Реестр — это репозиторий с открытым исходным кодом, имеющий разрешительную лицензию Apache.

Реестр Docker также помогает улучшить управление правами доступа и безопасность образов Docker, хранящихся в его репозитории. Он управляет распространением образов, а также может интегрироваться с рабочими процессами разработки приложений. Разработчики могут настроить собственный реестр Docker или использовать сервис размещения, например, Docker Hub, Oracle Container Registry, Azure Container Registry и т.д.



-Docker Flow - процесс работы Рисунок 1. Docker Flow — процесс работы

Docker Hub — это размещенный реестр Docker, управляемый Docker. Docker Hub содержит более 100 000 образов контейнеров от поставщиков программного обеспечения, а также проекты с открытым исходным кодом и сообщества. Docker Hub содержит программное обеспечение и приложения из официальных репозиториев, таких как NGINX, Apache, MySQL, Ubuntu, Python, Flask и т.д.

Он позволяет нам извлекать и добавлять Docker образы в Docker Hub. Его также можно сравнить с GitHub, где мы получаем и отправляем наш исходный код, но в случае Docker Hub мы работаем с образами Docker контейнеров.

Облачный онлайн-репозиторий «Docker Hub» использует два типа репозиториев:

- общедоступный репозиторий;
- частный репозиторий.

Общедоступные репозитории доступны всем пользователям сервиса Docker Hub, в то время как частный репозиторий доступен только его владельцу (аналогично репозиториям на GitHub).

При запуске контейнера Docker по умолчанию автоматически извлекает соответствующий образ из общедоступного Docker Hub, если он недоступен локально. Вы также можете создавать собственные

образы и отправлять их в Docker Hub в общедоступный или частный репозиторий.

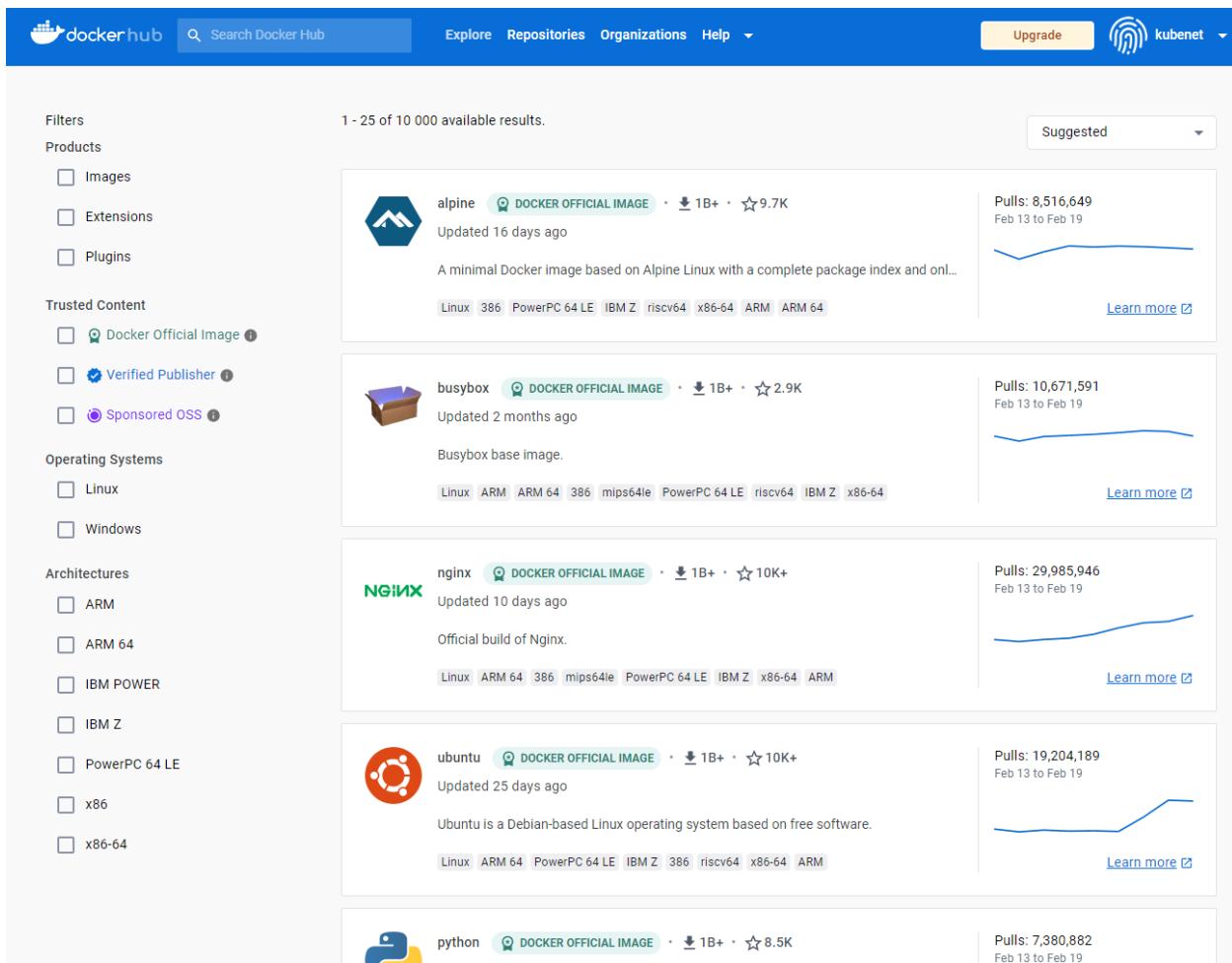


Рисунок 2. Снимок экрана реестра Docker

### 27.1.1 Создание первого Docker репозитория

Первым делом требуется авторизоваться в Docker Hub, используя ваши учетные данные. Если у вас нет учетной записи, вы можете создать ее, нажав кнопку Register (Регистрация) на странице <https://hub.docker.com>. После того как вы авторизовались на сервисе, вы можете создать репозиторий, нажав на кнопку «Create repository» (Создать репозиторий).

## toolchain, Выпуск 1.0

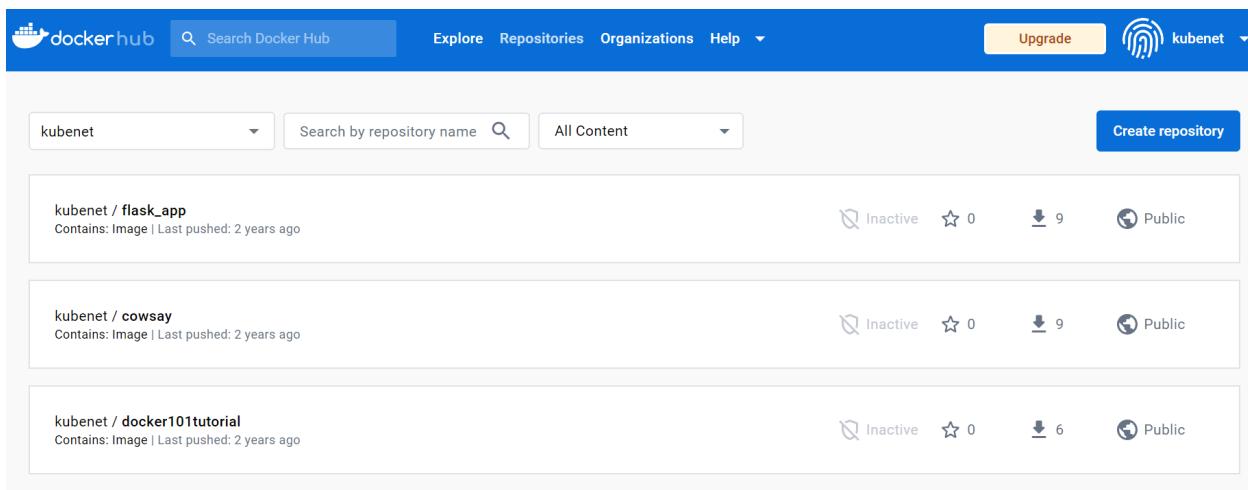


Рисунок 3. Создание Docker Hub репозитория

Откроется окно, в котором предлагается заполнить краткую информацию о новом репозитории. В качестве примера создадим открытый (публичный) репозиторий с названием «**my\_web\_service**».

Docker Hub предоставляет нам только один частный репозиторий бесплатно. Если нам нужно больше частных репозиториев, мы можем обновить нашу учетную запись до платной.

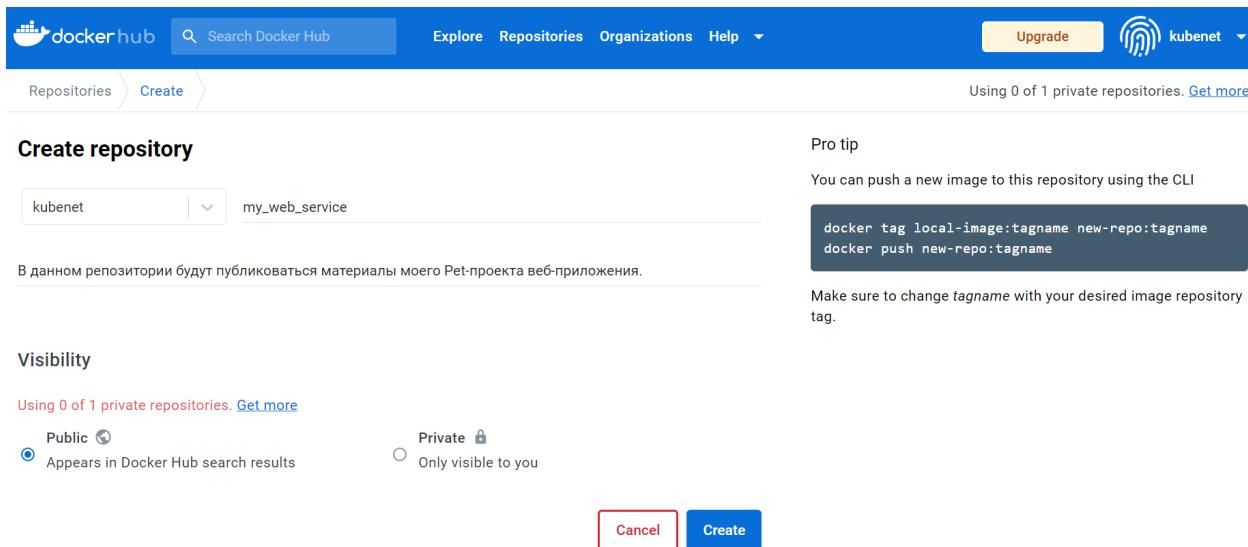


Рисунок 4. Параметры нового репозитория Docker Hub

Как только все будет сделано, нажмите Создать (Create). Поздравляем! Вы создали свой первый репозиторий, который будут выглядеть следующим образом.

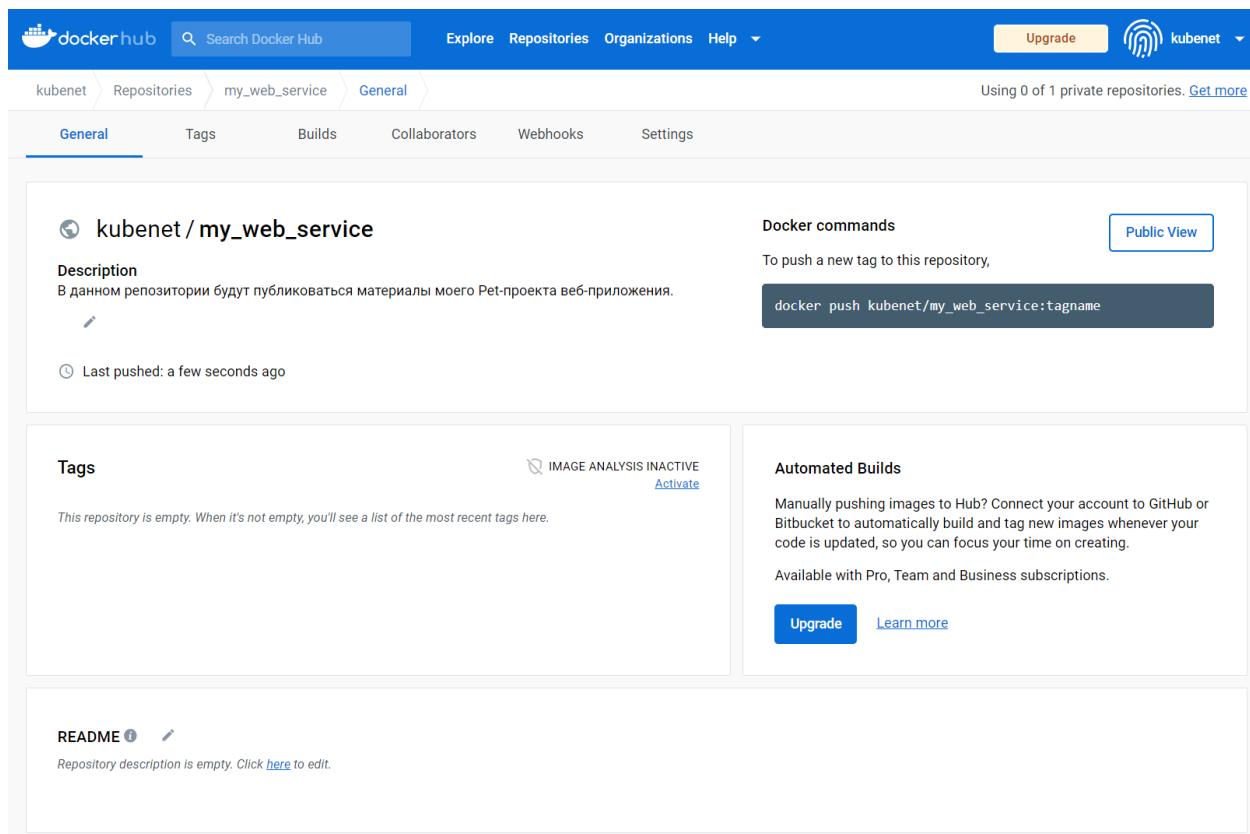


Рисунок 5. Созданный репозиторий Docker Hub

Теперь откройте приложение Docker Desktop tool или консоль (terminal) Docker и войдите в Docker Hub с помощью команды `docker login`.

### 27.1.2 Поиск Docker образа

Для поиска образа на Docker Hub нам доступны 2 способа:

- Поиск на веб-сайте Docker Hub через браузер. Для этого надо перейти в раздел «Explore» в верхнем меню Docker Hub.
- Через командную строку (терминал). В таком случае, если мы хотим найти образ Flask, мы можем воспользоваться следующей командой: `docker search mysql`.

The screenshot shows the Docker Hub search interface for the query 'Flask'. The search bar at the top contains 'Flask'. Below it, the search results are displayed under the heading '1 - 25 of 10 000 results for Flask.' The results are filtered by 'Images' and sorted by 'Best Match'. Each result card includes the publisher's logo, the repository name, the publisher status (e.g., VERIFIED PUBLISHER), the number of pulls, the last update time, and a pull volume chart. The results are as follows:

- ibmcom/flask-ldap-login-ppc64le** (By IBM) - VERIFIED PUBLISHER, 1.0K pulls, updated 4 years ago. Description: Docker image for flask-ldap-login-ppc64le. Tags: Linux, ppc64le.
- rapidfort/flaskapp** (By RapidFort, Inc.) - VERIFIED PUBLISHER, 5.4K pulls, updated a day ago. Description: Docker image for flaskapp. Tags: Linux, x86-64.
- okteto/flask-cockroachdb** (By Okteto) - VERIFIED PUBLISHER, 184 pulls, updated 3 years ago. Description: Docker image for flask-cockroachdb. Tags: Linux, x86-64.
- openpolicyagent/flaskapp** (By openpolicyagent) - SPONSORED OSS, 1.9K pulls, updated 4 years ago. Description: Docker image for flaskapp. Tags: Linux, x86-64.

Преимуществом поиска Docker образа в веб-браузере является возможность поиска по ключевым словам и выбора наиболее подходящего образа. Есть возможность ознакомиться с его описанием, документацией и содержимым, а также скопировать команду для загрузки найденного Docker образа на свой компьютер (Docker Pull Command).

The screenshot shows the Docker Hub interface for the repository `ibmcom/flask-ldap-login-ppc64le`. At the top, there's a navigation bar with links for Explore, Repositories, Organizations, Help, Upgrade, and kubenet. A message in Russian says "Чтобы выйти из полноэкранного режима, нажмите F11". Below the header, the repository name is displayed with a verified publisher badge and a star icon. It shows 1.1K pulls. On the left, there's an IBM logo and a "Image" button. The main content area has tabs for Overview and Tags. The Overview tab is selected and contains sections for Supported Tags and respective Dockerfile links, Author (snehlata mohite smohite@us.ibm.com), and Usage Information. The Usage Information section includes Docker build and run commands. To the right, there's a Docker Pull Command section with a code block containing `docker pull ibmcom/flask-ldap...`.

### 27.1.3 Загрузка Docker образа из реестра Docker Hub

Чтобы загрузить Docker образ из Docker Hub, нужно использовать команду `pull` следующим образом: `docker pull <image_name>`. Например, если требуется скачать последнюю стабильную версию Docker образа `MySQL`, можно использовать следующую команду: `docker pull mysql`

Если образ `mysql` уже есть на нашем компьютере, то приведенная выше команда автоматически обновит образ до последней версии. Здесь нужно помнить одну вещь: если мы выполним поиск на Docker Hub, нам отобразится список различных образов `MySQL`. Нам нужно найти подходящий вариант в зависимости от решаемой задачи. Допустим, мы хотим получить образ `bitnami/mysql`. `docker pull bitnami/mysql`

### 27.1.4 Создание Docker образа

Этот процесс требует Dockerfile. Мы можем рассматривать Dockerfile как руководство по эксплуатации, которое говорит Docker, что собирать (подобие requirements.txt в venv для python). Проще говоря, это файл конфигурации, в котором хранится инструкция по сборке.

### 27.1.5 Как это работает?

Docker читает инструкцию из Dockerfile и автоматически создает образы. Образ Docker является многоуровневой файловой системой и состоит из нескольких слоев, доступных только для чтения, и каждый слой образа Docker представляет инструкции файла Docker. Давайте выполним следующие шаги, чтобы создать образ с помощью Dockerfile.

### 27.1.6 Создайте Dockerfile, который определяет конфигурацию нашего приложения

sudo vim Dockerfile \*Примечание. Имя файла должно быть Dockerfile с большой латинской буквы D.

```
FROM ubuntu:16.04
MAINTAINER someuser@somedomain.com
RUN apt-get update
RUN apt-get install -y mysql
CMD echo "My first image created."
```

Давайте посмотрим на некоторые важные ключевые слова, используемые в Dockerfile

Мы можем использовать символы #, чтобы добавить комментарий в Dockerfile Ключевое слово FROM определяет базовый образ, которое будет использоваться. Ключевое слово MAINTAINER — это пользователь, который будет поддерживать этот образ. Ключевое слово RUN используется для запуска инструкции, приведенной для образа. В нашем случае сначала обновите систему, а затем установите MySQL. Ключевое слово CMD используется для выполнения команды после запуска контейнера. Ключевое слово COPY можно использовать для копирования файла из нашей хост операционной системы в док-контейнер. Ключевое слово EXPOSE используется для указания номера порта, на котором контейнер будет выполнять свой процесс. Запустите команду ниже, чтобы создать наш образ Docker. Синтаксис docker build -t /repo-name

docker build -t <user\_name>/<repo\_name> . В приведенной выше команде имя изображения и символ точки « . » указывает текущий рабочий каталог. Это означает, что Docker ищет текущий каталог файла Docker. Опция -t используется для обозначения образа.

Вывод будет похож на:

```
docker build
```

Теперь давайте проверим наш образ, запустив его с помощью команды запуска docker.

```
docker run asadali08527/first-repo
```

Выполнение вышеупомянутой команды попросит ваш Dockerfile образ успешно установить MySQL на ваш компьютер со всеми необходимыми обновлениями, и, наконец, он также отобразит сообщение.

Заливаем образ на Docker Hub. Как только наш образ был успешно создан и запущен, мы можем отправить его в Docker Hub с помощью команды push.

```
docker push asadali08527/first-repo
```

Мы можем проверить теги образа и статус на Docker Hub, который будет выглядеть примерно так.

```
artem@pc:~$ sudo docker search asadali08527/first-repo
NAME                  DESCRIPTION          STARS      OFFICIAL   AUTOMATED
asadali08527/first-repo          0
artem@pc:~$
```

### 27.1.7 Образ на Docker Hub

Что такое DOCKER CERTIFIED образы? Это официальные образы поставщиков или участников. Образ может быть сертифицирован Docker Hub, только если его содержимое соответствует правилам, стандартам и законам, предусмотренным Docker Hub. Проще говоря, этот образ должен пройти определенные базовые тесты.

Docker Hub предоставляет инструмент `inspectDockerImage`, с помощью которого поставщик может самостоятельно сертифицировать образы и плагины (обычно поставщик или участник публикует свои плагины для регистрации томов и сетей).

**Примечание:** Чтобы опубликовать наш образ в качестве сертифицированного Docker образа, сначала мы должны самостоятельно сертифицировать и протестировать наш образ и плагин с помощью инструмента `inspectDockerImage`. После того как мы сертифицировали и успешно протестировали, после отправки Docker Hub подтвердит его. Если наш контент требует несертифицированной среды, он не может быть опубликован как сертифицированный ни при каких обстоятельствах.

#### Рекомендации

- Ищите конкретную версию Docker образа, используя теги, в основном последние. Это необходимо, чтобы избежать возможных конфликтов и сбоев в случае несовместимости пакетов в вашем проекте и библиотек, модулей в составе Docker образа.
- Выберите тот, который имеет максимальные загрузки и звезды.
- Проверьте его обновления (когда он был обновлен последний раз).
- Если возможно, проверьте его тип: проверенный издаватель или официальный (Docker Certified).