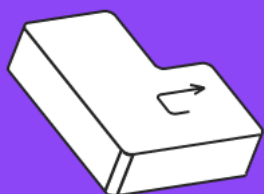


# Юнит-тестирование в других языках

Введение в юнит-тестирование



# Оглавление

Введение	4
Термины, используемые в лекции	5
Общая картина юнит-тестирования	5
На чем писать тесты	6
Python	7
Инструменты тестирования в Python	8
Пример работы с линтером Flake8	11
Пример работы с линтером Pylint	13
Юнит-тесты в Python	14
assert	15
Фреймворки тестирования Python	16
Pytest	16
Инструкция по установке Pytest	17
Практика PyTest	18
PyUnit	27
Robot Framework	28
Doctest	29
Mocking в Python	32
TDD	34
Выбор фреймворка для тестирования в Python	34
Code coverage	35
JavaScript	36
Golang (Go)	40
Выводы: какой язык выбрать для написания тестов	44

Антипаттерны тестирования	46
Автоматизация тестирования	49
ИИ и нейросети для юнит-тестирования	51
Итоги курса	52
Глоссарий курса	54
Домашнее задание	55
Полезные материалы	55

# Введение

На заключительном уроке мы узнаем, как проводить юнит-тестирование на разных языках программирования.

Юнит-тестирование — это процесс тестирования отдельных блоков кода (юнитов) для проверки на корректность и соответствие требованиям. Помогает обеспечить качество ПО и уменьшить риски, связанные с его разработкой и сопровождением.

В ходе этого курса вы изучили основы юнит-тестирования и научились писать тесты на Java. Однако разработчики используют множество языков программирования. Чтобы успешно проходить тестирование, нужно уметь писать тесты на том языке, на котором написан проект.

На этом уроке мы узнаем, как писать тесты на Python, JavaScript и Golang. Обсудим особенности тестирования на каждом из них. Разберемся, как выбрать подходящий язык в зависимости от задачи.

## На этом уроке:

- Посмотрим на общую картину юнит-тестирования.
- Узнаем, как проводить юнит-тестирование на языке Python.
- Изучим инструменты, которые помогут провести тестирование на Python эффективно.
- Рассмотрим фреймворки тестирования Python: Pytest, PyUnit, Robot Framework и Doctest.
- Попрактикуемся в использовании PyTest.
- Узнаем, как проводить юнит-тестирование на языке JavaScript и Golang (Go).
- Поймём, как выбрать подходящий язык для написания тестов.
- Познакомимся с антипаттернами тестирования, чтобы избежать распространенных ошибок.
- Узнаем, как автоматизировать тестирование, и как это может улучшить процесс разработки.
- Подведем итоги курса и проведем обзор основных тем, которые мы изучили.

# Термины, используемые в лекции

**Асинхронные тесты** — тесты, которые проходят не последовательно, а параллельно. Обычно их используют для проверки системы на ошибки, которые могут возникнуть при одновременном доступе к данным. Например, конфликт может произойти при обновлении данных.

**Антипаттерны** — это проектные решения, которые сперва кажутся приемлемыми, но по мере разработки приводят к нежелательным последствиям. Могут существенно ухудшить качество продукта и замедлить процесс разработки, поэтому их нужно избегать.

**Docstring** — это комментарий в программном коде, который используется для документирования.

Docstring начинается с трёх двойных кавычек (""") и описывает функцию, класс или модуль. Docstring рассказывает, что делает код, помогает другим разработчикам понять и использовать его.

## Общая картина юнит-тестирования

**Юнит-тестирование** — это процесс проверки программного кода с целью убедиться, что он работает правильно и не содержит ошибок.

Независимо от выбранного языка программирования процесс тестирования примерно одинаковый: нужно написать код, который будет проверять работу основного кода и создавать тестовые сценарии для каждого блока кода. Когда все тесты написаны, нужно запустить их и исправить ошибки, если они обнаружатся. Также важно проверить, что основной код полностью покрыт тестами.

### Преимущества тестирования:

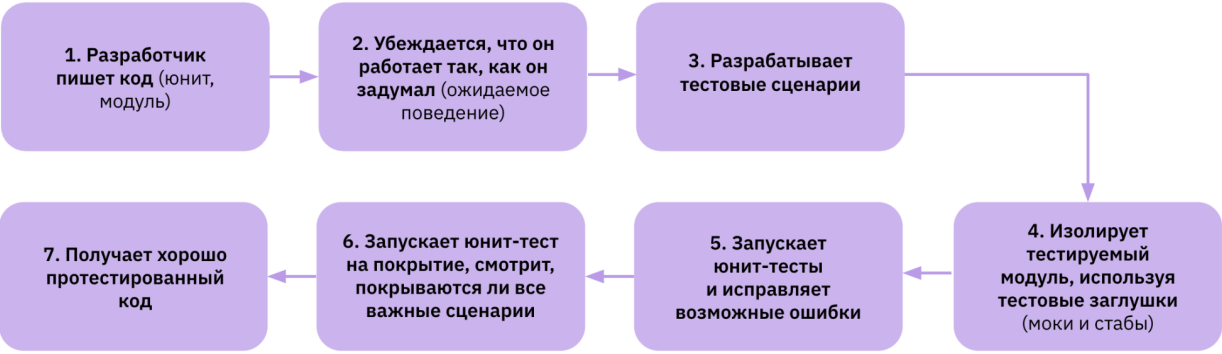
- проверяет работоспособность кода;
- позволяет быстрее разрабатывать;
- является всегда актуальной документацией к коду (иначе тесты не будут работать);
- может успокоить: если код вдруг сломается, вы сразу об этом узнаете.



Все это позволяет добиться стабильного качества при изменении разрабатываемого продукта.

Чтобы упростить процесс тестирования, можно использовать TDD — методологию разработки через тестирование: сначала пишутся тесты, а уже потом код, который должен пройти эти тесты.

Модульное (или юнит-) тестирование кода включает в себя следующие шаги:














Можно использовать TDD-методологию, когда сначала пишутся тесты на код, а потом сам код.

# На чем писать тесты

Unit-тесты принято писать на том языке, на котором реализована тестируемая функциональность.

Все современные языки программирования поддерживают удобные инструменты для тестирования кода. Где-то они встроены в язык, где-то есть удобные подключаемые библиотеки и фреймворки.

Feb 2023	Feb 2022	Change	Programming Language		Ratings	Change
1	1			Python	15.49%	+0.16%
2	2			C	15.39%	+1.31%
3	4	▲		C++	13.94%	+5.93%
4	3	▼		Java	13.21%	+1.07%
5	5			C#	6.38%	+1.01%
6	6			Visual Basic	4.14%	-1.09%
7	7			JavaScript	2.52%	+0.70%
8	10	▲		SQL	2.12%	+0.58%
9	9			Assembly language	1.38%	-0.21%
10	8	▼		PHP	1.29%	-0.49%
11	11			Go	1.11%	-0.12%

Статистика по языкам программирования [TIOBE Index](#)

Статистика **Tiobe** — это индекс популярности языков программирования на основе поисковых запросов.

🔥 Tiobe — не признак лучшего языка программирования или языка, на котором написано больше всего строк кода.

**Индекс можно использовать, чтобы проверить актуальность ваших навыков, или чтобы принять стратегическое решение о том, какой язык программирования применять для создания новой программной системы.**

Для большинства популярных высокоуровневых языков и языков без встроенной поддержки модульного тестирования существуют специальные инструменты и библиотеки

Некоторые языки напрямую поддерживают модульное тестирование. Их грамматика допускает прямое объявление модульных тестов без импорта библиотеки.

Основная идея всех инструментов тестирования в том, чтобы при изменении логики или структуры кода процесс тестирования оставался быстрым и понятным.

## Python

Python популярен среди тестировщиков и разработчиков (это очевидно из рейтинга Tiobe). А чем больше популярность, тем больше релевантность.



- высокоуровневый
- объектно-ориентированный
- интерпретируемый
- с динамической типизацией

Python – это язык программирования с открытым исходным кодом. Представлен миру в 1991 году. У него удобный, читаемый, элегантный и минимально нагроможденный синтаксис. На нем можно писать что угодно: от простых скриптов до сложных приложений.

Основные задачи, которые люди решают с помощью Python: разработка веб-приложений, работа с базами данных, создание интерфейсов, скриптинг, автоматизация задач, разработка игр, а также исследования и анализ данных.

**Плюсы:**

**Минусы:**

✓ **Широкая область применения:**

Python подходит не только для написания тестов, но и для автоматизации процессов, настройки окружения, сканирования портов, проведения тестов на безопасность, настройки CI и многого другого.

✓ **Простота использования и чтения.**

Для решения одной задачи в Python нужно написать примерно в 10 раз меньше строк кода, чем в Java. Так что разработчики сэкономят время и усилия при написании кода и тестов.

✓ **Понятные обучающие материалы**  
в свободном доступе в интернете.

— **Ограниченная поддержка.**

Python был разработан простым, универсальным, позволяющим написать скрипты непосредственно из интерпретатора. Он не слишком хорошо взаимодействует с IDE, в отличие от Java. Python настолько простой и универсальный, что IDE не может уловить, что вы создаете объекты и передаете их. Это существенный недостаток, который нужно учитывать.

— **Сложность включения библиотек.**

У новичков это может сказаться на скорости и эффективности разработки.

## Инструменты тестирования в Python

Как и в Java в Python есть инструменты, которые подскажут об ошибках до того, как код начнёт выполняться.

🔥 В компилируемых языках программист может получить полезную обратную связь о коде уже на этапе компиляции.

Компилятор проверяет, что код валиден и что его можно скомпилировать. И рекомендует, как сделать его лучше.

Python — интерпретируемый язык, так что этап компиляции в нём отсутствует. Поэтому линтеры (своего рода компиляторы кода на Python) имеют особое значение. Они дают информацию о том, валиден ли Python-код, независимо от того, запускался ли он раньше.

В 1979 году термин «lint» впервые использовали для обозначения программы, которая анализировала код на C и предупреждала о языковых структурах, которые плохо переносятся на другие платформы. С тех пор «линтерами» называют статические анализаторы, которые помогают устранять распространенные ошибки, устанавливая согласованность и улучшая читаемость. Это шуточный термин, адаптированный от слова «lint» (соринки на одежде).

На Python есть множество линтеров. Рассмотрим основные:



- **PEP 8** (pycodestyle) — самый распространенный кодекс стиля для программирования на Python. Представляет собой набор рекомендаций и правил. Был создан, чтобы программисты использовали один и тот же стиль написания кода, несмотря на разные подходы.

В кодексе стиля зафиксированы определенные соглашения для именования переменных, написания комментариев и так далее. Следуйте кодексу, чтобы другие программисты могли легко читать и понимать ваш код.

Подробнее о кодексе PEP 8 можно почитать на официальном сайте Python: [PEP 8 – Style Guide for Python Code](#).

- **Flake8** — это инструмент, который проверяет Python-код на соответствие стандартам, ищет в нем ошибки и проблемы с оптимизацией. Проверяет и по стандарту PEP 8, и по другим правилам, которые определил пользователь. Даёт ряд простых инструментов для автоматизации проверки кода.

Подробнее об инструменте — на официальном сайте: [Flake8](#).

- **Pylint** — инструмент, который проверяет правильность Python-кода и дает полезные советы для его улучшения.

Pylint анализирует код на соответствие стандартам стиля, проверяет правильность имен, предлагает решения по исправлению проблем и предоставляет отчет о качестве кода. Он полезен для разработчиков, которые хотят писать код по стандарту, улучшить читаемость, поддерживаемость и общее качество Python-кода.

Руководство пользователя, описание функций и другая полезная информация — на официальном сайте: [Pylint](#).

В интернете можно множество ресурсов, посвященных линтерам на языке Python. Примеры:

- [Stack Overflow](#)
- [Python](#) (Reddit)
- [Python Software Foundation](#).
- [Real Python](#)



Линтеры — неотъемлемая часть разработки. Они помогают улучшить качество кода, уменьшить число синтаксических и стилевых ошибок. Это своего рода тесты, которые работают до исполнения кода. Они помогут не тратить силы на элементарные ошибки.

Рассмотрим распространенные ошибки, которые могут выявить линтеры в Python.

1. **Ошибки синтаксиса:** неправильное использование операторов, отсутствие двоеточия в блоке кода и подобное.

Например, если в коде пропущено двоеточие после инструкции if:

```
if a == b
    print("a equals b")
```

PyLint выдаст ошибку: *EOL while scanning string literal* (конец строки во время сканирования строкового литерала).

2. **Несоответствие стандартам кодирования:** линтеры могут проверять код по стандартам (например, PEP8) и выдавать предупреждение, если код им не соответствует.

Например, если строка слишком длинная:

```
my_long_string = "This is a really long string that should be shortened to fit within 79 characters per line in accordance with PEP 8 guidelines."
```

Flake8 выдаст предупреждение: *E501 line too long (116 > 79 characters)* (строка слишком длинная (116 > 79 символов)).

3. **Неопределенные переменные:** линтеры могут выдавать предупреждение, если переменная не определена или используется несколько раз.

Например, если переменная x не определена в коде:

```
print(x)
```

Pyflakes (еще один популярный линтер) выдаст предупреждение: *undefined name 'x'* (неопределенное имя 'x').

## Пример работы с линтером Flake8

Flake8, как и любой другой Python-пакет, устанавливается через pip.

Нужно выполнить команду:

```
pip install flake8
```

Чтобы убедиться, что линтер установлен, запросим версию:

```
flake8 --version
```

Чтобы начать использовать Flake8, укажем директорию, которую нужно проверять. Например, проверить один файл можно командой, в которой указано название файла:

```
$ flake8 Main.py
```

Проверить директорию рекурсивно:

```
$ flake8 src/
```

Проверить текущую директорию рекурсивно можно командой с точкой:

```
$ flake8 .
```

Чтобы наглядно показать работу, рассмотрим код, написанный с использованием плохих практик:

```
from math import *
import itertools

def CalculateSquareRoot (Number ):
    return sqrt(Number )

def append_item(item, l=[]):
    l.append(item)
    return l

while True:
    try:
        your_number=float(input('Enter your number: '))
        print("Square root is:",
CalculateSquareRoot(your_number))
```

```
break
except:
    pass
```

Здесь есть следующие ошибки:

- `import *` — импортирование всех имен из модуля, хотя используется из них только одно;
- `import itertools` — ненужный импорт;
- во многих местах пробелы лишние или их нет;
- название функции написано в стиле PascalCase;
- кое-где используются табы для отступов;
- используется список (изменяемый объект) в качестве значения аргумента функции по умолчанию;
- используется слишком широкое выражение `except:` без указания конкретного исключения.

Проверим код с помощью `flake8` (команда `flake8 bad_code.py`):

```
flake8 bad_code.py
bad_code.py:1:1: F403 'from math import *' used; unable to detect
undefined names
bad_code.py:2:1: F401 'itertools' imported but unused
bad_code.py:4:1: E302 expected 2 blank lines, found 1
bad_code.py:4:4: E271 multiple spaces after keyword
bad_code.py:4:25: E211 whitespace before '('
bad_code.py:4:33: E202 whitespace before ')'
bad_code.py:5:1: W191 indentation contains tabs
bad_code.py:5:8: E271 multiple spaces after keyword
bad_code.py:5:10: F405 'sqrt' may be undefined, or defined from
star imports: math
bad_code.py:5:21: E202 whitespace before ')'
bad_code.py:7:1: E302 expected 2 blank lines, found 1
bad_code.py:7:23: E741 ambiguous variable name 'l'
bad_code.py:8:1: E101 indentation contains mixed spaces and tabs
bad_code.py:9:1: E101 indentation contains mixed spaces and tabs
...
```

Как видим, Flake8 проверил код и нашёл множество ошибок. К сожалению, он не умеет чинить их — придётся исправлять вручную.

Каждая строка имеет свой код (E101, W291 и так далее) и указывает на строку и символ, где произошла ошибка. Коды можно использовать для включения и отключения правил.

Однако это не все ошибки. Чтобы искать дополнительные ошибки в Flake8, нужно установить плагины.

Плагины можно найти через поисковую систему или списки плагинов. Для популярных фреймворков и библиотек также доступны соответствующие плагины.

- **flake8-aaa** — проверяет соответствие шаблону утверждения Arrange-Act-Assert
- **flake8-assertive** — проверяет утверждения (assert)
- **flake8-mock** — обеспечивает проверку тестовых заглушек
- **flake8-pytest-style** — проверяет общие проблемы со стилем или несоответствия с тестами на основе pytest.
- **flake8-pytest** — применяет pytest-утверждения
- **flake8-pytestrail** — проверяет идентификаторы тестовых наборов TestRail

Есть плагины, которые проверяют соответствие тестов определенным шаблонам, есть проверки для моков, есть расширения, поддерживающие фреймворк pytest (аналог Junit для Java). Плагины устанавливаются так же, как и сам flake8.

Установка плагина flake8-aaa:

```
$ pip install flake8-aaa
```

Проверка установки (покажет все плагины):

```
flake8 --version
```

## Пример работы с линтером Pylint

Более умный и продвинутый линтер, чем Flake8, — Pylint. Он достаточно строгий: содержит множество правил и рекомендаций, активированных по умолчанию.

Pylint можно расширить, хотя экосистема плагинов у него скудная.

Каждый раз, когда Pylint запускается, он выдаёт оценку качества кода (до 10) и следит, чтобы это значение улучшилось. Достичь 10 сложно, но это цель, к которой нужно стремиться. Установка Pylint принципиально ничем не отличается от установки Flake8.

Установка:

```
pip install pylint
```

Проверка:

```
$ pylint --version
```

Использование:

```
# проверить один файл  
$ pylint Main.py
```

Выполним проверку с помощью Pylint:

```
pylint bad_code.py  
***** Module bad_code  
bad_code.py:5:0: W0311: Bad indentation. Found 1 spaces, expected 4  
(bad-indentation)  
bad_code.py:17:0: C0304: Final newline missing (missing-final-newline)  
bad_code.py:1:0: C0114: Missing module docstring (missing-module-docstring)  
bad_code.py:1:0: W0622: Redefining built-in 'pow' (redefined-builtin)  
bad_code.py:1:0: W0401: Wildcard import math (wildcard-import)  
bad_code.py:4:0: C0116: Missing function or method docstring  
(missing-function-docstring)  
bad_code.py:4:0: C0103: Function name "CalculateSquareRoot" doesn't conform  
to snake_case naming style (invalid-name)  
bad_code.py:4:26: C0103: Argument name "Number" doesn't conform to snake_case  
naming style (invalid-name)  
...  
-----  
Your code has been rated at 0.00/10
```

## Юнит-тесты в Python

Для модульных, интеграционных, системных, приемочных и прочих тестов нам нужно исполнять код. Такое тестирование проводится с использованием входных и выходных динамических данных, и называется динамическим.

Как и в Java, для проверки функции с различными параметрами нужно вернуть систему в исходное состояние и проверить, что возвращаемое значение соответствует ожиданиям.

**Типичный сценарий юнит-теста:**

1. Есть исходный код программы с функциями, которые нужно протестировать. Приводим систему в начальное положение.

2. Пишем новую программу, которая вызывает функции и смотрит на результат.
3. Проверяем, что функция возвращает ожидаемое значение.
4. Если результат совпадает с тем, что должно быть, тест считается пройденным. Если результат не совпадает, тест не пройден, нужно разбираться.

## Структура юнит-теста



### Arrange:

Подготовка среды,  
в которой выполняется код:  
  
Создать переменные a и b со  
значениями 5 и 8 соответственно



### Act:

Тестирование кода:  
  
Вычислить сумму двух  
переменных с помощью  
функции калькулятора



### Assert:

Убеждаемся, что результат теста  
именно тот, что мы ожидали:  
  
Проверить, что результат  
равен 13

## assert

Оператор `assert` в Python используется для проверки истинности утверждения. Если утверждение неверно, `assert` вызывает `AssertionError`. С помощью оператора `assert` мы можем указать, что условия, предъявляемые к коду, должны быть выполнены.

Оператор `assert` может использоваться для проверки условий:

```
x = 10
assert x > 5, "Значение x должно быть больше 5"
```

В этом примере оператор `assert` проверяет, что значение `x` больше 5. Если условие выполняется, `assert` ничего не делает. Если условие не выполняется, `assert` будет выдавать `AssertionError`.

```
def check_password(password):
    if len(password) < 8:
        raise AssertionError("Длина пароля должна быть не меньше
8 символов")
    # ...другие условия...
    assert True, "Пароль удовлетворяет требованиям"
```

В примере **assert True, "Пароль удовлетворяет требованиям"** — это указания, что если условия, указанные выше, удовлетворяются, то assert-выражение True (выдаст AssertionError).



Функционал аналогичен assert в Java. У него тот же недостаток: нельзя использовать для отладки более сложных алгоритмов, так как assert не может выводить сообщения, которые указывают на причину сбоя.

Поэтому чаще используют более продвинутые инструменты тестирования: библиотеки и фреймворки.

## Фреймворки тестирования Python

Популярные фреймворки для модульного тестирования на Python:

- Pytest
- PyUnit (unittest)
- Robot Framework

Рассмотрим их поподробнее.

### Pytest

Pytest — мощный и гибкий фреймворк для автоматизации юнит-тестирования в Python.

- **Поддерживает разные типы тестов:** модульные, интеграционные. Поддерживает использование мок-объектов.
- **Предоставляет инструменты для отчетов:** позволяет получать подробную информацию о тестировании.
- **Поддерживает параллельное тестирование:** следовательно, значительно сокращает время тестирования.
- **Работает со множеством фреймворков:** Django, Flask и другими.



Особенности Pytest:



- **Простота.** У Pytest простой API, который делает тестирование быстрым и легким.
- **Большое сообщество.** Pytest — один из самых популярных фреймворков тестирования. Есть много ресурсов, учебных материалов, форумов и сообществ, которые помогут с ним разобраться.
- **Расширяемость.** API Pytest можно расширить, чтобы улучшить функциональность.
- **Встроенные функции.** Pytest предлагает набор удобных инструментов: fixture, parametrization, assert rewriting, monkeypatching/mocking, log capturing, performance testing.

## Инструкция по установке Pytest

Чтобы установить Pytest:

1. Откройте командную строку (терминал) на компьютере.
2. Установите Pytest с помощью команды:

```
pip install pytest
```

Чтобы проверить Pytest:

1. Создайте новый файл test\_example.py в любой директории на вашем компьютере.
2. Введите следующий код в файл test\_example.py:

```
def func(x):  
    return x + 1  
  
def test_answer():  
    assert func(3) == 5
```

3. Сохраните файл.
4. В командной строке (терминале) перейдите в директорию, где находится файл test\_example.py.
5. Выполните следующую команду для запуска теста:

```
pytest
```

После выполнения теста вы должны увидеть следующий результат:

```
===== test session starts =====
platform win32 -- Python 3.9.7, pytest-6.2.5, py-1.10.0, pluggy-1.0.0
rootdir: C:\Users\user\Documents\pytest_example
collected 1 item
test_example.py F [100%]
===== FAILURES =====
_____ test_answer _____

    def test_answer():
>         assert func(3) == 5
E         assert 4 == 5
E         + where 4 = func(3)

test_example.py:5: AssertionError
===== short test summary info =====
FAILED test_example.py::test_answer - assert 4 == 5
===== 1 failed in 0.12s =====
```

6. Тест должен завершиться неудачей, так как утверждение в функции `test_answer` неверно. Если тест проходит успешно, вы должны увидеть успех в выводе командной строки.

🔥 Чтобы понять лучше, не забудьте повторить действия самостоятельно.

## Практика PyTest

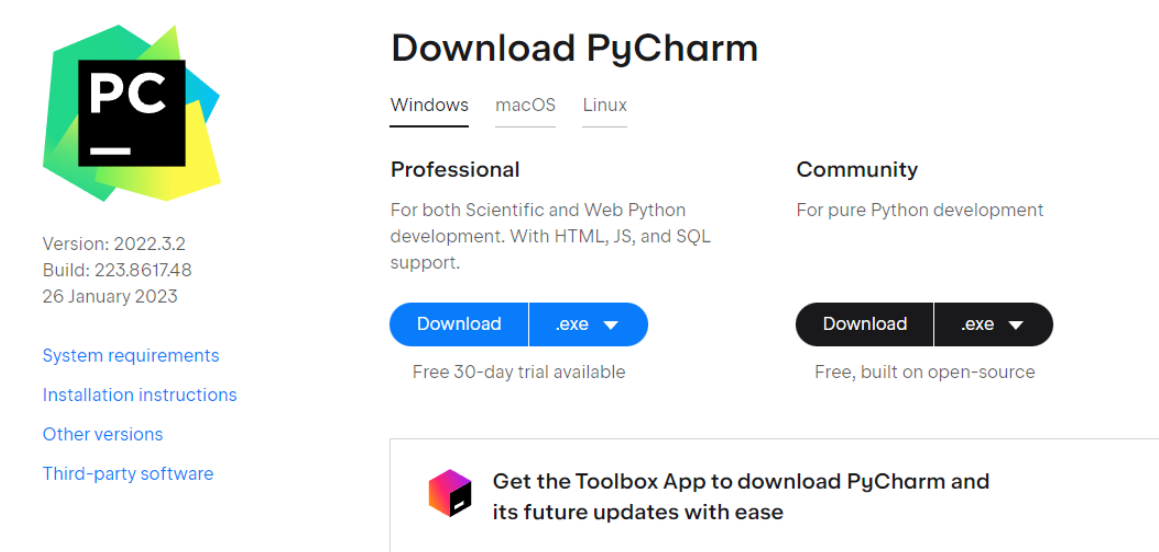
Попрактикуемся использовать PyTest для написания тестов на Python.

**Скачаем последнюю версию Python** с [официального сайта](#). В выпадающем меню вкладки Downloads по умолчанию появится последняя версия для операционной системы пользователя. Для Windows скачиваем установочный файл в формате \*.exe, для macOS — pkg-файл, а для Linux — deb-пакет.



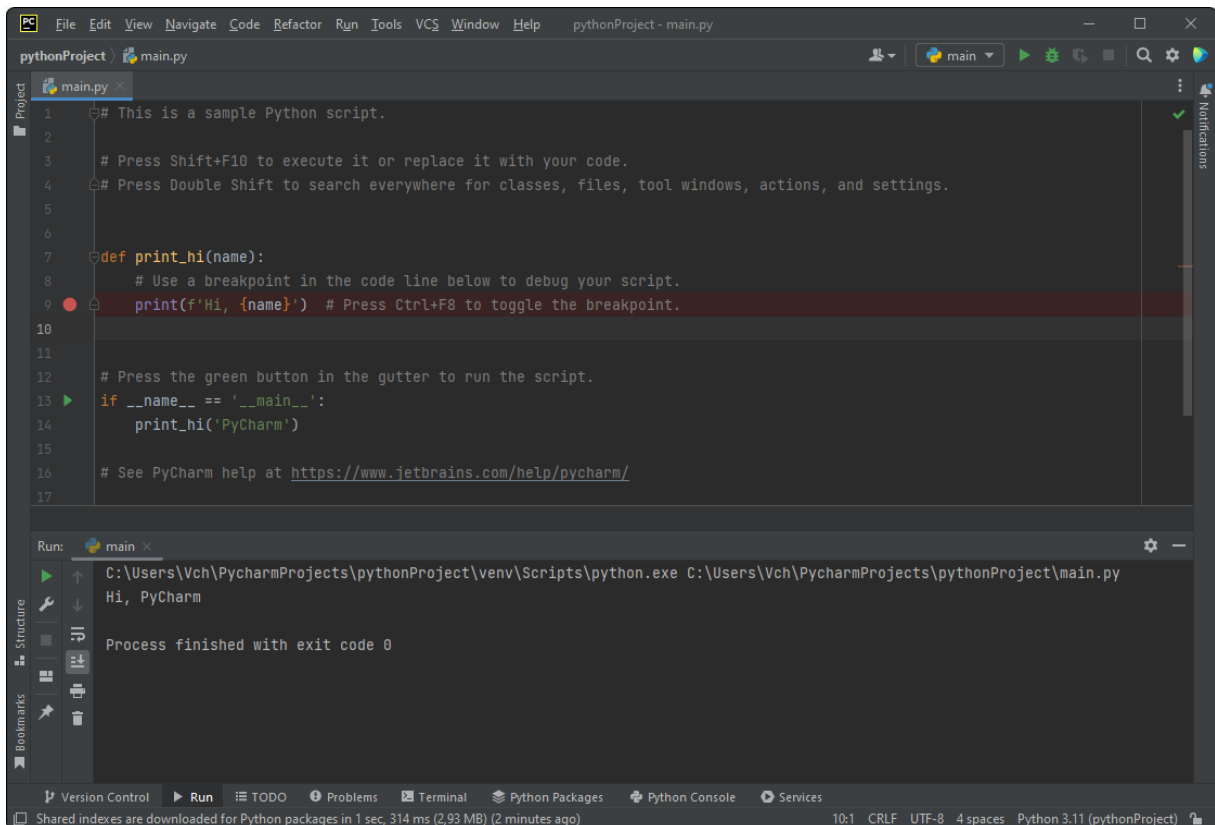
**Установим IDE PyCharm.**

1. Переходим [на официальный сайт PyCharm](#).
2. Скачиваем установочный пакет и устанавливаем IDE.

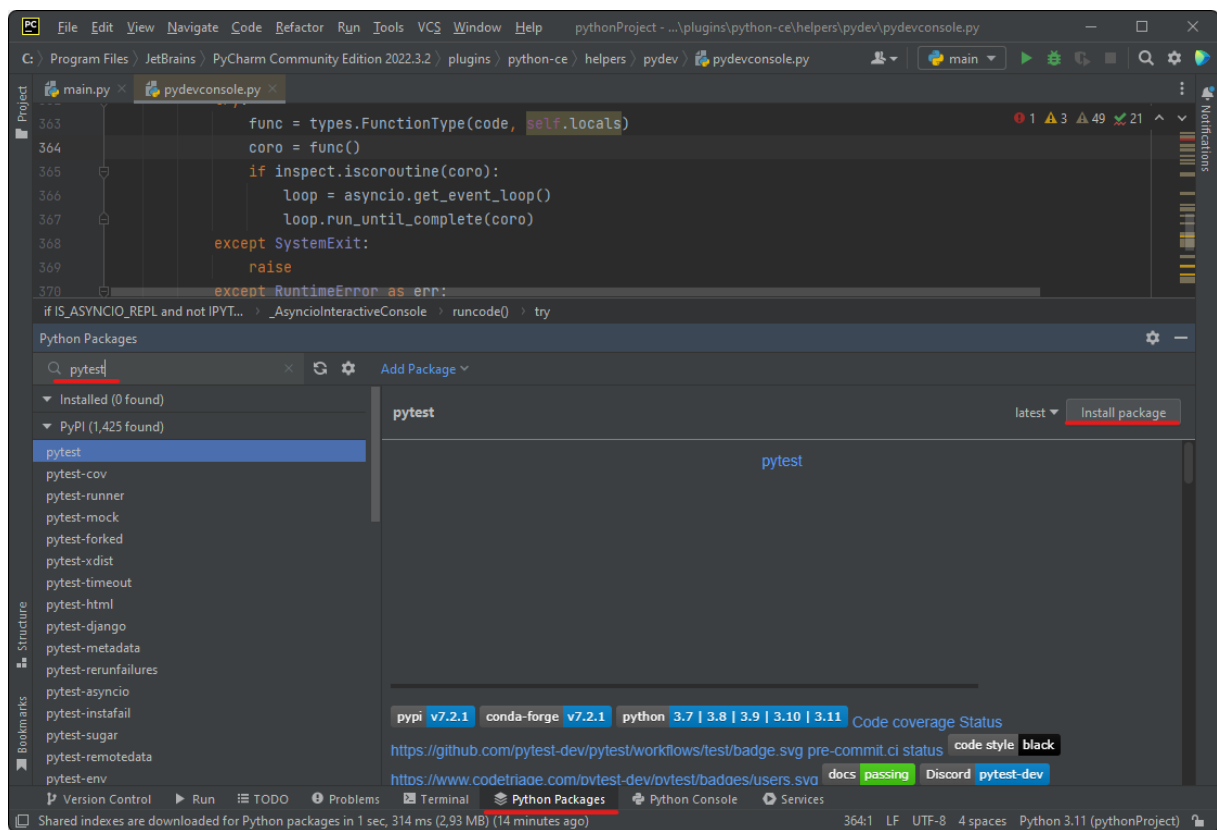


The image shows the official PyCharm download page. On the left is the PyCharm logo (a green and yellow hexagon with 'PC' and a minus sign) and version information: Version: 2022.3.2, Build: 223.8617.48, 26 January 2023. Below this are links for 'System requirements', 'Installation instructions', 'Other versions', and 'Third-party software'. The main section is titled 'Download PyCharm' and has tabs for 'Windows', 'macOS', and 'Linux'. Under 'Windows', there are two options: 'Professional' (for both Scientific and Web Python development) and 'Community' (for pure Python development). Each has a 'Download' button and a '.exe' dropdown. Below the buttons, it says 'Free 30-day trial available' for Professional and 'Free, built on open-source' for Community. At the bottom, there is a box with the JetBrains Toolbox App logo and text: 'Get the Toolbox App to download PyCharm and its future updates with ease'.

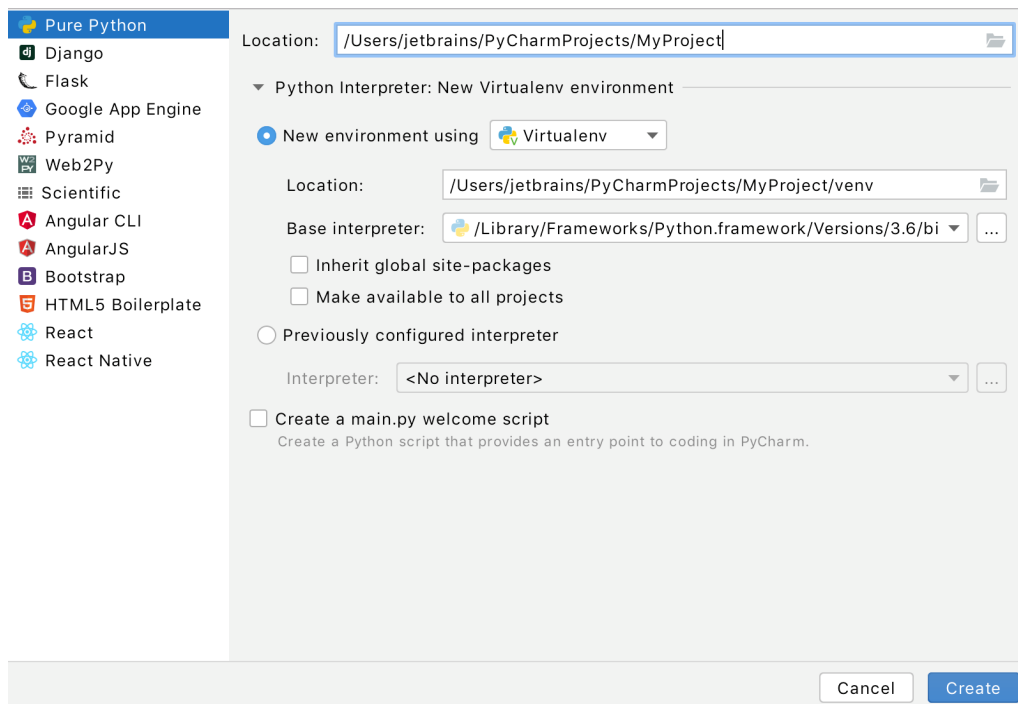
3. После установки можем проверить, что все работает, запустив тестовый пример:



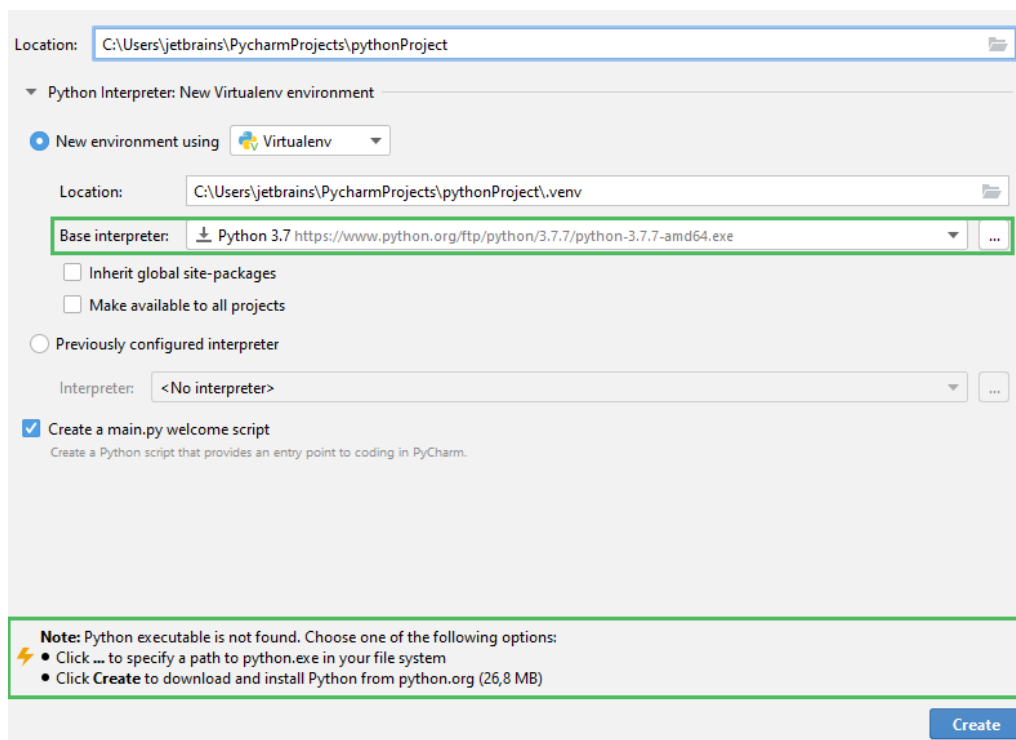
4. **Теперь установим PyTest.** Это можно сделать через интерфейс PyCharm для загрузки дополнительных пакетов либо в терминале:



5. Создадим новый проект. Если вы на экране приветствия, нажмите «Новый проект». Если проект уже открыт, выберите File → New Project в меню:

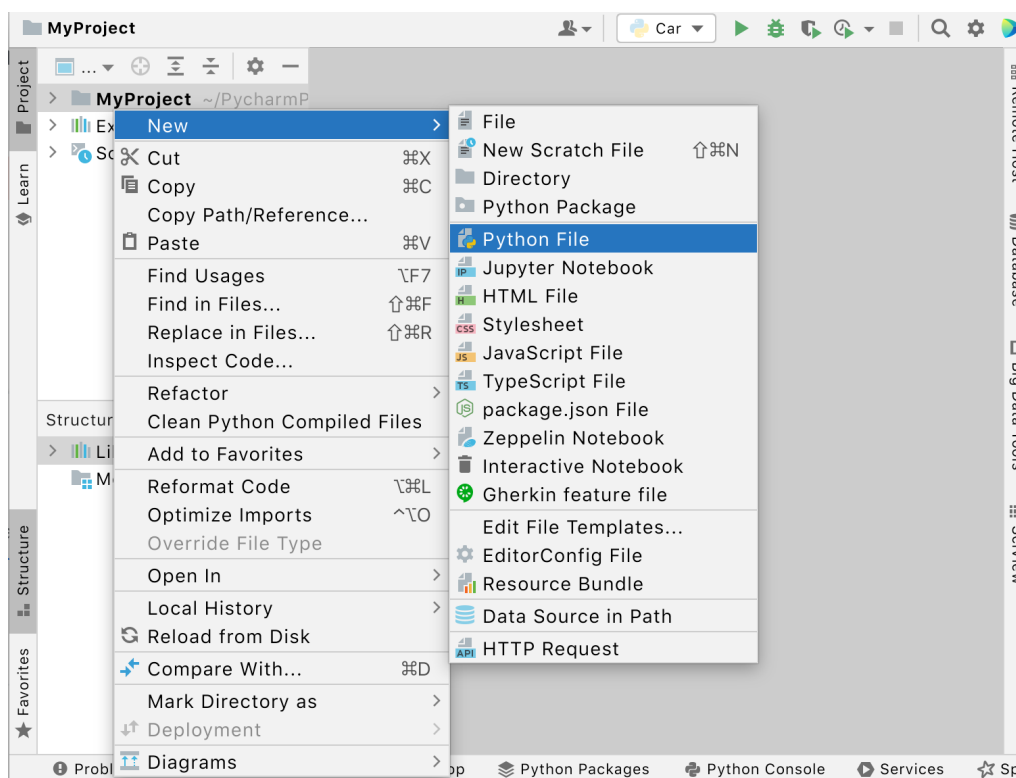


6. Выберите расположение проекта и версию интерпретатора Python. Нажмите кнопку «Обзор» рядом с полем Location и укажите каталог для проекта.

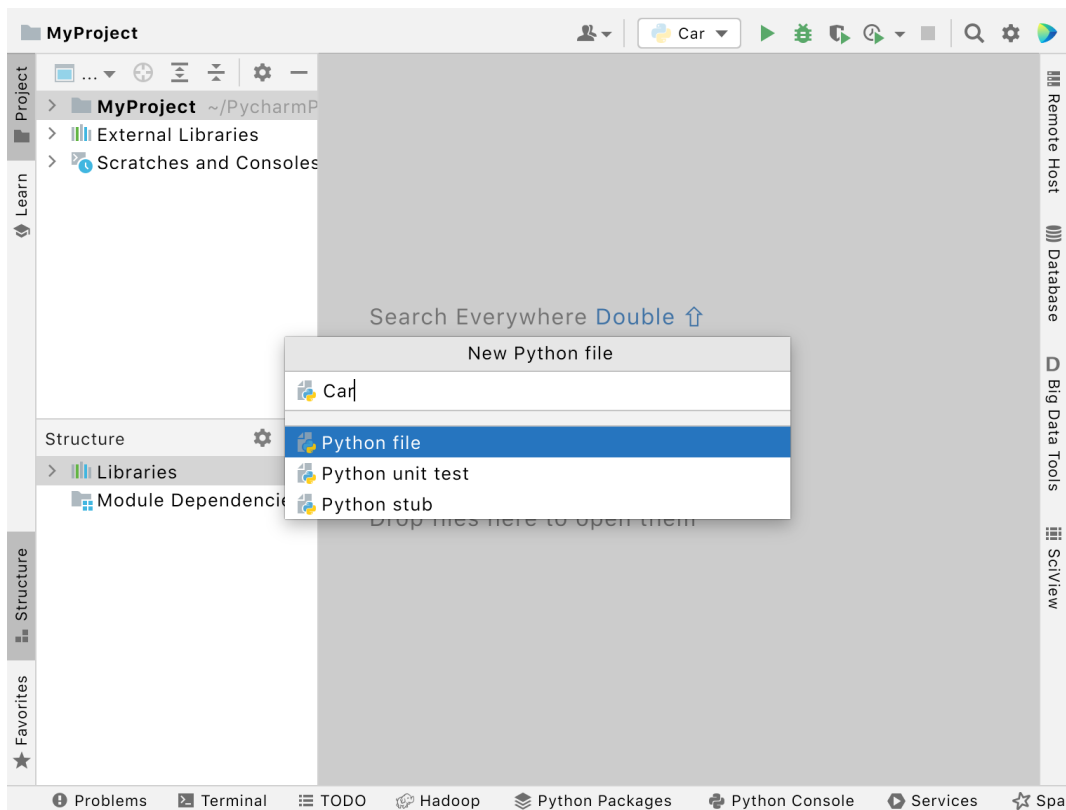


7. Создайте файл Python.

8. В окне Project tool выберите корень проекта (обычно это корневой узел в дереве проекта). Щелкните его правой кнопкой мыши и выберите File → New → Python File.



9. Выберите опцию Python File в контекстном меню. Введите новое имя файла:



10. PyCharm создаст новый файл Python и откроет его для редактирования. Ниже исходный код для файла Car.py. Мы будем использовать этот проект управления машиной для тестирования:

```
class Car:
    def __init__(self, speed=0):
        self.speed = speed
        self.odometer = 0
        self.time = 0

    def say_state(self):
        print("I'm going {} kph!".format(self.speed))

    def accelerate(self):
        self.speed += 5

    def brake(self):
        if self.speed < 5:
            self.speed = 0
        else:
            self.speed -= 5

    def step(self):
        self.odometer += self.speed
        self.time += 1
```

```

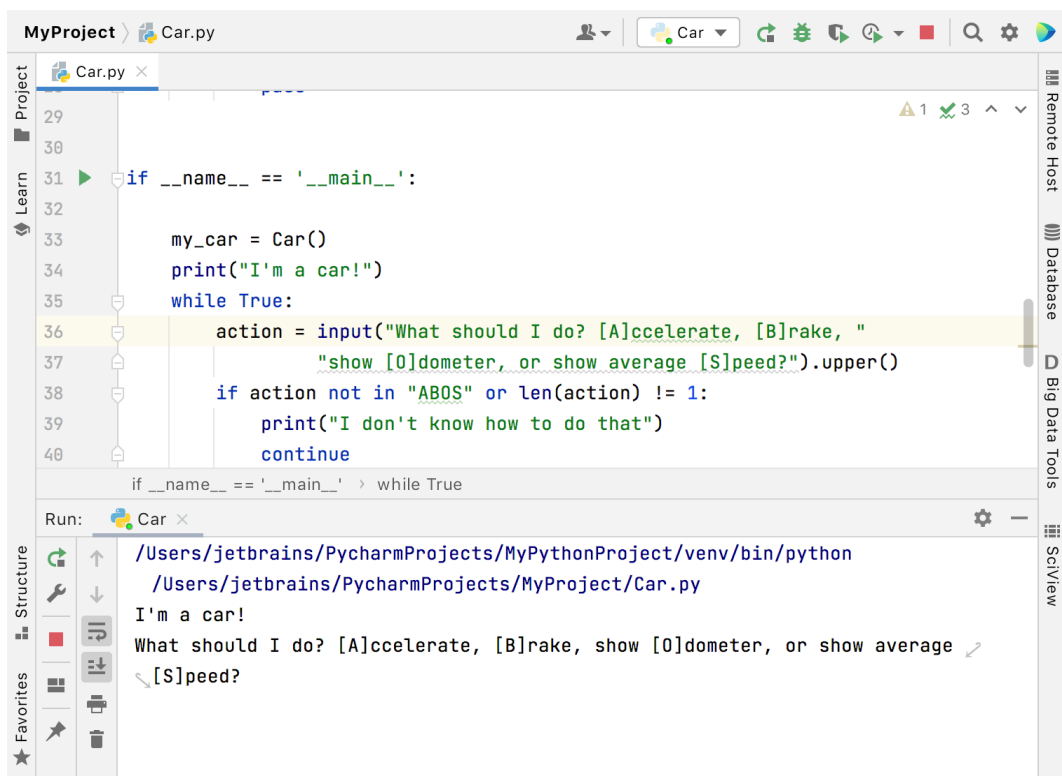
def average_speed(self):
    if self.time != 0:
        return self.odometer / self.time
    else:
        pass

if __name__ == '__main__':

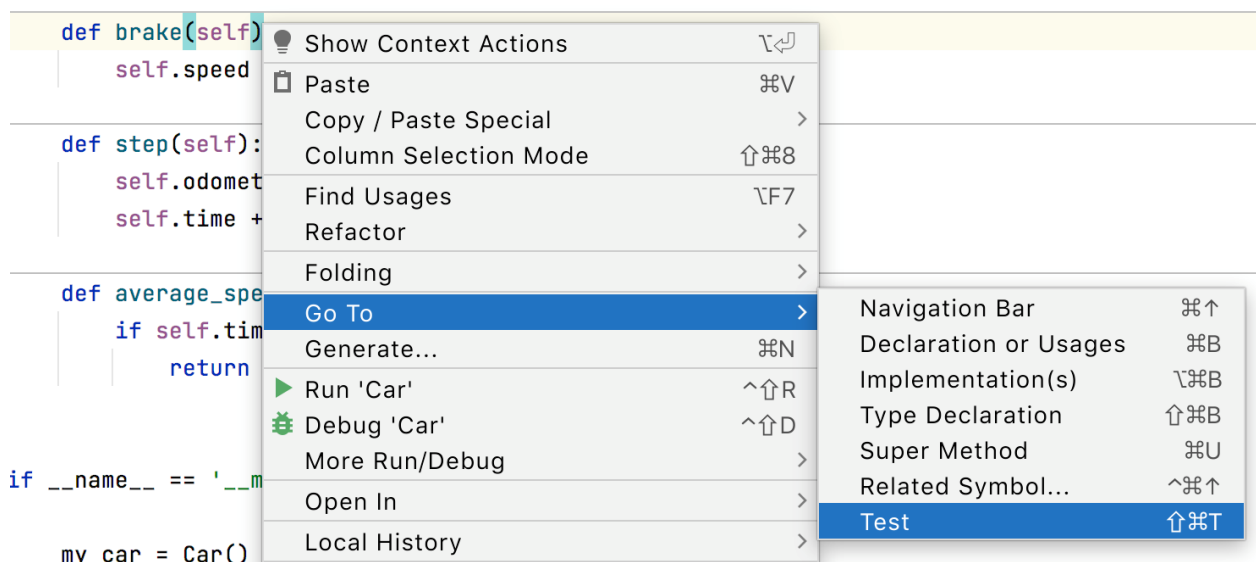
    my_car = Car()
    print("I'm a car!")
    while True:
        action = input("What should I do? [A]ccelerate, [B]rake,
"
                        "show [O]dometer, or show average
[S]peed?").upper()
        if action not in "ABOS" or len(action) != 1:
            print("I don't know how to do that")
            continue
        if action == 'A':
            my_car.accelerate()
        elif action == 'B':
            my_car.brake()
        elif action == 'O':
            print("The car has driven {}
kilometers".format(my_car.odometer))
        elif action == 'S':
            print("The car's average speed was {}
kph".format(my_car.average_speed()))
        my_car.step()
        my_car.say_state()

```

11. Программа взаимодействует с пользователем, ожидая команду из консоли:



12. Быстрый способ создания тестов — автоматическая генерация. Для этого нужно открыть Car.py, затем щелкнуть правой кнопкой мыши на названии класса, выбрать пункт, к которому нужно перейти, а затем выбрать Test: PyCharm автоматически обнаруживает программу запуска тестов, установленную в вашем интерпретаторе Python, и использует ее. Если специальный тестовый модуль не установлен, PyCharm использует unittest:



13. Появится всплывающее окно с предложением создать новый тест:



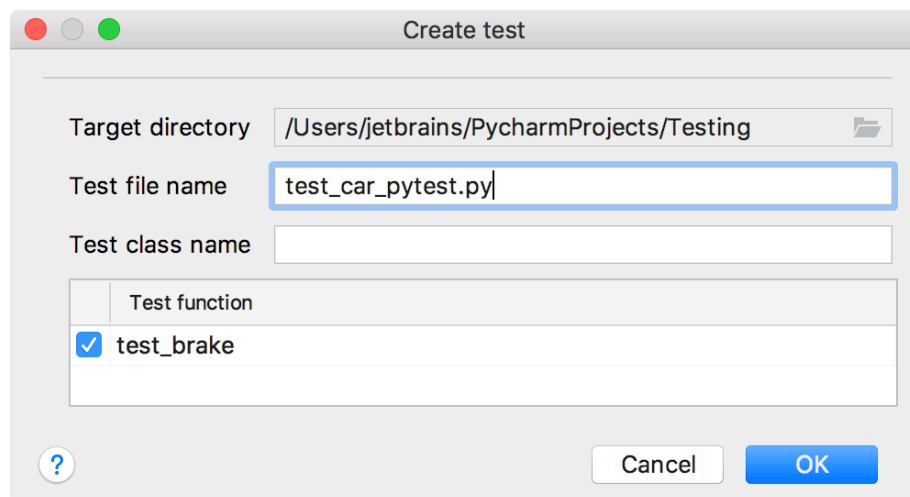
```
def brake(self):
    self.speed

def step(self):
    self.odometer += self.speed
    self.time += 1
```

Choose Test for **brake** (0 found)

Create New Test...

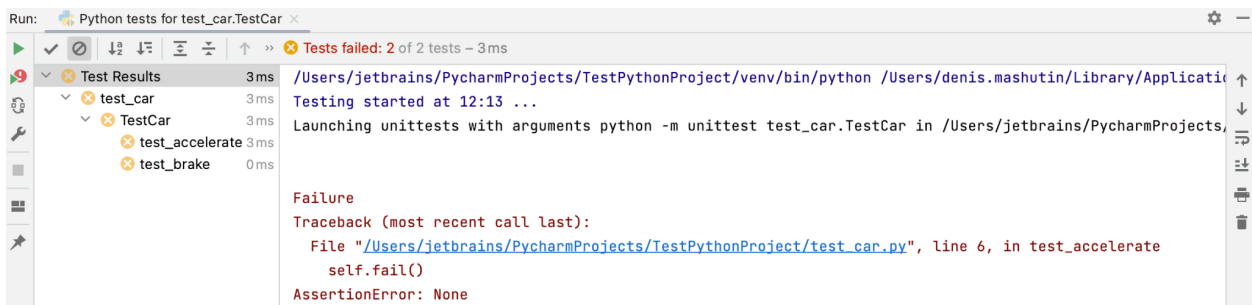
14.Проверяем, способен ли наш автомобиль ускоряться и тормозить. Для этого установим флажки:



15.Создастся новый тестовый класс Python. Вы можете запустить тест, щелкнув значок запуска рядом с определением класса.

```
car.py x test_car.py x
1 from unittest import TestCase
2
3
4 class TestCar(TestCase):
5     def test_accelerate(self):
6         self.fail()
7
8     def test_brake(self):
9         self.fail()
10
```

16.Конфигурация запуска / отладки будет создана автоматически. Мы видим, что по умолчанию тест завершается неудачно:



17. Теперь напомним реальные тесты. Часть кода ниже. Мы просто описываем сценарии движения машины.

```
class TestCar(unittest.TestCase):
    def setUp(self):
        self.car = Car()

class TestInit(TestCar):
    def test_initial_speed(self):
        self.assertEqual(self.car.speed, 0)

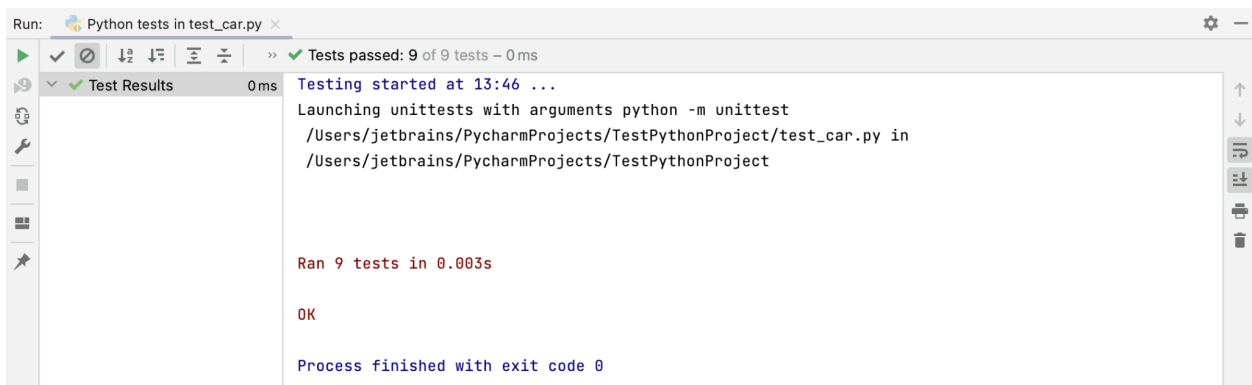
    def test_initial_odometer(self):
        self.assertEqual(self.car.odometer, 0)

    def test_initial_time(self):
        self.assertEqual(self.car.time, 0)

class TestAccelerate(TestCar):
    def test_accelerate_from_zero(self):
        self.car.accelerate()
        self.assertEqual(self.car.speed, 5)

    def test_multiple_accelerates(self):
        for _ in range(3):
            self.car.accelerate()
            self.assertEqual(self.car.speed, 15)
```

18. Запускаем тесты и видим, что они проходят. Таким образом, можно использовать PyCharm для написания тестов с любым фреймворком



## PyUnit

PyUnit (Unititest) – фреймворк для модульного тестирования на Python. Его преимущества:

- Входит в стандартную библиотеку Python, поэтому дополнительные модули ставить не нужно. Все поставляется из коробки, поэтому большинство разработчиков начинают свой путь в тестировании с него.
- Быстрая генерация отчетов о тестах в XML и unittest-sml-reporting.



PyUnit разработан с учетом стандарта xUnit, а именно JUnit. Он обеспечивает автоматизацию тестирования, общий код настройки и завершения работы для тестов, агрегирование тестов в коллекции и независимость тестов от среды отчетности.

Фреймворк PyUnit предоставляет широкий инструментарий для создания и запуска тестов. Даже небольшого набора инструментов будет достаточно для большинства пользователей.

Короткий скрипт для тестирования двух простых методов:

```
import unittest

class TestStringMethods(unittest.TestCase):

    def test_upper(self):
        self.assertEqual('foo'.upper(), 'FOO')

    def test_isupper(self):
        self.assertTrue('FOO'.isupper())
        self.assertFalse('Foo'.isupper())
```

```
if __name__ == '__main__':  
    unittest.main()
```

Набор тестов формируется путем создания подклассов, происходящих от `unittest.TestCase`. Три отдельных теста определяются с помощью методов, имена которых начинаются с `test`. Это соглашение об именовании информирует тестировщика о том, какие методы представляют тесты.

**Суть каждого теста** — это вызов `assertEqual()` для проверки ожидаемого результата; `assertTrue()` или `assertFalse()` для проверки условия; `assertRaises()` для проверки того, что возникает конкретное исключение. Эти методы используются вместо `assert`-инструкции, чтобы тестировщик мог накапливать все результаты тестирования и создавать отчет.

## Robot Framework

Об этом фреймворке мы уже упоминали на прошлой лекции. Его ядро написано на Python, но может запускаться на Jython (Java-реализация Python) и IronPython (Python для .NET framework).



Robot Framework — инструмент с открытым исходным кодом для автоматизации тестирования. Он использует высокоуровневые словарные условия, чтобы помочь пользователям сосредоточиться на тестируемых функциях, а не на реализации.

Robot Framework поставляется со множеством библиотек, которые позволяют работать с различными фреймворками, языками, API, базами данных, GUI-приложениями, веб-приложениями и так далее. Он основывается на подходе Keyword-driven testing (KDT), что позволяет нам легко создавать тест-кейсы с использованием удобных для восприятия ключевых слов (не требует опыта написания кода).

Тестовые примеры Robot Framework создаются с использованием простого табличного синтаксиса. Например, ниже приведены два теста:

- пользователь может создать учетную запись и войти в систему;
- пользователь не может войти в систему с неверным паролем.

```
*** Test Cases ***  
User can create an account and log in
```

```
Create Valid User fred P4ssw0rd
Attempt to Login with Credentials fred P4ssw0rd
Status Should Be Logged In

User cannot log in with bad password
Create Valid User betty P4ssw0rd
Attempt to Login with Credentials betty wrong
Status Should Be Access Denied
```

Тестовые примеры создаются из ключевых слов и их возможных аргументов. Синтаксис требует, чтобы ключевые слова и аргументы, а также настройки и их значения разделялись как минимум двумя пробелами или символом табуляции. Обычно рекомендуется использовать четыре пробела, чтобы сделать разделитель более четким, а в некоторых случаях выравнивание аргументов или других значений может облегчить понимание данных.

## Doctest

**Doctest** — модуль, включенный в стандартную библиотеку Python. Он считывает примеры из документации и автоматически запускает их. Помогает программистам проверить, что функции выполняются правильно и что код действительно работает.

Доктесты в Python пишутся в виде обычных комментариев, но помещаются внутри тройных кавычек. Иногда в доктесты включают пример функции и ожидаемого результата, иногда — также комментарий о том, для чего предназначена функция.

Комментарии помогают программистам подробно описать свои цели, чтобы люди, читающие код, могли все понять (таким человеком может оказаться и автор кода).

Ниже приведен математический пример доктеста для функции **add(a, b)**, которая складывает два числа:

```
import doctest

def add(a, b):
    """
    Given two integers, return the sum.

    :param a: int
    :param b: int
    :return: int

    >>> add(2, 3)
    5
```

```
"""  
    return a + b  
  
doctest.testmod()
```

Рассмотрим пример тестирования функции вычисления факториала: [Test interactive Python examples](#). Это модуль на языке Python, который определяет функцию **factorial(n)**, которая вычисляет факториал заданного неотрицательного целого числа n. В модуле также содержится серия тестов, демонстрирующих правильное поведение функции factorial() в разных ситуациях.

Функция factorial() сначала проверяет, что входной аргумент n является неотрицательным целым числом, вызывая **ValueError**, если хотя бы одно условие не выполняется. Затем она вычисляет факториал с помощью простого цикла, умножая все числа от 2 до n. Функция возвращает результат как **int** или **long**, в зависимости от размера результата.

Блок **if \_\_name\_\_ == "\_\_main\_\_":** в конце модуля запускает серию тестов с помощью встроенного модуля Doctest Python. Эти тесты проверяют поведение функции factorial() в различных сценариях (например, ввод разных типов данных) и проверяют, что функция возвращает ожидаемые результаты.

Запуск модуля как скрипта выполнит эти тесты и напечатает результаты в консоль.

```
"""  
This is the "example" module.  
  
The example module supplies one function, factorial().  For  
example,  
  
>>> factorial(5)  
120  
"""  
  
def factorial(n):  
    """Return the factorial of n, an exact integer >= 0.  
  
    If the result is small enough to fit in an int, return an  
    int.  
    Else return a long.  
  
    >>> [factorial(n) for n in range(6)]  
    [1, 1, 2, 6, 24, 120]  
    >>> [factorial(long(n)) for n in range(6)]
```

```
[1, 1, 2, 6, 24, 120]
>>> factorial(30)
2652528598121910586363084800000000L
>>> factorial(30L)
2652528598121910586363084800000000L
>>> factorial(-1)
Traceback (most recent call last):
...
ValueError: n must be >= 0
```

Factorials of floats are OK, but the float must be an exact integer:

```
>>> factorial(30.1)
Traceback (most recent call last):
...
ValueError: n must be exact integer
>>> factorial(30.0)
2652528598121910586363084800000000L
```

It must also not be ridiculously large:

```
>>> factorial(1e100)
Traceback (most recent call last):
...
OverflowError: n too large
"""
```

```
import math
if not n >= 0:
    raise ValueError("n must be >= 0")
if math.floor(n) != n:
    raise ValueError("n must be exact integer")
if n+1 == n: # catch a value like 1e300
    raise OverflowError("n too large")
result = 1
factor = 2
while factor <= n:
    result *= factor
    factor += 1
return result

if __name__ == "__main__":
    import doctest
    doctest.testmod()
```

# Mocking в Python

Процесс mocking в Python предназначен для подмены объектов или функций для использования при тестировании. Позволяет имитировать поведение объектов или функций, которые могут быть дорогими или сложными в получении или использовании, но не несут важных для процесса тестирования данных. Так тестирующие могут избегать ненужных зависимостей, приостанавливать и изменять поведение процесса, который они тестируют.

Для создания тестовых заглушек в Python можно использовать библиотеки:

- **unittest.mock** — модуль включен в стандартную библиотеку, начиная с Python 3.3;
- **PyMock** — библиотека, которая позволяет создавать и использовать заглушки для изолирования частей кода для тестирования. Она также поддерживает подмену значений аргументов функций, позволяя проверить поведение кода в разных ситуациях.
- **Pytest-mock** — пакет для Python, позволяющий автоматизировать и упростить процесс мокинга в тестах.

Пример использования unittest.mock:

```
from unittest.mock import MagicMock
from my_module import function_to_test

def test_function():
    # Создаем мок-объект для замены зависимости
    dependency_mock = MagicMock(return_value=42)
    # Подменяем зависимость в тестируемой функции
    function_to_test.dependency = dependency_mock
    # Вызываем функцию и проверяем результат
    result = function_to_test(10)
    assert result == 420
    # Проверяем, что зависимость была вызвана с правильным
    аргументом
    dependency_mock.assert_called_once_with(10)
```

Пример использования PyMock:

```
from pymock import PyMock
from my_module import function_to_test

def test_function():
```



```
# Создаем заглушку для зависимости
dependency_mock = PyMock()
# Подменяем зависимость в тестируемой функции
function_to_test.dependency = dependency_mock
# Задаем ожидаемое значение для вызова зависимости
dependency_mock.expect_call_with(10).and_return(42)
# Вызываем функцию и проверяем результат
result = function_to_test(10)
assert result == 420
# Проверяем, что вызов зависимости произошел с правильным
аргументом
dependency_mock.assert_call_with(10)
```

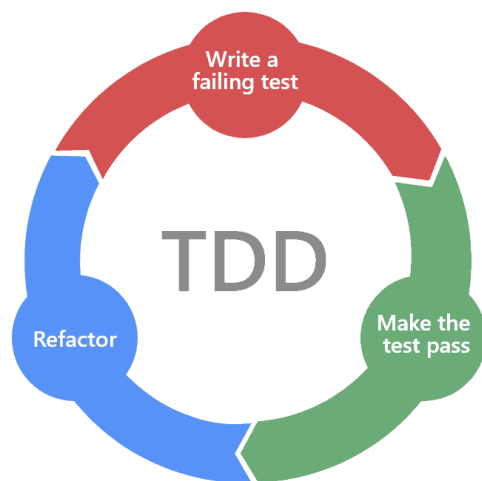
Пример использования pytest-mock:

```
from pytest_mock import mocker
from my_module import function_to_test

def test_function(mocker):
    # Создаем заглушку для зависимости
    dependency_mock = mocker.MagicMock(return_value=42)
    # Подменяем зависимость в тестируемой функции
    function_to_test.dependency = dependency_mock
    # Вызываем функцию и проверяем результат
    result = function_to_test(10)
    assert result == 420
    # Проверяем, что зависимость была вызвана с правильным
    аргументом
    dependency_mock.assert_called_once_with(10)
```

## TDD

**TDD** (Test-Driven Development) — это принцип разработки ПО, при котором сначала пишут тесты, а затем код, который проходит эти тесты. Подход позволяет писать код, который будет достаточно хорошо работать и проходить все тесты.



В Python есть множество инструментов для проведения тестирования TDD, включая `unittest`, `pytest`, `nosetest` и `mock`. Каждый из них позволяет писать тесты для проверки правильности кода и взаимодействия различных компонентов.

## Выбор фреймворка для тестирования в Python

Выбор фреймворка для тестирования на Python зависит от многих факторов: целей и задач проекта, опыта команды, предпочтений и доступности инструментов.

Один из самых популярных фреймворков для тестирования на Python — `Pytest`. У него простой синтаксис и хорошая документация. Он позволяет создавать тесты на основе функций, классов и методов, а также поддерживает множество расширений и интеграций с другими инструментами. Все это делает его очень гибким и удобным для использования.

Если ваш проект основан на фреймворке Django, то Django Testing Framework подойдёт лучше. Этот фреймворк позволяет создавать тесты для приложений Django и дает множество полезных инструментов для тестирования веб-приложений.

Для тестирования API можно использовать фреймворки `Requests` и `Pyresttest`, которые позволяют отправлять HTTP-запросы и проверять ответы.

Если вам нужно тестировать визуальный интерфейс веб-приложения, хорошим выбором будет `Selenium`. Он позволяет автоматизировать тестирование, воспроизводя действия пользователя и проверяя результаты.

Есть множество других фреймворков и инструментов для тестирования на Python (`unittest`, `nose`, `tox` и другие). При выборе ориентируйтесь на задачи проекта и потребности команды.



**Чтобы правильно выбрать инструмент для тестирования, ответьте на вопросы:**

1. На каком языке будет написан тестируемый код?
2. Какой фреймворк популярен среди разработчиков? Активно ли его поддерживают?
3. Какой тип тестирования будет проводиться с использованием этого фреймворка?
4. Какие ограничения есть при использовании этого фреймворка?

## Code coverage

Существуют разные инструменты для оценки покрытия кода в Python: Coverage.py, Pytest-cov, Nose и другие. Они позволяют измерять покрытие тестами для каждой строки кода в вашем приложении и создавать отчеты, которые помогут найти протестированные участки.



**Code coverage в Python** — это метрика, которая измеряет, как много строк кода в приложении было выполнено во время тестирования. Используется для оценки того, насколько хорошо ваш тестовый набор охватывает ваше приложение.

Чтобы использовать инструменты, их сначала нужно установить через менеджер пакетов, например, pip. Затем можно запустить тесты и собрать отчет о покрытии кода. Например, с помощью инструмента Coverage.py можно запустить следующую команду:

```
coverage run my_script.py
```

Она выполнит my\_script.py и соберет отчет о покрытии кода. Затем можно создать отчет с помощью команды:

```
coverage report -m
```

Отчёт покажет процент покрытия кода для каждого модуля, а также для каждой строки кода в модуле.

Также можно использовать инструменты вместе с фреймворками для тестирования, такими как Pytest. Например, для запуска тестов с покрытием можно использовать команду:

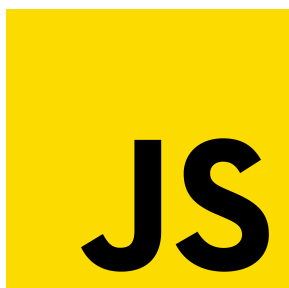
```
pytest --cov=my_package tests/
```

Она запустит все тесты в пакете `my_package` и соберет отчет о покрытии кода.

В целом, code coverage в Python — полезный инструмент. С его помощью можно оценить, насколько хорошо ваш тестовый набор охватывает приложение и выявить участки кода, которые не были протестированы.

## JavaScript

JavaScript — это мультипарадигменный язык программирования для создания интерактивных веб-страниц, веб-приложений, мобильных приложений и многого другого. Он похож на Java, но отличается более гибкой синтаксической структурой. Его также можно использовать для построения бэкенда сайта и для предоставления доступа к базе данных.



- Интерпретируемый
- Мультипарадигменный
- Без строгой типизации
- Применяется для бэкенда

Разберем особенности языка.

- **JavaScript — интерпретируемый язык**, в отличие от большинства современных компилируемых языков. Для работы с кодом, написанным на JS не требуется компиляция. Он передается программе-интерпретатору для исполнения: это облегчает процесс разработки, но требует интерпретатора для запуска кода (интерпретатор JavaScript встроен во все современные браузеры, так что с запуском кода не будет проблем).
- **JavaScript подходит для разных парадигм программирования:** объектной, функциональной и императивной. В объектной парадигме объекты связываются между собой, в функциональной кнопка представляется в виде набора действий, а в императивной парадигме код пишется как набор инструкций, выполняющихся последовательно. JS позволяет программистам использовать любой из этих подходов.

- **JavaScript — язык с динамической типизацией.** В таких языках переменные могут хранить значения разных типов. Например, в JS можно сравнивать строки и числа, результат будет выдаваться. Противоположность — языки со статической типизацией, где при создании переменной нужно указать её тип (например, целое число), а затем хранить в ней только значения этого типа.
- **JavaScript можно запускать и в браузере, и на сервере.** Разработчики могут создавать и реализовывать не только интерактивные элементы веб-страниц, но и серверную часть сайта, включающую в себя функции обработки данных и вычислений. Для этого нужно использовать Node.js — движок, который позволяет запускать JavaScript на сервере.

В JavaScript модульное тестирование основано на использовании специальных библиотек. Самая популярная из них — Jest. Она предоставляет удобный интерфейс для написания, запуска и анализа результатов модульных тестов.

Одна из особенностей модульного тестирования в JavaScript — тесты могут выполняться не только в браузере, но и на стороне сервера с использованием Node.js. Кроме того, в JavaScript используются специальные инструменты для мокирования (например, Sinon.js) и подмены зависимостей (например, Rewire), что позволяет эффективно тестировать отдельные модули и компоненты приложения.

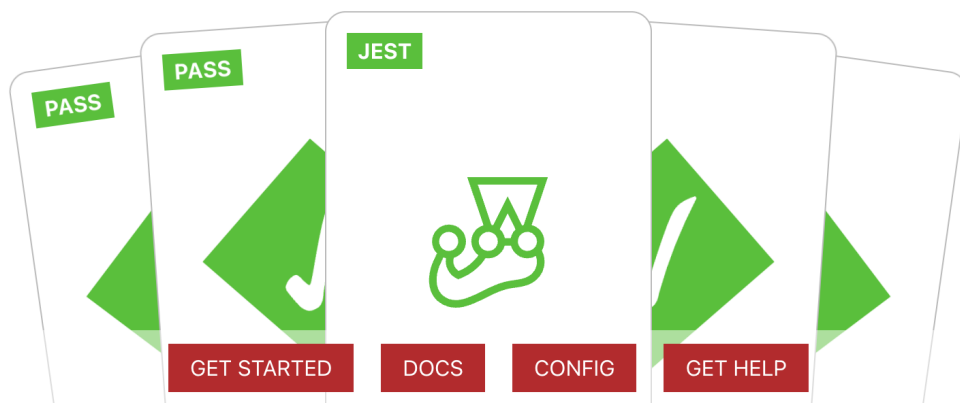
Особенности написания тестов на JavaScript:

- **JavaScript — язык с динамической типизацией. Это значит, что тип переменной может изменяться во время выполнения программы. Поэтому в модульных тестах важно учитывать все возможные значения переменных, которые могут быть переданы в тестируемые функции.**
- В JavaScript функции могут быть переданы в качестве аргументов другим функциям. Это позволяет создавать более гибкие и масштабируемые приложения, но также усложняет их тестирование. В модульных тестах необходимо учитывать все возможные комбинации аргументов, которые могут быть переданы в функцию.
- В языке JavaScript используется механизм асинхронного программирования с помощью колбэков, промисов и `async/await`. В модульных тестах необходимо учитывать все возможные варианты обработки ошибок, а также проверять, что асинхронный код выполняется корректно.
- JavaScript имеет многообразие платформ и сред исполнения. Тесты должны запускаться в разных средах, таких как браузер или Node.js, и проверять, что код работает корректно во всех условиях.

- Важно следить за производительностью тестов. В JavaScript модульные тесты могут занимать много времени, особенно если выполняются на больших объемах данных или в условиях высокой нагрузки. Поэтому нужно оптимизировать тесты и убедиться, что они не замедляют работу приложения.

Рассмотрим инструменты для работы с JavaScript.

1. Линтеры: JSLint, JSHint и ESLint. Они нужны для проверки синтаксиса, управления памятью и безопасностью, а также для создания систем автоматических замен. Иногда программисты могут использовать несколько линтеров одновременно, чтобы полнее анализировать код.
- 1) Популярные фреймворки и библиотеки тестирования JavaScript: **Jest**, **Jasmine** и **Mocha (мока)**.



**Jest** стал самой популярной библиотекой для тестирования в 2017 году. С тех пор остается на первом месте.

- Предоставляет мощный инструмент автоматизации тестирования, который позволяет создавать и запускать тесты в несколько разных браузерах.
- Предлагает поддержку асинхронных тестов и приложений для простого подключения к серверу.
- Позволяет протестировать приложения, использующие базу данных.

**Jasmine** — библиотека для тестирования браузерных и нативных приложений. Поддерживает асинхронные тесты и позволяет использовать различные материалы для тестирования. Предоставляет подробные отчеты для отслеживания прогресса тестирования.

**Mocha** обеспечивает гибкие возможности тестирования, поддержку асинхронных тестов и возможность запуска тестов в разных окружениях, включая браузеры и

средства командной строки. Поддерживает разные типы тестов, включая асинхронные тесты и тесты производительности.

Пример — несколько тестов, которые используют методы сложения, вычитания, умножения и деления в объекте `calculator` для проверки его функциональности:

```
test('adding two numbers with the add method should return the correct sum', () => {
  const calculator = new Calculator();
  expect(calculator.add(5, 10)).toBe(15);
  expect(calculator.add(-5, 10)).toBe(5);
});

test('subtracting two numbers with the subtract method should return the correct difference', () => {
  const calculator = new Calculator();
  expect(calculator.subtract(10, 5)).toBe(5);
  expect(calculator.subtract(-5, 10)).toBe(-15);
});

test('multiplying two numbers with the multiply method should return the correct product', () => {
  const calculator = new Calculator();
  expect(calculator.multiply(5, 10)).toBe(50);
  expect(calculator.multiply(-5, 10)).toBe(-50);
});

test('dividing two numbers with the divide method should return the correct quotient', () => {
  const calculator = new Calculator();
  expect(calculator.divide(10, 2)).toBe(5);
  expect(calculator.divide(10, 3)).toBeCloseTo(3.333, 3);
});
```

Ключевое слово **expect** связывает объект, который мы тестируем, с ключевым словом **toBe**, которое указывает ожидаемый результат. Чтобы больше узнать о возможностях ключевого слова `expect` в Jest, изучите раздел **matchers** в его [документации](#).

## Golang (Go)

Это компилируемый многопоточный язык с открытым исходным кодом. В основном его применяют в веб-сервисах и клиент-серверных приложениях.

Для него характерны:



- Простота и читаемость
- Эффективность
- Безопасность
- Кроссплатформенность
- Встроенная поддержка многопоточности
- Строгая типизация
- Сборка мусора
- Совместимость с C
- Расширенная стандартная библиотека

Golang — язык программирования от компании Google, разработанный в 2007 году. Был создан с целью предоставить программистам простой, быстрый и масштабируемый язык для разработки приложений для веб-сервисов.

- У Go приятный и лаконичный синтаксис. Его легко использовать и понимать.
- Go имеет встроенную параллельную производительность. Она позволяет масштабировать приложения на нескольких платформах и поддерживает многозадачность.
- Go также имеет статическую типизацию и безопасность. Они помогают разработчикам создавать стабильные приложения.

У Go много характерных черт. Некоторые уникальны, другие заимствованы из языков программирования:

- **Простота и читаемость:** Go разработан так, чтобы быть легким для чтения и изучения.
- **Эффективность:** Go обеспечивает быструю скорость компиляции и имеет встроенную поддержку параллелизма, что позволяет легко писать конкурентный код.
- **Безопасность:** Go имеет встроенную поддержку безопасности памяти и типов, что уменьшает количество ошибок, связанных с утечкой памяти или неправильным использованием типов данных.
- **Кроссплатформенность:** Go может быть скомпилирован для разных операционных систем и архитектур процессоров.



- **Встроенная поддержка многопоточности:** Go обеспечивает простой и эффективный способ создания конкурентного кода с помощью горутин и каналов.
- **Строгая типизация:** Go позволяет избежать ошибок, связанных с несоответствием типов данных.
- **Сборка мусора:** В Go есть встроенный сборщик мусора. Это упрощает управление памятью и избавляет разработчиков от необходимости управлять памятью вручную.
- **Совместимость с C:** Go может вызывать функции из библиотек, написанных на C.
- **Расширенная стандартная библиотека:** у Go обширная стандартная библиотека со множеством полезных пакетов для работы с сетью, вводом-выводом, шифрованием и другими задачами.

Модульное тестирование в Go — важная практика для обеспечения качества кода.

Некоторые особенности модульного тестирования в Golang:

- **Встроенный пакет тестирования Testing** обеспечивает возможность писать модульные тесты для функций и пакетов.
- **Поддержка таблиц тестов**, которые позволяют определить набор тестовых данных и ожидаемый результат для каждого теста в удобном формате.
- **Поддержка бенчмарков** позволяет измерять производительность функций и пакетов.
- **Простота и скорость компиляции:** тесты в Golang компилируются вместе с основным кодом и могут быть запущены без необходимости установки дополнительных инструментов или библиотек.
- **Поддержка параллельного выполнения тестов** может значительно сократить время выполнения тестов в больших проектах.
- **Встроенная поддержка Code coverage** позволяет измерять покрытие кода тестами.

Рассмотрим популярные фреймворки и библиотеки для тестирования в Golang:

- **Testing** — встроенный в Golang фреймворк для модульного тестирования. Предоставляет функции для написания тестов и проверки результатов, а также возможность создать бенчмарки для измерения производительности.

- **GoConvey** — фреймворк, который обеспечивает удобный интерфейс для написания тестов и проверки результатов. Поддерживает функциональное и интеграционное тестирование.
- **Ginkgo** — библиотека для BDD-тестирования (Behavior Driven Development) на Golang. Предоставляет функции для написания тестов, описывающих поведение приложения, а также удобный интерфейс для проверки результатов.
- **Testify** — библиотека с множеством удобных функций для написания тестов и проверки результатов. Также поддерживает функциональное и интеграционное тестирование.
- **Gomega** — библиотека, которая обеспечивает удобный интерфейс для написания тестов и проверки результатов. Предоставляет расширенные возможности для проверки сложных типов данных, таких как структуры и массивы.

У каждого из этих фреймворков и библиотек есть свои особенности и преимущества. Какой выбрать — зависит от задачи и потребностей разработчика.

Пример модульного теста с использованием встроенной библиотеки для функции Add калькулятора, который складывает два числа и возвращает результат:

```
package calculator

import "testing"

func TestAdd(t *testing.T) {
    // Arrange
    num1 := 5
    num2 := 10
    expected := 15

    // Act
    result := Add(num1, num2)

    // Assert
    if result != expected {
        t.Errorf("Add(%d, %d) = %d; expected %d", num1, num2,
result, expected)
    }
}
```

В примере мы определили тестовые данные, вызвали функцию Add с этими данными и сравнили полученный результат с ожидаемым. Если результаты не совпадают, тест будет провален и выведется сообщение об ошибке. Если результаты совпадают, тест будет пройден успешно.

Это простой пример тестирования функции Add калькулятора. Его можно использовать в качестве отправной точки для более сложных модульных тестов для других функций.

🔥 Go не предоставляет утверждений (assert). Вот как создатели языка это объясняют: они, несомненно, удобны, но наш опыт показывает, что программисты используют их как опору, чтобы не думать о правильной обработке ошибок и отчетности.

Правильная обработка ошибок означает, что серверы продолжают работать, а не выходят из строя после неустранимой ошибки.

Правильное сообщение об ошибках означает, что ошибки являются прямыми и точными, что избавляет программиста от интерпретации большой трассировки сбоев.

Точные ошибки особенно важны, когда программист, видящий ошибки, не знаком с кодом.

## Выводы: какой язык выбрать для написания тестов

Несколько рекомендаций, которые помогут вам определиться с выбором:

- **Рассмотрите язык, на котором написано приложение.** Если приложение написано на конкретном языке, то тесты, написанные на нем же, могут лучше интегрироваться с приложением. Писать тесты будет легче.
- **Исследуйте доступность и поддержку библиотек и фреймворков** для тестирования на выбранном языке. Хорошо развитая экосистема тестирования может значительно упростить процесс написания и запуска тестов.
- **Учитывайте опыт команды разработчиков.** Если команда знакома с определенным языком, используйте его. Процесс написания тестов ускорится, они будут качественными.

- **Определите сложность и масштаб проекта.** Для масштабных проектов с большим объемом кода может быть целесообразно использовать языки с более высоким уровнем абстракции и возможностями для модульного тестирования.
- **Рассмотрите возможности интеграции с инструментами для автоматизации** тестирования, такими как CI/CD-платформы. Они могут значительно упростить процесс запуска и поддержки тестов.
- **Учитывайте предпочтения.** Некоторые разработчики предпочитают определенные языки программирования и фреймворки. На них им легче писать тесты, так что эффективность тестирования будет выше.

Таблица с плюсами и минусами инструментов тестирования различных языков программирования. Среди них и те, которые мы не рассматривали подробно:

Язык	Преимущества	Недостатки
Java	Множество инструментов и библиотек для тестирования, широко используется для корпоративных проектов	Сложный в использовании, большой объем кода, может быть медленным
Python	Прост в использовании, хорошо подходит для автоматизации, большое сообщество пользователей, богатые библиотеки	Низкая производительность, может потребоваться дополнительная настройка окружения, сложность в распределенном тестировании
Java Script	Распространенный язык, хорошо подходит для тестирования веб-приложений	Сложности в тестировании на стороне сервера, множество версий, отсутствие строгой типизации, сложность в обработке ошибок
Golang	Быстрый и эффективный, подходит для масштабируемых проектов, сильная типизация, хорошая поддержка конкурентности	Не имеет стандартной библиотеки для тестирования, меньшее количество библиотек и фреймворков для тестирования по сравнению с другими языками

Ruby	Легко читаемый код, простой синтаксис, хорошо подходит для тестирования веб-приложений	Низкая производительность, недостаточно поддержки многопоточности, неудобство в работе с современными фреймворками, не подходит для масштабирования
C#	Хорошо подходит для тестирования Windows-приложений, отличная интеграция с Visual Studio	Требует наличия Windows, сложный для использования на других платформах

Во многих языках программирования есть встроенная поддержка инструментов модульного тестирования. Это позволяет разработчикам быстро и эффективно протестировать код и убедиться, что он правильный до того, как он будет внедрен в более широкую среду. Ниже часть таблицы, в которой описана степень поддержки разными языками инструментов тестирования:

Язык программирования	Встроенная поддержка тестирования
C#	Да
Java	Да
Python	Да
Ruby	Да
Swift	Да
JavaScript	Частично
PHP	Частично
Go	Нет
Rust	Нет

C++	Нет
Kotlin	Да
Objective-C	Да
Scala	Да
TypeScript	Да

## Антипаттерны тестирования

**Антипаттерны** — это проектные решения, которые сперва кажутся приемлемыми, но по мере разработки приводят к нежелательным последствиям. Могут существенно ухудшить качество продукта и замедлить процесс разработки, поэтому нужно их избегать.

Разберем основные антипаттерны юнит-тестирования с примерами на Python:

1. **Чрезмерное тестирование (Eager Test)** — проверка результата теста с помощью утверждения `assert`. Некоторые разработчики могут злоупотреблять этой конструкцией: например, писать длинные утверждения. Этот антипаттерн относится к написанию тестов, которые слишком много тестируют и перепроверяют одну и ту же функциональность, что может привести к избыточности тестов и увеличению времени их выполнения.

```
def test_multiply():  
    assert multiply(3, 4) == 12  
    assert multiply(0, 4) == 0  
    assert multiply(2, 2) == 4  
    assert multiply(-2, -2) == 4  
    assert multiply(-2, 2) == -4 # опасное отрицание!
```

- Функция `test_multiply()` проверяет различные входные данные для функции `multiply()` и утверждает, что результат равен ожидаемому значению с помощью `assert`.

- Однако используется слишком много утверждений `assert`. Это может привести к избыточности тестов и увеличению времени их выполнения.
- В этом примере можно было проверить только несколько входных значений, чтобы убедиться, что функция работает правильно, и не тратить время на проверку всех возможных значений.

2. **Копипаста (Copy-Paste)** — дублирование кода в разных тестах. Может привести к тому, что изменения в одном тесте не будут применены в других тестах.

```
def test_add():
    assert add(1, 2) == 3
    assert add(3, 4) == 7
    assert add(4, 5) == 9

def test_subtract():
    assert subtract(1, 2) == -1
    assert subtract(3, 4) == -1
    assert subtract(4, 5) == -1 # копия!
```

- Функция `test_add()` и функция `test_subtract()` используют похожие утверждения `assert` для разных функций.
- Дублирование кода может привести к тому, что изменения в одном тесте не будут применены в других тестах.
- Лучше использовать общие функции и переменные, чтобы уменьшить дублирование кода и облегчить его поддержку.

3. **Тест-ничего-не-делает (Do-Nothing Test)** — написание тестов, которые не делают ничего, кроме вызова функции, которую они тестируют.

```
def test_multiply():
    multiply(3, 4)
    multiply(0, 4)
    multiply(2, 2)
    multiply(-2, -2)
```

- Функция `test_multiply()` вызывает функцию `multiply()` для разных входных данных, но не проверяет результат с помощью утверждений `assert`.

- Тест бессмысленный, он не проверяет правильность работы функции `multiply()` и не дает обратной связи о том, прошел тест или нет.
- Лучше использовать утверждения `assert` для проверки результатов работы функций и обеспечения правильности работы кода.

4. **Непонятные имена (Unclear Names)** — в тесте используются неинформативные имена. Это может привести к тому, что другие разработчики не смогут понять, что в нем тестируется.

```
def test_foo():  
    assert foo(3) == 6  
    assert foo(4) == 8  
    assert foo(5) == 10 # непонятное имя!
```

- У функции `test_foo()` неинформативное имя. Оно не описывает, что тестируется.
- Лучше использовать информативные имена для тестов. Они помогут другим разработчикам понять, что тестируется и какие входные данные используются для проверки кода.

5. **Ненадежные данные (Unreliable Data)** — использование случайных данных для тестирования. Может привести к непредсказуемым результатам.

```
def test_multiply():  
    x = random.randint(0, 10)  
    y = random.randint(0, 10)  
    assert multiply(x, y) == x * y # ненадежные данные!
```

- Функция `test_multiply()` использует случайные данные для тестирования функции `multiply()`.
- Использование ненадежных данных может привести к непредсказуемым результатам, так как мы не можем гарантировать, что случайные данные будут проверять все возможные случаи.
- Лучше использовать определенные входные данные, которые покрывают все возможные случаи и обеспечивают более надежное тестирование.

6. **Тестирование зависимостей (Testing Dependencies)** вместо тестируемой функциональности. Например, тестирование функции, которая использует базу данных, может привести к тому, что тест будет зависеть от работы базы данных.



```
def test_get_user():
    assert get_user(1) == {"name": "John", "age": 30} #
    тестирование зависимостей!
```

- Тест проверяет функцию `get_user`, используя фиксированное значение в качестве аргумента. Однако, как написано в комментарии, этот тест является тестированием зависимостей, он не тестирует функциональность.
- Тестирование зависимостей может быть полезно, чтобы проверить: зависимости (например, база данных) правильно настроены и доступны. Но оно не гарантирует, что функциональность работает правильно во всех возможных случаях. Кроме того, тестирование зависимостей может быть нестабильным и непредсказуемым из-за непостоянства или изменчивости внешних зависимостей.
- В идеале для тестирования функции, использующей базу данных, нужно использовать тестовую базу данных, которая может быть создана, настроена и запущена автоматически во время выполнения теста. Тестирование будет надежным и предсказуемым.

Антипаттерны могут привести к тому, что тесты будут менее эффективными и труднее поддерживаемыми. Чтобы этого избежать, нужно следовать лучшим практикам и обращаться к библиотекам. Они помогут сократить время написания тестов и улучшить их качество.

## Автоматизация тестирования

Непрерывная интеграция (CI) — это метод автоматического сбора и тестирования кода в репозитории при каждом изменении. Автоматизация тестирования и непрерывная интеграция — важные аспекты разработки ПО. Они позволяют ускорить процесс разработки, улучшить качество и надежность кода.

Python — отличный выбор для автоматизации тестирования и CI благодаря своей простоте, удобству, мощным библиотекам и большому сообществу разработчиков. Для Python вы можете использовать такие инструменты, как Jenkins, Travis CI и CircleCI.

Пример конфигурации Travis CI для автоматического запуска тестов при каждом изменении в коде:

```
language: python
```

```
python:
  - "3.6"
  - "3.7"
  - "3.8"
  - "3.9"

install:
  - pip install -r requirements.txt

script:
  - python -m unittest discover
```

Эта конфигурация говорит Travis CI, что используется Python, какие версии Python должны быть проверены, как установить необходимые зависимости и как запустить тесты.

```
pipeline {
  agent any
  stages {
    stage('Build') {
      steps {
        sh 'python setup.py install'
      }
    }
    stage('Test') {
      steps {
        sh 'python -m pytest tests/'
      }
    }
  }
}
```

Этот код описывает pipeline (конвейер) в Jenkins для автоматической сборки, тестирования и развертывания приложения. Он состоит из двух этапов: Build (сборка) и Test (тестирование).

На этапе Build выполняется команда **python setup.py install**, которая устанавливает приложение и его зависимости.

На этапе Test выполняется команда **python -m pytest tests/**, которая запускает тесты приложения, расположенные в папке tests/.

Конвейер описывается с помощью декларативного синтаксиса, который позволяет определить последовательность этапов и команд для их выполнения.

Агент `any` указывает, что конвейер может запускаться на любом агенте, доступном в Jenkins.

`Stages` (этапы) описывают отдельные этапы работы конвейера, а `steps` (шаги) определяют команды, которые нужно выполнить на каждом этапе.

Этот код — пример хорошей практики использования CI/CD для автоматической сборки и тестирования приложения при каждом изменении кода, что позволяет ускорить процесс разработки и обеспечивает более стабильную работу приложения.

## ИИ и нейросети для юнит-тестирования

С развитием информационных технологий и программного обеспечения создание надежных и качественных приложений все более актуально.

Поговорим о применении искусственного интеллекта (ИИ) и нейросетей для юнит-тестов в разных языках программирования.

- Современные нейросетевые архитектуры, такие как GPT-4 от OpenAI, могут анализировать большие объемы данных и обучаться на основе выявленных закономерностей. Это позволяет создавать инструменты, способные автоматически генерировать юнит-тесты, учитывая особенности разных языков программирования.
- Специализированные нейросети могут быть обучены для работы с разными языками программирования (Java, Python, JavaScript, C++ и другими). Они могут учитывать особенности синтаксиса и структуры данных каждого языка, обеспечивая корректность и эффективность тестов.
- ИИ-инструменты для генерации юнит-тестов могут быть интегрированы с популярными фреймворками тестирования (JUnit, Pytest и другими). Это облегчает процесс создания и выполнения тестов, а также позволяет разработчикам использовать привычные средства контроля и отчетности.
- Для удобства разработчиков существуют плагины для различных интегрированных сред разработки (IDE), которые предоставляют возможность генерации юнит-тестов с помощью ИИ и нейросетей. Это ускоряет процесс написания тестов и делает его более гибким, позволяя разработчикам сосредоточиться на основных задачах.

Testing



# Diffblue Cover - Create complete JUnit tests with AI



Diffblue Ltd

🔥 Несмотря на преимущества автоматической генерации юнит-тестов с использованием ИИ и нейросетей, разработчикам все равно важно понимать код, который они тестируют. Искусственный интеллект может создать тесты на основе выявленных закономерностей и шаблонов, однако ответственность за качество и надежность программного обеспечения лежит на разработчиках.

Разработчики должны проверять автоматически сгенерированные тесты, чтобы убедиться в их корректности и полноте покрытия кода. Также следует уделить внимание ручному тестированию и анализу кода, поскольку они позволяют обнаружить возможные проблемы и уязвимости, которые могут быть упущены автоматизированными методами.

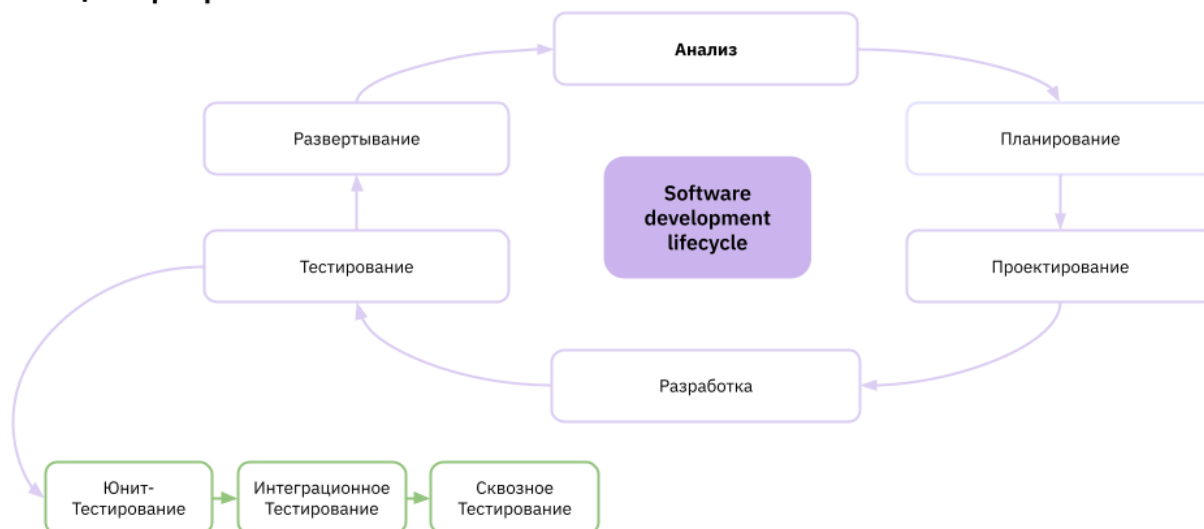
Искусственный интеллект и нейросети дают новые возможности для улучшения качества и надежности программного обеспечения. Однако разработчикам важно сохранять активное участие в процессе тестирования, обеспечивать корректность и полноту проверок.

## Итоги курса

На курсе мы рассмотрели цикл разработки программного обеспечения, включающий этапы анализа требований, проектирования, реализации, тестирования и сдачи продукта.



## Цикл разработки. SDLC



Мы разобрали принципы тестирования программного обеспечения, включая планирование, использование разных видов тестирования, обеспечение полного покрытия тестами и повторяемость тестов.

Рассмотрели разные типы тестирования: модульное, интеграционное, сквозное и тестирование пограничных случаев. Узнали, какими инструментами можно измерить покрытие тестами.

Изучили методологии разработки через тестирование (TDD) и через поведение (BDD). Разобрали понятие зависимостей, их типы и использование test doubles для изоляции тестируемого кода.

Узнали, что такое тестирование по принципу чёрного и белого ящика, и что такое пирамида тестов, определяющая соотношение между разными типами тестов, включая юнит-, интеграционные и функциональные тесты.

Рассмотрели принципы и практики Agile-методологий: Scrum и Kanban. Они помогают команде разработчиков организовать процесс разработки программного обеспечения, разбивая его на короткие итерации и активно взаимодействуя с заказчиком, чтобы обеспечить максимально возможную ценность продукта.

Узнали про принципы непрерывной интеграции и непрерывной доставки, которые позволяют автоматизировать процессы тестирования и доставки ПО, что ускоряет и упрощает разработку.

В целом, курс позволяет получить базовые знания о процессе разработки программного обеспечения, о методологиях, инструментах и практиках,

используемых в этом процессе, а также о том, как правильно планировать и проводить тестирование ПО, чтобы обеспечить его качество и надёжность.



Спасибо за участие в курсе! Желаем успехов в карьере и надеемся, что приобретенные знания и опыт будут полезны на профессиональном пути.

Помните, что сильное сообщество разработчиков и специалистов всегда готово помочь вам в изучении новых тем и в решении проблем. Не стесняйтесь обращаться за помощью, делиться успехами и сотрудничать с коллегами по отрасли.

## Глоссарий курса

**Модульное тестирование** — один из основных типов тестирования ПО, который позволяет проверить работу отдельных модулей кода.

**Интеграционное тестирование** — тестирование, которое проверяет правильность взаимодействия между модулями или компонентами программы.

**Сквозные (end-to-end) тесты** — тесты, которые проверяют работу программы в целом.

**Пограничные случаи** — тестовые сценарии, которые проверяют правильность работы программы при неправильных или неожиданных входных данных.

**Утверждения (Assert)** — инструкции, которые проверяют правильность выполнения определенных условий в тесте.

**Принципы чёрного ящика и белого ящика** — два подхода к тестированию. Отличаются тем, какую информацию о программе использует тестировщик.

**Инструменты для измерения покрытия тестами** — инструменты, измеряющие, какое количество кода протестировано.

**Разработка через тестирование** (Test-driven development или TDD) — методология разработки, при которой сначала создаются тесты, а затем код, который проходит эти тесты.

**Разработка через поведение** (Behavior-driven development или BDD) — методология разработки, которая ставит целью описать поведение системы на языке, понятном заказчику.

**Зависимости** — это объекты или компоненты, которые используются внутри других объектов или компонентов. Бывают разных типов: зависимости от классов, зависимости от внешних компонентов и зависимости от конфигурационных файлов.

**Test double** — это объекты, которые заменяют реальные зависимости в тестах, чтобы изолировать тестируемый код. Чтобы заменять зависимости в тестах, используют разные типы test doubles: заглушки (stubs), имитаторы (fakes), моки (mocks), шпионы (spies).

**Mocking Frameworks** — инструменты, которые упрощают создание мок-объектов и управление ими в тестах.

**Пирамида тестов** — методика, которая определяет соотношение между разными типами тестов. В ней тесты делятся на три уровня: юнит-тесты, интеграционные тесты и функциональные (или сквозные) тесты.

## Домашнее задание

1. **Установите последнюю версию Python.** Загрузите установщик Python [с официального сайта](#) и следуйте инструкциям по установке.
2. **Установите IDE PyCharm.** Загрузите установщик PyCharm [с официального сайта](#) и следуйте инструкциям по установке.
3. **Установите Pytest и линтеры.** После установки Python и PyCharm установите пакет Pytest и один из линтеров (например, Flake8 или PyLint). Для установки пакетов можно использовать утилиту pip, запустив команду из лекции в терминале.
4. **Напишите код калькулятора.** После установки всех инструментов, напишите код для простого калькулятора на Python. Достаточно функции сложения, вычитания, умножения и деления.
5. **Напишите тесты.** Когда код калькулятора будет готов, напишите тесты для его функций.
6. **Запустите тесты и проверьте код с помощью линтера.** Убедитесь, что все тесты проходят успешно, а код соответствует рекомендациям по стилю.

## Полезные материалы

Список книг с полезными рекомендациями для модульного тестирования на разных языках программирования:

1. **xUnit Test Patterns: Refactoring Test Code** by Gerard Meszaros — в книге описаны шаблоны проектирования и лучшие практики для модульного тестирования. Автор рассматривает разные типы тестов, включая юнит-, функциональные и интеграционные тесты, и показывает, как использовать их для обеспечения качества кода.
2. **Test Driven Development: By Example** by Kent Beck — в книге описана методология разработки TDD (Test-Driven Development). Автор объясняет, как использовать модульное тестирование для написания кода, который будет проходить тесты, а также как это поможет улучшить качество кода и сократить время на отладку.
3. **Effective Unit Testing: A guide for Java Developers** by Lasse Koskela — в книге рассмотрены лучшие практики и методы для написания эффективных юнит-тестов на языке Java. Автор показывает, как использовать модульное тестирование для обеспечения качества кода, повышения надежности и улучшения производительности.
4. **Python Testing with pytest** by Brian Okken — в книге описано, как использовать инструмент pytest для написания и выполнения тестов на Python. Автор объясняет, как использовать pytest для создания и запуска тестовых наборов, а также для проверки результатов тестирования.
5. **JavaScript Testing with Jasmine: JavaScript Behavior-Driven Development** by Evan Hahn — в книге рассматривается инструмент Jasmine для модульного тестирования JavaScript-приложений. Автор объясняет, как использовать Jasmine для создания и выполнения тестовых наборов, а также для проверки результатов тестирования.
6. **The Art of Unit Testing: With Examples in .NET** by Roy Oshero — в книге рассматриваются лучшие практики и методы для написания эффективных юнит-тестов на платформе .NET. Автор объясняет, как использовать модульное тестирование для улучшения качества кода, сокращения времени на отладку и улучшения производительности.
7. **Go Testing and Debugging** by John Arundel — в книге описаны методы и инструменты для модульного тестирования на языке Go. Автор показывает, как использовать модульное тестирование для обеспечения качества кода и улучшения производительности.
8. **JUnit in Action** by Petar Tahchiev, Felipe Leme, Vincent Massol, and Gary Gregory — в книге описан инструмент JUnit для модульного тестирования на Java. Авторы показывают, как использовать JUnit для создания и



выполнения тестовых наборов, а также для проверки результатов тестирования.

9. **Practical Unit Testing with TestNG and Mockito** by Tomek Kaczanowski — в книге рассматриваются инструменты TestNG и Mockito для модульного тестирования на Java. Автор показывает, как использовать их для создания и выполнения тестовых наборов, а также для проверки результатов тестирования.
10. **Test-Driven Development with Python** by Harry J.W. Percival — в книге описана методология разработки ПО TDD с использованием языка Python. Автор показывает, как использовать модульное тестирование для создания и выполнения тестовых наборов, а также для проверки результатов тестирования.

**Ссылки на статьи**, в которых подробно раскрывается тема этого урока:

1. [Общая картина модульного тестирования](#)
2. [Модульное тестирование](#)
3. [How To Select The Right Test Automation Tool - TestProject](#) — подробнее про выбор инструмента тестирования
4. [Линтеры в Python](#) — линтеры Python
5. [PEP 8 – Style Guide for Python Code](#) — документация PEP8
6. [Full pytest documentation](#) — документация фреймворка PyTest
7. [Python Unit Testing Framework](#) — документация фреймворка PyUnit
8. [ROBOT FRAMEWORK](#)
9. [8 лучших фреймворков для тестирования с помощью Python в 2021 году](#) — про тестовые фреймворки Python
10. [Step 1. Create and run your first Python project | PyCharm Documentation](#)
11. [Code coverage | PyCharm Documentation](#)
12. [An Overview of JavaScript Testing in 2022 by Vitali Zaidman | Welldone Software](#) — про тестирование в JavaScript
13. [Анти-паттерны Test Driven Development](#) — про антипаттерны
14. [How To Select The Right Test Automation Tool - TestProject](#) — про выбор инструмента