|  |
| --- |
| МИНОБРНАУКИ РОССИИ |
| Федеральное государственное бюджетное образовательное учреждение высшего образования **«МИРЭА  Российский технологический университет»**  **РТУ МИРЭА** |

**Институт информационных технологий (ИИТ)**

**Кафедра инструментального и прикладного программного обеспечения (ИиППО)**

**ОТЧЁТ ПО УЧЕБНОЙ ПРАКТИКЕ**

**Ознакомительная практика**

приказ Университета о направлении на практику от «12» февраля 2025 г. №1427-C

|  |  |  |
| --- | --- | --- |
| Отчет представлен к  рассмотрению:  Студент группы ИКБО-10-24 | «\_\_» июня 2025 | Лесовой К.Р.  (подпись и расшифровка подписи) |
| Отчет утвержден.  Допущен к защите: |  |  |
| Руководитель практики  от кафедры | «\_\_» июня 2025 | Маличенко С.В.  (подпись и расшифровка подписи) |
|  |  |  |

Москва 2025 г.

|  |
| --- |
| МИНОБРНАУКИ РОССИИ |
| Федеральное государственное бюджетное образовательное учреждение высшего образования **«МИРЭА  Российский технологический университет»**  **РТУ МИРЭА** |

**Институт информационных технологий (ИИТ)**

**Кафедра инструментального и прикладного программного обеспечения (ИиППО)**

**ИНДИВИДУАЛЬНОЕ ЗАДАНИЕ НА УЧЕБНУЮ ПРАКТИКУ**

**Ознакомительная практика**

**Студенту 1 курса учебной группы ИКБО-10-24**

**Лесовому Кириллу Романовичу**

**Место и время практики:** РТУ МИРЭА кафедра ИиППО, с 10 февраля 2025 г. по 31 мая 2025 г.

**Должность на практике:**  студент

**1. СОДЕРЖАНИЕ ПРАКТИКИ:**

1.1. Изучить: основные методы отладки программ на языке Python, включая использование встроенных и сторонних инструментов, и подходы к выявлению ошибок в коде.

1.2. Практически выполнить: подготовить примеры программ, которые нужно проанализировать, и отладить их с целью научиться пользоваться инструментами отладки кода.

1.3. Ознакомиться: с библиотеками для анализа работы кода и современными инструментами отладки, такими как интегрированные среды разработки (IDE) с поддержкой дебаггинга, и их функционалом для упрощения процесса.

**2. ДОПОЛНИТЕЛЬНОЕ ЗАДАНИЕ:** подготовить доклад на научно-техническую конференцию студентов и аспирантов РТУ МИРЭА или иную конференцию, подготовить презентационный материал

**3. ОРГАНИЗАЦИОННО-МЕТОДИЧЕСКИЕ УКАЗАНИЯ:** В процессе практики рекомендуется использовать периодические издания и отраслевую литературу годом издания не старше 5 лет от даты начала прохождения практики

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Руководитель практики от кафедры  «10» февраля 2025 г. | | | *Подпись* | | (Маличенко С.В.) |
| Задание получил | |  |  | |  |
| «10» февраля 2025 г. | |  | *Подпись* | | (Лесовой К.Р.) |
|  | |  |  | |  |
|  | |  |  | |  |
| **СОГЛАСОВАНО:** | |  |  | |  |
| Заведующий кафедрой: | | | | | |
| «10» февраля 2025 г. | | | *Подпись* | | (Болбаков Р.Г.) |
| **Проведенные инструктажи:** |  | | |  | |
| Охрана труда: |  | | | «10» февраля 2025 г. | |
| Инструктирующий | *Подпись* | | | Болбаков Р.Г., зав. кафедрой ИиППО | |
| Инструктируемый | *Подпись* | | | Лесовой К.Р. | |
|  |  | | |  | |
| Техника безопасности: |  | | | «10» февраля 2025 г. | |
| Инструктирующий | *Подпись* | | | Болбаков Р.Г., зав. кафедрой ИиППО | |
| Инструктируемый | *Подпись* | | | Лесовой К.Р. | |
|  |  | | |  | |
| Пожарная безопасность: |  | | | «10» февраля 2025 г. | |
| Инструктирующий | *Подпись* | | | Болбаков Р.Г., зав. кафедрой ИиППО | |
| Инструктируемый | *Подпись* | | | Лесовой К.Р. | |
|  | | | |  | |
| С правилами внутреннего распорядка ознакомлен: | | | | «10» февраля 2025 г. | |
|  | *Подпись* | | | Лесовой К.Р. | |

|  |
| --- |
| МИНОБРНАУКИ РОССИИ |
| Федеральное государственное бюджетное образовательное учреждение высшего образования **«МИРЭА  Российский технологический университет»**  **РТУ МИРЭА** |

**РАБОЧИЙ ГРАФИК ПРОВЕДЕНИЯ  
ОЗНАКОМИТЕЛЬНОЙ ПРАКТИКИ**

Студента Лесового К.Р. 1 курса группы ИКБО-10-24 очной формы обучения, обучающегося по направлению подготовки 09.03.04 Программная инженерия

|  |  |  |  |
| --- | --- | --- | --- |
| **Неделя** | **Сроки**  **выполнения** | **Этап** | **Отметка о выполнении** |
| 1 | 10.02.2025 | Подготовительный этап, включающий в себя организационное собрание (Вводная лекция о порядке организации и прохождения ознакомительной практики, инструктаж по технике безопасности, получение задания на практику) |  |
| 5 | 10.03.2025 | Участие в круглом столе на тему «Проблема достоверности информации в современном мире» |  |
| 6 | 20.03.2025 | Участие в круглом столе на тему «Информационно-коммуникационные технологии для организации информационного процесса» |  |
| 8 | 31.03.2025 | Участие в круглом столе на тему «Информационная и библиографическая культура» |  |
| 8 | 01.04.2025 | Исследовательский этап (Поиск, отбор и анализ материалов для выполнения задания по практике) |  |
| 10 | 18.04.2025 | Согласование с руководителем доклада на научно-техническую конференцию студентов и аспирантов РТУ МИРЭА |  |
| 11 | 21.04.2025 | Представление руководителю структурированного материала: аналитический обзор предметной области |  |
| 11 | 22.04.2025 | Технологический этап (разработка программного продукта) |  |
| 15 | 19.05.2025 | Представление разработанного программного продукта |  |
| 16 | 31.05.2025 | Подготовка окончательной версии отчета и программного продукта (Оформление материалов отчета в полном соответствии с требованиями на оформление ГОСТ 7.32-2017) |  |

Руководитель практики от  
кафедры \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_/Маличенко С.В. к.т.н., ассистент/

Обучающийся \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_/Лесовой К.Р./

**Согласовано:**

Заведующий кафедрой \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_/Болбаков Р.Г., к.т.н., доцент/

# РЕФЕРАТ

Отчёт 83 с., 35 рис., 53 листингов, 2 табл., 0 источн., 10 прил.

МЕТОДЫ И ИНСТРУМЕНТЫ ОТЛАДКИ И ПРОФИЛИРОВАНИЯ ПРОГРАММ ПРИ ИСПОЛЬЗОВАНИИ ЯЗЫКА ПИТОН

Объектом исследования являются методы и инструменты отладки и профилирования программ при использовании языка Python.

Цель работы - анализ методов и инструментов отладки и профилирования, выбор наиболее эффективных из них и их применение для отладки приложения.

В процессе работы проводились изучение и сравнение различных инструментов отладки (pdb, PyCharm, VS Code, Jupyter Notebook), профилирования (cProfile, py-spy, line\_profiler, memory\_profiler) и линтеров.

В результате работы был проведен анализ инструментов отладки и профилирования, что позволит оптимизировать процесс разработки.

Область применения результатов - разработка программного обеспечения на языке Python, анализ и оптимизация Python-программ.

# СОДЕРЖАНИЕ

[РЕФЕРАТ 8](#_Toc197763086)

[СОДЕРЖАНИЕ 9](#_Toc197763087)

[ТЕРМИНЫ И ОПРЕДЕЛЕНИЯ 10](#_Toc197763088)

[ПЕРЕЧЕНЬ СОКРАЩЕНИЙ И ОБОЗНАЧЕНИЙ 12](#_Toc197763089)

[Основная часть 14](#_Toc197763090)

[1. Научный поиск 14](#_Toc197763091)

[2. Ознакомление 21](#_Toc197763092)

[3. Тестирование и оценка результатов выполнения работы. 71](#_Toc197763093)

[Заключение 72](#_Toc197763094)

[ПРИЛОЖЕНИЯ 73](#_Toc197763095)

# ТЕРМИНЫ И ОПРЕДЕЛЕНИЯ

|  |  |
| --- | --- |
| **Термин** | **Определение** |
| Отладка | Процесс поиска и устранения ошибок (багов) в программе. |
| Профилирование | Анализ работы программы для выявления узких мест и оптимизации производительности. |
| Линтер | Инструмент статического анализа кода для выявления стилистических и логических ошибок. |
| Точка остановки (брейкпоинт) | Место в коде, где выполнение программы приостанавливается для анализа состояния. |
| Статистический профайлер | Инструмент, собирающий данные о выполнении программы через периодические снимки. |
| Событийный профайлер | Инструмент, отслеживающий каждое событие (например, вызов функции) для точного анализа. |
| Постмортем-отладка | Анализ программы после возникновения ошибки для определения её причины. |
| Flame Graph | Визуализация стека вызовов, показывающая, где программа тратит больше времени. |
| Тепловая карта | Графическое представление активности кода, где интенсивность цвета указывает на время выполнения. |

# ПЕРЕЧЕНЬ СОКРАЩЕНИЙ И ОБОЗНАЧЕНИЙ

В настоящем отчёте о НИР применяют следующие сокращения и обозначения.

|  |  |
| --- | --- |
| **Сокращение** | **Расшифровка** |
| IDE | Integrated Development Environment (Интегрированная среда разработки) |
| pdb | Python Debugger (Отладчик Python) |
| PEP 8 | Python Enhancement Proposal 8 (Предложение по улучшению Python 8, стандарт стиля кода) |
| JIT | Just-In-Time (Компиляция во время выполнения) |
| CI/CD | Continuous Integration/Continuous Deployment (Непрерывная интеграция/непрерывное развертывание) |
| API | Application Programming Interface (Программный интерфейс приложения) |
| CAPI | Python/C API (Интерфейс программирования приложений для интеграции Python и C) |
| CLI | Command Line Interface (Интерфейс командной строки) |
| PID | Process Identifier (Идентификатор процесса) |

# Основная часть

## 1. Научный поиск

В рамках данной работы были изучены и проанализированы современные методы и инструменты, используемые для отладки, профилирования и статического анализа кода программ, написанных на языке Python. Основное внимание уделялось инструментам, которые помогают разработчикам выявлять ошибки, оптимизировать производительность и улучшать качество кода. Рассмотренные инструменты разделены на три категории: отладчики, профайлеры и линтеры.

#### Отладчики

Отладчик - инструмент, который помогает отлаживать (искать ошибки) программу.

Отладка - это процесс, при котором разработчик ищет и устраняет ошибки (баги) в программе.  
Даже опытные программисты сталкиваются с ошибками, и эффективные инструменты отладки помогают быстрее находить их источник.  
В Python доступен широкий спектр методов и инструментов, от простых, таких как вывод сообщений в консоль, до продвинутых отладчиков в интегрированных средах разработки (IDE).

* pdb (Python Debugger) - встроенный отладчик Python, предоставляющий интерактивный интерфейс для пошагового выполнения кода. Поддерживает установку брейкпоинтов через pdb.set\_trace() или breakpoint(), просмотр стека вызовов и значений переменных. Прост в использовании, но требует ручного вмешательства в код.
* print - встроенная функция для вывода данных в консоль. Часто используется для быстрой отладки благодаря простоте, однако не подходит для сложных сценариев из-за отсутствия интерактивности.
* logging - модуль стандартной библиотеки для структурированного логирования. Позволяет записывать события с различными уровнями важности (DEBUG, INFO и т.д.), обеспечивая потокобезопасность и гибкость.
* pytest - фреймворк для тестирования, который может использоваться для отладки через автоматическое обнаружение тестов и анализ ошибок. Поддерживает фикстуры и параметризацию.
* unittest - встроенный фреймворк для модульного тестирования. Обеспечивает автоматизацию тестов и отладку через проверку утверждений.
* PyCharm - интегрированная среда разработки (IDE) с мощным отладчиком. Поддерживает условные точки остановки, удаленную отладку и интерактивную консоль.
* VS Code - легковесный редактор с расширением для Python, включающий отладчик с поддержкой точек остановки и логпойнтов.
* Jupyter Notebook - интерактивная среда, популярная для анализа данных. Поддерживает отладку через магические команды IPython (%debug) и интеграцию с pdb.

#### Профайлеры

Профилирование - это процесс анализа работы программы для выявления узких мест и возможностей для оптимизации.  
Обычно профилирование применяют, когда программа работает медленнее, чем ожидалось, или потребляет слишком много ресурсов, таких как процессорное время или память.

Для чего нужно профилирование? Оно позволяет собрать данные о поведении программы, чтобы понять, какие её части требуют улучшения.  
Среди характеристик, которые можно измерить:

* Время выполнения отдельных строк кода.
* Частота вызовов функций и их продолжительность.
* Иерархия вызовов (какие функции вызывают другие).
* Участки кода, где программа тратит больше всего времени («hot spots», «горячие точки»).
* Использование процессора, памяти или других системных ресурсов (например, доступ к файлам).

Профайлеры делятся на два основных типа: **статистические** (statistical) и **событийные** (deterministic, event-based).  
Каждый из них имеет свои особенности, преимущества и ограничения.

##### Статистические профайлеры

Статистические профайлеры собирают данные, периодически «заглядывая» в программу через небольшие интервалы времени.  
Они фиксируют, какая инструкция выполняется в данный момент, и сохраняют эту информацию (так называемые «сэмплы»).  
Это похоже на моментальные снимки, которые затем анализируются, чтобы выявить наиболее активные участки кода.

###### Пример работы

Представьте, что профайлер делает снимок каждую миллисекунду.  
Если какая-то функция выполняется часто или долго, она будет чаще попадать в эти снимки, что укажет на её «вес» в программе.

###### Проблемы и ограничения

* Недостаточная точность. Если интервал между снимками слишком большой, профайлер может пропустить короткие, но частые вызовы функций. Например, функция, которая выполняется быстро, но вызывается тысячи раз, может остаться незамеченной.
* Сложность оценки времени. Статистический профайлер не измеряет время выполнения напрямую, поэтому трудно понять, вызывается ли функция часто или просто работает долго.
* Долгий сбор данных. Для получения достоверной картины нужно много снимков, что требует времени.
* Ограниченный инструментарий. Инструментов для анализа данных статистических профайлеров меньше, чем для событийных.

###### Преимущества

Несмотря на недостатки, статистические профайлеры отлично справляются с поиском «горячих точек» - мест, где программа тратит больше всего ресурсов.  
Их главное достоинство - минимальное влияние на работу программы.  
Это делает их подходящими даже для использования в реальных условиях (например, на серверах в продакшене).  
В Python такие профайлеры могут собирать полные стектрейсы, что даёт более глубокое понимание происходящего в коде.

##### Событийные профайлеры

Событийные профайлеры работают иначе: они отслеживают каждое значимое событие в программе, такое как вызов функции, её завершение или возникновение исключения.  
Они записывают, сколько времени прошло между этими событиями, и сохраняют данные для анализа.

###### Пример работы

Если программа вызывает функцию calculate(), событийный профайлер зафиксирует момент входа в функцию, момент выхода и точное время выполнения.  
Также он учтёт, сколько раз функция была вызвана.

###### Проблемы и ограничения

* Сильное влияние на программу. Так как профайлер вмешивается в каждый шаг выполнения, программа может замедляться в разы. Это делает событийные профайлеры почти непригодными для продакшена.
* Изменение поведения. В редких случаях вмешательство профайлера может даже повлиять на логику работы программы.

###### Преимущества

Событийные профайлеры дают полную картину работы программы: точное время выполнения каждой функции, количество вызовов, граф зависимостей между функциями.  
Это помогает не только найти узкие места, но и выявить проблемы в архитектуре или алгоритмах.  
Удобные интерфейсы для анализа и обилие инструментов делают их популярными среди разработчиков.

* py-spy - статистический профайлер, написанный на Rust. Имеет низкий накладной расход, подходит для продакшена, генерирует flame graphs для визуализации.
* cProfile - встроенный событийный профайлер с высокой точностью. Подходит для больших приложений, но требует дополнительных инструментов для интерпретации результатов.
* timeit - модуль для измерения времени выполнения небольших фрагментов кода. Идеален для бенчмаркинга, но ограничен в детальном анализе.
* line\_profiler - инструмент для построчного профилирования. Показывает время выполнения каждой строки, что помогает точно локализовать узкие места.
* memory\_profiler - инструмент для анализа потребления памяти. Выявляет утечки и избыточное использование ресурсов.
* pyheat - инструмент для визуализации профилирования через тепловые карты. Полезен для интуитивного анализа, но менее поддерживаем.

#### Линтеры

Линтеры - это инструменты статического анализа кода, которые помогают разработчикам находить потенциальные ошибки, стилистические проблемы и несоответствия стандартам ещё до запуска программы.  
Они сканируют исходный код, выявляя такие проблемы, как неправильное форматирование, неиспользуемые переменные, потенциальные баги или отклонения от принятых в команде правил.  
Линтеры особенно популярны в языках программирования, таких как Python, где свобода синтаксиса может привести к неоднородному или небезопасному коду.

Основная задача линтеров - повысить качество и читаемость кода, а также снизить вероятность ошибок на этапе выполнения.  
Они интегрируются в редакторы кода (например, VS Code или PyCharm) или процессы непрерывной интеграции (CI/CD), предоставляя мгновенную обратную связь.  
Например, линтер может указать на отсутствие пробела после запятой или предупредить о сложной конструкции, которая может быть трудно читаемой.

Линтеры также экономят время, позволяя исправлять мелкие недочёты автоматически или до начала отладки.  
Популярные линтеры для Python, такие как flake8, ruff или mypy, предлагают гибкую настройку под нужды проекта, поддерживая стандарты, такие как PEP 8.  
Использование линтеров помогает не только улучшить код, но и воспитать у разработчиков привычку писать аккуратный и надёжный код, что особенно важно в крупных и долгосрочных проектах.

* flake8 - инструмент для проверки стиля и качества кода. Объединяет PEP 8, pyflakes и mccabe, предлагая гибкую настройку.
* ruff - высокопроизводительный линтер и форматтер, написанный на Rust. Быстрее традиционных инструментов, поддерживает автоматическое исправление ошибок.
* mypy - статический проверщик типов. Использует аннотации типов для выявления ошибок типизации на этапе разработки.

#### Дополнительные технологии

В дополнение к основным инструментам отладки и профилирования, существуют технологии, которые используются реже из-за своей специфичности и редкой необходимости.  
Эти технологии открывают новые возможности для оптимизации и анализа производительности программ.  
Они особенно полезны в высокопроизводительных вычислениях, где стандартные инструменты могут быть недостаточно эффективны, или в проектах, требующих глубокого взаимодействия с низкоуровневыми языками.

* PyPy - альтернативная реализация Python с JIT-компиляцией для повышения производительности.
* Cython - язык для создания C-расширений, ускоряющий выполнение Python-кода.
* Интеграция с C - использование C для оптимизации производительности через C-расширения.

## 2. Ознакомление

#### Отладчики

##### pdb (Python Debugger)

###### Описание

pdb - встроенный интерактивный отладчик Python, который позволяет разработчикам пошагово выполнять код, устанавливать точки остановки (брейкпоинты) и анализировать значения переменных для выявления ошибок.

###### Основные возможности

* Установка брейкпоинтов для приостановки выполнения кода.
* Пошаговое выполнение.
* Просмотр значений переменных и стека вызовов.
* Интерактивная консоль для выполнения команд во время отладки.
* Поддержка постмортем (посмертной) отладки.
* Автоматический перезапуск с сохранением состояния (точек остановки).
* Поддержка табуляции для автодополнения команд.
* Возможность создания псевдонимов для команд.
* Удобные переменные, такие как $\_frame, $\_retval, $\_exception.

###### Установка

pdb входит в стандартную библиотеку Python, поэтому установка не требуется.

###### Использование

Достаточно вставить import pdb; pdb.set\_trace() или breakpoint() в код, чтобы начать отладку, или запустить скрипт с помощью python -m pdb script.py.

Навигация по коду:

* w (where) - выводит информаию о позиции в которой сейчас находитесь.
* s (step) - перейти во внутрь вызова объекта, если это возможно, иначе перейти к следующей строке кода.
* n (next) - перейти к следующей строке кода.
* unt (until) - перейти к следующей строке кода, но гарантировано чтобы номер строки был больше чем текущий.
* r (return) - завершить ("выйти из") текущую функцию.
* u (up) - подняться на один стек-фрейм вверх.
* d (down) - опуститься на один стек-фрейм вниз.
* j (jump) <line> - перепрыгнуть на указанную строку кода, не выполняя код находящийся между текущей позицией и указанной. Исключение составляют циклы for и код в блоке finally (т.к. должен быть обязательно выполнен). Также, вы можете перепрыгивать только внутри текущего фрейма (т.е. нижнего фрейма).

Команды:

* h (help) [<command>] - помощь по командам.
* p <var or expression> - вывести значение переменной по имени или значение выражения.

###### Примеры

Листинг 1 - pdb\_breakpoints.py

def too\_much\_conditions(a: int, b: int) -> int:  
 breakpoint()  
 if b == 0:  
 breakpoint()  
 return 1  
 if b == 1:  
 breakpoint()  
 return a \*\* 1  
 if b == 2:  
 breakpoint()  
 return a \*\* 2  
 if b == 3:  
 breakpoint()  
 return a \*\* 3  
 if b == 4:  
 breakpoint()  
 return a \*\* 4  
 breakpoint()  
 return a \*\* 5  
  
for b in range(0, 6):  
 too\_much\_conditions(2, b)

Листинг 2 - pdb\_pm.py

def zero\_divison(a: int) -> int:  
 return a // 0  
  
zero\_divison(1)

###### Плюсы

* Не требует дополнительной установки.
* Прост в использовании для базовой отладки.
* Поддерживает посмертную отладку.
* Автоматическое сохранение состояния при перезапуске.
* Поддержка табуляции и псевдонимов для удобства.

###### Минусы

* Ограниченная документация по классу Pdb.
* Необходимо проставлять breakpoint в коде.

###### Ссылки

* [Официальная документация pdb](https://docs.python.org/3/library/pdb.html)
* [Документация на docs-python.ru](https://docs-python.ru/standart-library/modul-pdb-python)

##### print

###### Описание

print - встроенная функция Python, которая выводит объекты на стандартный поток вывода, разделенные пробелами и заканчиваясь символом новой строки. Она часто используется для быстрого вывода информации во время отладки.

###### Основные возможности

* Вывод объектов на стандартный вывод.
* Возможность указания разделителя (sep), окончания (end), файла (file) и флага очистки буфера (flush).

###### Установка

print входит в стандартную библиотеку Python, поэтому установка не требуется.

###### Использование

print(\*objects, sep=' ', end='\n', file=sys.stdout, flush=False),  
где objects - любые объекты;  
sep - разделитель между объектами (по умолчанию - пробел);  
end - символ, который будет после всего вывода (по умолчанию - символ переноса строки);  
file - место вывода, может быть любой объект, поддерживающий .write() (по умолчанию - стандартный поток вывода);  
flush - нужно ли принудительно очищать поток, чтобы буферизированный вывод благополучно добрался до точки назначения (по умолчанию - False).

Листинг 3 - Использование print

print("tralalelo", "tralala")  
print("bombordiro", "crocodile", sep="---")  
  
print("brr brr", end="+")  
print("patapim")  
  
with open("temp.txt", "w") as f:  
 print("frigo camelo",  
 sep=" | ", end=" = buffo fardello", file=f, flush=True)

###### Плюсы

* Простота использования.
* Удобно для быстрого вывода информации во время отладки.

###### Минусы

* Не предназначен для структурированного логирования.
* Может загромождать вывод в продакшене.

###### Ссылки

* [Официальная документация print](https://docs.python.org/3/library/functions.html#print)
* [Документация на pythonru.com](https://pythonru.com/osnovy/python-print)

##### logging

###### Описание

logging - модуль Python, который предоставляет гибкую систему ведения журнала событий для приложений и библиотек, позволяя интегрировать сообщения из различных модулей в единый лог.

###### Основные возможности

* Иерархическая структура логгеров.
* Настраиваемые уровни логирования (DEBUG, INFO, WARNING, ERROR, CRITICAL).
* Компоненты: логгеры, обработчики, фильтры, форматтеры.
* Потокобезопасность.
* Интеграция с модулем warnings.

###### Установка

logging входит в стандартную библиотеку Python, поэтому установка не требуется.

###### Использование

Создать логгер (logging.getLogger(\_\_name\_\_)), настроить его (logging.basicConfig()), использовать методы debug(), info(), warning(), error(), critical() для логгирования.

Листинг 4 - Использование logging

import logging  
  
logging.basicConfig(level=logging.DEBUG)  
logger = logging.getLogger("logger\_name")  
logger.debug("Это дебаг лог")  
logger.info("Это инфо лог")  
logger.warning("Это варнинг лог")  
logger.error("Это лог про ошибку")  
logger.critical("Этот лог про критическую ошибку")

###### Плюсы

* Стандартизированное логирование по всему приложению.
* Иерархическая структура для детального контроля.
* Гибкая конфигурация через уровни, обработчики, фильтры, форматтеры.
* Потокобезопасность.
* Интеграция с модулем warnings.

###### Минусы

* Возможны дублированные сообщения, если обработчики прикреплены к нескольким логгерам.
* Требует тщательной конфигурации для сложных приложений.
* Не подходит для асинхронных обработчиков сигналов из-за проблем с блокировкой потоков.

###### Ссылки

* [Официальная документация logging](https://docs.python.org/3/library/logging.html)
* [Документация на docs-python.ru](https://docs-python.ru/standart-library/paket-logging-python)

##### pytest

###### Описание

pytest - популярный фреймворк для тестирования Python, который упрощает написание и выполнение тестов, поддерживая как простые, так и сложные функциональные тесты. Его возможности делают его полезным для отладки через тестирование.

###### Основные возможности

* Автоматическое обнаружение тестов в файлах, названных test\_\*.py или \*\_test.py.
* Поддержка детального анализа утверждений для лучших сообщений об ошибках.
* Управление исключениями с помощью pytest.raises.
* Группировка тестов в классы для организации и обмена фикстурами.
* Встроенные фикстуры, такие как tmp\_path для временных директорий.
* Параметризация фикстур и функций тестов.
* Поддержка пропуска и ожидания сбоев тестов.
* Многочисленные плагины для расширения функциональности.

###### Установка

Требуется Python 3.8+. Установка с помощью pip install pytest. Проверить версию можно с помощью pytest --version.

###### Использование

Запуск тестов происходит с помощью команды pytest. В файлах, начинающихся с test\_, или заканчивающихся на \_test.py, фреймворк ищет функции, начинающиеся с test\_, и запускает их как тесты.

В файлах conftest.py можно указать фикстуры, которые будут применены при запуске тестов.

Листинг 5 - Использование pytest

import pytest  
  
@pytest.fixture  
def dictionary():  
 return {}  
  
def test\_dict(dictionary):  
 assert len(dictionary) == len(dictionary.keys()) == 0  
 dictionary["key"] = "value"  
 assert dictionary["key"] == "value"  
 assert len(dictionary) == 1  
 del dictionary["key"]  
 with pytest.raises(KeyError):  
 dictionary["key"]

###### Плюсы

* Детальный анализ утверждений для облегчения отладки.
* Легкое обнаружение и организация тестов.
* Поддержка группировки тестов в классы.
* Встроенные фикстуры для функционального тестирования.
* Режим тихого вывода для краткого отчета.

###### Минусы

* Атрибуты класса общаются между тестами, что может привести к проблемам изоляции, если не обращаться внимательно.
* Нет специального Mock класса как в unittest.

###### Ссылки

* [Официальная документация pytest](https://docs.pytest.org/en/stable/)
* [Документация на docs-python.ru](https://docs-python.ru/packages/frejmvork-pytest)

##### unittest

###### Описание

unittest - встроенный фреймворк для модульного тестирования в Python, вдохновленный JUnit, который поддерживает автоматизацию тестов, общее кодовое обеспечение для настройки и завершения, агрегацию тестов и независимость от фреймворков отчетности.

###### Основные возможности

* Автоматизация тестов.
* Общее кодовое обеспечение для настройки и завершения.
* Агрегация тестов.
* Независимость от фреймворков отчетности.
* Концепции: тестовый фикстура, тестовый случай, набор тестов, запускатель тестов.
* Возможность пропуска тестов и ожидания сбоев (с версии 3.1).
* Подтесты для детального тестирования итераций (с версии 3.4).
* Автоматическое обнаружение тестов (с версии 3.2).
* Изолированное асинхронное тестирование (с версии 3.8).
* Отображение длительности тестов (с версии 3.12).

###### Установка

unittest входит в стандартную библиотеку Python, поэтому установка не требуется.

###### Использование

Создаются классы тестов, наследующие от unittest.TestCase. Определяются методы, начинающиеся с test\_. Дальше используются методы-утверждения assertEqual, assertTrue, assertRaises для проверки поведения тестируемого объекта. Тесты запускаются с помощью unittest.main() или через командную строку с python -m unittest.

Листинг 6 - Использование unittest

import unittest  
  
class TestDict(unittest.TestCase):  
 def setUp(self):  
 self.dictionary = {}  
  
 def test\_dict(self):  
 self.assertEqual(len(self.dictionary), 0)  
 self.dictionary["key"] = "value"  
 self.assertEqual(self.dictionary["key"], "value")  
 self.assertEqual(len(self.dictionary), 1)  
 del self.dictionary["key"]  
 with self.assertRaises(KeyError):  
 self.dictionary["key"]  
  
if \_\_name\_\_ == "\_\_main\_\_":  
 unittest.main()

###### Плюсы

* Поддерживает автоматизацию тестов.
* Обеспечивает повторное использование кода для настройки и завершения.
* Позволяет агрегировать тесты.
* Независим от фреймворков отчетности.
* Предоставляет обширный набор методов утверждения.
* Поддерживает пропуск тестов и ожидание сбоев.
* Подтесты для детального тестирования итераций.
* Автоматическое обнаружение тестов.
* Поддержка командной строки с различными опциями.
* Фикстуры на уровне класса и модуля.
* Обработка сигналов для контроля-C (с версии 3.2).

###### Минусы

* Общие фикстуры могут нарушать изоляцию тестов.
* Не идеален для параллелизации.
* Возможны несколько вызовов фикстур с рандомизированным порядком.
* Сложность в настройке пользовательской загрузки тестов.
* Возвращение значений из методов тестов устарело (с версии 3.11).

###### Ссылки

* [Официальная документация unittest](https://docs.python.org/3/library/unittest.html)
* [Документация на pythonworld.ru](https://pythonworld.ru/moduli/modul-unittest.html)

##### Pycharm

###### Описание

PyCharm - профессиональная IDE от JetBrains, созданная для Python-разработки. Её отладчик поддерживает точки остановки, пошаговое выполнение, анализ переменных и условные остановки.

###### Основные возможности

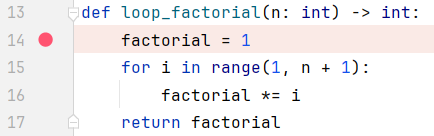
* Установка брейкпоинтов для приостановки выполнения кода.
* Пошаговое выполнение (step over, step into, step out).
* Просмотр стека вызова функций.
* Просмотр и изменение значений переменных в реальном времени.
* Условные точки остановки для остановки при выполнении условий.
* Точки остановки на исключениях для анализа ошибок.
* Интерактивная консоль отладки для выполнения кода в контексте.
* Поддержка удаленной отладки.

###### Установка

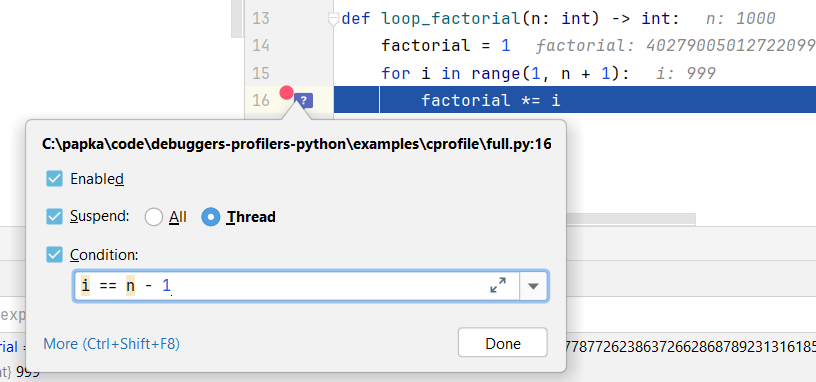
1. Скачать PyCharm с официального сайта [JetBrains](https://www.jetbrains.com/pycharm/).
2. Установите программу, следуя инструкциям установщика.

###### Использование

1. Открыть PyCharm, создать новый проект или открыть существующий.
2. Создать или открыть Python-файл в редакторе.
3. Установить точку остановки, щелкнув ЛКМ на левой границе редактора рядом с номером строки. *Опционально* добавить условие на брейкпоинт.

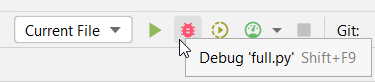


*Рисунок 1 - Установка брейкпоинта в PyCharm*



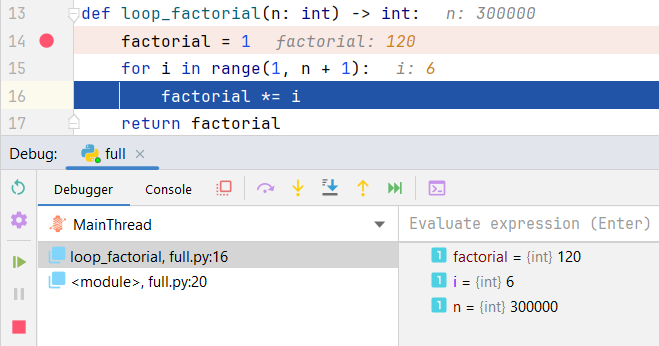
*Рисунок 2 - Установка условия на брейкпоинт в PyCharm*

1. Нажать кнопку "Debug" в верхней панели инструментов или использовать сочетание клавиш Shift+F9 для запуска отладки.



*Рисунок 3 - Запуск отладки в PyCharm*

1. Использовать панель отладки для пошагового выполнения, просмотра переменных и управления выполнением.

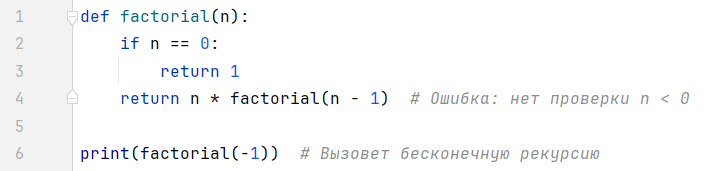


*Рисунок 4 - Панель отладки в PyCharm*

###### Пример

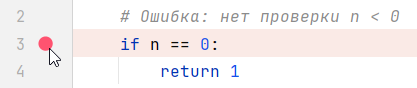
Рассмотрим функцию вычисления факториала, которая не проверяет отрицательные входные значения, что приводит к бесконечной рекурсии и RecursionError.

**Шаг 1.** Создадим файл factorial.py в PyCharm с функцией вычисления факториала:



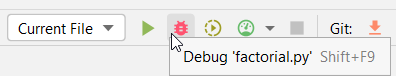
*Рисунок 5 - Код функции вычисления факториала*

**Шаг 2.** Установим точку остановки на строке if n == 0:, щелкнув на левой границе редактора.



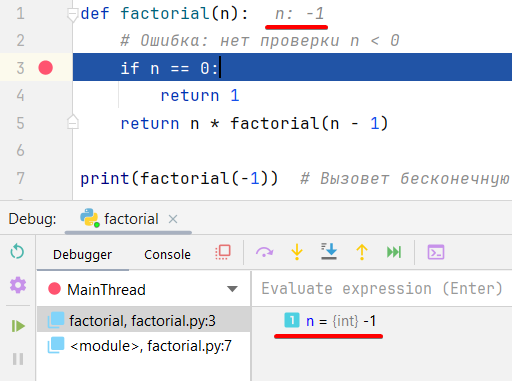
*Рисунок 6 - Установка брейкпоинта на строке с условием*

**Шаг 3.** Нажмём кнопку "Debug" (иконка жука) для запуска отладки.



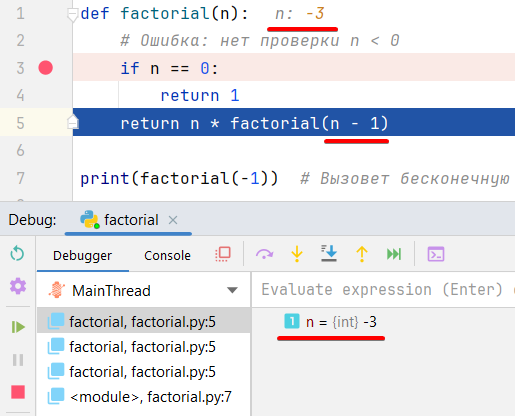
*Рисунок 7 - Запуск отладки factorial.py*

**Шаг 4.** Выполнение остановится на условии. В панели "Variables" проверим значение n. Увидим, что n = -1, что указывает на проблему.

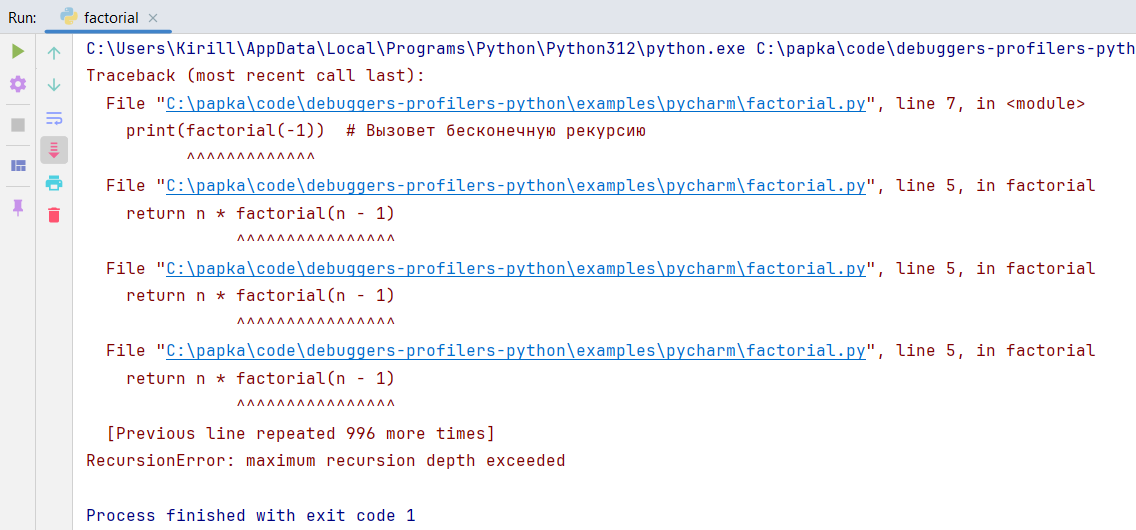


*Рисунок 8 - Просмотр значения переменной n*

**Шаг 5.** Используем кнопку "Step Into" (F7) для перехода в рекурсивный вызов. Обратим внимание, что n становится -2, -3 и так далее, указывая на бесконечную рекурсию.

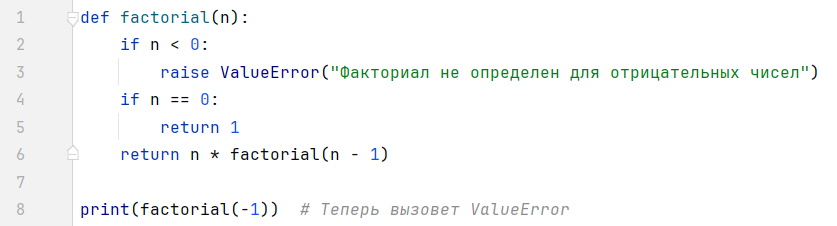


*Рисунок 9 - Переменная n уменьшается*



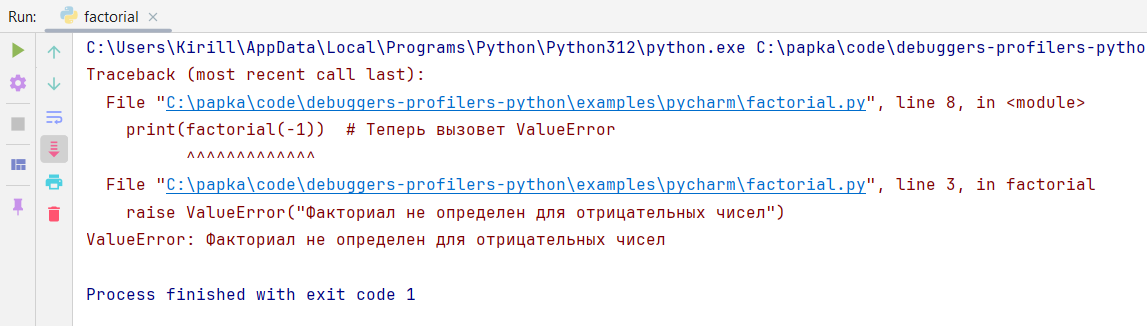
*Рисунок 10 - Ошибка вечной рекурсии*

**Шаг 6.** Исправим код, добавив проверку отрицательных чисел.



*Рисунок 11 - Проверка на отрицательные числа*

**Шаг 7.** Перезапустим программу и убедимся, что код вызовет ValueError с понятным сообщением.



*Рисунок 12 - Понятная ошибка при передаче отрицательного числа*

###### Плюсы

* Мощный отладчик с поддержкой условных точек остановки и удаленной отладки.
* Интеграция с автодополнением, рефакторингом и Git.
* Регулярные обновления и профессиональная поддержка.

###### Минусы

* Высокое потребление ресурсов.
* Сложный интерфейс для новичков.
* Некоторые функции доступны только в платной версии Professional.

###### Ссылки

* [Официальный сайт PyCharm](https://www.jetbrains.com/pycharm/)
* [Документация PyCharm](https://www.jetbrains.com/help/pycharm/)

##### VS Code

###### Описание

Visual Studio Code (VS Code) - легковесный редактор кода от Microsoft, который с расширением Python становится полноценной IDE. Его отладчик прост в использовании, поддерживает точки остановки, просмотр стека вызовов, переменных и интерактивную консоль.

###### Основные возможности

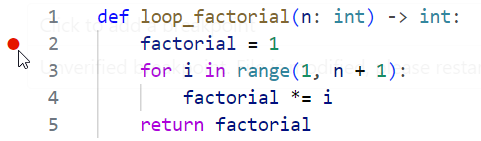
* Установка брейкпоинтов для приостановки выполнения.
* Условные точки остановки для остановки при определенных условиях.
* Пошаговое выполнение кода (step over, step into, step out).
* Панель переменных для просмотра и изменения значений.
* Журналирование значений (logpoints) без остановки выполнения.
* Просмотр стека вызовов и поддержка многопоточной отладки.
* Интерактивная консоль для выполнения команд во время отладки.

###### Установка

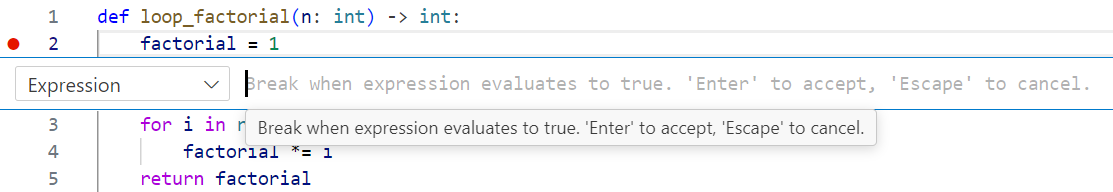
1. Скачайте VS Code с [официального сайта](https://code.visualstudio.com/download).
2. Установите редактор, следуя инструкциям.
3. Откройте VS Code, перейдите в раздел "Extensions" (Ctrl+Shift+X), найдите расширение "Python" от Microsoft и установите его.

###### Использование

1. Открыть VS Code, создать новый проект или открыть существующий.
2. Создать или открыть Python-файл в редакторе.
3. Установить точку остановки, щелкнув ЛКМ на левом поле рядом с номером строки. *Опционально* добавить условие на брейкпоинт.

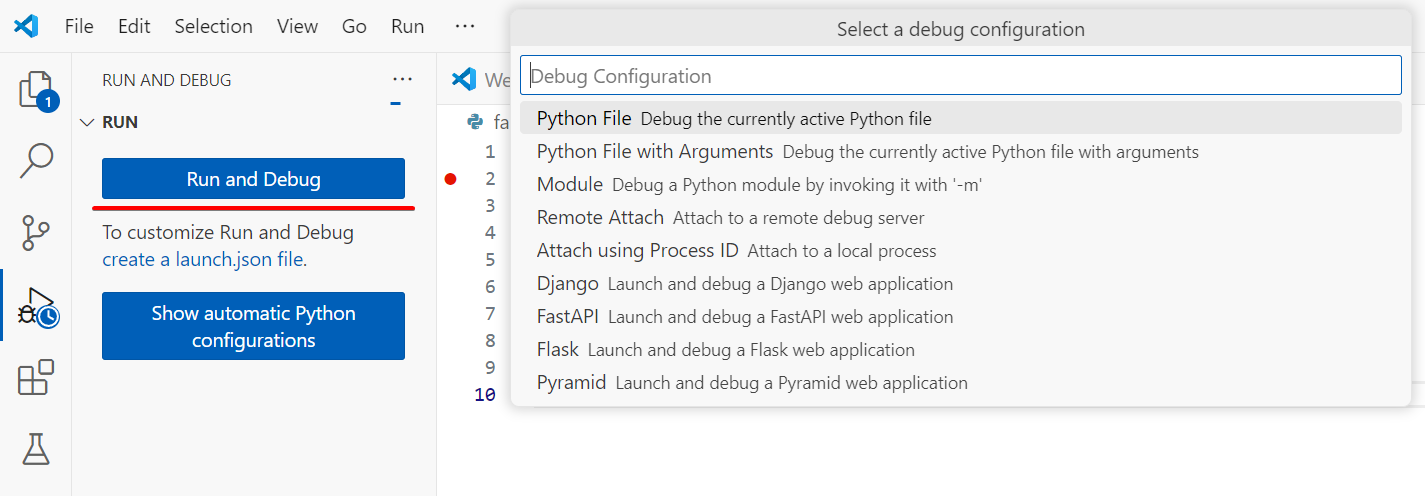


*Рисунок 13 - Установка брейпоинта в VS Code*



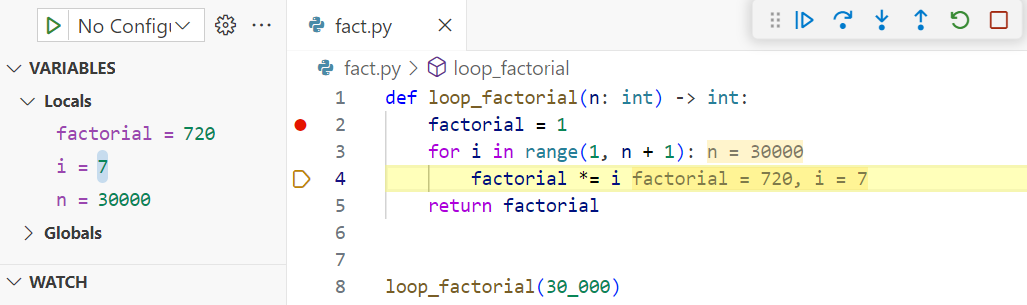
*Рисунок 14 - Установка условия на брейкпоинт в VS Code*

1. Перейти в панель "Run and Debug" (Ctrl+Shift+D), выбрать конфигурацию "Python: Current File" и нажать "Start Debugging" (F5)



*Рисунок 15 - Запуск отладки в VS Code*

1. Использовать панель отладки для пошагового выполнения, просмотра переменных и управления выполнением.

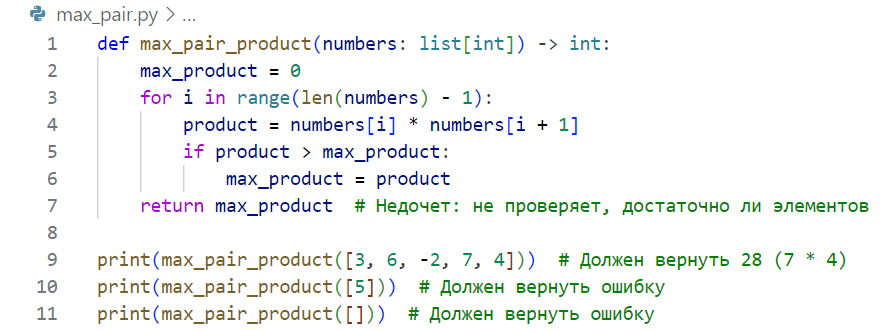


*Рисунок 16 - Панель отладки в VS Code*

###### Пример

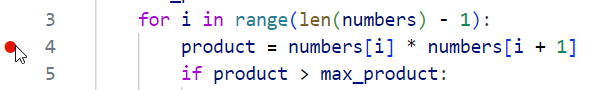
Рассмотрим функцию, которая вычисляет максимальное произведение двух соседних элементов в списке чисел. Ошибка заключается в том, что функция предполагает, что список всегда содержит как минимум два элемента, что приводит к некорректному результату (например, возвращает 0 для списка из одного элемента).

**Шаг 1.** Создадим файл max\_pair.py в папке проекта в VS Code с кодом этой функции.



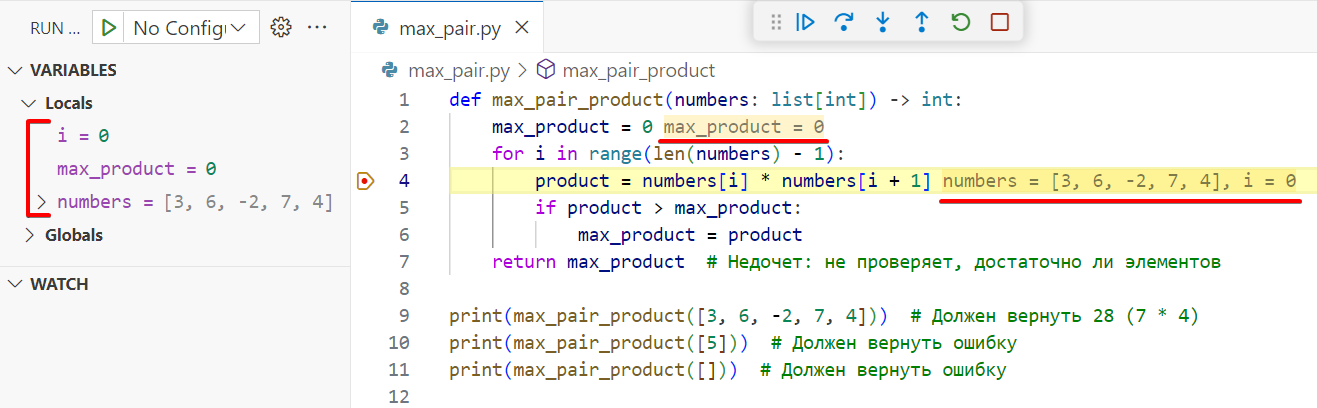
*Рисунок 17 - Код функции поиска наибольшего произведения пары*

**Шаг 2.** Установим точку остановки на строке product = numbers[i] \* numbers[i + 1], щелкнув на левом поле рядом с номером строки.



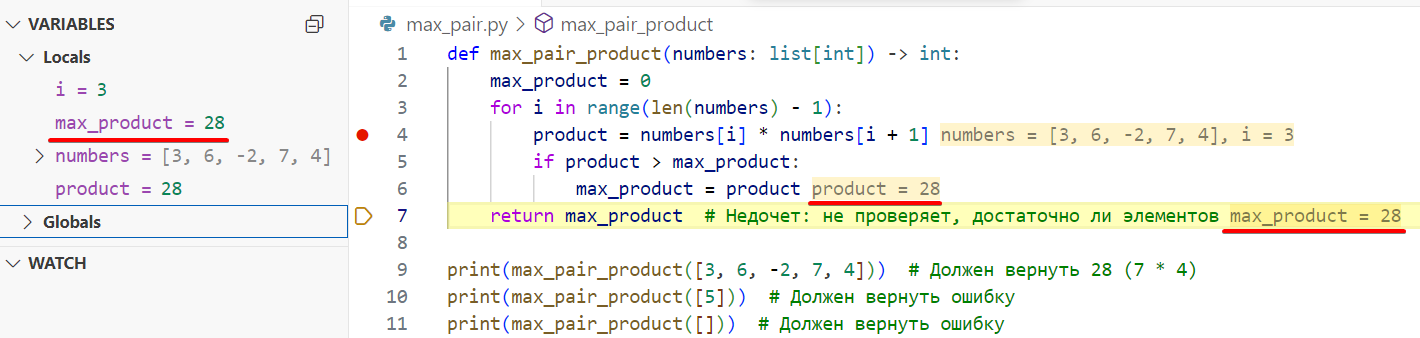
*Рисунок 18 - Установка брейкпоинта на строке с умножением*

**Шаг 3.** Запустим отладку через F5. Выполнение остановится при первом вызове max\_pair\_product([3, 6, -2, 7, 4]).



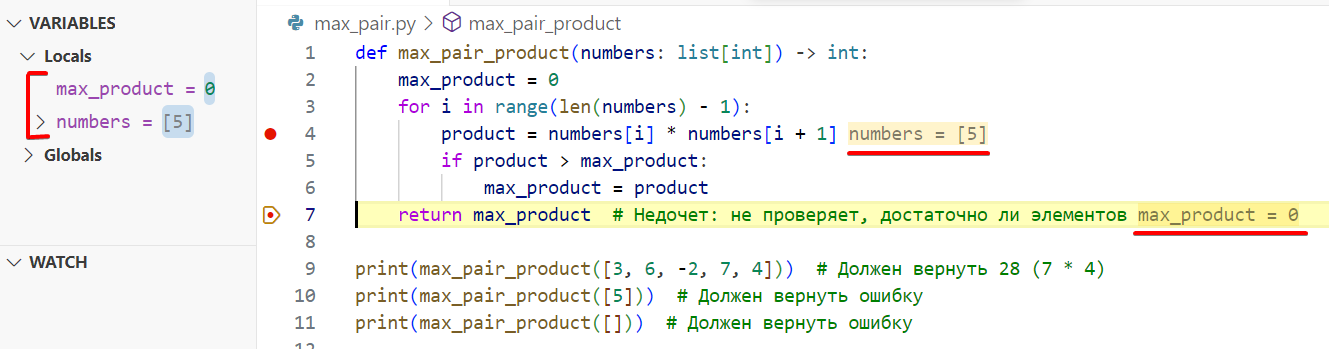
*Рисунок 19 - Остановка в первый раз*

**Шаг 4.** Используем кнопку "Step Over" (F10) для прохождения цикла. Наблюдаем, как product принимает значения: 3 \* 6 = 18, 6 \* -2 = -12, -2 \* 7 = -14, 7 \* 4 = 28. Проверим, что max\_product обновляется до 18, затем до 28. Результат корректен.

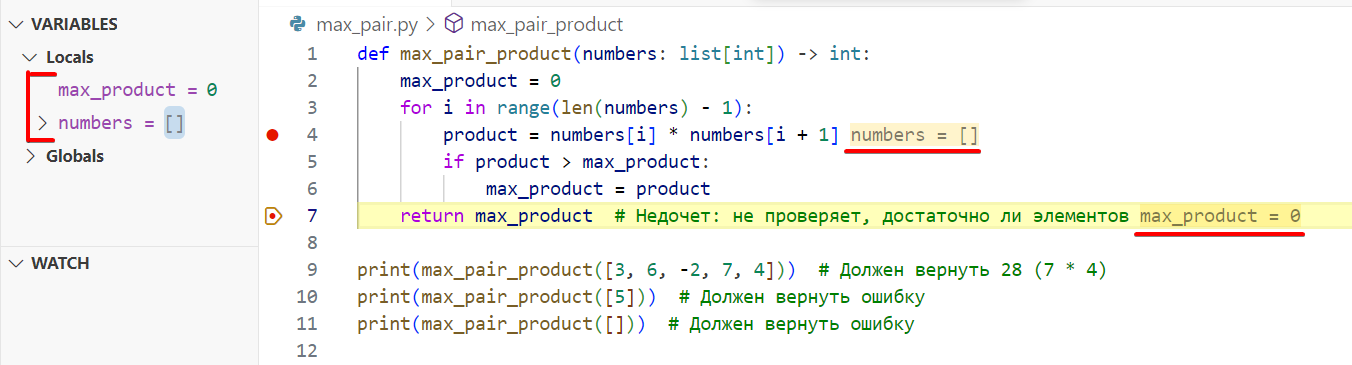


*Рисунок 20 - Первый результат функции*

**Шаг 5.** Продолжим отладку (F5) для второго вызова (max\_pair\_product([5])) и третьего (max\_pair\_product([])). Отладчик не остановится на точке остановки, так как цикл не выполняется (range(0) пуст). В панели "Variables" увидим, что max\_product = 0, что неверно, так как функция должна сигнализировать об ошибке для списка с одним элементом.

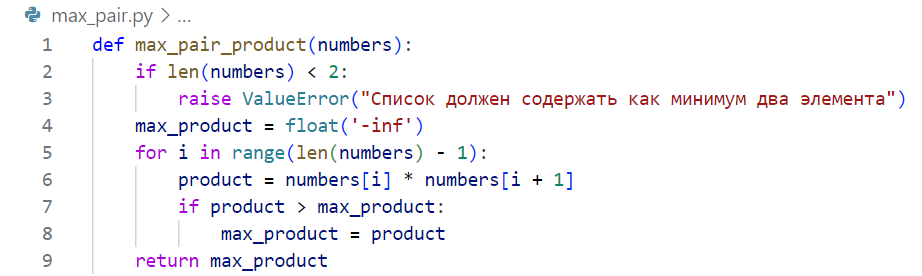


*Рисунок 21 - Второй результат функции*



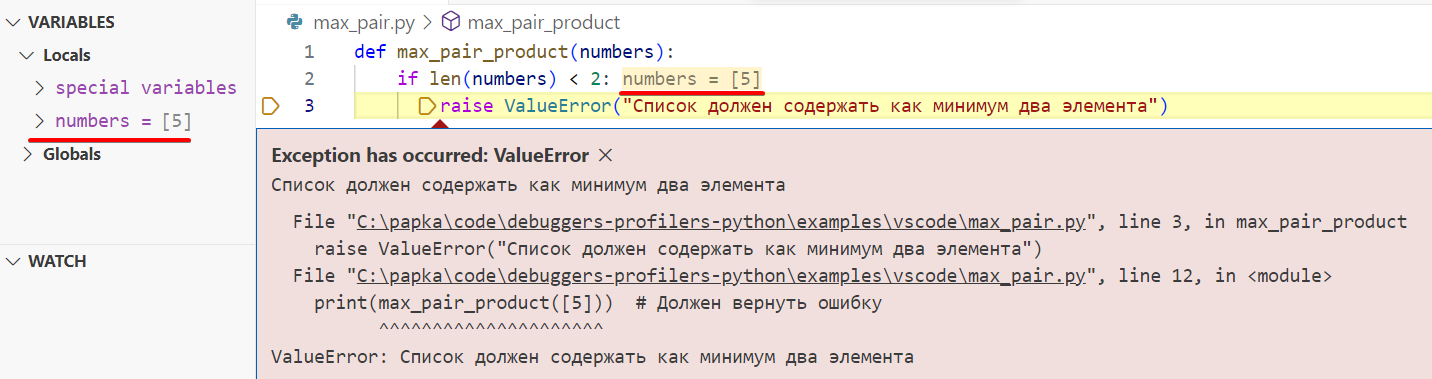
*Рисунок 22 - Третий результат функции*

**Шаг 6.** Исправим код, добавив проверку количества элементов.



*Рисунок 23 - Условие на два минимум два элемента*

**Шаг 7.** Перезапустим программу и увидим ошибку при вызове max\_pair\_product([5]).



*Рисунок 24 - Ошибка при передаче короткого списка*

###### Ссылки

* [Официальный сайт VS Code](https://code.visualstudio.com/)
* [Документация VS Code](https://code.visualstudio.com/docs)

##### Jupyter Notebook

###### Описание

Jupyter Notebook - это интерактивная среда, широко используемая для анализа данных, машинного обучения и научных исследований. Она поддерживает отладку через магические команды IPython, такие как %debug, для анализа ошибок post-mortem, и встроенный модуль pdb для установки точек остановки с помощью pdb.set\_trace().

###### Основные возможности

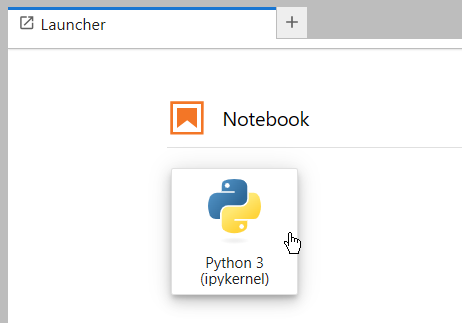
* Магическая команда %debug для запуска отладчика после возникновения исключений.
* Интеграция с pdb для пошаговой отладки через pdb.set\_trace().
* Выполнение кода по ячейкам с немедленным просмотром результатов.
* Интерактивная консоль для проверки переменных и экспериментов.
* Поддержка отображения данных и визуализаций в ячейках

###### Установка

Jupyter Notebook можно установить через [официальный сайт](https://jupyter.org/install), [anaconda](https://anaconda.org/anaconda/jupyter) или [Docker](https://jupyter-docker-stacks.readthedocs.io/en/latest/).

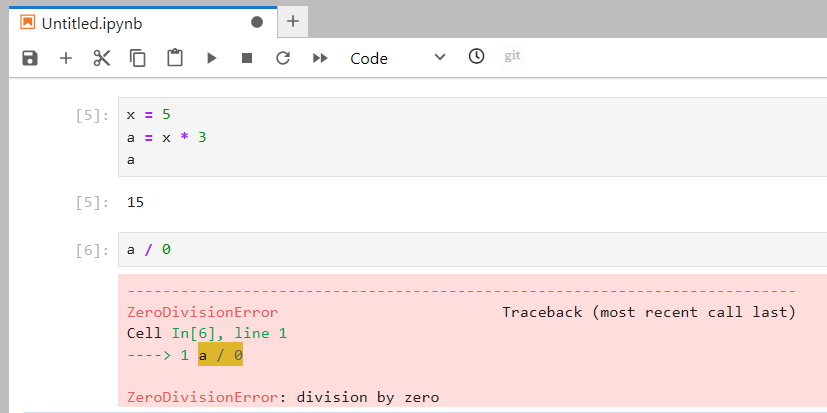
###### Использование через pdb

1. Открыть Jupyter Notebook в браузере и создать новый блокнот.



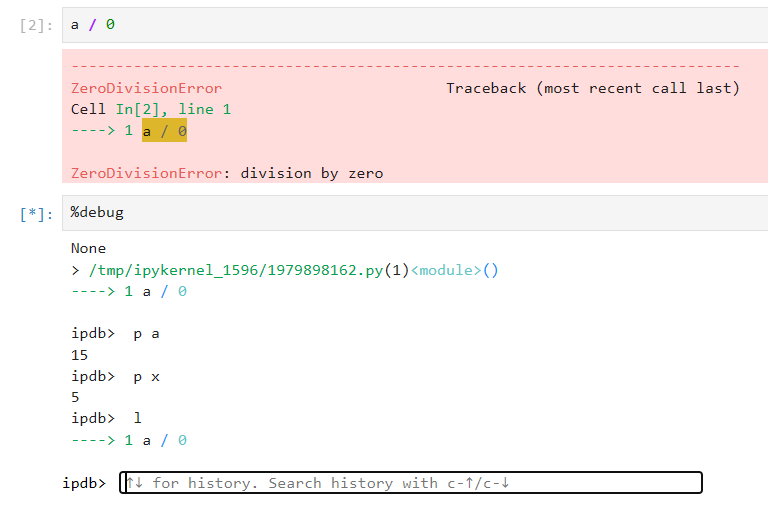
*Рисунок 25 - Создание нового блокнота в Jupyter Notebook*

1. Вставить код в ячейки и выполнить их с помощью Shift+Enter.



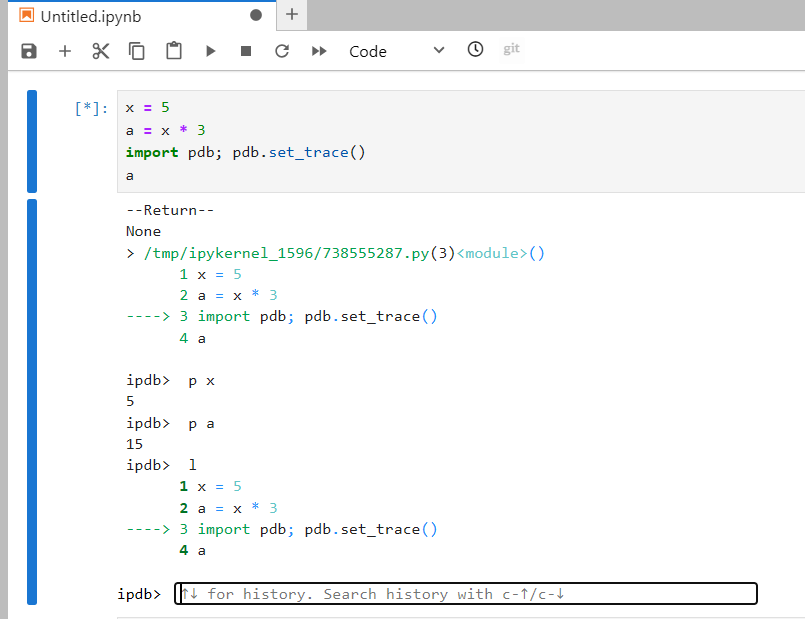
*Рисунок 26 - Запуск кода без отладки*

1. Для анализа ошибки после исключения надо создать новую ячейку, ввести %debug и выполнить её.



*Рисунок 27 - %debug после ZeroDivisionError*

1. Для proactive отладки нужно вставить import pdb; pdb.set\_trace() в код и выполнить ячейку.

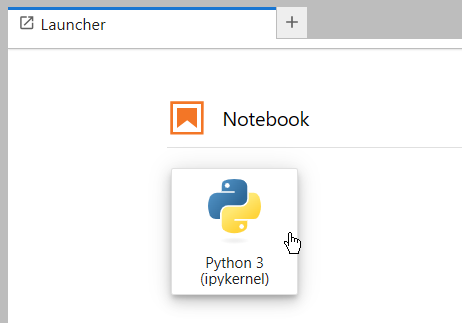


*Рисунок 28 - pdb в середине блока кода*

1. В отладчике используйте команды pdb, такие как n (next), s (step), p variable (print variable), для анализа состояния.

###### Использование через debug в ipykernel

1. Открыть Jupyter Notebook в браузере и создать новый блокнот.

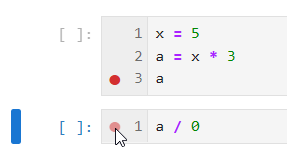


*Рисунок 29 - Создание нового блокнота в Jupyter Notebook*

1. Включить режим отладки и поставить точки остановки.

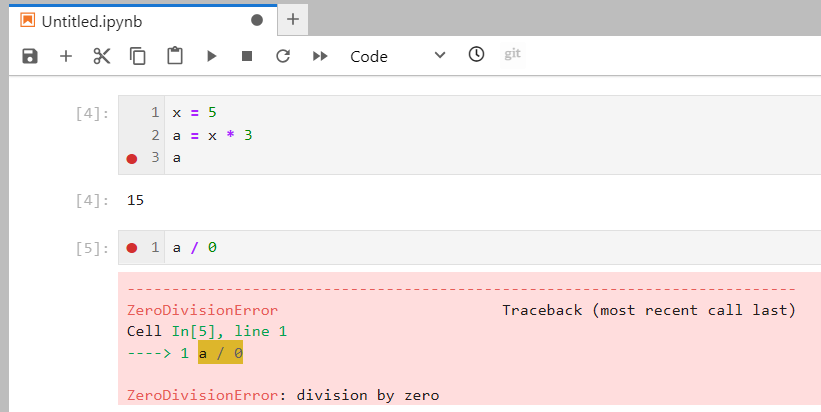


*Рисунок 30 - Включение отладки в Jupyter Notebook*



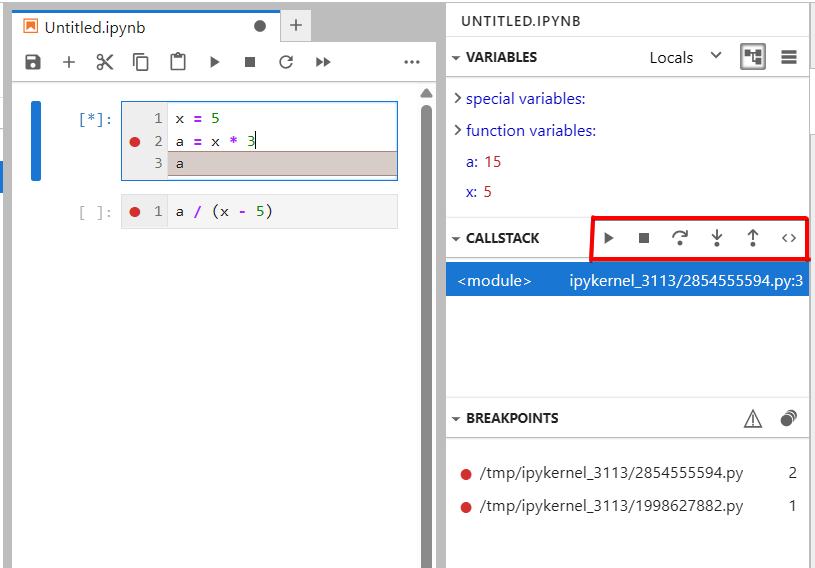
*Рисунок 31 - Установка брейкпоинтов в Jupyter Notebook*

1. Выполнить код с помощью Shift+Enter.

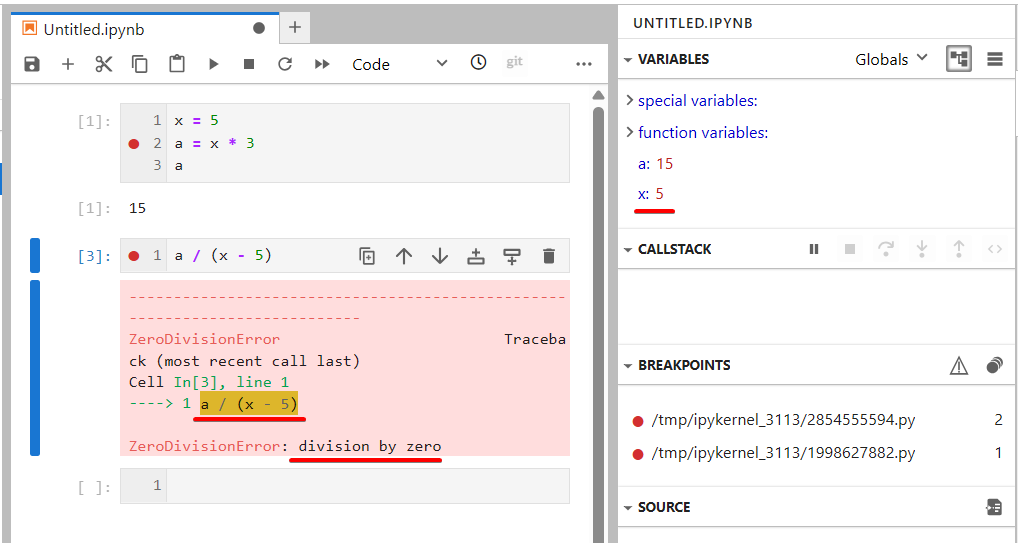


*Рисунок 32 - Запуск кода в режиме отладки*

1. Использовать панель отладки для пошагового выполнения, просмотра переменных и управления выполнением.



*Рисунок 33 - Панель отладки в Jupyter Notebook*



*Рисунок 34 - Ошибка ZeroDivisionError при отладке в Jupyter Notebook*

###### Ссылки

[Описание отладки через pdb в Jupyter Notebook](https://www.cambridge.org/core/resources/pythonforscientists/jupyterdb)  
[Документация отладчика Jupyter Notebook](https://jupyterlab.readthedocs.io/en/latest/user/debugger.html)

#### Профайлеры

##### py-spy

###### Описание

py-spy - статистический профайлер для Python, написанный на Rust. Он анализирует работающие программы без необходимости их перезапуска или модификации кода.

###### Основные возможности

* Статистическое профилирование: выборочный анализ стека вызовов.
* Низкий накладной расход, подходит для продакшена.
* Профилирование работающих процессов по PID.
* Генерация flame graph для визуализации.

###### Установка

Установка через pip: pip install py-spy

###### Использование

Для профилирования процесса: py-spy top --pid <pid>

Для создания flame graph: py-spy record --flame profile.svg --pid <pid>

###### Плюсы

* Минимальное влияние на производительность.
* Поддержка профилирования без изменений кода.
* Визуализация через flame graph.

###### Минусы

* Статистический подход может пропустить редкие события.
* Требует установки стороннего пакета.

###### Ссылки

* [GitHub py-spy](https://github.com/benfred/py-spy)
* [PyPI py-spy](https://pypi.org/project/py-spy/)

##### cProfile

###### Описание

cProfile - встроенный модуль Python для детерминированного профилирования, измеряющий точное время, затраченное на каждый вызов функции. Реализованный на C, он минимизирует накладные расходы, что делает его подходящим для анализа производительности длительных программ.

###### Основные возможности

* Детерминированное профилирование: точное измерение времени выполнения функций.
* Низкий накладной расход благодаря реализации на C.
* Поддержка запуска из командной строки или внутри программ.
* Вывод статистики: количество вызовов, общее и кумулятивное время.
* Интеграция с модулем pstats для анализа результатов.

###### Установка

cProfile входит в стандартную библиотеку Python, установка не требуется.

###### Использование

Для профилирования скрипта можно использовать команду python -m cProfile script.py или через код:

Листинг 7 - Работа с cProfile внутри программы

>>> import cProfile, re  
>>> cProfile.run('re.compile("foo|bar")', sort="ncalls")  
  
 231 function calls (224 primitive calls) in 0.001 seconds  
  
 Ordered by: call count  
  
 ncalls tottime percall cumtime percall filename:lineno(function)  
 48 0.000 0.000 0.000 0.000 {method 'append' of 'list' objects}  
 34 0.000 0.000 0.000 0.000 {built-in method builtins.isinstance}  
 30/27 0.000 0.000 0.000 0.000 {built-in method builtins.len}  
 18 0.000 0.000 0.000 0.000 \_parser.py:168(\_\_getitem\_\_)

###### Плюсы

* Высокая точность измерений.
* Низкий накладной расход, подходит для больших приложений.
* Простота интеграции без дополнительных зависимостей.

###### Минусы

* Не анализирует время выполнения отдельных строк.
* Вывод может быть сложным для интерпретации без инструментов визуализации, таких как [SnakeViz](https://jiffyclub.github.io/snakeviz/).

###### Ссылки

* [Официальная документация cProfile](https://docs.python.org/3/library/profile.html#module-cProfile)

##### timeit

###### Описание

timeit - модуль стандартной библиотеки Python для измерения времени выполнения небольших фрагментов кода. Он идеально подходит для бенчмаркинга и сравнения производительности различных реализаций.

###### Основные возможности

* Точное измерение времени выполнения небольших кодовых фрагментов.
* Многократный запуск кода для повышения точности.
* Отключение сборки мусора для минимизации накладных расходов.
* Поддержка командной строки и программного использования.

###### Установка

timeit входит в стандартную библиотеку Python, установка не требуется.

###### Использование

Листинг 8 - Использование timeit через код

import timeit  
  
print(timeit.timeit("list(map(str, r))", setup="r=range(100)"))  
# 7.257156100000429  
print(timeit.timeit("[str(x) for x in r]", setup="r=range(100)"))  
# 6.102027400000225

Листинг 9 - Использование timeit через командную строку

>>> python -m timeit --number=1\_000\_000 --setup="r=range(100)" "list(map(str, r))"  
1000000 loops, best of 5: 6.24 usec per loop  
  
>>> python -m timeit --number=1\_000\_000 --setup="r=range(100)" "[str(x) for x in r]"  
1000000 loops, best of 5: 5.24 usec per loop

###### Плюсы

* Простота использования для бенчмаркинга.
* Высокая точность благодаря многократным запускам.
* Не требует дополнительных зависимостей.

###### Минусы

* Не подходит для профилирования больших программ.
* Ограничен анализом времени без детальной статистики.

###### Ссылки

* [Официальная документация timeit](https://docs.python.org/3/library/timeit.html)
* [Документация на docs-python.ru](https://docs-python.ru/standart-library/modul-timeit-python)

##### line\_profiler

###### Описание

line\_profiler - инструмент для построчного профилирования Python-кода. Он показывает время, затраченное на каждую строку функции, что помогает выявить точные узкие места.

###### Основные возможности

* Построчное измерение времени выполнения кода.
* Использование декоратора @profile для выбора функций.
* Подробные отчеты с процентом времени на каждую строку.
* Интеграция с инструментами, такими как [Spyder](https://www.spyder-ide.org/).

###### Установка

Установка через pip: pip install line\_profiler

###### Использование

Листинг 10 - Использование line\_profiler

from line\_profiler import profile  
  
@profile  
def fibonacci(n):  
 if n < 2:  
 return n  
 return fibonacci(n - 1) + fibonacci(n - 2)  
  
fibonacci(30)

###### Плюсы

* Детальный анализ времени выполнения строк.
* Простота выбора функций для профилирования.

###### Минусы

* Накладные расходы выше, чем у cProfile.
* Требует установки стороннего пакета.

###### Ссылки

* [GitHub line\_profiler](https://github.com/pyutils/line_profiler)
* [PyPI line-profiler](https://pypi.org/project/line-profiler/)

##### memory\_profiler

###### Описание

memory\_profiler - инструмент для мониторинга потребления памяти Python-программ. Он отслеживает использование памяти построчно, помогая выявить утечки и оптимизировать код.

###### Основные возможности

* Построчный анализ потребления памяти.
* Декоратор @profile для профилирования функций.
* Поддержка отчетов о максимальном использовании памяти.
* Интеграция с psutil для точных измерений.

###### Установка

Установка через pip: pip install memory\_profiler

###### Использование

Листинг 11 - Использование memory\_profiler

from memory\_profiler import profile  
  
@profile  
def main():  
 n = 100\_000\_000  
 a = tuple(range(n))  
 b = list(range(n))  
 del a  
 del b  
  
main()

###### Плюсы

* Выявляет участки кода с высоким потреблением памяти.
* Полезен для обнаружения утечек памяти.

###### Минусы

* Может замедлять выполнение из-за мониторинга памяти.
* Требует установки стороннего пакета.

###### Ссылки

* [GitHub memory\_profiler](https://github.com/pythonprofilers/memory_profiler)
* [PyPI memory-profiler](https://pypi.org/project/memory-profiler/)

##### pyheat

###### Описание

pyheat - инструмент для профилирования Python-кода с визуализацией результатов в виде тепловой карты. Он использует pprofile и matplotlib для отображения горячих зон кода.

###### Основные возможности

* Визуализация профилирования через тепловые карты.
* Простой CLI-интерфейс для анализа Python-файлов.
* Экспорт тепловых карт в изображения.
* Поддержка прокрутки для больших файлов.

###### Установка

Установка через pip: pip install py-heat

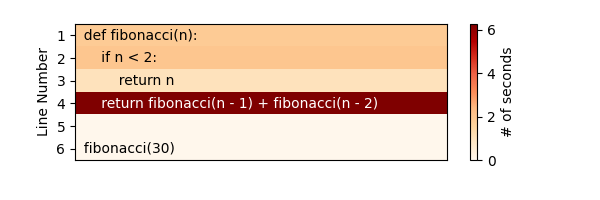
###### Использование

Для профилирования и просмотра тепловой карты: pyheat <path\_to\_python\_file>

Для сохранения в файл: pyheat <path\_to\_python\_file> --out filename.png

Листинг 12 - Код для анализа с помощью pyheat

def fibonacci(n):  
 if n < 2:  
 return n  
 return fibonacci(n - 1) + fibonacci(n - 2)  
  
fibonacci(30)



*Рисунок 35 - Выходное изображение pyheat*

###### Плюсы

* Интуитивная визуализация горячих зон кода.
* Простота использования для быстрого анализа.

###### Минусы

* Менее известен и потенциально менее поддерживаем.
* Ограниченная функциональность по сравнению с другими профайлерами.
* Устарел на текущий момент

###### Ссылки

* [GitHub pyheat](https://github.com/csurfer/pyheat)

#### Линтеры

#### Остальное

##### PyPy

###### Описание

PyPy - это альтернативная реализация Python, использующая just-in-time (JIT) компиляцию для повышения производительности.  
Он совместим с большинством Python-кода и может значительно ускорить выполнение программ по сравнению с CPython.

Для анализа производительности в PyPy часто используют встроенные инструменты, такие как pypyjit, или внешние профайлеры, совместимые с JIT, например, vmprof.

###### Основные возможности

* JIT-компиляция для оптимизации кода во время выполнения.
* Высокая совместимость с CPython-кодом и библиотеками.
* Поддержка cffi, cppyy и популярных библиотек, таких как NumPy, через слой совместимости.
* Stackless-режим для микропотоков и массовой параллельности.
* Оптимизированное использование памяти для ресурсоемких приложений.

###### Установка

PyPy доступен для загрузки на [официальном сайте](https://pypy.org/). Рекомендуется использовать готовые бинарные файлы, так как сборка из исходников требует значительных ресурсов.

###### Использование

PyPy используется как замена CPython. Достаточно запустить скрипт с помощью команды: pypy filename.py

Листинг 22 - Сравнение скорости PyPy и Python

import timeit  
  
def fib(n):  
 if n <= 1:  
 return n  
 return fib(n - 1) + fib(n - 2)  
  
n = 30  
print(timeit.timeit("fib(n)", globals=locals(), number=1000))  
# python: 89.13345369999297  
# pypy: 11.242766199997277

Листинг 23 - Сравнение скорости PyPy и Python

>>> python  
Python 3.12.10 (tags/v3.12.10:0cc8128, Apr 8 2025, 12:21:36) [MSC v.1943 64 bit (AMD64)] on win32  
Type "help", "copyright", "credits" or "license" for more information.  
>>> import timeit  
>>> timeit.timeit("sum(range(1000))")  
7.1561374000002616  
  
  
>>> pypy  
Python 3.11.11 (0253c85bf5f8, Feb 26 2025, 10:43:25)  
[PyPy 7.3.19 with MSC v.1941 64 bit (AMD64)] on win32  
Type "help", "copyright", "credits" or "license" for more information.  
>>> import timeit  
>>> timeit.timeit("sum(range(1000))")  
3.236320699999851

###### Примеры

Листинг 24 - py\_py.py

import timeit  
  
def fib(n):  
 if n <= 1:  
 return n  
 return fib(n - 1) + fib(n - 2)  
  
n = 30  
print(timeit.timeit("fib(n)", globals=locals(), number=1000))  
# python: 89.13345369999297  
# pypy: 11.242766199997277

###### Плюсы

* В среднем в 3 раза быстрее CPython 3.11.
* Эффективное использование памяти для больших программ.
* Поддержка stackless-режима для параллельности.
* Совместимость с большинством Python-библиотек.

###### Минусы

* Некоторые C-расширения CPython могут не работать.
* Сборка из исходников сложна и ресурсоемка.
* Ограниченная поддержка некоторых специализированных библиотек.

###### Ссылки

* [Официальный сайт PyPy](https://pypy.org/)
* [Документация PyPy](https://doc.pypy.org/en/latest/)
* [Репозиторий GitHub PyPy](https://github.com/pypy/pypy)

##### Cython

###### Описание

Cython - это язык, упрощающий создание C-расширений для Python. Код, похожий на Python, компилируется в C, что позволяет ускорить выполнение критических участков программы.

Cython преобразует Python-код в C, что требует использования отладчиков вроде gdb для анализа скомпилированного кода, а профилирование может проводиться с помощью инструментов C, таких как gprof.

###### Основные возможности

* Синтаксис, близкий к Python, с добавлением статических типов.
* Ускорение от 2x до 1000x при использовании типизации.
* Интеграция с C/C++ библиотеками, включая стандартную библиотеку C++.
* Поддержка вызова существующих C-функций.
* Генерация оптимизированного C-кода для Python-модулей.

###### Установка

Установите Cython через pip:

Листинг 25

pip install Cython

Также требуется C-компилятор (например, GCC).

###### Использование

Создайте файл .pyx, добавьте код с типами, скомпилируйте с помощью Cython и импортируйте как Python-модуль:

###### Примеры

Листинг 26 - simple.c

int add(int a, int b) {  
 return a + b;  
}  
  
int multiply(int a, int b) {  
 return a \* b;  
}

Листинг 27 - simple.h

#ifndef SIMPLE\_H  
#define SIMPLE\_H  
  
int add(int a, int b);  
int multiply(int a, int b);  
  
#endif

Листинг 28 - cython\_file.pyx

cdef extern from "simple.h":  
 int add(int a, int b)  
 int multiply(int a, int b)  
  
  
cdef int subtract(int a, int b):  
 return a - b  
  
  
def py\_add(int a, int b) -> int:  
 return add(a, b)  
  
def py\_multiply(int a, int b) -> int:  
 return multiply(a, b)  
  
def py\_subtract(int a, int b) -> int:  
 return subtract(a, b)

Листинг 29 - cython\_setup.py

# python cython\_setup.py build\_ext --inplace --compiler=mingw32  
from setuptools import setup  
from Cython.Build import cythonize  
from setuptools.extension import Extension  
  
extensions = [  
 Extension(  
 "cython\_file",  
 sources=["cython\_file.pyx", "simple.c"],  
 include\_dirs=["."],  
 )  
]  
  
setup(  
 name="cython\_integration",  
 ext\_modules=cythonize(extensions, compiler\_directives={'language\_level': "3"}),  
)

Листинг 30 - cython\_run.py

import cython\_file  
  
result\_add = cython\_file.py\_add(5, 3)  
print(f"Сложение: 5 + 3 = {result\_add}")  
  
result\_multiply = cython\_file.py\_multiply(4, 5)  
print(f"Умножение: 4 \* 5 = {result\_multiply}")  
  
result\_substract = cython\_file.py\_subtract(10, 5)  
print(f"Разность: 10 - 5 = {result\_multiply}")

###### Плюсы

* Значительное ускорение численных вычислений.
* Простая интеграция с C/C++ библиотеками.
* Синтаксис, близкий к Python, упрощает обучение.

###### Минусы

* Требуется компиляция, что усложняет разработку.
* Код с типами менее читаем.
* Ограниченная поддержка динамических функций Python.

###### Ссылки

* [Официальный сайт Cython](https://cython.org/)
* [Документация Cython](https://cython.readthedocs.io/en/latest/)
* [Репозиторий GitHub Cython](https://github.com/cython/cython)

##### Интеграция с C

###### Описание

Интеграция Python с C позволяет использовать производительность C для ускорения Python-программ. Это достигается через создание C-расширений, вызываемых из Python.

###### Основные возможности

* Вызов C-функций из Python.
* Создание высокопроизводительных Python-модулей.
* Доступ к существующим C-библиотекам.
* Оптимизация критических участков кода.

###### Установка

Требуются заголовочные файлы Python (включены в стандартную установку) и C-компилятор (например, GCC).

###### Использование

Напишите C-код, определите функции, скомпилируйте в общую библиотеку (.so/.dll) и импортируйте в Python с помощью ctypes, cffi или Python/C API.

###### Примеры

Листинг 31 - c\_simple.c

int add(int a, int b) {  
 return a + b;  
}  
  
int multiply(int a, int b) {  
 return a \* b;  
}

Листинг 32 - c\_ctypes.py

# Windows: gcc -shared -o simple.dll simple.c  
# Linux: gcc -shared -o simple.os -fPIC simple.c  
import ctypes  
  
# Windows: simple.dll  
# Linux: simple.so  
lib = ctypes.CDLL('./simple.dll')  
  
lib.add.argtypes = [ctypes.c\_int, ctypes.c\_int]  
lib.add.restype = ctypes.c\_int  
  
result = lib.add(5, 3)  
print(f"Результат: {result}") # Результат: 8

Листинг 33 - c\_cffi.py

# Windows: gcc -shared -o simple.dll simple.c  
# Linux: gcc -shared -o simple.os -fPIC simple.c  
from cffi import FFI  
  
ffi = FFI()  
ffi.cdef("int multiply(int a, int b);")  
  
# Windows: simple.dll  
# Linux: simple.so  
lib = ffi.dlopen("simple.dll")  
  
result = lib.multiply(4, 5)  
print(f"Результат: {result}") # Результат: 20

Листинг 34 - c\_capi.c

#include <Python.h>  
  
static PyObject\* add\_numbers(PyObject\* self, PyObject\* args) {  
 int a, b;  
 if (!PyArg\_ParseTuple(args, "ii", &a, &b)) {  
 return NULL;  
 }  
 return Py\_BuildValue("i", a + b);  
}  
  
static PyMethodDef integration\_methods[] = {  
 {"add\_numbers", add\_numbers, METH\_VARARGS, "Add two numbers."},  
 {NULL, NULL, 0, NULL}  
};  
  
static struct PyModuleDef integration\_module = {  
 PyModuleDef\_HEAD\_INIT,  
 "integration",  
 "A module to add numbers.",  
 -1,  
 integration\_methods  
};  
  
#ifdef \_WIN32  
\_\_declspec(dllexport)  
#endif  
PyMODINIT\_FUNC PyInit\_integration(void) {  
 return PyModule\_Create(&integration\_module);  
}

Листинг 35 - c\_capi\_setup.py

# python c\_capi\_setup.py build\_ext --inplace --compiler=mingw32  
from setuptools import setup, Extension  
  
module = Extension(name='integration', sources=['capi.c'])  
  
setup(  
 name='integration',  
 version='1.0',  
 description='A simple Python C extension module',  
 ext\_modules=[module]  
)

Листинг 36 - c\_capi\_run.py

# python capi\_setup.py build\_ext --inplace --compiler=mingw32  
import integration # Собственный модуль, скомпилированный через Python/C API  
  
result = integration.add\_numbers(5, 3)  
print(result) # Вывод: 8

###### Плюсы

* Высокая производительность для численных задач.
* Доступ к обширным C-библиотекам.
* Повторное использование существующего C-кода.

###### Минусы

* Требуется знание C.
* Сложность управления C-кодом и компиляцией.
* Отладка требует знаний C и Python.
* Требуется опыта работы с Python/C API.

###### Ссылки

* [Документация Python/C API](https://docs.python.org/3/extending/extending.html)

## 3. Тестирование и оценка результатов выполнения работы.

Тестирование и оценка результатов выполненной работы на тему отладка и профилирование программ с использованием языка Питон

# Заключение

В ходе данной работы был проведен анализ методов и инструментов отладки, профилирования и статического анализа кода для программ на языке Python. В первой главе, посвященной научному поиску, были рассмотрены основные категории инструментов: отладчики, профайлеры и линтеры, а также дополнительные технологии. Для каждой категории были описаны ключевые инструменты, их возможности, преимущества и недостатки, что позволило сформировать представление об их применимости в различных сценариях разработки.

Во второй главе были детально проанализированы выбранные инструменты с точки зрения их установки, использования и практических примеров. Этот анализ дал возможность не только теоретически изучить их функциональность, но и оценить их эффективность на практике. В результате были выделены сильные и слабые стороны каждого инструмента, что может служить основой для выбора подходящего инструмента в зависимости от задачи - будь то поиск ошибок, оптимизация производительности или повышение качества кода.

Итогом работы стало углубленное понимание инструментария Python для разработки, отладки и оптимизации программ. Полученные знания могут быть применены для улучшения качества кода, ускорения выполнения приложений и обучения эффективным практикам программирования.

# ПРИЛОЖЕНИЯ

Листинг 43 - Пример работы с pdb breakpoints

def too\_much\_conditions(a: int, b: int) -> int:  
 breakpoint()  
 if b == 0:  
 breakpoint()  
 return 1  
 if b == 1:  
 breakpoint()  
 return a \*\* 1  
 if b == 2:  
 breakpoint()  
 return a \*\* 2  
 if b == 3:  
 breakpoint()  
 return a \*\* 3  
 if b == 4:  
 breakpoint()  
 return a \*\* 4  
 breakpoint()  
 return a \*\* 5  
  
for b in range(0, 6):  
 too\_much\_conditions(2, b)

Листинг 44 - Пример работы с pdb postmortem

def zero\_divison(a: int) -> int:  
 return a // 0  
  
zero\_divison(1)

Листинг 45 - Пример работы с print

tralalelo = "tralala"  
bombordiro = "crocodilo"  
brr\_brr = "patapim"  
  
print(tralalelo, bombordiro, brr\_brr)  
# Вывод: tralala crocodilo patapim  
print(tralalelo, bombordiro, brr\_brr, sep="---")  
# Вывод: tralala---crocodilo---patapim  
print(tralalelo, bombordiro, brr\_brr, sep=", ")  
# Вывод: tralala, crocodilo, patapim  
  
print(tralalelo, end=", ")  
print(bombordiro, end=", ")  
print(brr\_brr)  
# Вывод: tralala, crocodilo, patapim  
  
with open("temp.txt", "w") as f:  
 print(tralalelo, bombordiro, brr\_brr,  
 sep=" | ", end=" = frigo camelo", file=f, flush=True)  
# Содержимое файла temp.txt: tralala | crocodilo | patapim = frigo camelo  
  
def calculator(a: float, b: float, operation: str) -> float | None:  
 print("Переданные значения: a=", a, ", b=", b, ", operation=", operation, sep="")  
  
 if operation == "+":  
 print("Складываю", a, "и", b)  
 print("Результат:", a + b, end="\n\n")  
 return a + b  
 if operation == "-":  
 print("Вычитаю из", a, "число", b)  
 print("Результат:", a - b, end="\n\n")  
 return a - b  
 if operation == "\*":  
 print("Умножаю", a, "и", b)  
 print("Результат:", a \* b, end="\n\n")  
 return a \* b  
  
 print("Не нашёл подходящее действия для оператора", operation, "возвращаю None", end="\n\n")  
 return None  
  
calculator(1, 2, "+")  
calculator(3, 4, "-")  
calculator(5, 6, "\*")  
calculator(7, 8, "/")  
  
"""  
Переданные значения: a=1, b=2, operation=+  
Складываю 1 и 2  
Результат: 3  
  
Переданные значения: a=3, b=4, operation=-  
Вычитаю из 3 число 4  
Результат: -1  
  
Переданные значения: a=5, b=6, operation=\*  
Умножаю 5 и 6  
Результат: 30  
  
Переданные значения: a=7, b=8, operation=/  
Не нашёл подходящее действия для оператора / возвращаю None  
"""

Листинг 46 - Пример работы с logging

import logging  
  
logger = logging.getLogger(\_\_name\_\_)  
  
logging.basicConfig(  
 filemode="a",  
 format="%(asctime)s - [%(levelname)s] - %(name)s - (%(filename)s).%(funcName)s(%(lineno)d) - %(message)s",  
 datefmt="%H:%M:%S",  
 style="%",  
 level=logging.DEBUG,  
)  
  
tralalelo = "tralala"  
bombordiro = "crocodilo"  
brr\_brr = "patapim"  
  
logger.debug("%s %s %s", tralalelo, bombordiro, brr\_brr)  
logger.info("%s---%s---%s", tralalelo, bombordiro, brr\_brr)  
logger.warning("%s, %s, %s", tralalelo, bombordiro, brr\_brr)  
"""  
00:51:35 - [DEBUG] - \_\_main\_\_ - (logger.py).<module>(17) - tralala crocodilo patapim  
00:51:35 - [INFO] - \_\_main\_\_ - (logger.py).<module>(18) - tralala---crocodilo---patapim  
00:51:35 - [WARNING] - \_\_main\_\_ - (logger.py).<module>(19) - tralala, crocodilo, patapim  
"""  
  
def calculator(a: int, b: int, operation: str) -> int | None:  
 logger.debug("Переданные значения: a=%d, b=%d, operation=%s", a, b, operation)  
  
 if operation == "+":  
 logger.debug("Складываю %d и %d", a, b)  
 logger.debug("Результат: %d\n", a + b)  
 return a + b  
 if operation == "-":  
 logger.debug("Вычитаю из %d число %d", a, b)  
 logger.debug("Результат: %d\n", a - b)  
 return a - b  
 if operation == "\*":  
 logger.debug("Умножаю %d и %d", a, b)  
 logger.debug("Результат: %d\n", a \* b)  
 return a \* b  
  
 logger.warning("Не нашёл подходящее действия для оператора %s возвращаю None", operation)  
 return None  
  
calculator(1, 2, "+")  
calculator(3, 4, "-")  
calculator(5, 6, "\*")  
calculator(7, 8, "/")  
  
"""  
00:51:35 - [DEBUG] - \_\_main\_\_ - (logger.py).calculator(26) - Переданные значения: a=1, b=2, operation=+  
00:51:35 - [DEBUG] - \_\_main\_\_ - (logger.py).calculator(29) - Складываю 1 и 2  
00:51:35 - [DEBUG] - \_\_main\_\_ - (logger.py).calculator(30) - Результат: 3  
  
00:51:35 - [DEBUG] - \_\_main\_\_ - (logger.py).calculator(26) - Переданные значения: a=3, b=4, operation=-  
00:51:35 - [DEBUG] - \_\_main\_\_ - (logger.py).calculator(33) - Вычитаю из 3 число 4  
00:51:35 - [DEBUG] - \_\_main\_\_ - (logger.py).calculator(34) - Результат: -1  
  
00:51:35 - [DEBUG] - \_\_main\_\_ - (logger.py).calculator(26) - Переданные значения: a=5, b=6, operation=\*  
00:51:35 - [DEBUG] - \_\_main\_\_ - (logger.py).calculator(37) - Умножаю 5 и 6  
00:51:35 - [DEBUG] - \_\_main\_\_ - (logger.py).calculator(38) - Результат: 30  
  
00:51:35 - [DEBUG] - \_\_main\_\_ - (logger.py).calculator(26) - Переданные значения: a=7, b=8, operation=/  
00:51:35 - [WARNING] - \_\_main\_\_ - (logger.py).calculator(41) - Не нашёл подходящее действия для оператора / возвращаю None  
"""

Листинг 47 - Пример работы с pytest

import pytest  
  
UserId = int  
UserName = str  
User = tuple[UserId, UserName]  
  
class Database:  
 def \_\_init\_\_(self):  
 self.data: dict[UserId, User] = {}  
  
 def add(self, user: User) -> None:  
 self.data[user[0]] = user  
  
 def get(self, user: UserId) -> User | None:  
 return self.data.get(user)  
  
 def delete(self, user: UserId) -> bool:  
 try:  
 del self.data[user]  
 return True  
 except KeyError:  
 return False  
  
 def len(self) -> int:  
 return len(self.data)  
  
  
@pytest.fixture(scope="session")  
def database() -> Database:  
 return Database()  
  
def test\_add\_get\_delete(database: Database) -> None:  
 user = (1, "Kirill")  
 database.add(user)  
 assert database.get(1) == user  
 assert database.delete(1) is True  
 assert database.delete(1) is False  
 assert database.get(1) is None  
  
@pytest.mark.parametrize(  
 "user, count",  
 [  
 ((1, "Kirill"), 1),  
 ((2, "Nikita"), 2),  
 ((3, "Egor"), 3),  
 ((4, "Maks"), 4),  
 ]  
)  
def test\_len(user: User, count: int, database: Database) -> None:  
 assert database.len() == count - 1  
 database.add(user)  
 assert database.len() == count

Листинг 48 - Пример работы с unittest

import unittest  
from unittest.mock import Mock  
from typing import Tuple, Dict, Optional  
  
UserId = int  
UserName = str  
User = Tuple[UserId, UserName]  
  
class Database:  
 def \_\_init\_\_(self):  
 self.data: Dict[UserId, User] = {}  
  
 def add(self, user: User) -> None:  
 self.data[user[0]] = user  
  
 def get(self, user: UserId) -> Optional[User]:  
 return self.data.get(user)  
  
 def delete(self, user: UserId) -> bool:  
 try:  
 del self.data[user]  
 return True  
 except KeyError:  
 return False  
  
 def len(self) -> int:  
 return len(self.data)  
  
class TestDatabase(unittest.TestCase):  
 @classmethod  
 def setUpClass(cls) -> None:  
 cls.database = Database()  
  
 @classmethod  
 def tearDownClass(cls) -> None:  
 cls.database.data.clear()  
  
 def test\_add\_get\_delete(self) -> None:  
 user = (1, "Kirill")  
 self.database.add(user)  
 self.assertEqual(self.database.get(1), user)  
 self.assertTrue(self.database.delete(1))  
 self.assertFalse(self.database.delete(1))  
 self.assertIsNone(self.database.get(1))  
  
 def test\_len(self) -> None:  
 test\_cases = [  
 ((2, "Nikita"), 1),  
 ((3, "Egor"), 2),  
 ((4, "Maks"), 3),  
 ((5, "Alex"), 4),  
 ]  
 initial\_len = self.database.len()  
 for user, count in test\_cases:  
 with self.subTest(user=user, count=count):  
 self.assertEqual(self.database.len(), initial\_len + count - 1)  
 self.database.add(user)  
 self.assertEqual(self.database.len(), initial\_len + count)  
  
 def test\_mock\_database\_get(self) -> None:  
 mock\_database = Mock(spec=Database)  
 mock\_database.get.return\_value = (1, "Kirill")  
 result = mock\_database.get(1)  
 self.assertEqual(result, (1, "Kirill"))  
 mock\_database.get.assert\_called\_once\_with(1)  
  
if \_\_name\_\_ == '\_\_main\_\_':  
 unittest.main()

Листинг 49 - Пример работы с cProfile

import cProfile  
import sys  
import math  
  
sys.setrecursionlimit(1\_000\_000)  
  
def recursive\_factorial(n: int) -> int:  
 if n == 0:  
 return 1  
 return n \* recursive\_factorial(n - 1)  
  
def loop\_factorial(n: int) -> int:  
 factorial = 1  
 for i in range(1, n + 1):  
 factorial \*= i  
 return factorial  
  
cProfile.run('recursive\_factorial(300\_000)')  
cProfile.run('loop\_factorial(300\_000)')  
cProfile.run('math.factorial(300\_000)')  
  
"""  
 300004 function calls (4 primitive calls) in 27.304 seconds  
  
 Ordered by: standard name  
  
 ncalls tottime percall cumtime percall filename:lineno(function)  
 1 0.000 0.000 27.304 27.304 <string>:1(<module>)  
 300001/1 27.304 0.000 27.304 27.304 cprofile.py:9(recursive\_factorial)  
 1 0.000 0.000 27.304 27.304 {built-in method builtins.exec}  
 1 0.000 0.000 0.000 0.000 {method 'disable' of '\_lsprof.Profiler' objects}  
"""  
"""  
 4 function calls in 26.556 seconds  
  
 Ordered by: standard name  
  
 ncalls tottime percall cumtime percall filename:lineno(function)  
 1 0.000 0.000 26.556 26.556 <string>:1(<module>)  
 1 26.556 26.556 26.556 26.556 cprofile.py:15(loop\_factorial)  
 1 0.000 0.000 26.556 26.556 {built-in method builtins.exec}  
 1 0.000 0.000 0.000 0.000 {method 'disable' of '\_lsprof.Profiler' objects}  
"""  
"""  
 4 function calls in 0.808 seconds  
  
 Ordered by: standard name  
  
 ncalls tottime percall cumtime percall filename:lineno(function)  
 1 0.000 0.000 0.808 0.808 <string>:1(<module>)  
 1 0.000 0.000 0.808 0.808 {built-in method builtins.exec}  
 1 0.808 0.808 0.808 0.808 {built-in method math.factorial}  
 1 0.000 0.000 0.000 0.000 {method 'disable' of '\_lsprof.Profiler' objects}  
"""

Листинг 50 - Пример работа с timeit: сравнение списочных выражений и map с lambda

import timeit  
  
n = 1\_000\_000  
  
print(  
 timeit.timeit("list(map(str, r))",  
 setup="r=range(100)", number=n)  
)  
# 7.257156100000429  
  
print(  
 timeit.timeit("[str(x) for x in r]",  
 setup="r=range(100)", number=n)  
)  
# 6.102027400000225  
  
print(  
 timeit.timeit("list(map(lambda x: x \* 2, r))",  
 setup="r=range(100)", number=n)  
)  
# 4.217808600002172  
  
print(  
 timeit.timeit("[x \* 2 for x in r]",  
 setup="r=range(100)", number=n)  
)  
# 1.9810508999980811  
  
"""  
python -m timeit --number=1\_000\_000 --setup="r=range(100)" "list(map(str, r))"  
1000000 loops, best of 5: 6.24 usec per loop  
  
python -m timeit --number=1\_000\_000 --setup="r=range(100)" "[str(x) for x in r]"  
1000000 loops, best of 5: 5.24 usec per loop  
  
python -m timeit --number=1\_000\_000 --setup="r=range(100)" "list(map(lambda x: x \* 2, r))"  
1000000 loops, best of 5: 4.23 usec per loop  
  
python -m timeit --number=1\_000\_000 --setup="r=range(100)" "[x \* 2 for x in r]"  
1000000 loops, best of 5: 1.98 usec per loop  
"""

Листинг 51 - Пример работы с line\_profiler

>>> kernprof -l short.py  
Wrote profile results to short.py.lprof  
Inspect results with:  
C:\Users\Kirill\AppData\Local\Programs\Python\Python312\python.exe -m line\_profiler -rmt "short.py.lprof"  
  
>>> python -m line\_profiler short.py.lprof  
Timer unit: 1e-06 s  
  
Total time: 1.97951 s  
File: short.py  
Function: fibonacci at line 3  
  
Line # Hits Time Per Hit % Time Line Contents  
==============================================================  
 3 @profile  
 4 def fibonacci(n):  
 5 2692537 487317.6 0.2 24.6 if n < 2:  
 6 1346269 232607.4 0.2 11.8 return n  
 7 1346268 1259582.0 0.9 63.6 return fibonacci(n - 1) + fibonacci(n - 2)

Листинг 52 - Пример работы с memory\_profiler

>>>python -m memory\_profiler ./examples/memory\_profiler/short.py  
Filename: ./examples/memory\_profiler/short.py  
  
Line # Mem usage Increment Occurrences Line Contents  
=============================================================  
 3 24.3 MiB 24.3 MiB 1 @profile  
 4 def main():  
 5 24.3 MiB 0.0 MiB 1 n = 100\_000\_000  
 6 3851.3 MiB 3827.0 MiB 1 a = tuple(range(n))  
 7 7678.3 MiB 3827.0 MiB 1 b = list(range(n))  
 8 3853.4 MiB -3824.9 MiB 1 del a  
 9 28.4 MiB -3824.9 MiB 1 del b