|  |
| --- |
| 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 82 с., 37 рис., 25 листингов, 2 табл., 27 источников

МЕТОДЫ И ИНСТРУМЕНТЫ ОТЛАДКИ И ПРОФИЛИРОВАНИЯ ПРОГРАММ ПРИ ИСПОЛЬЗОВАНИИ ЯЗЫКА PYTHON

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, 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 7](#_Toc199557612)

[Содержание 8](#_Toc199557613)

[Термины и определения 10](#_Toc199557614)

[Перечень сокращений и обозначений 11](#_Toc199557615)

[Основная часть 13](#_Toc199557616)

[1. Научный поиск 13](#_Toc199557617)

[Отладчики 13](#_Toc199557618)

[Профайлеры 14](#_Toc199557619)

[Статистические профайлеры 15](#_Toc199557620)

[Событийные профайлеры 16](#_Toc199557621)

[Линтеры 18](#_Toc199557622)

[2. Ознакомление 20](#_Toc199557623)

[Отладчики 20](#_Toc199557624)

[pdb (Python Debugger) 20](#_Toc199557625)

[print 23](#_Toc199557626)

[logging 25](#_Toc199557627)

[pytest 27](#_Toc199557628)

[unittest 29](#_Toc199557629)

[PyСharm 32](#_Toc199557630)

[VS Code 40](#_Toc199557631)

[Jupyter Notebook 46](#_Toc199557632)

[Профайлеры 53](#_Toc199557633)

[cProfile 53](#_Toc199557634)

[timeit 55](#_Toc199557635)

[line\_profiler 57](#_Toc199557636)

[memory\_profiler 59](#_Toc199557637)

[pyheat 61](#_Toc199557638)

[Линтеры 63](#_Toc199557639)

[flake8 63](#_Toc199557640)

[ruff 65](#_Toc199557641)

[mypy 67](#_Toc199557642)

[3. Тестирование и оценка результатов выполнения работы. 69](#_Toc199557643)

[Задача 69](#_Toc199557644)

[Первая версия 69](#_Toc199557645)

[Отладка с помощью PyCharm 71](#_Toc199557646)

[Отладка с помощью cProfile 74](#_Toc199557647)

[Форматирование и аннотации типов 76](#_Toc199557648)

[Оценка результатов 78](#_Toc199557649)

[Вывод 78](#_Toc199557650)

[Заключение 80](#_Toc199557651)

[Источники 81](#_Toc199557652)

# 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. |
| Тепловая карта | Графическое представление активности кода, где интенсивность цвета указывает на время выполнения. |

# Перечень сокращений и обозначений

В настоящем отчёте о НИР применяются следующие сокращения и обозначения.

|  |  |
| --- | --- |
| **Сокращение** | **Расшифровка** |
| IDE | Integrated Development Environment (Интегрированная среда разработки) |
| pdb | Python Debugger (Отладчик Python) |
| PEP8 | 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.

* 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 PEP8.  
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 PEP8, pyflakes и mccabe, предлагая гибкую настройку.
* ruff - высокопроизводительный линтер и форматтер, написанный на Rust. Быстрее традиционных инструментов, поддерживает автоматическое исправление ошибок.
* mypy - статический анализатор типов. Использует аннотации типов для выявления ошибок типизации на этапе разработки.

## 2. Ознакомление

### Отладчики

#### pdb (Python Debugger)

pdb[1] - 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.

##### Использование

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.

#### print

print[2] - 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.

#### logging

logging[3] - 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.

#### pytest

pytest[4] - 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.

#### unittest

unittest[5] - 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: тестовые фикстуры, тестовые случаи, наборы тестов, запуск 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.

#### PyСharm

PyCharm[6] - профессиональная IDE от JetBrains, созданная для Python-разработки. Её отладчик поддерживает точки остановки, пошаговое выполнение, анализ переменных и условные остановки.

##### Основные возможности

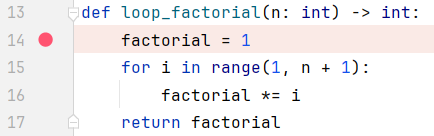
* Установка брейкпоинтов для приостановки выполнения кода.
* Пошаговое выполнение (step over, step into, step out).
* Просмотр стека вызова функций.
* Просмотр и изменение значений переменных в реальном времени.
* Условные точки остановки для остановки при выполнении условий.
* Точки остановки на исключениях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

##### Установка

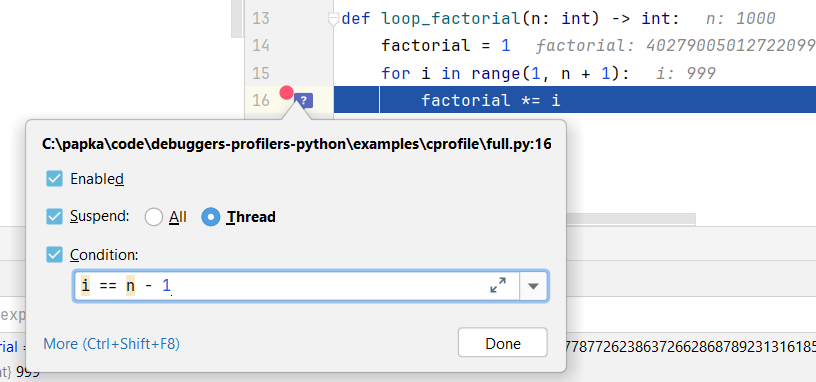
1. Скачать PyCharm с официального сайта JetBrains[7].
2. Установите программу, следуя инструкциям установщика.

##### 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.

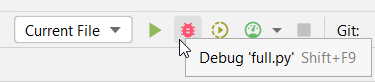


*Рисунок 1 - Установка брейкпоинта в PyCharm*



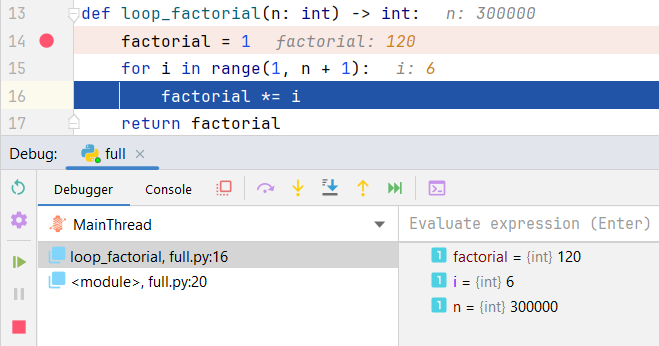
*Рисунок 2 - Установка условия на брейкпоинт в PyCharm*

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.

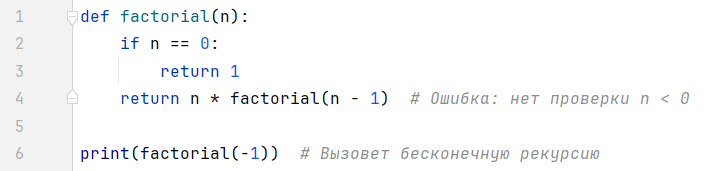


*Рисунок 4 - Панель отладки в PyCharm*

##### Пример

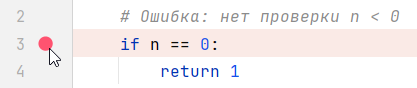
Рассмотрим функцию вычисления факториала, которая не проверяет отрицательные входные значения, что приводит к бесконечной рекурсии и RecursionError.

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



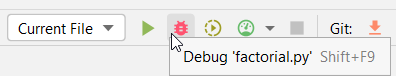
*Рисунок 5 - Код функции вычисления факториала*

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



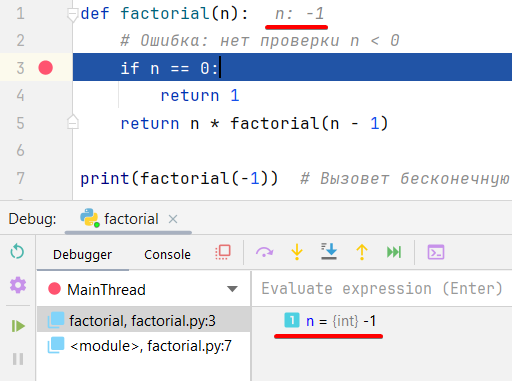
*Рисунок 6 - Установка брейкпоинта на строке с условием*

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



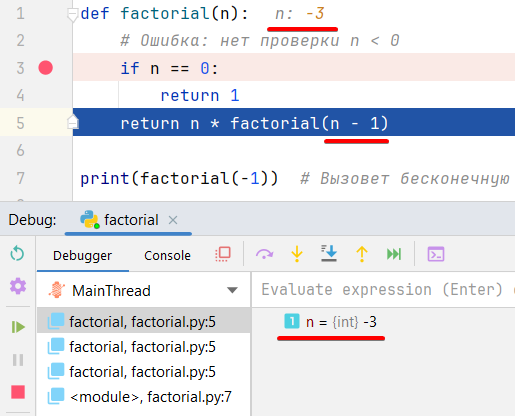
*Рисунок 7 - Запуск отладки factorial.py*

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

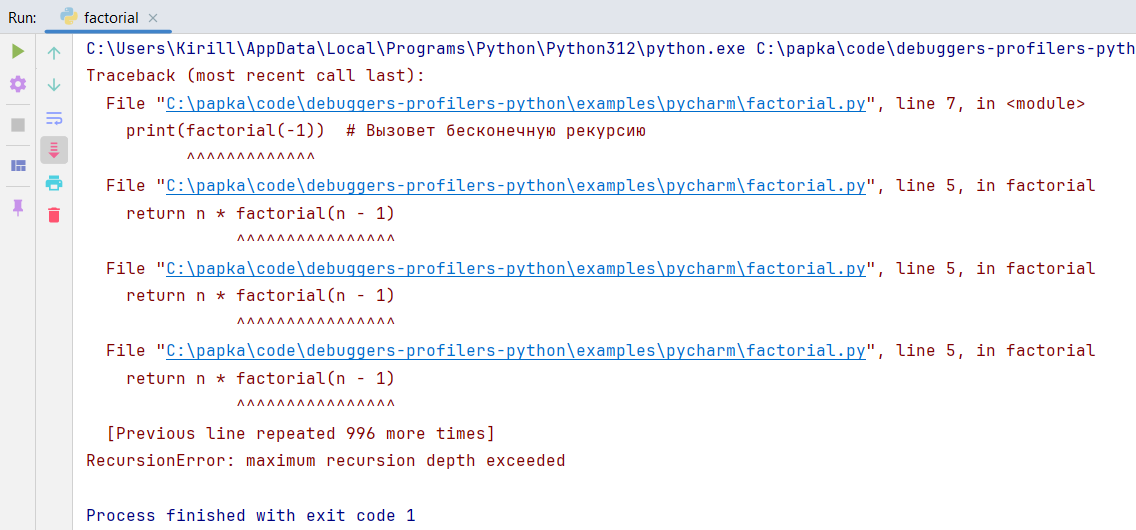


*Рисунок 8 - Просмотр значения переменной n*

**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.

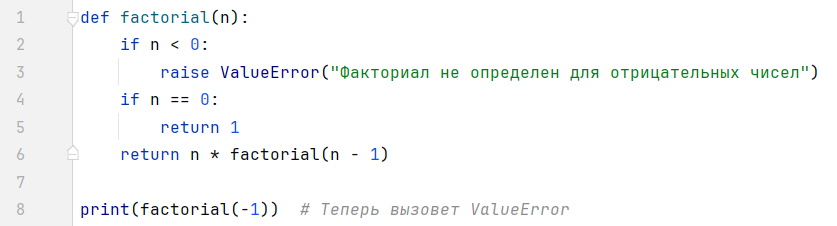


*Рисунок 9 - Переменная n уменьшается*



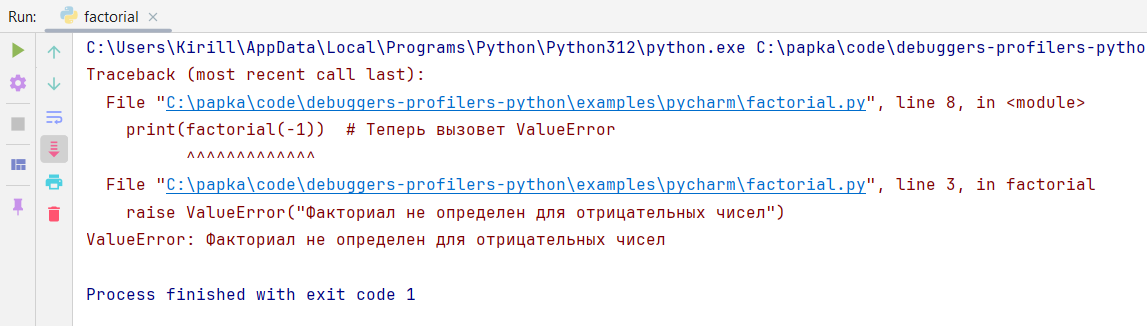
*Рисунок 10 - Ошибка вечной рекурсии*

**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.

#### VS Code

Visual Studio Code (VS Code)[8] - 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

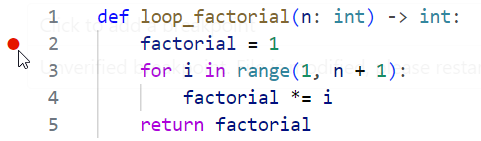
* Установка брейкпоинтов для приостановки выполнения.
* Условные точки остановки для остановки при определенных условиях.
* Пошаговое выполнение кода (step over, step into, step out).
* Панель переменных для просмотра и изменения значений.
* Журналирование значений (logpoints) без остановки выполнения.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

##### Установка

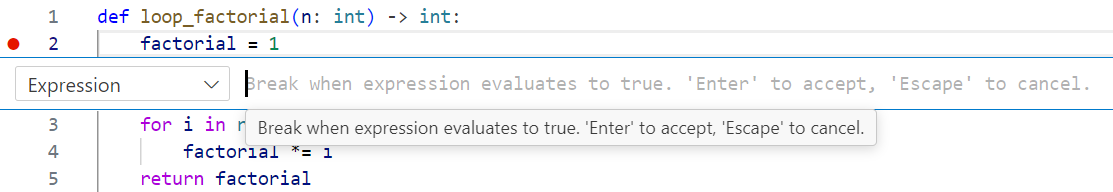
1. Скачайте VS Code с официального сайта[9].
2. Установите редактор, следуя инструкциям.
3. Откройте VS Code, перейдите в раздел "Extensions" (Ctrl+Shift+X), найдите расширение "Python" от Microsoft и установите его.

##### 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.

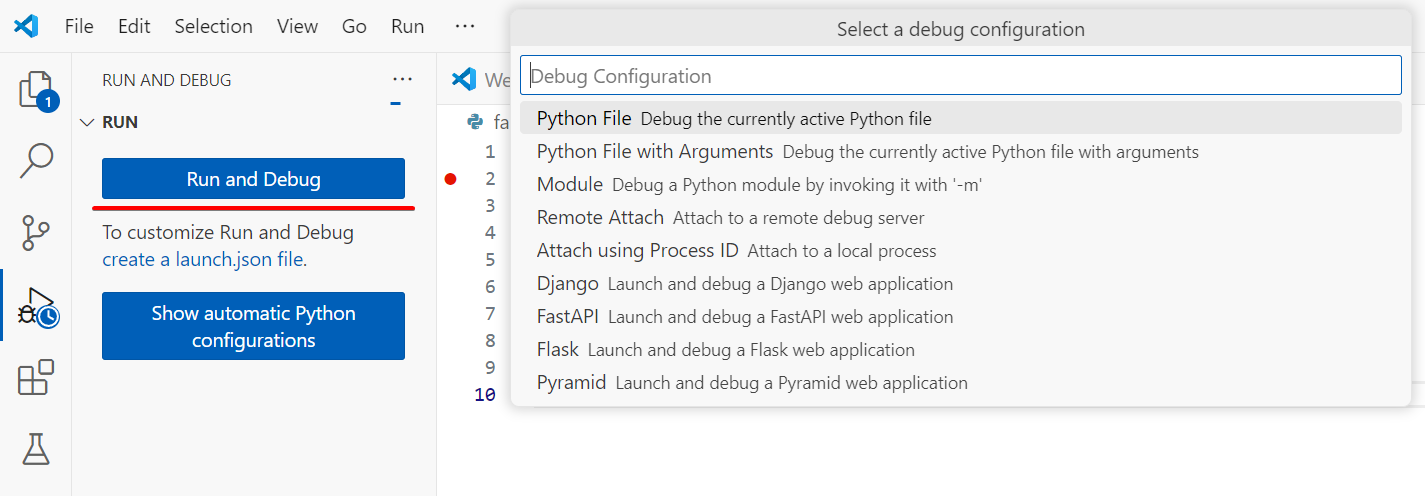


*Рисунок 13 - Установка брейпоинта в VS Code*



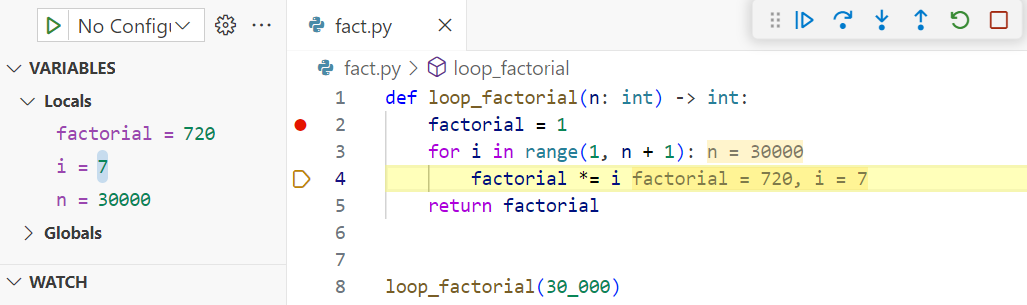
*Рисунок 14 - Установка условия на брейкпоинт в VS Code*

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.

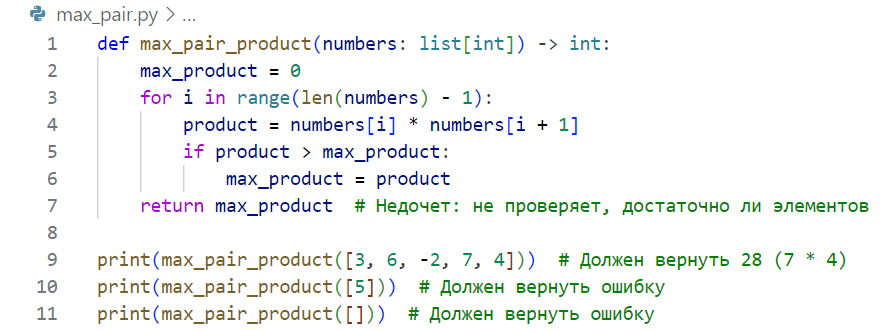


*Рисунок 16 - Панель отладки в VS Code*

##### Пример

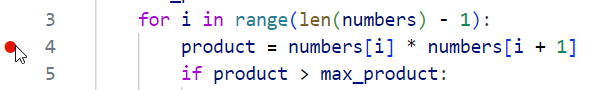
Рассмотрим функцию, которая вычисляет максимальное произведение двух соседних элементов в списке чисел. Ошибка заключается в том, что функция предполагает, что список всегда содержит как минимум два элемента, что приводит к некорректному результату (например, возвращает 0 для списка из одного элемента).

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



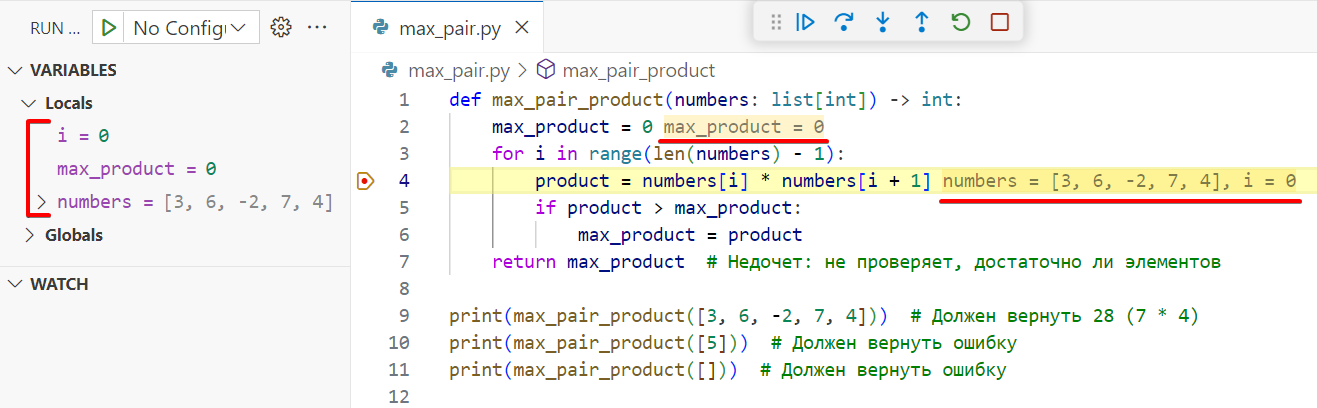
*Рисунок 17 - Код функции поиска наибольшего произведения пары*

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



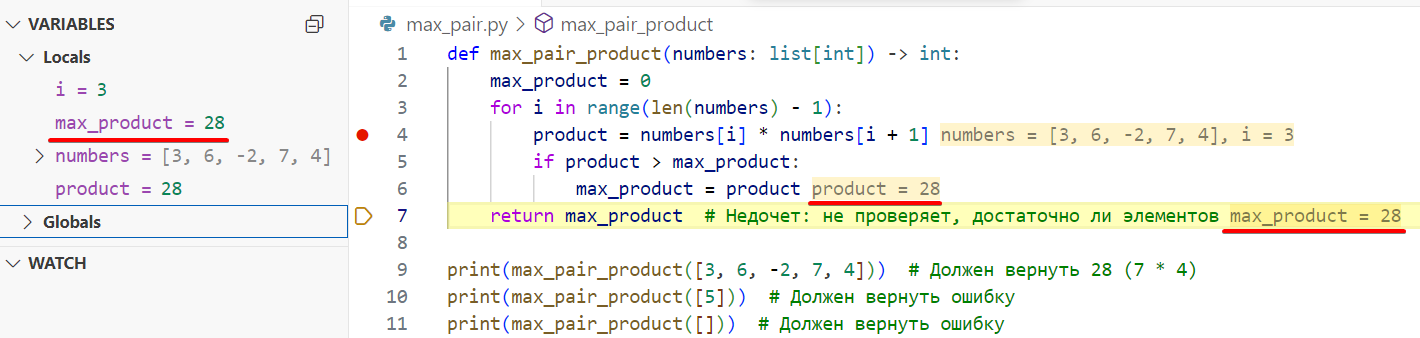
*Рисунок 18 - Установка брейкпоинта на строке с умножением*

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



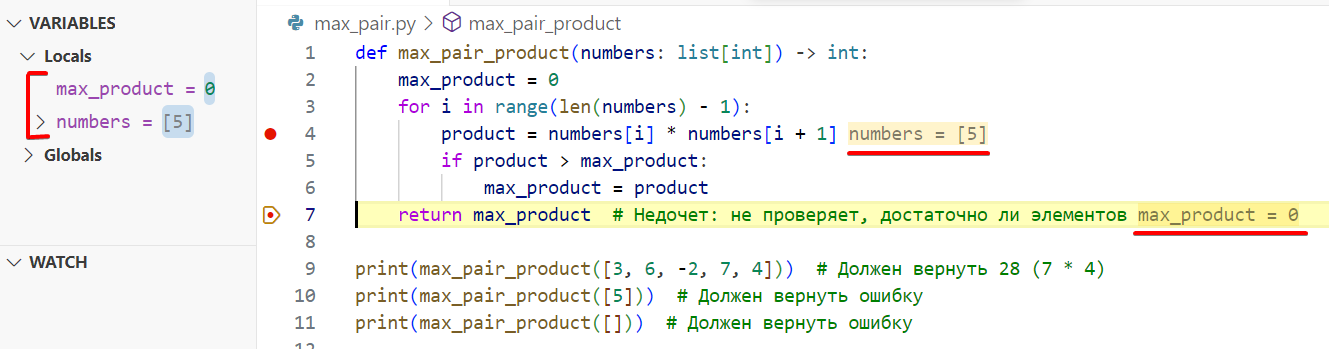
*Рисунок 19 - Остановка в первый раз*

**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.

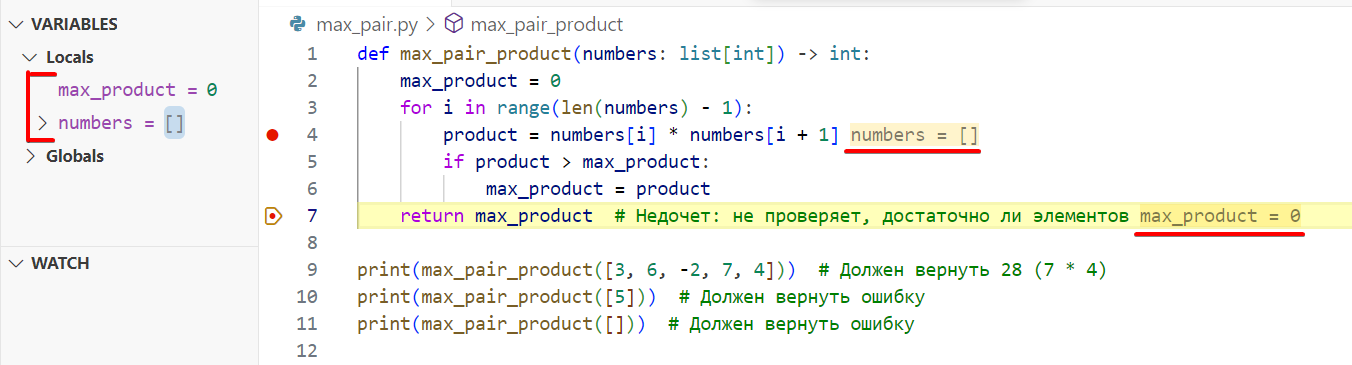


*Рисунок 20 - Первый результат функции*

**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.

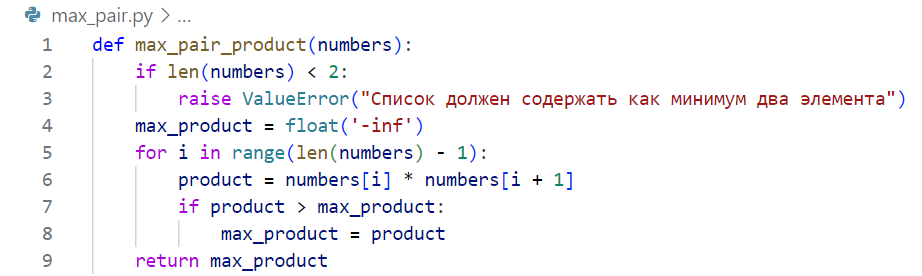


*Рисунок 21 - Второй результат функции*



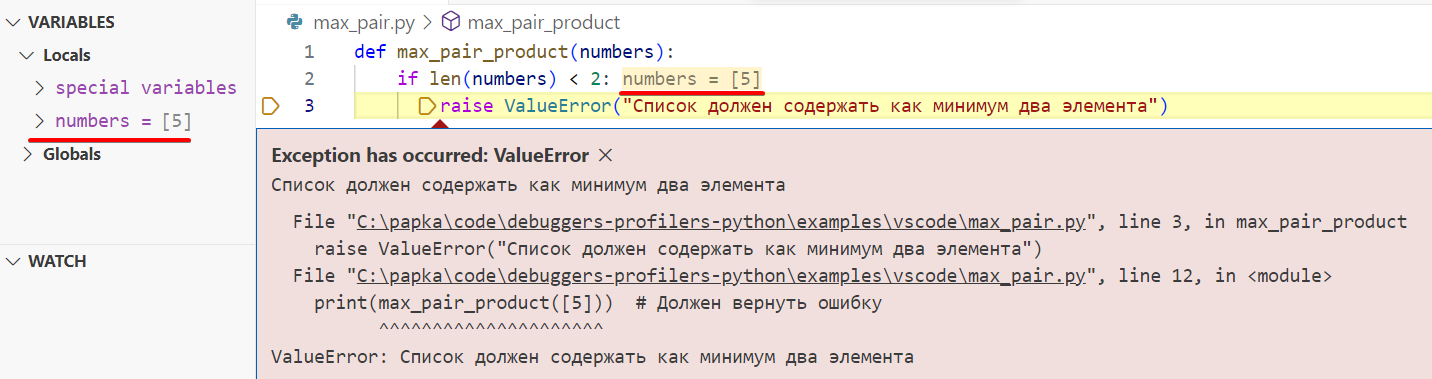
*Рисунок 22 - Третий результат функции*

**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*

#### Jupyter Notebook

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

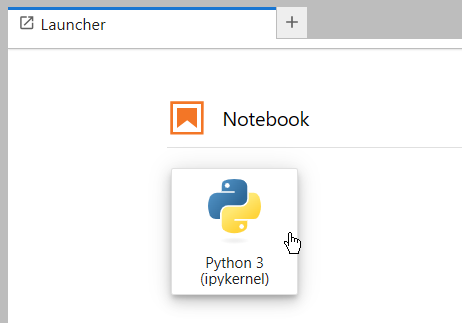
* Магическая команда %debug[10] для запуска отладчика после возникновения исключений.
* Интеграция с pdb для пошаговой отладки через pdb.set\_trace().
* Выполнение кода по ячейкам с немедленным просмотром результатов.
* Интерактивная консоль[11] для проверки переменных и экспериментов.
* Поддержка отображения данных и визуализаций в ячейках

##### Установка

Jupyter Notebook можно установить через официальный сайт[12], anaconda[13] или Docker[14].

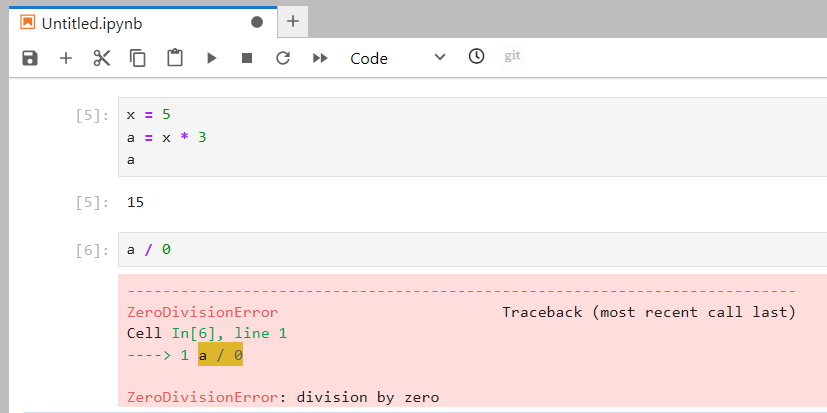
##### Использование через pdb

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



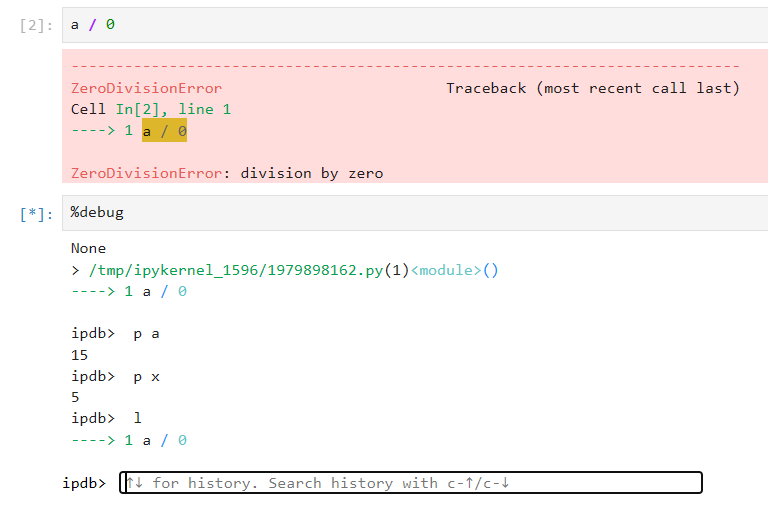
*Рисунок 25 - Создание нового блокнота в Jupyter Notebook*

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



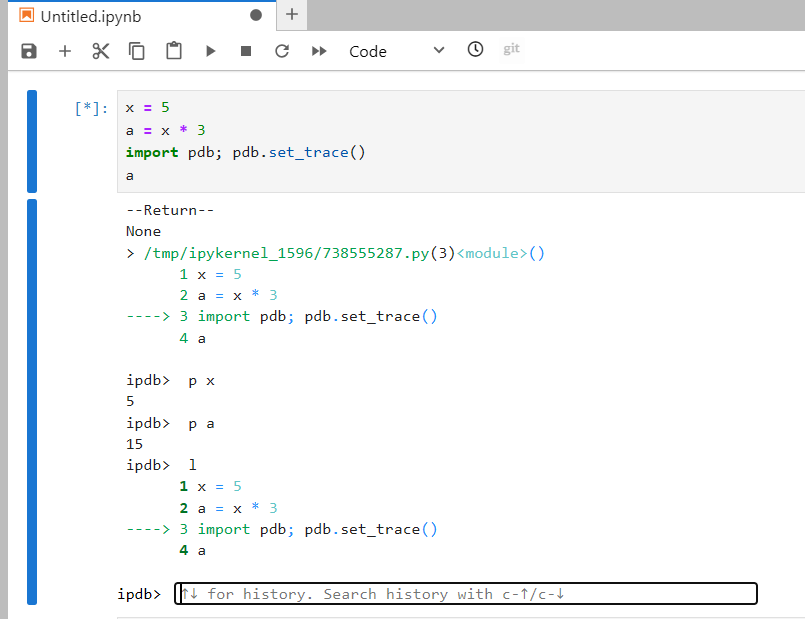
*Рисунок 26 - Запуск кода без отладки*

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.

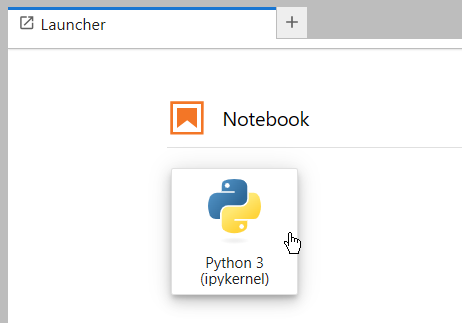


*Рисунок 28 - pdb в середине блока кода*

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.

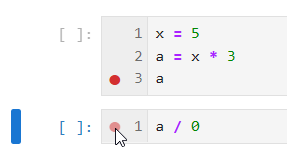


*Рисунок 29 - Создание нового блокнота в Jupyter Notebook*

1. Включить режим отладки и поставить точки остановки.

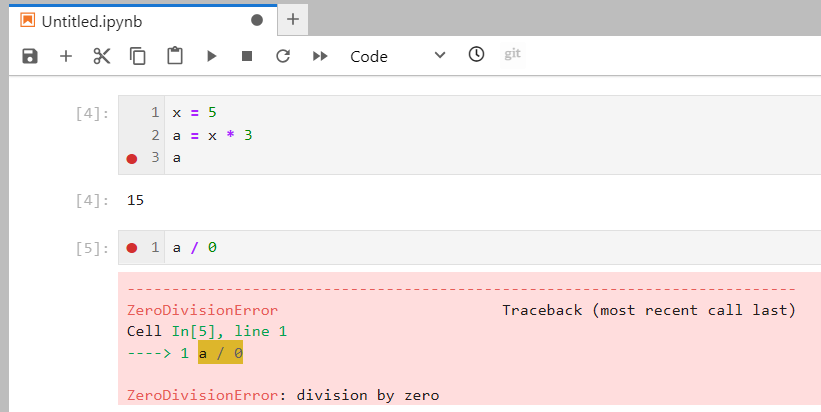


*Рисунок 30 - Включение отладки в Jupyter Notebook*



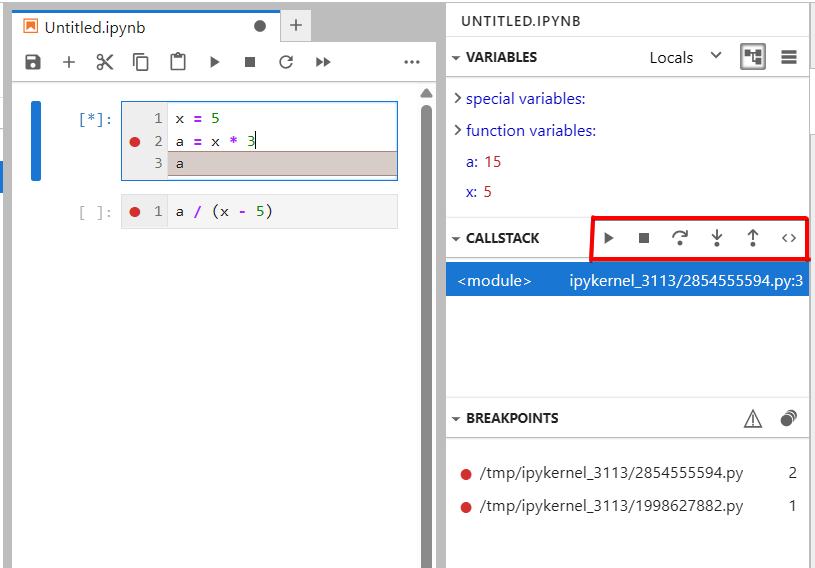
*Рисунок 31 - Установка брейкпоинтов в Jupyter Notebook*

1. Выполнить код с помощью Shift+Enter.

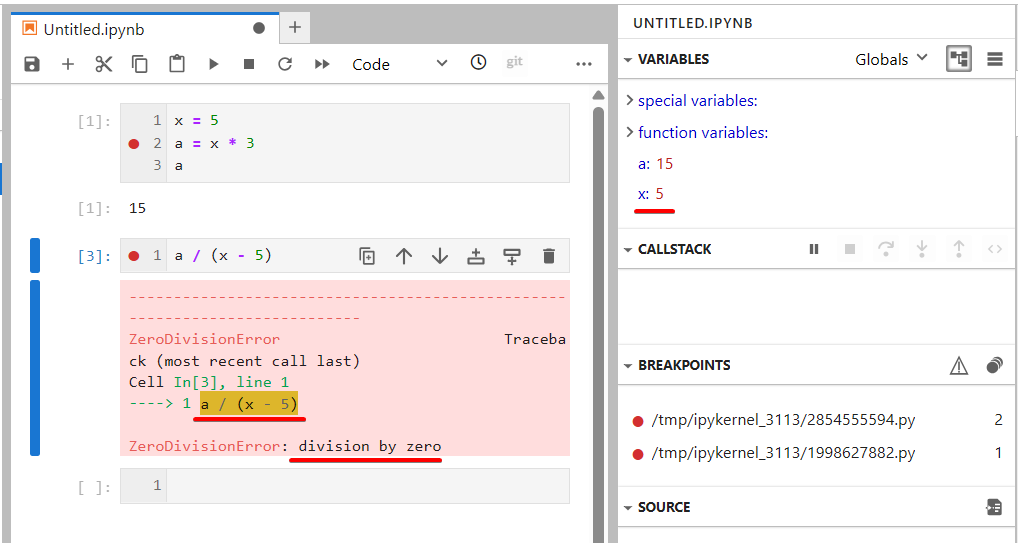


*Рисунок 32 - Запуск кода в режиме отладки*

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*

### Профайлеры

#### cProfile

cProfile[15] - 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[16].

#### timeit

timeit[17] - 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.

#### line\_profiler

line\_profiler[18] - 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 для профилирования.
* Подробные отчеты с процентом времени на каждую строку.

##### 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.

#### memory\_profiler

memory\_profiler[19] - 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.

#### pyheat

pyheat[20] - 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

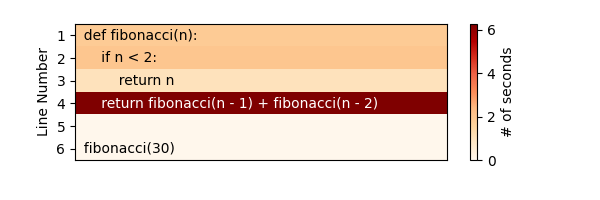
##### Использование

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

### Линтеры

#### flake8

flake8[21] - это инструмент для проверки стиля и качества Python-кода. Он объединяет pycodestyle (проверка PEP8), pyflakes (логические ошибки) и mccabe (сложность кода), предоставляя единый интерфейс для анализа кода.

##### Основные возможности

* Проверка соответствия стандартам стиля (PEP8).
* Выбор конкретных предупреждений/ошибок с помощью --select.
* Игнорирование предупреждений/ошибок с помощью --extend-ignore.
* Конфигурация через командную строку или файлы конфигурации (например, .flake8).

##### Установка

Установка через pip: pip install flake8

##### Использование

Листинг 13 - Использование flake8

>>> flake8 ./examples/logging  
./examples/logging/full.py:26:1: E302 expected 2 blank lines, found 1  
./examples/logging/full.py:65:4: W292 no newline at end of file  
./examples/logging/short.py:3:21: E201 whitespace after '('  
>>> flake8 ./examples/memory\_profiler/short.py  
./examples/memory\_profiler/short.py:11:1: E305 expected 2 blank lines after class or function definition, found 1

##### Плюсы

* Быстрая проверка стиля и логики кода.
* Гибкая конфигурация под нужды проекта.
* Поддержка плагинов для расширения функциональности.

##### Минусы

* Может требовать дополнительных плагинов[22] для сложных проверок.

#### ruff

ruff[23] - это высокопроизводительный линтер и форматтер кода для Python, написанный на Rust. Он в 10-100 раз быстрее альтернатив, таких как flake8 и black, и заменяет множество инструментов, включая flake8, isort, pydocstyle и другие.

##### Основные возможности

* Скорость: 10-100x быстрее существующих линтеров и форматтеров.
* Поддержка более 800 правил, включая реализации плагинов flake8.
* Автоматическое исправление ошибок (например, удаление неиспользуемых импортов).
* Встроенное кэширование для ускорения повторных проверок.
* Плагины для интеграции с редакторами, такими как VS Code[24] и PyCharm[25].

##### Установка

Установка с pip: pip install ruff

##### Использование

Листинг 14 - Код с ошибкой для ruff

import random  
import math  
  
print(math. exp(random.random( )) )

Листинг 15 - Использование ruff

>>> ruff check --select=I ./examples/ruff/short.py  
examples/ruff/short.py:1:1: I001 [\*] Import block is un-sorted or un-formatted  
 |  
1 | / import random  
2 | | import math  
 | |\_\_\_\_\_\_\_\_\_\_\_^ I001  
3 |  
4 | print(math. exp(random.random( )) )  
 |  
 = help: Organize imports  
  
Found 1 error.  
[\*] 1 fixable with the `--fix` option.  
  
>>> ruff check --select=I --fix ./examples/ruff/short.py  
Found 1 error (1 fixed, 0 remaining).

##### Плюсы

* Исключительная скорость (до 1000x быстрее в некоторых случаях).
* Широкое принятие в проектах, таких как FastAPI и Pandas.
* Обширный набор правил и автоматические исправления.
* Интеграция с разными IDE.

##### Минусы

* Нет поддержки пользовательских плагинов (наборов правил).

#### mypy

mypy[26] - это статический анализатор типов для Python, который выявляет ошибки типизации, используя аннотации типов (PEP 484[27]). Он проверяет код без его запуска, что делает его полезным для раннего обнаружения проблем.

##### Основные возможности

* Поддержка вывода типов, generics, callable types, tuple types, union types и структурного подтипирования (Protocol).
* Постепенное введение типов (gradual typing), позволяющее добавлять аннотации постепенно.
* Совместимость с динамической типизацией Python.

##### Установка

Установка с pip: pip install mypy

##### Использование

Листинг 16 - Использование mypy

# examples/mypy/short.py:  
number: int = input("Любимое число?")  
  
>>> mypy ./examples/mypy/short.py  
examples/mypy/short.py:1: error:  
Incompatible types in assignment  
(expression has type "str", variable has type "int")  
[assignment]

##### Плюсы

* Упрощает понимание, отладку и поддержку кода.
* Поддерживает постепенное добавление типов.

##### Минусы

* Возможны редкие нарушения обратной совместимости.
* Требует аннотаций типов для максимальной эффективности.

## 3. Тестирование и оценка результатов выполнения работы.

В этой главе мы практически применим инструменты отладки, профилирования и статического анализа, описанных в предыдущих главах, на примере около реального сценария. Возьмём скрипт для обработки числовых данных из csv-файла, который содержит ошибки и возможность оптимизации. Используя встроенный в PyCharm отладчик, cProfile, logging, ruff и mypy, попробуем устранить проблемы, увеличить производительность и улучшить качество кода.

### Задача

Рассмотрим программу, которая читает файл с числовыми данными (одно число в строке), вычисляет статистические показатели (среднее, медиану, стандартное отклонение) и записывает результаты в другой файл.

### Первая версия

Листинг 17 - Первая версия программы

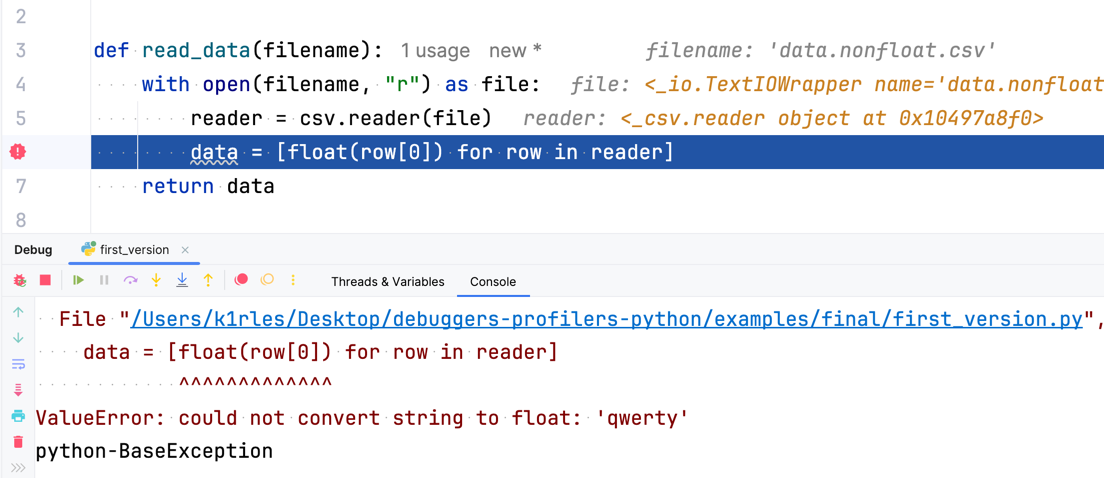
import csv  
  
def read\_data(filename):  
 with open(filename, "r") as file:  
 reader = csv.reader(file)  
 data = [float(row[0]) for row in reader]  
 return data  
  
def calculate\_mean(data):  
 total = 0  
 for num in data:  
 total += num  
 mean = total / len(data)  
 return mean  
  
def calculate\_median(data):  
 sorted\_data = sorted(data)  
 n = len(sorted\_data)  
 if n % 2 == 0:  
 median = (sorted\_data[n // 2 - 1] + sorted\_data[n // 2]) / 2  
 else:  
 median = sorted\_data[n // 2]  
 return median  
  
def calculate\_std\_dev(data):  
 mean = calculate\_mean(data)  
 variance = sum((x - mean) \*\* 2 for x in data) / len(data)  
 std\_dev = variance \*\* 0.5  
 return std\_dev  
  
def write\_results(filename, mean, median, std\_dev):  
 with open(filename, "w") as file:  
 file.write(f"Среднее: {mean}\n")  
 file.write(f"Медиана: {median}\n")  
 file.write(f"Стандартное отклонение: {std\_dev}\n")  
  
if \_\_name\_\_ == "\_\_main\_\_":  
 data = read\_data("data.empty.csv")  
 mean = calculate\_mean(data)  
 median = calculate\_median(data)  
 std\_dev = calculate\_std\_dev(data)  
 write\_results("results.txt", mean, median, std\_dev)

Скрипт заранее содержит следующие проблемы:  
- Ошибки в расчётах: стандартное отклонение считается неправильно, а нечисловые данные в файле ломают программу.  
- Нет проверки на отсутствие данных в исследуемом файле.  
- Вопросы к производительности: чтение больших файлов с помощью csv.reader выполняется медленно.

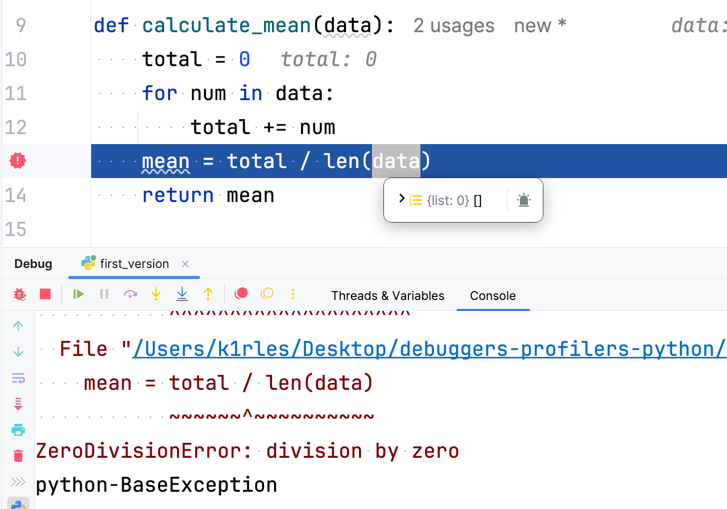
Предположим, что файл data.csv содержит числа, но некоторые строки содержат нечисловые данные, например, текст "Н/д". Это вызовет сбой программы.  
Или data.csv не содержит данных, тогда программа завершится с ошибкой ZeroDivisonError.

### Отладка с помощью PyCharm

Для поиска и устранения ошибок используем отладчик PyCharm. Запустим скрипт в режим отладки, тогда PyCharm покажет нам исключение и строку, на которой оно произошло.



*Рисунок 36 - Исключение ValueError*



*Рисунок 37 - Исключение ZeroDivisonError*

Исправление кода: Модифицируем функцию read\_data, добавив обработку нечисловых значений. Также добавим условие - если файл пустой, то сразу запишем в ответ нули.

Листинг 18 - Вторая версия программы

import csv  
import logging  
import timeit  
  
logging.basicConfig(  
 level=logging.INFO,  
 format='%(asctime)s - %(levelname)s - %(message)s',  
)  
  
def read\_data(filename):  
 data = []  
 skip = 0  
 with open(filename, "r") as file:  
 reader = csv.reader(file)  
 for row in reader:  
 try:  
 num = float(row[0])  
 data.append(num)  
 except (ValueError, IndexError):  
 skip += 1  
 logging.info("Пропущено %d строк", skip)  
 return data  
  
def calculate\_mean(data):  
 total = 0  
 for num in data:  
 total += num  
 mean = total / len(data)  
 return mean  
  
def calculate\_median(data):  
 sorted\_data = sorted(data)  
 n = len(sorted\_data)  
 if n % 2 == 0:  
 median = (sorted\_data[n // 2 - 1] + sorted\_data[n // 2]) / 2  
 else:  
 median = sorted\_data[n // 2]  
 return median  
  
def calculate\_std\_dev(data):  
 mean = calculate\_mean(data)  
 variance = sum((x - mean) \*\* 2 for x in data) / len(data)  
 std\_dev = variance \*\* 0.5  
 return std\_dev  
  
def write\_results(filename, mean, median, std\_dev):  
 with open(filename, "w") as file:  
 file.write(f"Среднее: {mean}\n")  
 file.write(f"Медиана: {median}\n")  
 file.write(f"Стандартное отклонение: {std\_dev}\n")  
  
def main():  
 logging.info("Начало обработки файла")  
 data = read\_data("data.csv")  
 if data:  
 mean = calculate\_mean(data)  
 median = calculate\_median(data)  
 std\_dev = calculate\_std\_dev(data)  
 write\_results("results.txt", mean, median, std\_dev)  
 else:  
 logging.warning("Файл пуст или не содержит числовых данных")  
 write\_results("results.txt", 0, 0, 0)  
 logging.info("Обработка завершена")  
  
if \_\_name\_\_ == "\_\_main\_\_":  
 logging.info(timeit.timeit(main, number=1))

### Отладка с помощью cProfile

Предположим, что нам нужно посчитать файл на 100 миллионов чисел. Сейчас работа программы занимает ~20 секунд. Попробуем запустить cProfile и узнать, что выполняется дольше всего.

Листинг 19 - Вывод без cProfile

2025-05-31 02:34:37,219 - INFO - Начало обработки файла  
2025-05-31 02:34:49,189 - INFO - Пропущено 0 строк  
2025-05-31 02:35:00,179 - INFO - Обработка завершена  
2025-05-31 02:35:00,622 - INFO - 23.4025829169841

Листинг 20 - Вывод cProfile

>>> python -m cProfile second\_version.py  
INFO:root:39.309911207994446  
200223976 function calls (200223699 primitive calls) in 39.314 seconds  
Ordered by: cumulative time  
ncalls tottime percall cumtime percall filename:lineno(function)  
 1 17.390 17.390 21.359 21.359 second\_version.py:6(read\_data)  
 1 0.000 0.000 11.130 11.130 second\_version.py:36(calculate\_std\_dev)  
 2 3.852 1.926 3.852 1.926 second\_version.py:20(calculate\_mean)  
 1 0.000 0.000 3.529 3.529 second\_version.py:27(calculate\_median)

Сразу видим, что вместо 20 секунд программа выполнялась почти 40, то есть в два раза дольше. Так профайлер влияет на производительность.  
Заметим, что самые долгие функции - read\_data и calculate\_std\_dev, а calculate\_mean вызывается два раза. Попробуем взять более подходящий инструмент для анализа больших данных - pandas. Установим его через pip install pandas и перепишем наши функции:

Листинг 21 - Третья версия программы

import logging  
import timeit  
from functools import partial  
  
import pandas as pd  
from pandas import Series  
  
logging.basicConfig(level=logging.INFO, format='%(asctime)s - %(levelname)s - %(message)s')  
  
def process\_csv(input\_file, output\_file):  
 logging.info(f'Чтение файла {input\_file}')  
 df: pd.DataFrame = pd.read\_csv(input\_file, header=None, names=['value'], on\_bad\_lines='skip')  
 numbers: Series = df['value'].dropna()  
  
 if numbers.empty:  
 logging.warning("Файл пуст или не содержит числовых данных")  
 with open(output\_file, 'w') as f:  
 f.write("Среднее: 0\nМедиана: 0\nСтандартное отклонение: 0\n")  
 return  
  
 logging.info(f'Обработано {len(numbers):\_} чисел')  
 mean: float = numbers.mean()  
 median: float = numbers.median()  
 std\_dev: float = numbers.std()  
  
 logging.info('Вычисление завершено, запись результатов')  
 with open(output\_file, 'w') as f:  
 f.write(f"Среднее: {mean}\nМедиана: {median}\nСтандартное отклонение: {std\_dev}\n")  
 logging.info(f'Результаты записаны в {output\_file}')  
  
if \_\_name\_\_ == "\_\_main\_\_":  
 logging.info(  
 timeit.timeit(  
 partial(process\_csv, "data.csv", "results.txt"),  
 number=1  
 )  
 )

Листинг 22 - Вывод при работе программы с pandas

2025-05-31 02:31:32,453 - INFO - Чтение файла data.csv  
2025-05-31 02:31:35,630 - INFO - Обработано 100\_000\_000 чисел  
2025-05-31 02:31:36,811 - INFO - Вычисление завершено, запись результатов  
2025-05-31 02:31:36,812 - INFO - Результаты записаны в results.txt  
2025-05-31 02:31:36,832 - INFO - 4.3786993749963585

### Форматирование и аннотации типов

Теперь, когда программа работает за приемлемое время, можно подумать о качестве кода. Создадим .ruff.toml и .mypy.ini:

Листинг 23 - Пример конфига для ruff

line-length = 88  
lint.select = ["ALL"]  
lint.ignore = ["RUF001", "PTH", "LOG", "D"]

Листинг 24 - Пример конфига для mypy

[mypy]  
strict = True

Скопируем третью версию в другой файл и отформатируем его: ruff check --fix ./fourth\_version.py, mypy ./fourth\_version.py

После исправления предупреждений ruff и mypy, получим последнюю версию скрипта:

Листинг 25 - Четвёртая версия программы

import logging  
import timeit  
from functools import partial  
  
import pandas as pd  
from pandas import Series  
  
logging.basicConfig(  
 level=logging.INFO,  
 format="%(asctime)s - %(levelname)s - %(message)s",  
)  
  
def process\_csv(input\_file: str, output\_file: str) -> None:  
 logging.info("Чтение файла %s", input\_file)  
 df: pd.DataFrame = pd.read\_csv(  
 input\_file,  
 header=None,  
 names=["value"],  
 on\_bad\_lines="skip",  
 )  
 numbers: Series[float] = df["value"].dropna()  
  
 if numbers.empty:  
 logging.warning("Файл пуст или не содержит числовых данных")  
 with open(output\_file, "w") as f:  
 f.write("Среднее: 0\nМедиана: 0\nСтандартное отклонение: 0\n")  
 return  
  
 logging.info("Обработано %d:\_ чисел", len(numbers))  
 mean: float = numbers.mean()  
 median: float = numbers.median()  
 std\_dev: float = numbers.std()  
  
 logging.info("Вычисление завершено, запись результатов")  
 with open(output\_file, "w") as f:  
 f.write(  
 f"Среднее: {mean}\nМедиана: {median}\n"  
 f"Стандартное отклонение: {std\_dev}\n",  
 )  
 logging.info("Результаты записаны в %s", output\_file)  
  
if \_\_name\_\_ == "\_\_main\_\_":  
 logging.info(  
 timeit.timeit(  
 partial(process\_csv, "data.csv", "results.txt"),  
 number=1,  
 ),  
 )

### Оценка результатов

Мы довели скрипт до рабочего состояния, решив все основные проблемы:

* Ошибки: добавили обработку некорректных данных и пустых файлов.
* Производительность: Переход на pandas сократил время выполнения с 20 секунд до 5 секунд на файле с 100 миллионами строк.
* Качество кода: ruff привёл код в соответствие с PEP 8, а mypy подтвердил типобезопасность.
* Прозрачность: logging сделал процесс выполнения понятным, с чёткими сообщениями о каждом этапе.

Вклад каждого инструмента:

* PyCharm Debugger: Помог быстро найти ошибки в обработке данных и расчётах. Интуитивный интерфейс и пошаговое выполнение сэкономили время.
* cProfile: Показал, что чтение файла и ручные вычисления были узкими местами, что привело к переходу на pandas.
* logging: Сделал процесс прозрачным, особенно полезно для больших файлов, где нужно отслеживать пропущенные строки.
* ruff: Устранил стилистические недочёты, сделав код аккуратным и удобным для команды (а в команде без единого стиля туго).
* mypy: Гарантировал, что код не сломается из-за ошибок типов.

### Вывод

Этот пример показал, как комплексное использование инструментов отладки, профилирования и статического анализа превращает проблемный скрипт в надёжный и быстрый инструмент.

Полученные результаты применимы к реальным задачам, от анализа данных до серверных приложений. Эти инструменты не только решают конкретные проблемы, но и учат писать качественный код, который легче поддерживать и масштабировать. Теперь скрипт готов к использованию в реальных проектах, где важны скорость, надёжность и читаемость.

# 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.

# Источники

[1] pdb. – URL: https://docs.python.org/3/library/pdb.html (дата обращения: 31.05.2025).

[2] print. – URL: https://docs.python.org/3/library/functions.html#print (дата обращения: 31.05.2025).

[3] logging. – URL: https://docs.python.org/3/library/logging.html (дата обращения: 31.05.2025).

[4] pytest. – URL: https://docs.pytest.org/en/stable/ (дата обращения: 31.05.2025).

[5] unittest. – URL: https://docs.python.org/3/library/unittest.html (дата обращения: 31.05.2025).

[6] PyCharm. – URL: https://www.jetbrains.com/pycharm/ (дата обращения: 31.05.2025).

[7] JetBrains. – URL: https://www.jetbrains.com/pycharm/ (дата обращения: 31.05.2025).

[8] Visual Studio Code (VS Code). – URL: https://code.visualstudio.com/ (дата обращения: 31.05.2025).

[9] официального сайта. – URL: https://code.visualstudio.com/download (дата обращения: 31.05.2025).

[10] %debug. – URL: https://www.cambridge.org/core/resources/pythonforscientists/jupyterdb (дата обращения: 31.05.2025).

[11] Интерактивная консоль Jupyter Notebook. – URL: https://jupyterlab.readthedocs.io/en/latest/user/debugger.html (дата обращения: 31.05.2025).

[12] официальный сайт. – URL: https://jupyter.org/install (дата обращения: 31.05.2025).

[13] anaconda для Jupyter Notebook. – URL: https://anaconda.org/anaconda/jupyter (дата обращения: 31.05.2025).

[14] Docker для Jupyter Notebook. – URL: https://jupyter-docker-stacks.readthedocs.io/en/latest/ (дата обращения: 31.05.2025).

[15] cProfile. – URL: https://docs.python.org/3/library/profile.html#module-cProfile (дата обращения: 31.05.2025).

[16] SnakeViz. – URL: https://jiffyclub.github.io/snakeviz/ (дата обращения: 31.05.2025).

[17] timeit. – URL: https://docs.python.org/3/library/timeit.html (дата обращения: 31.05.2025).

[18] line\_profiler. – URL: https://pypi.org/project/line-profiler/ (дата обращения: 31.05.2025).

[19] memory\_profiler. – URL: https://pypi.org/project/memory-profiler/ (дата обращения: 31.05.2025).

[20] pyheat. – URL: https://pypi.org/project/py-heat/ (дата обращения: 31.05.2025).

[21] flake8. – URL: https://flake8.pycqa.org/en/latest/ (дата обращения: 31.05.2025).

[22] плагинов. – URL: https://github.com/DmytroLitvinov/awesome-flake8-extensions (дата обращения: 31.05.2025).

[23] ruff. – URL: https://docs.astral.sh/ruff/ (дата обращения: 31.05.2025).

[24] VS Code. – URL: https://github.com/astral-sh/ruff-vscode (дата обращения: 31.05.2025).

[25] PyCharm. – URL: https://plugins.jetbrains.com/plugin/20574-ruff (дата обращения: 31.05.2025).

[26] mypy. – URL: https://mypy.readthedocs.io/en/stable/ (дата обращения: 31.05.2025).

[27] PEP 484. – URL: (https://peps.python.org/pep-0484/) (дата обращения: 31.05.2025).