

MINI DOOM

Современный Python через игру

```
EL = [
    "#####",
    "#@..#.z.h.X#",
    "#.#.#.###..#",
    "#h..i.....#",
    "#####",

    taiclass(frozen=True)
    ss Pos:
        x: int
        y: int

        def step(self, dx: int, dy: int) -> Pos:
            return Pos(self.x + dx, self.y + dy)

        def near(self, other: Pos) -> bool:
            return abs(self.x - other.x) + abs(self.y - other.y) == 1

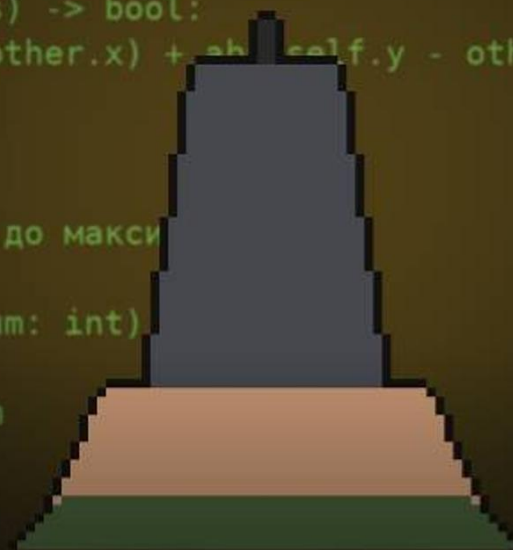
    ss Health:
        """Здоровье: всегда от 0 до макси

        def __init__(self, maximum: int):
            self._max = maximum
            self._value = maximum

        @property
        def value(self) -> int:
            return self._value

        @property
        def TOM 1 -> bool:
            return self._value > 0

        def value(self, value: int):
            self._value = value
```



ТОМ 1

Учимся держать оружие

Python 3.15 · pygame-ce · numpy



MINI DOOM Современный Python через игру

Алексей Примак

**MINI DOOM Современный
Python через игру Том 1
Учимся держать оружие**

«Автор»

2026

Примак А.

MINI DOOM Современный Python через игру Том 1 Учимся
держаться / А. Примак — «Автор», 2026 — (MINI DOOM
Современный Python через игру)

Первый том серии «MINI DOOM. Современный Python через игру». Ты с нуляходишь в программирование на Python 3.15: ставишь Python и uv, пишешь первую программу и рисуешь титульный экран будущей игры. Дальше — переменные и типы, условия и циклы, функции и модули, списки, словари и строки, дата классы, классы и протоколы, а в конце — как Python хранит объекты и почему два имени могут смотреть на один список. К концу тома у тебя консольный рогаик с картой, монстрами и боем. С первой главы код проверяют ruff и муру --strict, как в настоящих проектах. 8 глав, 113 листингов с настоящим выводом, задачи с ответами, шпаргалка по Python и разбор частых ошибок. Код — на GitHub Для тех, кто никогда не программировал или хочет освоить Python на живом проекте. На «ты», с юмором. 18+.

Содержание

Предисловие	5
Для кого эта книга	6
Почему Doom и почему Python	7
Как устроена книга	8
Как читать листинги	9
Цветные врезки	10
Что понадобится	11
Архив примеров	12
Поехали	13
Пять томов	14
Часть I. Учимся держать оружие	15
Глава 1. Hello, Doom	16
Что такое программа и зачем нужен интерпретатор	17
Ставим Python	18
Редактор и первый проект	19
Hello, Doom	20
Рисуем титульный экран	22
Когда Python ругается	24
Настраиваем строгость	28
Итоги главы	30
Упражнения	31
Задачи	32
Глава 2. Переменные и типы	33
Переменная — бирка на коробке	34
Какие бывают типы	36
Числа, которые не ломаются	38
Деление: три разных оператора	39
Дроби и округление	40
Конец ознакомительного фрагмента.	41

MINI DOOM Современный Python через игру Том 1 Учимся держать оружие

Предисловие

Привет. Если ты держишь эту книгу, значит, тебе хочется не просто «выучить Python», а **сделать что-то своё**. Отлично, потому что именно этим мы и займёмся. За двадцать девять глав мы с тобой с нуля напишем игру в духе Doom: с коридорами, дверями, импами, которые швыряются огнём, зомби с винтовками, дробовиком и статус-баром внизу экрана. Без готовых движков, без скачанных картинок и звуков — всё своими руками, строчка за строчкой.



Рисунок 0.1 — Так выглядит MINI DOOM в конце книги: титульный экран, имп в проёме, дробовик в деле, второй уровень. Ни одной картинки из интернета — всё нарисовано кодом

Посмотри на рисунок 0.1. Каждый пиксель на нём — результат кода, который ты напишешь. Текстура кирпича, рожа импа, огонь на титульнике и даже звук выстрела — всё это генерируется программой при запуске. Звучит как магия, но к концу книги ты будешь знать, как работает каждый фокус.

Для кого эта книга

Для тех, кто **ни разу не программировал** или пробовал, но бросил на третьей странице скучного учебника. Я не буду требовать от тебя знаний математики сверх школьной, а ту, что понадобится, — синус, косинус, векторы, — объясню на пальцах прямо тогда, когда она понадобится, и ни минутой раньше.

Если ты уже немного знаешь Python, тоже не уходи. Первую часть можешь пролистать по диагонали, а со второй начнётся то, чего в обычных учебниках нет: своё окно, пиксели на пипру, псевдо-3D, искусственный интеллект монстров, синтез звука, тесты и профилировщики. Здесь Python не «язык для скриптов», а язык, на котором пишут настоящую игру — и разбираются, почему она тормозит и как её ускорить.

Книга маркирована **18+**. Не потому, что тут кровь и кишки (хотя импы истекают пиксельной кровью), а потому что я разговариваю с тобой как со взрослым: иногда крепким словом, когда программа ведёт себя как последняя сволочь. Дозированно, как соль в супе.

Почему Doom и почему Python

Потому что Doom — идеальный учебный проект. Он достаточно простой, чтобы один человек мог написать его за книгу, и достаточно сложный, чтобы по дороге встретилось всё важное: списки и словари, классы и датаклассы, массивы `numpy` и алгоритмы, конечные автоматы, файлы и исключения, звук и тесты. А ещё потому, что стрелять по демонам веселее, чем считать среднюю зарплату сотрудников в отделе, как в половине учебников.

А Python — потому что на нём проще всего начать и дальше всего можно уйти. Говорят, что Python медленный и игры на нём не пишут. Отчасти правда: цикл по пикселям на чистом Python действительно в сотню раз медленнее, чем надо. Но мы научимся думать массивами и отдавать тяжёлую работу `numpy`, мерить время профилировщиком и находить, где тормозит на самом деле. Это самый ценный навык, который можно вынести из книги, — и он пригодится далеко за пределами игр: в анализе данных, машинном обучении, науке.

Лирика. Оригинальный Doom вышел в декабре 1993 года. Его написала крошечная студия id Software, а движок сделал Джон Кармак — человек, который выжимал трёхмерную графику из компьютеров, где у процессора не было даже нормальной арифметики с плавающей точкой. Наш MINI DOOM устроен проще настоящего: это рейкастер, ближе к его старшему брату Wolfenstein 3D. Зато мы напишем его на современном Python 3.15 — интерпретируемом языке, о котором в 1993 году ещё никто не слышал, — и он будет работать в реальном времени.

Как устроена книга

Книга разбита на пять частей. Каждая следующая опирается на предыдущую, поэтому читать лучше по порядку.

Часть	Что изучаем	Что появляется в игре
I. Учимся держать оружие	основы Python: переменные, условия, циклы, функции, коллекции, классы, ссылки	консольный рогалик: ходим по карте, бьём монстров
II. Окно в ад	pygame-се, пиксели на питру, игровой цикл, клавиатура и мышь, свой шрифт	окно, в котором что-то движется и слушается мыши
III. Третье измерение	векторы, рейкастинг, текстуры кодом, свет, профилирование, двери	трёхмерный лабиринт с текстурами и дверями
IV. Здесь кто-то есть	спрайты, z-буфер, @cache, hitscan, конечные автоматы, поиск в ширину	монстры, которые охотятся на тебя, и оружие, чтобы им ответить
V. Шум, свет и финальные титры	синтез звука, потоки и GIL, экраны, файлы и исключения, тесты	законченная игра из двух уровней

Таблица 0.1 — Пять частей книги

Внутри главы всё идёт по одному сценарию: сначала объясняю идею на пальцах, потом пишем код, потом смотрим на настоящий вывод программы и разбираем его, потом — грабли, на которые наступают все, и в конце упражнения и задачи. Ответы на них собраны в конце книги.

И главное правило: **каждая глава заканчивается работающей программой**. Не «допишем в следующей главе», а прямо сейчас: запустил — увидел результат. Я называю это контрольной точкой — как в игре, где можно сохраниться.

Как читать листинги

Код в книге выглядит как окно редактора. Сверху — имя файла и язык, слева — номера строк, на которые я ссылаюсь в тексте. Под кодом тёмная панель: это то, что программа напечатала, когда я её запустил. Вот пример — листинг 0.1.

```

1  """Как выглядит листинг в этой книге."""
2
3  hp = 100
4  for damage in (20, 35, 50):
5      hp -= damage
6      print(f"имп попал: -{damage}, здоровье {max(hp, 0)}%")
7  print("YOU DIED" if hp <= 0 else "живой")

$ python3 main.py
имп попал: -20, здоровье 80%
имп попал: -35, здоровье 45%
имп попал: -50, здоровье 0%
YOU DIED

```

Листинг 0.1 — Как выглядит листинг: код, номера строк и вывод программы

Первая зелёная строка в тёмной панели, «\$ python3 main.py», — команда запуска. На Windows та же команда пишется `python main.py`, а проще всего — `uv run main.py`: об этом в главе 1. Строки, начинающиеся с решётки, — мои комментарии, программа их не печатала.

Весь вывод в книге **настоящий**. Каждый пример я запускал на Python 3.15 и проверял двумя строгими сторожами: линтером `ruff` с большим набором правил и проверкой типов `tuuru` в строгом режиме. Если у тебя вывод отличается — скорее всего, отличается и код. Сверься с архивом примеров. Исключение — замеры времени: у тебя другой компьютер, и миллисекунды будут другими, а вот соотношения — такими же.

Если Python отказывается запускать программу, панель подписана «интерпретатор ругается». Такие листинги я показываю специально: читать трейсбеки — такой же навык, как писать код. Окна игры в книге сняты «роботом» — программой, которая нажимает клавиши и двигает мышь вместо человека, — поэтому каждый скриншот тоже настоящий и повторяемый. Как это устроено, ты узнаешь в главе 9.

Цветные врезки

По тексту разбросаны цветные врезки четырёх видов. Вот они все.

Грабли. Место, где все ломают ноги. Если видишь красную врезку — читай внимательно, тут сэкономишь себе вечер отладки.

Запомни. Короткое правило, которое стоит выучить наизусть. Например: в Python отступы — это часть синтаксиса, а не украшение.

Лирика. Байка, история или шутка. Можно пропустить, но не советую — иногда здесь объясняется, почему всё устроено именно так.

По-взрослому. Как это делают в настоящих проектах, в продакшене. Не обязательно для понимания, но отличает новичка от профи.

Что понадобится

- Компьютер с Windows, macOS или Linux. Игра работает везде, где есть Python.
 - Python 3.15 и менеджер uv — как их поставить, расскажу в главе 1.
 - Редактор Visual Studio Code с расширениями Python и Ruff. Можно и любой другой, но объяснять я буду на нём.
 - Две библиотеки: numru и rугame-се. Их uv поставит одной командой в главе 9.
 - Желание разобраться. Серьёзно, это важнее всего остального.
- Мышь тоже пригодится: в Doom без мыши играть можно, но это как есть суп вилкой. И колонки — в главе 26 игра заговорит.

Архив примеров

Весь код из книги лежит на GitHub: <https://github.com/RushSex/MiniDoom-Python>. Скачать его можно одним архивом — зелёная кнопка Code, затем Download ZIP — или командой `git clone`; дальше в книге я называю его просто архивом примеров. В папке `examples` там по папке на каждую главу — от `ch01` до `ch29`, — и в каждой папке программы в том виде, в каком они должны быть в конце главы. Готовая игра целиком — в папке `minidoom`: `uv sync`, `uv run main.py` — и вперёд.

Но не превращай архив в копиясту. Код, набранный своими пальцами, запоминается в десять раз лучше, а опечатки, которые ты при этом наделаешь, научат читать трейсбеки. Это не баг обучения, это его фиша.

Поехали

Хватит предисловий. Ставим Python, пишем первую программу и заставляем компьютер сказать «Hello, Doom!». Переворачивай страницу.

Пять томов

Книга «MINI DOOM. Современный Python через игру» выходит в пяти томах — по одному на каждую часть. Каждый заканчивается работающей программой, и в каждом свои ответы, глоссарий и указатель.

— **Том 1. Учимся держать оружие** — главы 1–8: основы Python в консоли, от первой программы до ссылок и ресурсов.

— **Том 2. Окно в ад** — главы 9–13: окно pygame-се, пиксели на пумру, игровой цикл, клавиатура и мышь, свой шрифт.

— **Том 3. Третье измерение** — главы 14–19: рейкастинг, стены, текстуры, пол и свет, профилировщик, двери.

— **Том 4. Здесь кто-то есть** — главы 20–25: спрайты, монстры, оружие, конечные автоматы, поиск пути, смерть.

— **Том 5. Шум, свет и финальные титры** — главы 26–29: звук, экраны, уровни в файлах, uv, ruff, муру и pytest.

Все исходники серии — на GitHub: <https://github.com/RushSex/MiniDoom-Python>. Папка `examples/chNN` для каждой главы, готовая игра — папка `minidoom`. Там же README: `uv sync`, `uv run main.py` — и вперёд.

Часть I. Учимся держать оружие

Прежде чем стрелять по демонам, надо научиться держать оружие. В этой части мы освоим основы Python в самой простой обстановке — в окне терминала. Никакой графики, никаких окон, только ты, код и интерпретатор. Зато к концу части у тебя будет консольный рогалик: герой бежит по карте, монстры бродят вокруг, а код устроен так, как пишут настоящие программисты.

Глава 1. Hello, Doom

Любое путешествие начинается с первого шага, а любой программист — с программы, которая печатает «Hello, World!». Мы поступим так же, только поздороваемся не с миром, а с Doom. В этой главе ты поставишь Python и редактор, создашь первый проект, настроишь его как профи, нарисуешь титульный экран игры буквами и, что не менее важно, научишься не бояться, когда Python начинает ругаться.

Что такое программа и зачем нужен интерпретатор

Процессор твоего компьютера — тупой, но очень быстрый исполнитель. Он понимает только машинный код: длинные последовательности чисел вроде «возьми число отсюда, прибавь вот это, положи туда». Писать так руками можно, люди так и делали в пятидесятых, но удовольствия примерно как от заполнения налоговой декларации на латыни.

Поэтому придумали языки программирования. Ты пишешь текст, понятный человеку, а специальная программа переводит его для машины. В языках вроде C++ переводчик — **компилятор** — переводит весь текст заранее и выдаёт готовый .exe. В Python всё устроено иначе: переводчик сидит рядом с поваром и читает ему рецепт вслух, строчка за строчкой, прямо во время готовки. Такой переводчик называется **интерпретатор**, и он тоже называется python.

Если заглянуть под капот, интерпретатор сначала быстро переводит твой файл в промежуточный **байт-код** — компактные команды для виртуальной машины Python — и потом выполняет их одну за другой. Весь путь показан на рисунке 1.1.



Рисунок 1.1 — Путь от текста программы до результата: перевод и выполнение идут сразу, при каждом запуске

У такого подхода есть плюс и минус, и оба нам важны. Плюс: написал строчку — сразу запустил, никакой долгой сборки. Минус: многие ошибки интерпретатор замечает, только когда до них **дойдёт**. Опечатка в пятой строке не мешает выполниться первым четырём, а программа упадёт посреди работы. В C++ такую ошибку поймал бы компилятор до запуска. Поэтому в Python мы позвоём на помощь двух сторожей — линтер и проверку типов. Знакомство с ними — в этой же главе.

Запомни. Интерпретатор выполняет программу строку за строкой. Ошибка в коде, до которого выполнение не дошло, никак себя не проявит — до поры до времени. Поэтому код проверяют до запуска специальными инструментами.

Ставим Python

Нам нужен свежий Python — версии 3.15, она вышла в октябре 2026 года. Подойдёт и любая более новая. Ставить его будем через **uv** — это современный менеджер Python от команды, которая сделала линтер Ruff. Он ставит сам Python нужной версии, инструменты для него и библиотеки, и делает это быстро. Старый способ — скачать установщик с python.org и не забыть галочку Add python.exe to PATH — тоже работает, но с uv меньше граблей.

— Открой PowerShell (на Windows) или терминал (на Linux и macOS).

— Поставь uv. На Windows: powershell -c «irm https://astral.sh/uv/install.ps1 | iex». На Linux и macOS: curl -LsSf https://astral.sh/uv/install.sh | sh. Точные команды всегда есть на сайте docs.astral.sh/uv.

— Закрой терминал и открой заново, чтобы он увидел uv. Потом поставь Python: uv python install 3.15.

— Поставь двух сторожей: uv tool install ruff и uv tool install mypy. О них — чуть ниже.

Проверить, что всё на месте, можно командами python --version, ruff --version и mypy --version. Каждая должна напечатать номер версии, а не «команда не найдена».

Грабли. На Windows команда python иногда открывает Microsoft Store вместо того, чтобы запустить Python. Это «заглушка» самой Windows: настоящего Python в системе ещё нет или он не прописан в PATH. Лечится так: Параметры → Приложения → Дополнительные параметры приложений → Псевдонимы выполнения приложений, и выключить там python.exe и python3.exe. После этого работает настоящий Python, который поставил uv.

Лирика. Python назван не в честь змеи, а в честь британского комедийного шоу «Летающий цирк Монти Пайтона». Его создатель Гвидо ван Россум читал сценарии шоу, пока придумывал язык, и хотел название короткое, загадочное и немного дурацкое. Змея на логотипе появилась позже — маркетологи не устояли.

Редактор и первый проект

Писать код можно хоть в «Блокноте», но мы возьмём **Visual Studio Code** — бесплатный редактор от Microsoft, стандарт для Python. Поставь его с сайта code.visualstudio.com, открой, слева на панели найди «Расширения» и поставь два: **Python** от Microsoft и **Ruff** от Astral. Первое учит редактор понимать Python, второе подсвечивает подозрительные места прямо по ходу набора.

Теперь проект. Создай где-нибудь папку `mini_doom`, в VS Code выбери «Файл → Открыть папку» и открой её. Создай файл `main.py` — в Python это обычное имя для главного файла программы. Запускать программу будем из встроенного терминала (меню «Терминал → Создать терминал») командой `python main.py`. Полезные клавиши собраны в таблице 1.1.

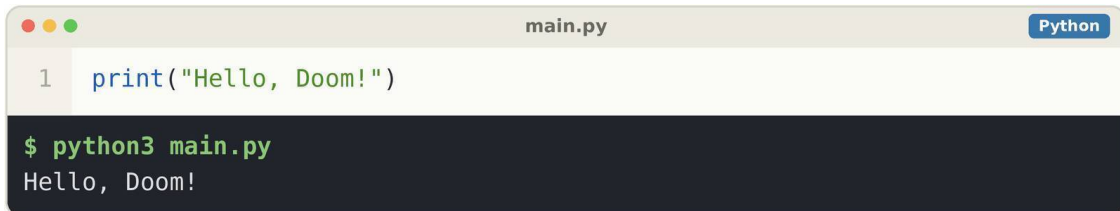
Клавиши	Что делают
<code>Ctrl+F5</code>	запустить текущий файл без отладчика
<code>F5</code>	запустить под отладчиком — пригодится, когда начнём ловить баги
<code>Ctrl+Ё</code>	показать или спрятать терминал (на английской раскладке это клавиша с обратной кавычкой)
<code>Ctrl+/ Shift+Alt+F</code>	закомментировать или раскомментировать выделенные строки
<code>Shift+Alt+F</code>	отформатировать файл (расширение Ruff расставит пробелы как положено)
<code>F12</code>	перейти к определению того, на чём стоит курсор
<code>Ctrl+Shift+P</code>	палитра команд: если не знаешь, где кнопка, ищи тут

Таблица 1.1 — Горячие клавиши VS Code, которые понадобятся сразу

По-взрослому. В книге команды запуска написаны как в Linux и macOS: `python3 main.py`. На Windows та же команда пишется `python main.py`, а можно и `uv run main.py` — тогда `uv` сам возьмёт нужную версию Python. Разницы для программы никакой.

Hello, Doom

Набери в `main.py` код из листинга 1.1. Именно набери руками, не копируй: пальцы тоже должны учиться.



```
main.py Python
1 print("Hello, Doom!")

$ python3 main.py
Hello, Doom!
```

Листинг 1.1 — Первая программа

Запусти. В терминале появится ровно то, что в тёмной панели под кодом: строка Hello, Doom!. Поздравляю, ты программист. Ну, почти. Одна строчка — и программа готова: ни заголовков, ни функции `main`, ни точки с запятой. За это Python и любят.

Разберём эту строку по частям. `print` — это **функция**, готовая команда, которая печатает в терминал то, что ей дали. Круглые скобки после имени означают «вызвать функцию». Внутри скобок — то, что ей передают, **аргумент**. Здесь это строка «Hello, Doom!»: текст в кавычках. Кавычки говорят Python, что это не команда, а просто буквы, которые надо оставить как есть. После печати `print` сам переводит курсор на новую строку.

Функции `print` можно передать сразу несколько значений через запятую — она напечатает их через пробел. А два особых именованных аргумента меняют поведение: `sep` задаёт, чем разделять значения, а `end` — чем закончить строку. Всё это в листинге 1.2.



```
main.py Python
1 print("HEALTH:", 100, "AMMO:", 50)
2 print("HEALTH:", 100, sep=" ")
3 print("Loading", end=" ")
4 print("...", end=" ")
5 print(" OK")
6 print()
7 print("2 + 2 =", 2 + 2)

$ python3 main.py
HEALTH: 100 AMMO: 50
HEALTH:100
Loading... OK

2 + 2 = 4
```

Листинг 1.2 — Возможности print: несколько значений, sep и end

Строка 1. Четыре значения через запятую — две строки и два числа. Числа пишутся без кавычек, и Python понимает, что это числа. `print` напечатал всё через пробел.

Строка 2. `sep=»»` — разделитель пустой, поэтому `HEALTH:` и `100` слились.

Строки 3–5. `end=»»` отменяет перевод строки в конце. Три вызова `print` напечатали одну строку `Loading... OK` — так рисуют индикаторы загрузки.

Строка 6. `print` без аргументов печатает пустую строку.

Строка 7. `«2 + 2 =»` в кавычках — это текст, его напечатали как есть. А `2 + 2` без кавычек — **выражение**: Python его посчитал и напечатал `4`. Как `print` раскладывает значения по строке, показано на рисунке 1.2.

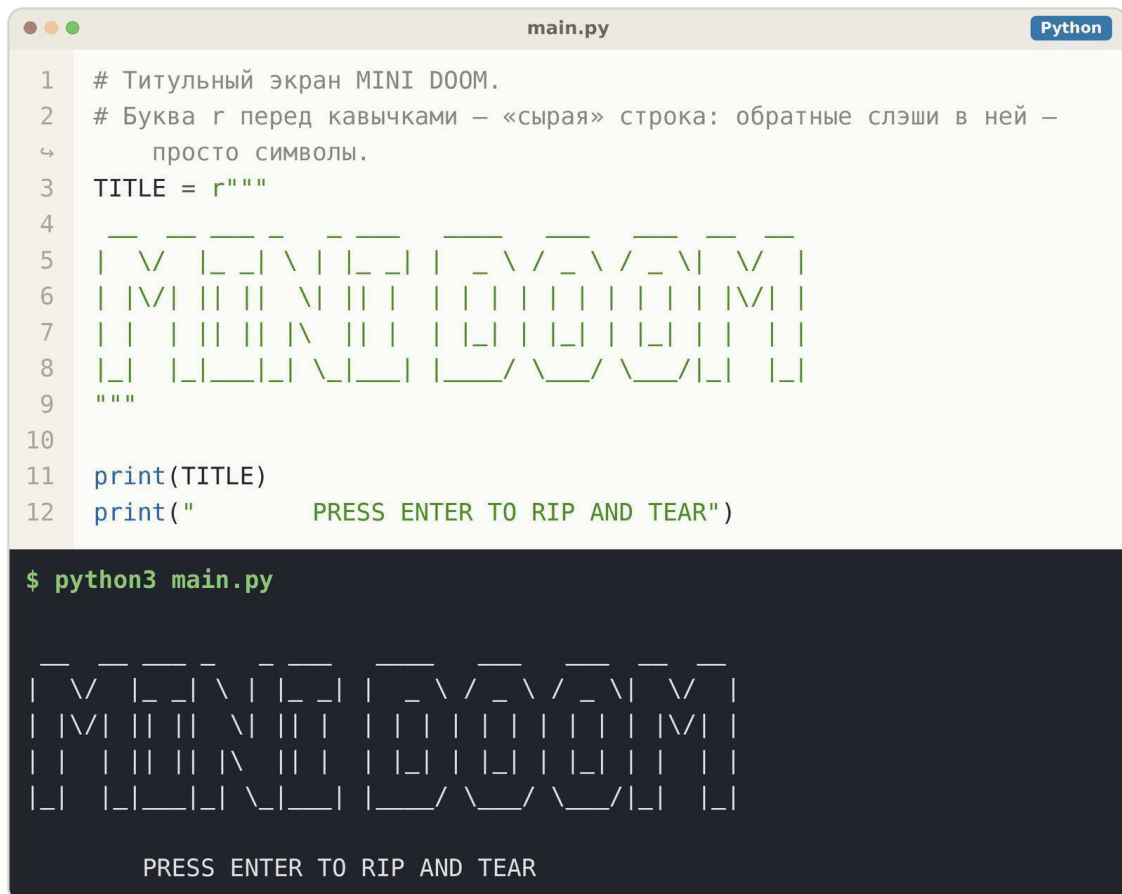


Рисунок 1.2 — Функция `print`: значения по очереди уезжают в терминал, между ними — разделитель

Запомни. Текст в кавычках — строка, его печатают как есть. Выражение без кавычек Python сначала вычисляет. `"2 + 2"` — это три символа и два пробела, а `2 + 2` — это `4`.

Рисуем титульный экран

Строка Hello, Doom! — это мило, но у настоящей игры должен быть титульный экран. Нарисуем его буквами, как делали в восьмидесятых, — это называется ASCII-арт. Код в листинге 1.3.



```

1  # Титульный экран MINI DOOM.
2  # Буква г перед кавычками — «сырая» строка: обратные слэши в ней —
   просто символы.
3  TITLE = r"""
4
5  |  \  |  _  |  _  |  _  |  _  |  _  |  _  |  _  |  _  |
6  |  \  |  |  |  \  |  |  |  |  |  |  |  |  |  |  |  |  |
7  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
8  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
9  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
10
11 print(TITLE)
12 print("          PRESS ENTER TO RIP AND TEAR")

```

```

$ python3 main.py

```

```

|  \  |  _  |  _  |  _  |  _  |  _  |  _  |  _  |  _  | | | | | | | | | | |
|  \  |  |  |  \  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
          PRESS ENTER TO RIP AND TEAR

```

Листинг 1.3 — Титульный экран MINI DOOM из символов

Строки 1 и 2 начинаются с решётки. Это **комментарии**: всё от # до конца строки Python пропускает. Комментарии пишут для людей — для коллег и для себя через полгода, когда забудешь, что тут происходит. А забудешь обязательно.

Строка 3. `TITLE = ...` — первая в книге **переменная**: имя, к которому привязано значение. Подробно о переменных — в следующей главе, а пока считай, что мы положили рисунок в коробку с надписью `TITLE` и в строке 11 достали его оттуда, чтобы напечатать. Имя написано заглавными буквами — так в Python договорились обозначать константы, значения, которые не меняются.

Сам рисунок — это строка в тройных кавычках «`»»»...»»»`. Обычная строка в одинарных кавычках должна уместиться в одну строку кода, а тройная может тянуться сколько угодно и сохраняет все переносы. Обрати внимание на первую строку вывода: она пустая. Это не баг: строка начинается сразу после открывающих кавычек, а там стоит перевод строки — он тоже попал в вывод.

Самое хитрое — буква `r` перед кавычками. В обычной строке обратный слэш `\` — особый символ: `\n` означает перевод строки, `\t` — табуляцию. А в нашем рисунке слэшей полно. Буква `r` делает строку «сырой» (raw): все символы в ней — просто символы. Что будет, если её забыть, видно в листинге 1.4: там рисунок без `r`, и мы проверили файл линтером Ruff.

```

main.py Python
1 # Забыли букву r перед кавычками.
2 TITLE = """
3 | \ / | _ _ | \ | | _ _ |
4 | ""
5
6 print(TITLE)

$ ruff check main.py
W605 [*] Invalid escape sequence: `\/`
--> main.py:3:4
|
1 | # Забыли букву r перед кавычками.
2 | TITLE = """
3 | | \ / | _ _ | \ | | _ _ |
  |     ^^
4 | ""
  |
Found 2 errors.
[*] 2 fixable with the `--fix` option.
# ... отчёт сокращён до сути

```

Листинг 1.4 — Забыли `r`: Ruff находит подозрительные слэши

Ruff говорит: в строке 3, в столбце 4 стоит неправильная escape-последовательность `\ /` — такой особой комбинации в Python нет. Таких мест в рисунке два, второе он нашёл в столбце 14, мы его сократили. Ниже Ruff даже подсказывает лекарство: поставить `r` перед кавычками. Сам Python в таком случае выдаст предупреждение `SyntaxWarning`, но картинку напечатает. А если бы в рисунке оказалось сочетание `\n`, на его месте молча появился бы перенос строки, и рисунок разъехался бы без всяких предупреждений.

Грабли. Захочешь написать на титульнике что-нибудь по-русски — в старой консоли Windows вместо букв могут вылезти кракозябры. Консоль по историческим причинам иногда работает не в той кодировке. В новом Терминале Windows и в терминале VS Code это обычно уже не проблема. А в части II у нас будет своё окно со своим шрифтом, и вопрос исчезнет сам. Поэтому пока все надписи в игре — по-английски, как в оригинальном Doom.

Когда Python ругается

Рано или поздно — скорее рано — ты запустишь программу, а вместо результата получишь простыню красного текста. Это нормально. Опытные программисты видят такое по сто раз на дню и не расстраиваются. Ошибки бывают трёх видов, и полезно понимать, кто именно ругается и когда.

Первый вид — синтаксическая ошибка. Ты написал то, что вообще не похоже на Python. Такие ошибки интерпретатор замечает сразу, на этапе перевода в байт-код, и не выполняет ни одной строки. Классика — незакрытая скобка в листинге 1.5.

```

1 print("MINI DOOM")
2 print("Hello, Doom!")
3 print("Press any key")

$ интерпретатор ругается:
  File "main.py", line 2
    print("Hello, Doom!")
    ^
SyntaxError: '(' was never closed

```

Листинг 1.5 — Забыли закрывающую скобку в строке 2

Сообщение выглядит пугающе, но устроено всегда одинаково. Разберём его по частям на рисунке 1.3.



Читай с конца: суть → тип → где.

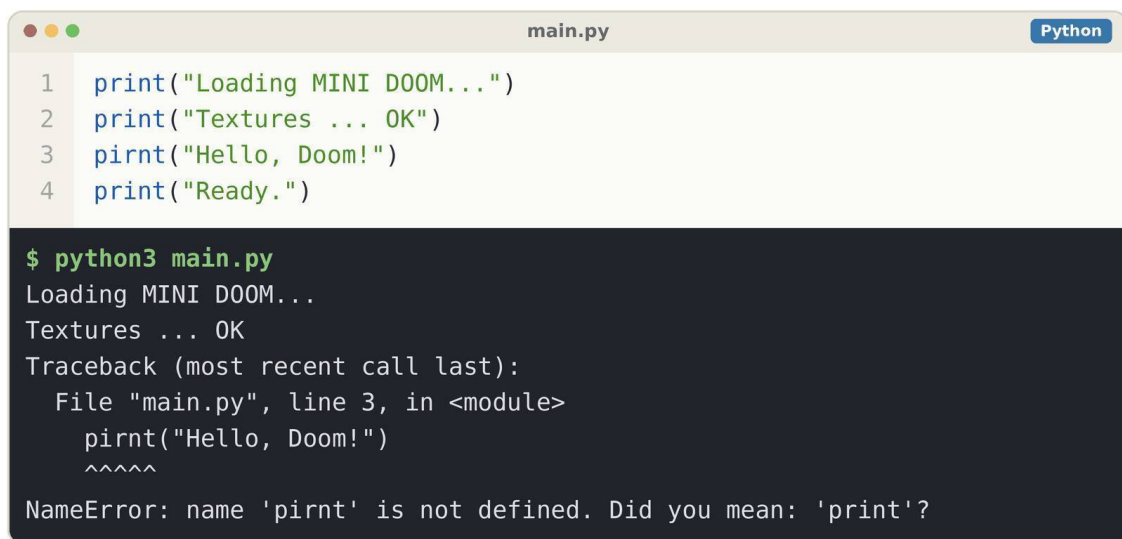
Скобку открыли в строке 2 и не закрыли — галочка ^ показывает, какую.

Рисунок 1.3 — Анатомия сообщения об ошибке: файл, строка, тип и суть

Python говорит: в файле `main.py`, в строке 2, открытая скобка так и не закрылась, и галочкой ^ показывает, какая именно. Обрати внимание: строка 1 тоже не выполнялась, MINI DOOM не напечатался. Синтаксическая ошибка останавливает всё ещё до старта.

Грабли. Иногда Python указывает не на ту строку, где ошибка, а на соседнюю. Он читает код подряд и спотыкается тогда, когда встречается что-то, что уже точно не лезет. Если строка из сообщения выглядит правильной — смотри на строку выше.

Второй вид — ошибка во время выполнения. Код написан грамматически правильно, но делает невозможное. Такие ошибки проявляются, только когда выполнение до них дойдёт. В листинге 1.6 вместо `print` написано `pirnt`.



```

main.py Python
1 print("Loading MINI DOOM...")
2 print("Textures ... OK")
3 pirnt("Hello, Doom!")
4 print("Ready.")

$ python3 main.py
Loading MINI DOOM...
Textures ... OK
Traceback (most recent call last):
  File "main.py", line 3, in <module>
    pirnt("Hello, Doom!")
    ^^^^^
NameError: name 'pirnt' is not defined. Did you mean: 'print'?

```

Листинг 1.6 — Опечатка в имени функции

Видишь? Первые две строки отработали и напечатали своё, а на третьей программа упала. Это **трейсбек** (traceback) — отчёт о том, где и почему всё сломалось. Читать его надо снизу: последняя строка — суть. `NameError`: имени `pirnt` не существует. Python даже догадывается, что ты имел в виду `print`. Строкой выше — сама виноватая строка, а над ней — файл и номер строки. Четвёртая строка, `Ready.`, так и не выполнилась.

Вот почему такие ошибки опасны. Представь, что опечатка не в третьей строке, а в коде, который срабатывает, только когда игрок подберёт секретную ракетницу на пятом уровне. Ты можешь сто раз запустить игру, и всё будет работать. А потом игрок дойдёт до ракетницы, и игра вылетит у него. Лучше бы поймать ошибку заранее. Для этого и нужен **линтер** — программа, которая читает код, не выполняя его, и ищет подозрительное. Мы поставили Ruff. Проверим им тот же файл в листинге 1.7.

```

main.py Python
1 print("Loading MINI DOOM...")
2 print("Textures ... OK")
3 pirnt("Hello, Doom!")
4 print("Ready.")

$ ruff check main.py
F821 Undefined name `pirnt`
--> main.py:3:1
|
1 | print("Loading MINI DOOM...")
2 | print("Textures ... OK")
3 | pirnt("Hello, Doom!")
  | ^^^^^
4 | print("Ready.")
  |

Found 1 error.

```

Листинг 1.7 — Тот же файл глазами Ruff: ошибка найдена без запуска

Ruff нашёл ошибку, не запуская программу: F821 — неопределённое имя `pirnt`, строка 3, столбец 1. Буква с цифрами в начале — код правила, по нему удобно искать объяснения в документации Ruff.

Третий вид — ошибка типов. Python строго следит, что с чем можно делать. Число можно сложить с числом, строку — со строкой, но строку с числом — нельзя: непонятно, что получится. В листинге 1.8 мы пытаемся приклеить к надписи число патронов.

```

main.py Python
1 ammo = 50
2 print("Shotgun ready")
3 print("AMMO: " + ammo)

$ python3 main.py
Shotgun ready
Traceback (most recent call last):
  File "main.py", line 3, in <module>
    print("AMMO: " + ammo)
                    ^~~~~
TypeError: can only concatenate str (not "int") to str

```

Листинг 1.8 — Строку нельзя сложить с числом

Снова трейсбек, и снова строка 2 успела выполниться. `TypeError`: склеивать можно только строку со строкой, а не с `int`, то есть целым числом. Волнистая линия под кодом показывает, какая операция не удалась, а галочка — где именно плюс.

Такие ошибки ловит второй сторож — **муру**, программа проверки типов. Она тоже читает код без запуска, но смотрит не на опечатки, а на то, сходятся ли типы. Результат — в листинге 1.9.

```

main.py Python
1 ammo = 50
2 print("Shotgun ready")
3 print("AMMO: " + ammo)

$ mypy --strict main.py
main.py:3: error: Unsupported operand types for + ("str" and "int")
[operator]
    
```

Листинг 1.9 — Ошибку типов находит муру

муру пишет: строка 3, неподдерживаемые типы для + — str и int, то есть строка и целое число. Как правильно печатать числа вместе с текстом, разберём в следующей главе. Самый простой способ ты уже знаешь: print(«АММО:», ammo) — через запятую, и print сам превратит число в текст.

Сводка — в таблице 1.2.

Ошибка	Когда заметна	Кто поймает заранее	Что случилось
<code>SyntaxError</code>	сразу, до запуска	Python, Ruff, редактор	текст не похож на Python: скобка, кавычка, двоеточие
<code>IndentationError</code>	сразу, до запуска	Python, Ruff, редактор	лишние или недостающие пробелы в начале строки
<code>NameError</code>	когда дойдёт	Ruff	имени не существует: опечатка или забыли создать переменную
<code>TypeError</code>	когда дойдёт	муру	операция не подходит к типу: строка плюс число

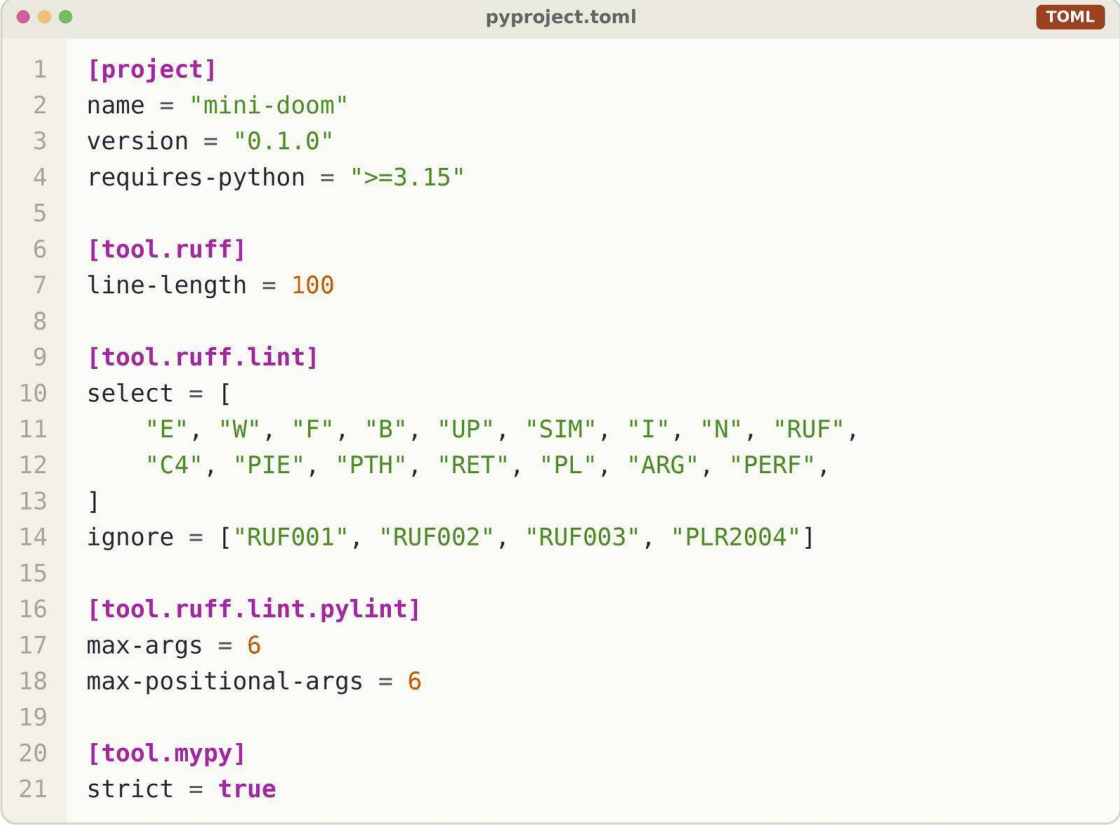
Таблица 1.2 — Первые ошибки, которые ты встретишь, и кто их ловит

Главные правила чтения ошибок такие:

- Читай трейсбек **снизу вверх**: последняя строка — что случилось, выше — где.
- Смотри на номер строки и на строку выше.
- Читай суть целиком, а не только слово Error. Python обычно говорит прямо, чего ему не хватает, и часто подсказывает: Did you mean...?
- Если ничего не понятно — скопируй последнюю строку и поищи её в интернете. С этой ошибкой до тебя уже встретились тысячи людей.

Настраиваем строгость

Ruff и муру по умолчанию добрые: проверяют немного и закрывают глаза на многое. Мы сделаем из них злых прапорщиков. Настройки проекта в Python живут в файле **pyproject.toml** в корне папки. Создай его рядом с `main.py` и заполни, как в листинге 1.10.



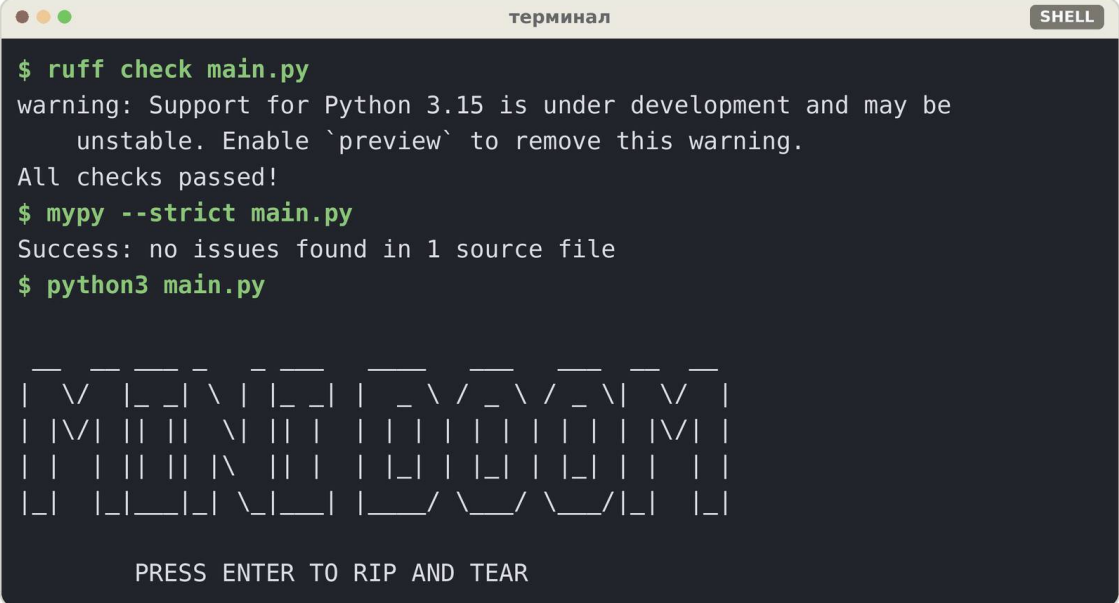
```

1  [project]
2  name = "mini-doom"
3  version = "0.1.0"
4  requires-python = ">=3.15"
5
6  [tool.ruff]
7  line-length = 100
8
9  [tool.ruff.lint]
10 select = [
11     "E", "W", "F", "B", "UP", "SIM", "I", "N", "RUF",
12     "C4", "PIE", "PTH", "RET", "PL", "ARG", "PERF",
13 ]
14 ignore = ["RUF001", "RUF002", "RUF003", "PLR2004"]
15
16 [tool.ruff.lint.pylint]
17 max-args = 6
18 max-positional-args = 6
19
20 [tool.mypy]
21 strict = true
  
```

Листинг 1.10 — Настройки проекта: строгий Ruff и строгий муру

Раздел `[project]` описывает сам проект: имя, версию и минимальную версию Python. Раздел `[tool.ruff.lint]` включает наборы правил Ruff: каждый код — отдельная группа проверок, от стиля (E, W) и явных ошибок (F) до подозрительных конструкций (B), устаревших приёмов (UP) и лишних сложностей (SIM, PL). Названия групп можно не запоминать: Ruff в каждом замечании пишет код правила, а по коду легко найти объяснение. Строка `ignore` выключает три правила, которые ругаются на русские буквы в комментариях, и одно, которое в учебных примерах только мешает. Раздел `[tool.ruff.lint.pylint]` чуть смягчает одно правило: Ruff по умолчанию считает подозрительной функцию больше чем с пятью параметрами, а в игре рисование прямоугольника — это четыре координаты, цвет и сам холст. А `strict = true` в разделе `[tool.mypy]` включает у муру самый строгий режим.

Каждый пример в этой книге проходит обе проверки без единого замечания, прежде чем попасть в листинг. Как это выглядит в терминале, показано в листинге 1.11: так стоит проверять свою программу перед каждым запуском.



```
$ ruff check main.py
warning: Support for Python 3.15 is under development and may be
       unstable. Enable `preview` to remove this warning.
All checks passed!
$ mypy --strict main.py
Success: no issues found in 1 source file
$ python3 main.py

  _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _
  | \  | _ _ | \  | _ _ | _ _ \ / _ _ \ / _ _ \  | \  | | | | | | | | | | | | | | | | |
  | \  | | | | \  | | | | | | | | | | | | | | | \  |
  | | | | | | | \  | | | | | | | | | | | | | | | |
  | | | | | | | \  | | | | | | | | | | | | | | | |

PRESS ENTER TO RIP AND TEAR
```

Листинг 1.11 — Проверка перед запуском: Ruff, туру и только потом программа

По-взрослому. В серьёзных командах код без чистого Ruff и туру просто не принимают: проверки запускаются автоматически при каждой отправке кода, и красный крестик блокирует изменения. Замечание сегодня — это баг завтра. Когда код проверяется начисто, любое новое замечание сразу бросается в глаза, а не прячется среди сотни старых, на которые все давно забили.

Итоги главы

- Интерпретатор Python выполняет программу строку за строкой, переводя её в байт-код при каждом запуске.
- Python ставим через `uv`, код пишем в VS Code с расширениями Python и Ruff.
- Программа на Python может состоять из одной строки: `print(«Hello, Doom!»)`.
- `print` печатает несколько значений через пробел; `sep` меняет разделитель, `end` — окончание строки.
- Тройные кавычки хранят многострочный текст, буква `r` делает строку сырой — идеально для ASCII-арта.
- Ошибки бывают синтаксические (видны сразу) и во время выполнения (видны, когда до них дойдёт). Вторые ловим заранее: опечатки — Ruff, типы — `myru`.
- Трейсбек читаем снизу вверх.

Контрольная точка: у тебя есть программа, которая печатает титульный экран MINI DOOM и проходит проверку Ruff и `myru`. В следующей главе на экране появятся числа — здоровье, патроны и урон, — и мы посчитаем, сколько выстрелов нужно, чтобы уложить импа.

Упражнения

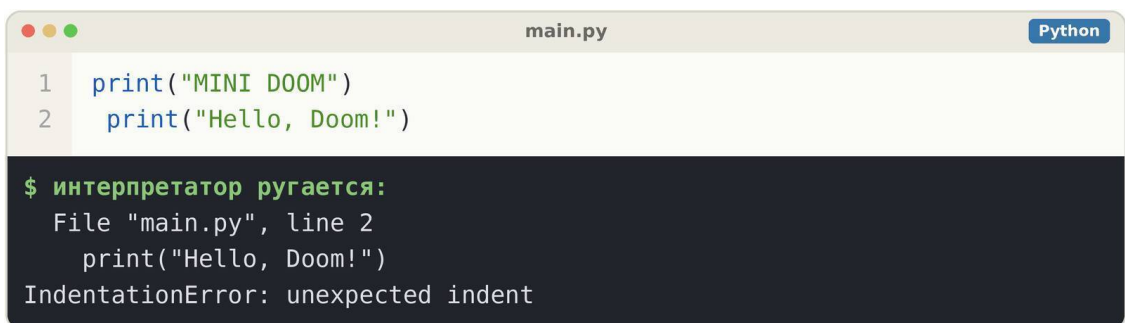
Упражнение 1.1. *Что выведет?* Не запуская программу из листинга 1.12, напиши на бумаге, что она напечатает. Потом проверь себя.



```
main.py Python
1 print("RIP", "AND", "TEAR", sep="- ")
2 print("Doom", end=" ")
3 print("Guy")
4 print("2 + 2")
5 print(2 + 2)
6 print("Imp" * 3)
```

Листинг 1.12 — Упражнение 1.1

Упражнение 1.2. *Найди баг.* Программа из листинга 1.13 не запускается. Почему? Попробуй догадаться до того, как запустишь её.



```
main.py Python
1 print("MINI DOOM")
2 print("Hello, Doom!")

$ интерпретатор ругается:
File "main.py", line 2
    print("Hello, Doom!")
IndentationError: unexpected indent
```

Листинг 1.13 — Упражнение 1.2

Упражнение 1.3. *Напиши сам.* Выведи надпись PRESS ENTER в рамке из символов +, — и |. Рамка должна быть ровной: углы на своих местах, стороны одной длины.

Упражнение 1.4. *Подумай.* Программа из листинга 1.6 успела напечатать две строки, прежде чем упасть. Программа из листинга 1.5 не напечатала ничего. Почему так? Какая из этих двух ошибок опаснее в большой игре?

Задачи

Задача 1.1. Экран загрузки. Напиши программу, которая печатает заголовок с версией игры и несколько строк вида «Loading textures ОК», где слово ОК во всех строках стоит строго в одном столбце. Закончи боевым кличем.

Задача 1.2. Свой баннер. Нарисуй в сырой строке титульник со своим ником. Потом попробуй закончить рисунок обратным слэшем прямо перед закрывающими кавычками. Что случится и как это обойти?

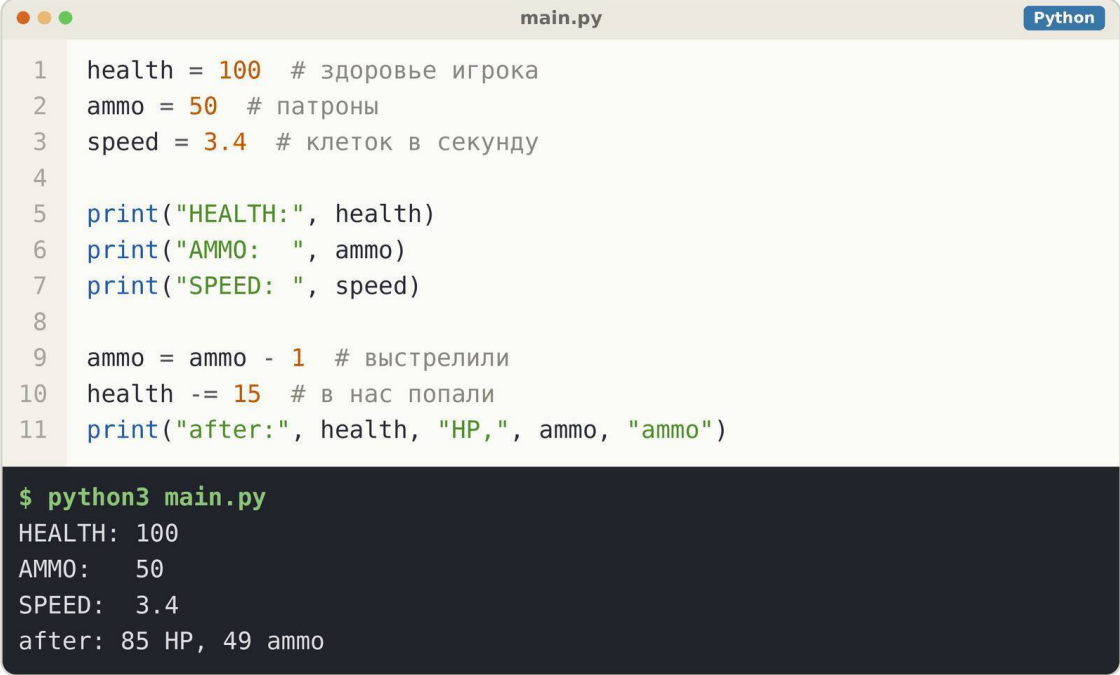
Задача 1.3. Коллекционер ошибок. Сломай программу из листинга 1.1 пятью разными способами: убери скобку, закавычь не то, напиши Print с большой буквы, добавь лишний пробел в начале и так далее. Для каждой поломки выпиши последнюю строку ошибки и объясни её своими словами. А потом проверь каждую сломанную версию Ruff: какие из ошибок он нашёл бы до запуска?

Глава 2. Переменные и типы

Игра — это числа. Здоровье, патроны, урон, координаты, время на уровне, процент найденных секретов. Даже красивая картинка в конце концов — горсть чисел, которые означают цвета. В этой главе мы научимся хранить числа и строки в переменных, считать с ними, выводить их красиво и не наступать на грабли, разложенные на этом пути. А в конце посчитаем, сколько выстрелов из пистолета нужно, чтобы уложить импа.

Переменная — бирка на коробке

Представь склад, заставленный коробками. В каждой коробке лежит значение: число 100, строка «imp», дробное 3.4. Сами коробки безымянные, зато на них можно повесить **бирку** с именем. Вот эта бирка и есть переменная в Python: имя, привязанное к значению. Запись `health = 100` читается так: «положи на склад коробку со значением 100 и повесь на неё бирку `health`». Знак `=` в программировании — не «равно», а «привязать».



```

1  health = 100 # здоровье игрока
2  ammo = 50 # патроны
3  speed = 3.4 # клеток в секунду
4
5  print("HEALTH:", health)
6  print("AMMO: ", ammo)
7  print("SPEED: ", speed)
8
9  ammo = ammo - 1 # выстрелили
10 health -= 15 # в нас попали
11 print("after:", health, "HP,", ammo, "ammo")

$ python3 main.py
HEALTH: 100
AMMO: 50
SPEED: 3.4
after: 85 HP, 49 ammo
  
```

Листинг 2.1 — Переменные игрока: создаём, печатаем, меняем

Строки 1–3 создают три переменные. Имена пишем маленькими буквами, слова разделяем подчёркиванием: `health`, `ammo`, `player_speed`. Так в Python принято, и Ruff проследит, чтобы ты не написал `PlayerSpeed` там, где ждут переменную. Имя не может начинаться с цифры и не может совпадать со словами языка вроде `if` или `for`.

Строка 9. `ammo = ammo — 1` — с точки зрения математики бред, а для программы нормальная команда: возьми то, на чём висит бирка `ammo`, отними единицу, а бирку перевесь на результат. **Строка 10** делает то же самое короче: `health -= 15` значит `health = health — 15`. Так же работают `+=`, `*=`, `/=` и остальные.

Бирки — не просто красивая метафора, Python действительно так устроен. Смотри листинг 2.2.

```

main.py Python
1 health = 100
2 armor = health # вторая бирка на ту же коробку
3 print(type(health), health is armor)
4
5 health -= 15 # бирку health перевесили на новую коробку 85
6 print(health, armor, health is armor)

$ python3 main.py
<class 'int'> True
85 100 False

```

Листинг 2.2 — Две бирки на одной коробке и что бывает при изменении

Строка 2. `armor = health` не копирует число: на ту же самую коробку 100 вешается вторая бирка. Проверка `health is armor` в строке 3 так и отвечает: `True`, это один и тот же объект. А `type(health)` говорит, что в коробке лежит `int` — целое число.

Строка 5. `health -= 15` не меняет число внутри коробки — числа в Python **неизменяемые**, как надпись на камне. Вместо этого на складе появляется новая коробка 85, и бирку `health` перевешивают на неё. `armor` продолжает висеть на старой коробке 100. Как это выглядит, показано на рисунке 2.1.

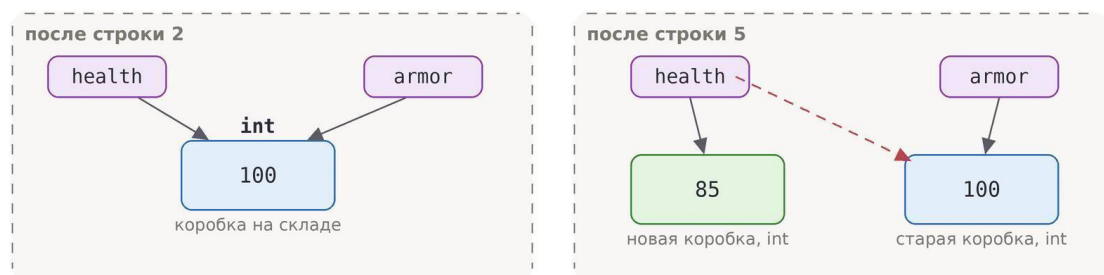


Рисунок 2.1 — Переменные — бирки на коробках. Изменение числа вешает бирку на новую коробку, а не переписывает старую

Запомни. Переменная в Python — это имя, привязанное к объекту. Присваивание ничего не копирует, а перевешивает бирку. Для чисел и строк это незаметно, потому что их нельзя изменить на месте. Для списков и других изменяемых объектов это станет важно — в главе 8 мы разберём, где тут ломают ноги.

Какие бывают типы

У каждой коробки на складе есть **тип** — что именно в ней лежит. От типа зависит, что с значением можно делать: числа складывают, строки склеивают, а сложить число со строкой, как мы видели в главе 1, нельзя. Узнать тип помогает функция `type`, а превратить значение одного типа в другой — функции с именами типов. Всё это в листинге 2.3.

```

main.py Python
1 health = 100
2 speed = 3.4
3 name = "imp"
4 alive = True
5 weapon = None
6 print(type(health), type(speed))
7 print(type(name), type(alive), type(weapon))
8 print(int("42") + 1, float("2.5") * 2, str(7) + "7")
9 print(bool(0), bool(15), bool(""), bool("imp"))

$ python3 main.py
<class 'int'> <class 'float'>
<class 'str'> <class 'bool'> <class 'NoneType'>
43 5.0 77
False True False True

```

Листинг 2.3 — Основные типы и превращения между ними

Строки 6 и 7 показывают пять типов, которые встречаются постоянно. Строка 8 — превращения: `int(«42»)` делает из строки число, к которому можно прибавить единицу; `float(«2.5»)` — дробное; `str(7)` — строку, которую можно склеить с другой строкой. В строке 9 — правило, которое пригодится в главе 3: ноль и пустая строка считаются ложью, всё остальное — правдой. Сводка — в таблице 2.1.

Тип	Что хранит	Пример в игре
<code>int</code>	целое число любой величины	здоровье <code>100</code> , патроны <code>50</code> , счёт <code>1_000_000</code>
<code>float</code>	дробное число, примерно 15–16 значащих цифр	скорость <code>3.4</code> , угол поворота <code>0.785</code>
<code>str</code>	строка, текст в кавычках	имя монстра <code>"imp"</code> , надпись <code>"GAME OVER"</code>
<code>bool</code>	правда или ложь: <code>True</code> или <code>False</code>	жив ли монстр, открыта ли дверь
<code>None</code>	ничего, пустое место	оружия в руках нет: <code>weapon = None</code>

Таблица 2.1 — Основные типы Python

Длинные числа можно разделять подчёркиваниями: 1_000_000 — то же, что 1000000, но читается без подсчёта нулей.

Числа, которые не ломаются

В большинстве языков у целого числа есть предел. Например, в C++ обычный `int` заканчивается на 2 147 483 647, и если к нему прибавить единицу, число перескакивает в огромный минус — это называется переполнение. Из-за таких историй в старых играх счётчики очков внезапно становились отрицательными. В Python у `int` предела нет, смотри листинг 2.4.

[illegible]

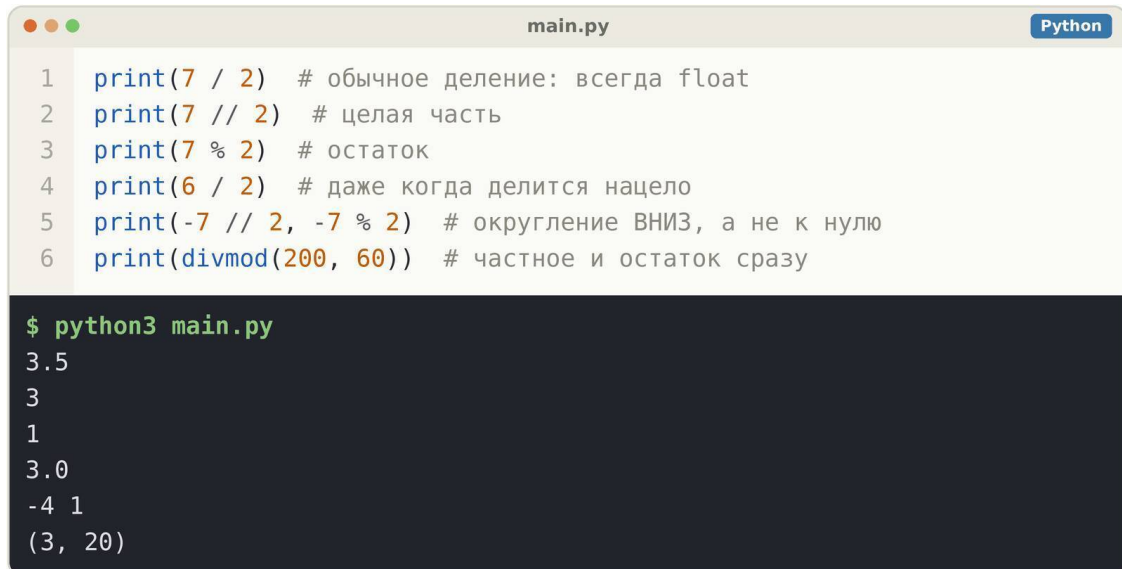
Листинг 2.4 — Целые числа Python растут без предела

Число `2**100` — это 2 в сотой степени, две звёздочки означают возведение в степень. А 2 в десяти тысячной степени содержит 3011 цифр, и Python посчитал его честно. Платим мы за это скоростью и памятью: чем больше число, тем дольше с ним работать. Для игры это не проблема, у нас числа скромные.

Лирика. В оригинальном Doom 1993 года счётчики и координаты были 32-битными целыми, а дробные координаты хранились хитрым способом — «числами с фиксированной точкой»: 16 бит на целую часть и 16 на дробную. Процессоры тех лет считали дроби мучительно медленно, а целые — быстро. Python тогда только появился и для игр был бы медленным, как черепаха. Сегодня у нас есть numpy, и в части III мы увидим, как Python рисует трёхмерный мир с нормальной скоростью.

Деление: три разных оператора

Со сложением, вычитанием и умножением всё как в школе: +, — и *. А вот делений в Python три, и путать их — верный способ получить баг. Листинг 2.5.



```

main.py Python
1 print(7 / 2) # обычное деление: всегда float
2 print(7 // 2) # целая часть
3 print(7 % 2) # остаток
4 print(6 / 2) # даже когда делится нацело
5 print(-7 // 2, -7 % 2) # округление ВНИЗ, а не к нулю
6 print(divmod(200, 60)) # частное и остаток сразу

$ python3 main.py
3.5
3
1
3.0
-4 1
(3, 20)

```

Листинг 2.5 — Обычное деление, целая часть и остаток

— / — обычное деление. Результат **всегда** float, даже если делится нацело: 6 / 2 даёт 3.0, а не 3.

— // — целая часть от деления: 7 // 2 даёт 3. Сколько полных обойм помещается в рюкзак.

— % — остаток от деления: 7 % 2 даёт 1. Сколько патронов останется россыпью.

— **divmod(a, b)** — частное и остаток одним махом: 200 секунд — это 3 минуты и 20 секунд.

Грабли. Целочисленное деление в Python округляет **вниз**, к минус бесконечности, а не к нулю. Поэтому -7 // 2 даёт -4, а не -3, как в C++ или Java, а остаток -7 % 2 — 1, а не -1. Для игр это скорее удобно: координата клетки на карте получается правильной и для отрицательных чисел. Но если переносишь формулу из другого языка — проверь.

Дроби и округление

Посчитаем процент убитых монстров для экрана конца уровня, как в настоящем Doom. Код — в листинге 2.6.



```
main.py Python
1 kills = 3
2 total = 14
3
4 print("percent:", kills / total * 100)
5 print("rounded:", round(kills / total * 100))
6 print("1 digit:", round(kills / total * 100, 1))
7
8 time = 83.9 # секунд на уровне
9 print("int(): ", int(time)) # дробь отрезается
10 print("round():", round(time))
11
12 print(round(0.5), round(1.5), round(2.5), round(3.5)) # банковское
    округление

$ python3 main.py
percent: 21.428571428571427
rounded: 21
1 digit: 21.4
int(): 83
round(): 84
0 2 2 4
```

Листинг 2.6 — Проценты, отрезание дроби и округление

Строка 4. `kills / total` — обычное деление, получаем дробь, умножаем на 100 — проценты. В C++ тут ждала бы ловушка: деление целых там отбрасывает дробь, и вышло бы 0. В Python оператор `/` всегда честный.

Конец ознакомительного фрагмента.

Текст предоставлен ООО «Литрес».

Прочитайте эту книгу целиком, [купив полную легальную версию](#) на Литрес.

Безопасно оплатить книгу можно банковской картой Visa, MasterCard, Maestro, со счета мобильного телефона, с платежного терминала, в салоне МТС или Связной, через PayPal, WebMoney, Яндекс.Деньги, QIWI Кошелек, бонусными картами или другим удобным Вам способом.