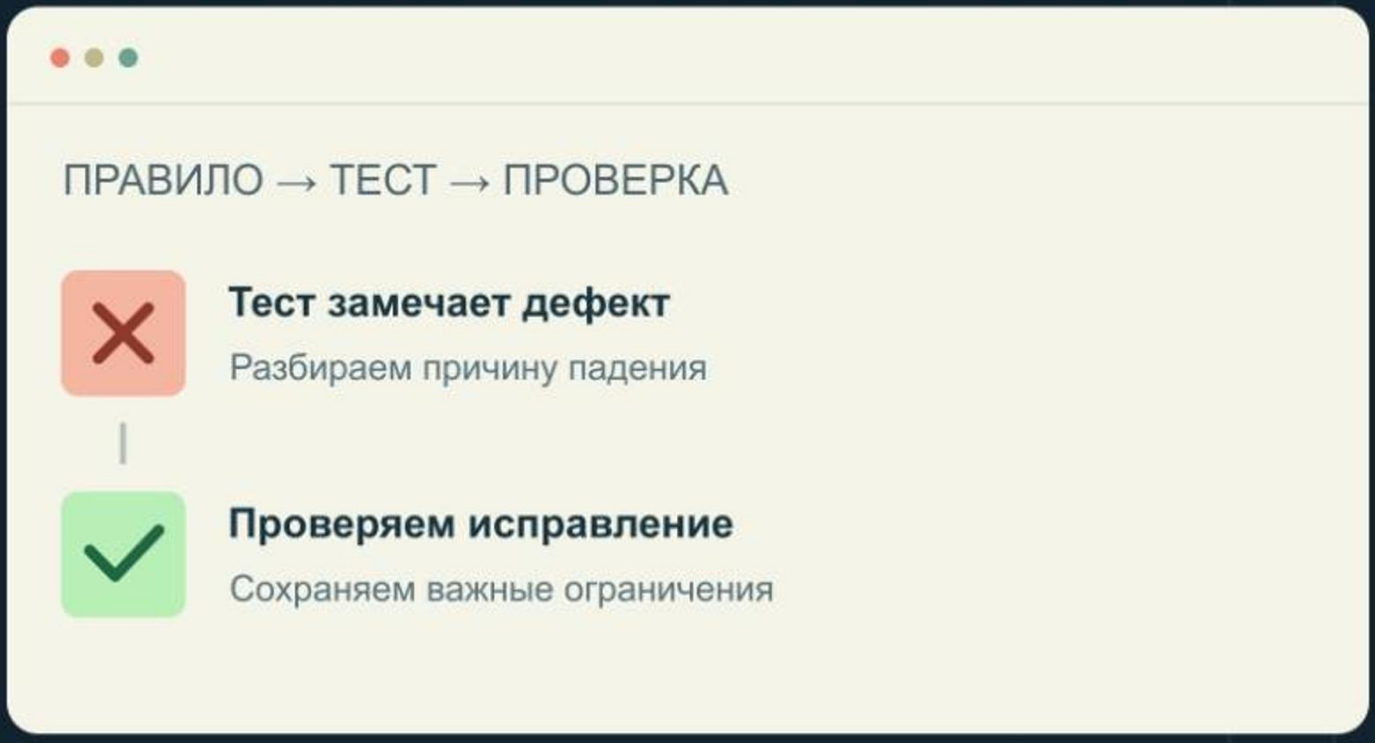
ПРАКТИКУМ С PYTEST

ТЕСТИРУЕМ PYTHON

12 лабораторных работ



Границы · Исключения · Фикстуры

Файлы · Зависимости · Состояния

КОД · РАЗБОРЫ · УПРАЖНЕНИЯ · ОТВЕТЫ

ГЛЕБ ЗАЙЦЕВ

Глеб Зайцев

Тестируем Python. 12 лабораторных работ с pytest

«Автор»
2026

Зайцев Г.

Тестируем Python. 12 лабораторных работ с pytest / Г. Зайцев — «Автор», 2026

Тест полезен не тем, что стал зелёным, а тем, что замечает нарушение правила. В этом практикуме — 12 самостоятельных лабораторных работ: границы, исключения, параметризация, изменяемые данные, фикстуры, временные файлы, зависимости и переходы состояний. В каждой работе есть точный учебный контракт, намеренно ошибочная реализация, тесты, разбор падения, исправленный код и упражнение с ответом. Примеры рассчитаны на Python 3.12 и pytest 8.3.5, не требуют веб-фреймворка, базы данных или аккаунта сервиса. Нужно уже понимать функции, импорты, списки и основы классов Python. Это практикум по небольшим изолированным проверкам, а не полный курс тестирования приложения.

Содержание

Глеб Зайцев
Тестируем Python. 12 лабораторных работ с pytest

Проверка должна замечать ошибку

От правила к проверке

Представьте две реализации одного правила. Одна ведёт себя правильно, другая ошибается только на границе. Если ваш тест принимает обе, зелёный результат почти ничего не сообщает именно об этой ошибке. Поэтому каждая работа начинается с точного условия, а намеренно испорченная версия нужна как контроль чувствительности теста. Мы не станем исправлять ожидаемый ответ под существующий код: сначала выясним, какой ответ требует условие.

Здесь нет одного большого приложения, которое нужно собрать, прежде чем увидеть первое падение. Каждая работа живёт в собственной папке и содержит небольшой файл logic.py с проверяемой логикой и файл test_logic.py с тестами. После разбора вы полностью замените учебный logic.py исправленной версией. Одноимённые файлы из разных работ не нужно объединять.

Книга рассчитана на читателя, который уже пишет функции, понимает импорт модулей, списки, словари и простые классы. Это следующий шаг от ручного вызова функции к воспроизводимой проверке её обещаний. Основы синтаксиса Python здесь не пересказываются. Стабы и наблюдатели будут небольшими функциями и вызываемыми объектами: механизм важнее большого набора инструментов.

Все ситуации и данные вымышлены. Книга подготовлена с помощью ИИ и прошла редакторскую проверку; исходные тесты и код, извлечённый из FB2, проверяются запуском в отдельных временных каталогах. Такая проверка охватывает конечный набор примеров, а не все возможные входы. Она не делает учебные реализации готовыми компонентами рабочего проекта.

Подготовка без рабочего проекта

Используйте отдельную новую папку, например pytest-practice. Не открывайте для этих опытов репозиторий работодателя, каталог с личными документами или существующее окружение проекта. Нужен установленный Python 3.12 и pytest версии 8.3.5. Эта версия выбрана для воспроизводимости примеров; мы не утверждаем, что она самая новая. Проверка книги выполнялась на Python 3.12.14.

Сначала проверьте установленный интерпретатор: python3.12 --version должен показать Python 3.12.x. В терминале новой папки создайте локальное окружение командой python3.12 -m venv .venv. Если интерпретатор установлен под другим именем, используйте его проверенный путь, а не случайную системную версию. В Windows аналогичная команда — py -3.12 -m venv .venv. В macOS/Linux активируйте окружение командой source .venv/bin/activate; в Windows PowerShell — .venv\Scripts\Activate.ps1. Если политика PowerShell запрещает активацию, не меняйте её ради книги: обращайтесь к .venv\Scripts\python.exe напрямую во всех следующих командах вместо python. Для macOS/Linux аналогичный прямой путь — .venv/bin/python.

После активации выполните python -m pip install pytest==8.3.5, затем python --version и python -m pytest --version. Установка загружает бесплатный пакет и его зависимости из настроенного у pip источника; интернет нужен на этом подготовительном шаге. Сами лабораторные работы не обращаются в сеть. Если Python или pip не найден, исправьте выбор интерпретатора до работы с примерами — это не ошибка тестируемой функции.

Внутри pytest-practice создайте через файловый менеджер или редактор отдельные папки lab01, lab02 и далее. Для выбранной работы откройте терминал именно в её папке. Не создавайте файлы с именами pytest.py, pathlib.py или datetime.py: они могут затенить библиотечные модули. Сохраняйте наши файлы в UTF-8, а не в формате текстового процессора. Один файл logic.py относится только к одной работе.

Как перенести код из читалки

В основных листингах каждая строка начинается с номера и разделителя, например 001 │ . Все обычные пробелы справа от черты показаны точками ·, чтобы читалка не скрыла отступы. Это печатная разметка, не Python. Для ручного восстановления удалите номер и черту, затем замените каждую точку · обычным пробелом, в том числе внутри строковых литералов. Точка . остаётся точкой. В исходных примерах символа · нет. Строка с одним лишь префиксом означает пустую строку кода.

Ниже дан небольшой вспомогательный скрипт без обязательных отступов и без служебных номеров. Сохраните его как restore_listing.py в отдельной учебной папке. Скопируйте ровно один номерованный листинг в copied.txt, сохранив UTF-8, и выполните python restore_listing.py. Скрипт проверяет последовательность номеров и создаёт новый restored.py. Он не запускает полученный код и откажется перезаписывать существующий restored.py. Просмотрите результат в редакторе, затем перенесите его под указанным именем logic.py, test_logic.py или test_extra.py в папку выбранной работы. Перед следующим преобразованием перенесите предыдущий restored.py, не затирая нужные файлы.

Если читалка добавила абзацные пробелы до номера, скрипт их уберёт. Визуальный перенос по ширине не является новой строкой кода. Если при копировании он превратился в настоящий разрыв, соедините части до следующего номера вручную: скрипт должен остановиться, а не угадывать. Не меняйте прямые кавычки на типографские и не применяйте общее обрезание отступов к уже восстановленному Python. Во вспомогательном скрипте все строки начинаются с левого края; удалите только добавленный читалкой начальный отступ.

Сначала создайте logic.py из блока «Версия с намеренной ошибкой», затем test_logic.py из следующего блока. Команда python -m pytest -q test_logic.py запускает только основной набор этой работы. Часть тестов должна упасть: ниже указано ожидаемое количество. Не переносите текст ожидаемого результата или номера строк в Python-файлы. Если появляется SyntaxError, IndentationError, ImportError или сообщение об ошибке сбора тестов, ещё рано обсуждать поведение функции — сначала проверьте копирование, имена файлов и текущую папку.

После разбора замените содержимое того же учебного logic.py блоком «Исправленная версия» и повторите ту же команду новым запуском. Все основные тесты должны пройти. Затем решите упражнение, не меняя исправленную реализацию. Ответ содержит отдельный test_extra.py: положите его рядом и выполните python -m pytest -q test_logic.py test_extra.py. Это намеренно явное перечисление файлов, чтобы случайные тесты из соседних работ не участвовали.

restore_listing.py — вспомогательный скрипт без номеров

```
    from pathlib import Path
    raw = Path("copied.txt").read_text(encoding="utf-8")
    rows = [s.strip() for s in raw.splitlines() if s.strip()]
    assert rows and all("│" in s for s in rows)
    parts = [s.split("│", 1) for s in rows]
    assert [p[0] for p in parts] == [f"{i:03d}" for i in range(1,
len(parts) + 1)]
    code = "\n".join(p[1].replace("·", " ") for p in parts) + "\n"
    out = open("restored.py", "x", encoding="utf-8")
    out.write(code)
    out.close()
```

Как читать красный и зелёный результат

Основной тест обычно состоит из подготовки данных, одного действия и проверки наблюдаемого результата. Ожидаемое значение берётся из контракта. Вычислять его тем же алгоритмом, который проверяется, опасно: одна и та же ошибка может оказаться по обе стороны сравнения. Короткие конкретные значения легче проверить глазами, а параметризация позволяет повторить одно правило на нескольких заранее выбранных входах.

В отчёте pytest слово FAILED указывает на неуспешный тест, а строка assert помогает увидеть, какие значения сравнивались. ERROR, особенно во время collection, означает иной тип проблемы: например, файл не удалось импортировать или не подготовилась фикстура. Намеренные дефекты этой книги должны давать именно ожидаемые провалы проверок, не аварии сбора. Время выполнения, абсолютные пути и оформление отчёта у вас будут другими.

Строка passed означает только, что собранные проверки прошли. Она не говорит, что тестов достаточно, что все важные входы рассмотрены или что внешний сервис работает. Даже маленький набор можно улучшить, задав вопрос: какую реалистичную неправильную реализацию он пока пропустит? Для каждого упражнения мы назовём такой пробел и его границы, а не будем обещать полную защиту от ошибок.

Часть работ проверяет только значения и состояние в памяти; работа с tmp_path действительно использует локальную файловую систему. Это небольшая проверка границы с файлом, а не доказательство поведения всех дисков и операционных систем. Подмена функции даты или справочника делает тест воспроизводимым, но не проверяет настоящий источник времени, удалённую базу или доставку сообщений. Для них нужны отдельные проверки вне этого практикума.

Выбрать лабораторную работу

Лабораторные работы

Работа 1. Последнее место: первое утверждение о границе

Условие и выбор проверок

В вымышленной мастерской нужно ответить, можно ли принять ещё одного участника. Функция ничего не бронирует: она получает два числа и возвращает решение. На этой маленькой задаче научимся превращать словесное правило в проверяемое утверждение и выбирать пример, который различает правильную и почти правильную формулы.

Контракт учебной функции

1. can_join(capacity, occupied) принимает ровно встроенные int; bool и любые другие типы вызывают TypeError.

2. capacity не меньше 1; occupied находится от 0 до capacity включительно. Нарушение этих границ вызывает ValueError.

3. Результат — bool: True, только когда занятых мест строго меньше вместимости; при полностью заполненной мастерской — False.

4. Функция не меняет внешнее состояние и не записывает участника. Тексты сообщений об ошибках не являются частью контракта.

Как выбрать проверки

Начнём не с оператора сравнения, а с вопроса к правилу. Если мастерская рассчитана на четыре человека и пришли трое, ответ положительный. Если пришли четверо, свободных мест уже нет. Обе ситуации допустимы как вход функции: полная мастерская не является ошибкой данных. Важно отделить обычный отрицательный ответ от исключения, означающего, например, невозможное число занятых мест.

Выберем три ситуации: никого нет, осталось одно место, свободных мест нет. Первая проверяет понятную исходную точку, вторая — ближайшее допустимое состояние перед заполнением, третья — саму границу. Десять проверок с полупустыми мастерскими могли бы дать меньше полезной информации: неверное сравнение способно соглашаться с правильным на всех таких примерах.

В test_logic.py каждая функция описывает одну ситуацию. Вызов can_join — действие, а assert фиксирует ожидаемый результат. Используем is True и is False, поскольку контракт обещает именно логическое значение: просто истинный список или число 1 не должны подменять True. Имена тестов называют поведение, поэтому по имени упавшей проверки легче понять, какое правило нарушено.

Ожидаемые значения записаны буквально. Не следует получать ожидаемый ответ выражением occupied <= capacity: так тест скопирует спорное решение из программы. Здесь независимым основанием служит описанная ситуация: четыре занятых места из четырёх означают запрет на нового участника. Тест и реализация должны встречаться в требовании, а не подтверждать друг друга одинаковым кодом.

У дефектной версии есть полезная особенность: два теста проходят. Это показывает, почему успешный запуск на одном удобном примере ещё не характеризует всю функцию. В отчёте нас интересует сравнение фактического True с ожидаемым False у полностью заполненной мастерской. Остальные две проверки удерживают положительную сторону правила и не позволяют исправить падение возвратом False для всех входов.

После замены реализации запускаем тот же файл тестов, не ослабляя утверждение. Меняется одна граница: равенство перестаёт означать разрешение. Пока мы не проверяем все ошибочные входы, хотя их обработка определена контрактом и присутствует в коде. Это осознанный фокус первой лабораторной: научиться обнаруживать конкретный дефект, не выдавая три примера за исчерпывающую проверку.

logic.py — версия с намеренной ошибкой

```
001|def·can_join(capacity,·occupied):
002|····if·type(capacity)·is·not·int·or·type(occupied)·is·not·int:
003|········raise·TypeError("capacity·and·occupied·must·be·integers
004|····if·capacity·<·1·or·not·0·<=·occupied·<=·capacity:
005|········raise·ValueError("invalid·capacity·or·occupancy")
006|····return·occupied·<=·capacity
```

test_logic.py — основной набор проверок

```
001|from·logic·import·can_join
002|
003|
004|def·test_empty_workshop_accepts_one_more():
005|····assert·can_join(4,·0)·is·True
006|
007|
008|def·test_one_free_place_accepts_one_more():
009|····assert·can_join(4,·3)·is·True
010|
011|
012|def·test_full_workshop_rejects_one_more():
013|····assert·can_join(4,·4)·is·False
```

Запуск, диагноз и исправление

В папке lab01 выполните python -m pytest -q test_logic.py. Для намеренно ошибочной версии ожидаются 3 проверок: неуспешных — 1, успешных — 2. Это контрольная ошибка поведения, а не ошибка импорта или сбора тестов.

Падает test_full_workshop_rejects_one_more: дефектная версия считает равенство допустимым для приёма участника. Проверка входных данных здесь не виновата — occupied == capacity разрешено передавать. Ошибка находится в решении, которое принимается для этого корректного входа.

Исправление заменяет <= на < только в строке возврата. Не нужно запрещать равенство валидатором: тогда вместо обещанного False функция выбросила бы исключение. Три успешных теста после исправления подтверждают три оговорённых наблюдения, включая поведение на границе.

logic.py — исправленная версия

```
001|def·can_join(capacity,·occupied):
002|····if·type(capacity)·is·not·int·or·type(occupied)·is·not·int:
003|········raise·TypeError("capacity·and·occupied·must·be·integers
004|····if·capacity·<·1·or·not·0·<=·occupied·<=·capacity:
005|········raise·ValueError("invalid·capacity·or·occupancy")
006|····return·occupied·<·capacity
```

Повторная проверка и самостоятельное упражнение

Замените только учебный logic.py исправленной версией и повторите основной запуск. Все 3 проверок должны пройти.

Не меняя контракт и исправленную функцию, проверьте мастерскую минимальной допустимой вместимости. Напишите отдельные тесты для пустой и полностью занятой мастерской на одного участника.

Эти тесты проверяют локальное решение по переданным числам. Они не доказывают корректность всех целых входов и не проверяют все обещанные исключения. Тем более они не обеспечивают реальное бронирование: между чтением числа участников и записью нового человека состояние могло бы измениться. Совместный доступ и хранение данных в этой модели отсутствуют.

Ответ к упражнению 1
К списку работ

Работа 2. Таблица границ: один тест, семь самостоятельных примеров

Условие и выбор проверок

В локальном планировщике учебных выступлений длительность получает одну из трёх меток. Составим маленькую таблицу правил и передадим её в pytest.mark.parametrize. Главная задача — не сократить строки кода, а сделать принадлежность каждой границы явной и получить отдельный результат для каждой пары «вход — ожидаемая метка».

Контракт учебной функции

1. talk_band(minutes) принимает ровно встроенный int, без bool; остальные типы вызывают TypeError.

2. Допустимы целые minutes от 0 до 120 включительно. Ноль обозначает ещё не заполненный по длительности черновик. Числа вне диапазона вызывают ValueError.

3. Для 0–5 минут включительно результат равен строке 'short'; для 6–20 — 'regular'; для 21–120 — 'long'.

4. Функция не округляет вход и не меняет внешнее состояние. Тексты исключений не закреплены контрактом.

Как выбрать проверки

Сначала раскладываем правило на непересекающиеся интервалы. Названия меток сами по себе не определяют границы: кому-то двадцатиминутное выступление покажется длинным, но в этой вымышленной задаче оно относится к regular. Тест должен защищать согласованное правило, а не личную оценку автора. Поэтому интервалы записаны числами, и обе стороны каждой границы названы явно.

Из первого интервала берём 0 и 5, из второго — 6, 19 и 20, из третьего — 21 и 120. Пары 5/6 и 20/21 проверяют смену категории на соседних целых значениях. Точка 19 помогает отличить ошибку только на последнем значении интервала от ошибки всей категории. Края 0 и 120 удерживают допустимый диапазон с обеих сторон; дробные значения здесь вообще не участвуют в классификации.

В декораторе два имени, minutes и expected, соответствуют двум элементам каждого кортежа. pytest создаёт отдельный проверяемый пример для каждой строки таблицы. Поэтому одна функция test_talk_band собирается как семь тестов. Это удобнее цикла внутри одного теста: там первое неуспешное утверждение остановило бы цикл и скрыло наблюдения для оставшихся строк.

Метки expected — независимые строковые литералы. Не вычисляем их вспомогательной функцией с теми же if, иначе две одинаковые ошибки дали бы зелёный отчёт. Само тело теста остаётся коротким: сделать один вызов и сравнить его со значением из согласованной таблицы. Менять таблицу следует при изменении требований, а не ради приспособления к текущему результату функции.

При чтении падения смотрим не только на имя функции теста, но и на её параметры. Если не прошла только строка с 20, а 19 и 21 прошли, это сужает поиск до перехода regular → long. Таблица превращает общий сигнал «классификация неправильна» в конкретное про-

тиворечие. При этом тест не требует, чтобы функция вообще была написана через if: важен её ответ.

Не будем складывать несколько декораторов для входа и ожидания: здесь нужны согласованные пары, а не все возможные сочетания. Не добавляем и xfail для известного дефекта: цель лабораторной — сначала получить обычное падение, затем устранить его. После ремонта запускаем все семь строк, чтобы убедиться, что исправление двадцатой минуты не переместило соседние значения.

logic.py — версия с намеренной ошибкой

```
001|def·talk_band(minutes):
002|····if·type(minutes)·is·not·int:
003|········raise·TypeError("minutes·must·be·an·integer")
004|····if·not·0·<=·minutes·<=·120:
005|········raise·ValueError("minutes·must·be·between·0·and·120")
006|····if·minutes·<=·5:
007|········return·"short"
008|····if·minutes·<·20:
009|········return·"regular"
010|····return·"long"
```

test_logic.py — основной набор проверок

```
001|import·pytest
002|
003|from·logic·import·talk_band
004|
005|
006|@pytest.mark.parametrize(
007|····"minutes,·expected",
008|····[
009|········(0,·"short"),
010|········(5,·"short"),
011|········(6,·"regular"),
012|········(19,·"regular"),
013|········(20,·"regular"),
014|········(21,·"long"),
015|········(120,·"long"),
016|····],
017|)
018|def·test_talk_band(minutes,·expected):
019|····assert·talk_band(minutes)·==·expected
```

Запуск, диагноз и исправление

В папке lab02 выполните python -m pytest -q test_logic.py. Для намеренно ошибочной версии ожидаются 7 проверок: неуспешных — 1, успешных — 6. Это контрольная ошибка поведения, а не ошибка импорта или сбора тестов.

Из семи строк падает одна: minutes == 20 возвращает 'long' вместо 'regular'. Первый if уже пропустил это число, а условие minutes < 20 тоже его отвергает. Так допустимая последняя минута среднего интервала попадает в оставшуюся ветку. Успех остальных шести строк не делает спорную границу менее обязательной.

Меняем второе сравнение на minutes <= 20. Нижняя граница regular не требует отдельного условия, потому что значения до 5 включительно уже вернулись из первой ветки. Тесты 5 и 6 поддерживают этот вывод наблюдением, а не только чтением кода. В исправленной версии проходят все семь собранных примеров.

logic.py — исправленная версия

```
001|def·talk_band(minutes):
002|····if·type(minutes)·is·not·int:
003|········raise·TypeError("minutes·must·be·an·integer")
004|····if·not·0·<=·minutes·<=·120:
005|········raise·ValueError("minutes·must·be·between·0·and·120")
006|····if·minutes·<=·5:
007|········return·"short"
008|····if·minutes·<=·20:
009|········return·"regular"
010|····return·"long"
```

Повторная проверка и самостоятельное упражнение

Замените только учебный logic.py исправленной версией и повторите основной запуск. Все 7 проверок должны пройти.

Добавьте отдельную параметризованную проверку трёх внутренних точек: 3, 12 и 70 минут. Сохраните ожидаемые метки явными литералами и заранее определите, сколько тестов добавится.

Параметризация меняет организацию примеров, но сама не обеспечивает полноту. Здесь нет перебора всех 121 допустимых чисел и нет проверок исключений для ошибочных входов. Таблица также не доказывает, что интервалы удачно выбраны для реального расписания: они заданы только для учебной модели и проверяются как требования этой модели.

Ответ к упражнению 2
К списку работ

Работа 3. Отказ тоже результат: проверяем нужное исключение

Условие и выбор проверок

Локальный инструмент делит макеты на пакеты заданного размера. До деления отдельная функция принимает и возвращает допустимый размер пакета. На её отказах разберём pytest.raises: тест должен принимать предусмотренную ошибку ввода и отвергать другие исключения, даже если программа в любом случае прервала выполнение.

Контракт учебной функции

1. require_batch_size(size) принимает ровно встроенный int от 1 до 50 включительно и возвращает то же числовое значение.

2. Любое целое вне диапазона вызывает исключение ровно класса ValueError, а не произвольного Exception и не подкласса.

3. Любой другой тип, включая bool, float, str и None, вызывает исключение ровно класса TypeError. Преобразования типов нет.

4. У функции нет внешних эффектов. Содержание сообщения исключения можно менять: контракт закрепляет тип, не текст.

Как выбрать проверки

Отказ имеет смысл лишь относительно допустимого входа. Число 0 — подходящий тип, но запрещённое значение; строка '12' — неподходящий тип, хотя человек мог бы прочитать её как число. Мы намеренно не делаем функцию парсером пользовательского текста. Такое разделение позволяет вызывающему коду различать две категории ошибок и избавляет тесты от догадок о неоговорённых преобразованиях.

Сначала фиксируем положительную часть: 1, 12 и 50 возвращаются без изменения. Проверки нижней и верхней допустимых границ нужны и здесь. Иначе реализация, которая бросает исключение для любого аргумента, могла бы удовлетворить только отрицательным примерам. Положительная и отрицательная части вместе описывают не просто способность падать, а правило принятия входа.

Для 0 и 51 используем with pytest.raises(ValueError). Внутри блока оставляем единственное действие — вызов проверяемой функции. Если она вернётся без исключения, тест не пройдёт; если выбросит Exception другого подходящего только по общему предку класса, это тоже будет падением. Подготовку данных и последующие утверждения держим снаружи, чтобы они не выдали свою ошибку за ожидаемый отказ проверяемого вызова.

Конец ознакомительного фрагмента.