Цели и подходы к тестированию. Создание "самодельных" тестов (без библиотек)

План урока

Аннотация

Мы узнаем, что такое тестирование и зачем оно нужно. Напишем несколько простых тестов и подготовимся к использованию более продвинутых инструментов (библиотек) для тестирования в Python.

1. Когда и зачем нужно тестирование?

Мы уже изучили несколько полезных практик, которые помогают в написании больших, сложных программ. Например, использование функций и объектов. Пришло время поговорить об ещё одной важной практике — тестировании.

Чтобы лучше усвоить, когда и зачем нужно тестирование, давайте разберём некоторые отличия между маленькими учебными программами (с которыми мы имели дело на уроках, контрольных и в домашних заданиях) и большими «промышленными» программами:

В чём различие	**Учебные программы**	**Промышленные программы**	**Следствия**
Размер	Несколько десятков строк	Сотни тысяч строк	Промышленную программу невозможно удерживать в голове целиком. Следовательно, нельзя полностью предсказать результаты изменений в коде
Изменяемость	Не изменяются, т.к. условие задачи не меняется	Постоянно меняются, т.к. меняются требования	Есть риски допустить ошибку при изменении программы (см. также предыдущий пункт)
Командная работа	Пишутся в одиночку (если вы, конечно, не списываете)	Пишутся несколькими разработчиками	Каждый разработчик может отвечать только за свои части кода и не знать, как устроен остальной код
Время жизни	Обычно — один урок	Месяцы и годы	В каждый момент времени нужна уверенность, что программа работает правильно

Мы выяснили, что промышленные программы, с одной стороны, гораздо сложнее учебных, а с другой стороны — должны работать правильно на протяжении гораздо более длительного времени, при этом постоянно меняясь.

Тестирование — это проверка корректности работы программы путём сравнения результата её работы с ожидаемым при заданных входных данных.

Тестирование нужно для того, чтобы снизить вероятность ошибок в программе при ее создании или изменении.

Один из способов тестирования программы — это юнит-тестирование, т.е. проверка правильности работы отдельных независимых компонентов программы (функций или методов). Юнит-тестирование полезно на всех этапах написания каждого из компонентов программы:

- Даже если компонент ещё не реализован, вы уже можете написать тесты, проверяющие его функциональность. Такой подход называется test-driven development;
- Если вы меняете уже написанный компонент (например, добавляете новую функциональность или исправляете найденную ошибку), то вы предварительно можете написать тесты, которые проверяют, что новая функциональность работает правильно

или что ошибка больше не повторяется. А если до этого уже были написаны тесты, то вы заодно сможете проверить, что ваши изменения не приводят к поломкам старой функциональности.

2. Пример написания тестов

Разберём совсем простой пример. Пусть нам нужно написать функцию, которая принимает на вход строку и «разворачивает» её в обратном порядке. Сперва напишем тесты к ней, а саму функцию пока не будем реализовывать:

```
def reverse(s):
    # Пока что наша функция ничего не делает
    pass
```

Общая рекомендация такая — всегда проверяйте как минимум три случая:

1. Неправильные входные данные. В нашем случае функция принимает на вход только строки, все остальные типы считаются ошибочными;
2. Граничные случаи. Например, пустые строки и массивы, или границы диапазона входных значений; Обычные случаи. Например, случайные корректные входные данные.
3. Мы подробнее разберём приёмы написания тестов на следующем уроке, после знакомства с инструментами для тестирования (библиотеки unittest и pytest).

А пока напишем тесты сами в виде функции test_reverse(). Она будет запускать функцию reverse() на разных входных данных и сравнивать полученный результат с ожидаемым. Если все такие сравнения успешны (т.е., если «тесты пройдены»), функция test_reverse() вернёт True, в противном случае — False.

```
def test_reverse():
    # Список тестов
    # Каждый тест – это пара (входное значение, ожидаемое выходное значение)
    test_data = (
        (42, None),               # неправильный тип входного аргумента, ни с чем
        (['a', 'b', 'c'], None),  # тоже неправильный входной аргумент, но он "по
                                  # (можно итерироваться и брать срезы)
        ('', ''),                 # "граничный" случай – пустая строка
        ('aba', 'aba'),           # "особый" случай – строка, которая не меняется
        ('a', 'a'),               # ещё один "особый" и почти "граничный" случай
        ('abc', 'cba'),           # "обычный" случай
    )

    for input_s, correct_output_s in test_data:
```

```
        try:
            # Вычисляем результат на входных данных
            # Есть вариант, что наша функция выбросит исключение, поэтому делаем
            output_s = reverse(input_s)
        except TypeError as E:
            if correct_output_s is None:
                # это исключение и ожидалось, продолжаем тестирование
                continue
            if type(input_s) == str:
                # вход корректный, но выброшено исключение TypeError — это ошибка
                print('Ошибка! Не удалось вычислить reverse("{}"). Ошибка: {}'.fo
                return False
        except Exception as E:
            # Выброшено неожиданное исключение — это ошибка
            print('Ошибка! Не удалось вычислить reverse("{}"). Ошибка: {}'.format
            return False
        else:
            if output_s != correct_output_s:
                # если ответ не совпал с ожидаемым, завершаем тестирование и возв
                print('Ошибка! reverse({}) равно {} вместо "{}"'.format(input_s,
                return False
    # тестирование успешно пройдено
    print('Все тесты пройдены успешно')
    return True
```

```
# пока что функция reverse() не реализована, и тесты не проходят
test_reverse()
```

Ошибка! reverse() равно None вместо ""

```
False
```

Попробуем следующую реализацию reverse:

```
def reverse(s):
    r = ''
    for c in s:
        r = c + r
    return r
# проверим, что теперь тесты проходят
test_reverse()
```

Ошибка! reverse(['a', 'b', 'c']) равно cba вместо "None"

```
False
```

Точно, мы забыли проверить входное значение на корректность.

Исправляем...

```
def reverse(s):
    if type(s) != str:
        raise TypeError()
    r = ''
    for c in s:
        r = c + r
    return r
# проверим, что теперь тесты проходят
test_reverse()
```

Все тесты пройдены успешно.

```
True
```

Давайте поупражняемся ещё немного на **рекурсивной** функции.

```
def reverse(s):
    if type(s) != str:
        raise TypeError()
    # если строка состоит из одного символа, то разворачивать её не нужно
    # кажется, логично...
    if len(s) == 1:
        return s
    return s[-1] + reverse(s[:-1])
test_reverse()
```

Ошибка! Не удалось вычислить reverse("")

```
False
```

Что не так? Конечно! Мы забыли про граничный случай — пустую строку. При написании рекурсивных функций очень легко ошибиться именно в граничных случаях.

Давайте исправим.

```
def reverse(s):
    if type(s) != str:
        raise TypeError()
    # если строка пустая или состоит из одного символа,
    # то разворачивать её не нужно
    if len(s) <= 0:
        return s
    return s[-1] + reverse(s[:-1])
test_reverse()
```

Все тесты пройдены успешно.

```
True
```

Кажется, нашу функцию можно сильно упростить. Тесты помогают проконтролировать, что после изменений ничего не сломалось.

```
def reverse(s):
    if type(s) != str:
        raise TypeError()
    return s[::-1]
test_reverse()
```

Все тесты пройдены успешно.

```
    True
```

Ну и давайте проверим, что наша test_reverse корректно отрабатывает исключительные ситуации в функции reverse:

```
def reverse(s):
    # пишем заведомо ошибочный код
    return 1/0
test_reverse()
```

Ошибка! Не удалось вычислить reverse("42"). Ошибка: division by zero

```
False
```

Кажется, нам всё удалось. Давайте решать задачи!

3. Обсуждение: почему неудобно писать тесты самостоятельно

В чём недостатки нашей тестовой функции test_reverse()?

- Код теста получился сложнее, чем код тестируемой функции! Конечно, отчасти это из-за того, что мы выбрали очень простую функцию reverse() в качестве примера. Но всё равно хотелось бы, чтобы по тесту было легко понять, что он делает. На самом деле, тесты — это ещё и неявный способ документирования, поэтому **писать и читать тесты должно быть просто**.

В частности, сложно **обрабатываются исключения**. Есть шанс допустить ошибку в тестирующей функции, а этого ни в коем случае не должно быть. Представьте, что у нас не одна функция reverse(), а, скажем, сто разных.

- К каждой из них придётся писать свою тестирующую функцию. Если каждая будет такой же сложной, то на тестирование станет уходить слишком много времени;
- При каждом изменении какой-то из функций нужно найти соответствующую ей тестовую функцию, запустить её и убедиться, что тесты по-прежнему проходят.

Иными словами, нет **автоматизации тестирования**.

Отчёт об ошибках тоже не очень удобен. Где именно произошла ошибка — при обработке неправильного входа, в граничном случае или в «обычном» случае?

Конечно, все указанные нами недостатки можно исправить самостоятельно. Но в этом нет необходимости, поскольку в языке Python есть удобные инструменты для тестирования, с которыми мы познакомимся на следующем уроке.

4. Резюме

- Мы узнали, что такое тестирование (проверка корректности путём сравнения выходных данных с ожидаемыми), и зачем оно нужно (чтобы защититься от ошибок при изменениях в больших программах);
- Мы познакомились с понятием юнит-тестирования (тестирование независимых компонент программы);

- Мы написали тесты для простой функции, с их помощью нашли ошибки и проверили корректность программы после внесения в неё изменений;
- Мы убедились, что для полноценного тестирования необходимы более удобные инструменты — о них мы узнаем на следующем уроке.
- Ваша практическая работа будет построена по следующему принципу:

    1. Сначала вам надо будет для уже созданной функции написать тесты

    2. А потом попытаться самостоятельно воссоздать исходную функцию