

ПМЗ Разработка модулей ПО.

РО 3.1 Понимать и применять принципы объектно-ориентированного и асинхронного программирования.

Тема 8. Вложенные циклы. Отладка программ.

Цель занятия:

Сформировать понимание принципа работы вложенных циклов, научить анализировать и отлаживать программы с несколькими уровнями циклов, а также применять вложенные циклы совместно с условными операторами для решения практических задач.

Учебные вопросы:

1. Определение вложенного цикла.
2. Вложенные циклы **for** и **while**
3. Комбинированные алгоритмы: циклы и условия
4. Решение задач с вложенными циклами.
5. Основы отладки программ.

1. Определение вложенного цикла

1. Определение вложенного цикла

Вложенный цикл — это цикл, который расположен внутри тела другого цикла.

Цикл, в котором находится другой цикл, называется внешним, а цикл внутри него — внутренним.

Общий принцип: На каждую итерацию внешнего цикла внутренний цикл выполняется полностью.

Общая схема вложенного цикла:

Порядок выполнения:

- Внешний цикл делает одну итерацию (**i = 0**);
- Внутренний цикл выполняется полностью (**j = 0, 1**);
- Внешний цикл переходит к следующей итерации (**i = 1**);
- Внутренний цикл снова выполняется полностью.

```
for i in range(3):      # внешний цикл
    for j in range(2):  # внутренний цикл
        print(i, j)
```

Пример:

```
for i in range(2):  
    for j in range(3):  
        print(f"i={i}, j={j}")
```

i=0, j=0

i=0, j=1

i=0, j=2

i=1, j=0

i=1, j=1

i=1, j=2

Область применения вложенных циклов

- Вложенные циклы используются, когда необходимо:
- обрабатывать таблицы и матрицы;
- перебирать двумерные структуры данных;
- сравнивать элементы между собой;
- генерировать узоры и таблицы;
- решать задачи, где один процесс зависит от другого.

Примеры задач:

- таблица умножения;
- обработка матрицы;
- поиск пар элементов;
- работа с координатами.

Ограничения и важные замечания

- Каждый уровень вложенности увеличивает сложность алгоритма.
- Глубокая вложенность затрудняет понимание кода.
- Следует избегать лишних вложенных циклов, если задачу можно решить проще.

Сложность алгоритмов при вложенных циклах

Сложность алгоритма показывает, как растёт время выполнения программы при увеличении объёма входных данных.

Для циклов:

- Один цикл с n итерациями имеет линейную сложность $O(n)$.
- Два вложенных цикла по n итераций имеют квадратичную сложность $O(n^2)$.
- Три уровня вложенности — $O(n^3)$, т.е. кубическая сложность.

Как сложность влияет на рост количества операций?

Сложность алгоритма показывает, как меняется количество операций при увеличении объёма входных данных.

Линейная сложность — $O(n)$

Если объём данных увеличивается в 2 раза:

$$n \rightarrow 2n$$

Количество операций тоже увеличивается в 2 раза:

$$O(n) \rightarrow O(2n)$$

Рост пропорциональный.

Квадратичная сложность — $O(n^2)$

При увеличении объёма данных в 2 раза:

$$n \rightarrow 2n$$

Количество операций:

$$n^2 \rightarrow (2n)^2 = 4n^2$$

То есть итераций становится в 4 раза больше.

Пример:

было 100 элементов \rightarrow 10 000 операций;

стало 200 элементов \rightarrow 40 000 операций.

Кубическая сложность — $O(n^3)$

При увеличении n в 2 раза:

$$n^3 \rightarrow (2n)^3 = 8n^3$$

Количество операций увеличивается в 8 раз

Выводы:

- Вложенные циклы позволяют решать более сложные задачи, чем одиночные циклы.
- Ключевой принцип — полное выполнение внутреннего цикла на каждой итерации внешнего.
- Вложенные циклы резко увеличивают время работы программы.
- При больших данных необходимо стремиться уменьшить количество уровней вложенности.

2. Вложенные циклы **for** и **while**

В Python можно вкладывать друг в друга любые циклы:

- for внутри for,
- while внутри while,
- for внутри while,
- while внутри for.

Главное правило остаётся неизменным:

- Внутренний цикл полностью выполняется на каждой итерации внешнего цикла.

Вложенные циклы for

Используются, когда заранее известно количество итераций.

Пример:

```
for i in range(3):      # внешний цикл
    for j in range(4):  # внутренний цикл
        print(f"i={i}, j={j}")
```

Внешний цикл управляет строками, внутренний — столбцами (часто применяется для таблиц и матриц).

Вложенные циклы while

Применяются, когда количество итераций определяется условием.

Пример:

```
i = 0
while i < 3:           # Внешний цикл
    j = 0
    while j < 2:       # Внутренний цикл
        print(i, j)
        j += 1
    i += 1
```

Важно:

Внутренний счётчик (j) необходимо инициализировать заново перед каждым запуском внутреннего цикла.

Смешанные вложенные циклы (for + while)

Возможны комбинированные конструкции.

Пример:

```
for i in range(3):    # Внешний цикл for
    j = 0
    while j < 2:      # Внутренний цикл while
        print(i, j)
        j += 1
```

Используются, когда:

- внешний цикл имеет фиксированное количество итераций;
- внутренний — зависит от условия.

3. Комбинированные алгоритмы: циклы и условия

Комбинированный алгоритм — это алгоритм, в котором циклы используются совместно с условными операторами **if**, **elif**, **else**.

Такие алгоритмы позволяют:

- выполнять повторяющиеся действия;
- принимать решения на каждой итерации;
- обрабатывать данные выборочно.

Условия внутри цикла.

Наиболее распространённый вариант — условие внутри цикла.

Пример:

```
for x in data:  
    if x > 0:  
        print(x)
```

Условие проверяется на каждой итерации цикла.

Условные операторы внутри цикла используются для:

- фильтрации данных;
- поиска элементов;
- подсчёта по заданному критерию;
- изменения поведения алгоритма.

Пример: подсчёт

```
count = 0
for x in nums:
    if x % 2 == 0:
        count += 1
```

В комбинированных алгоритмах условия могут находиться:

- во внутреннем цикле;
- во внешнем цикле;
- одновременно в обоих.

Пример:

```
for i in range(5):  
    for j in range(5):  
        if i == j:  
            print(1, end=" ")  
        else:  
            print(0, end=" ")  
    print()
```

Управляющие операторы `break` и `continue`.

Оператор `break`:

- Пропускает текущую итерацию и переходит к следующей.
- Во вложенных циклах `break` прерывает только тот цикл, в котором он расположен.

Пример, поиск первого отрицательного элемента в матрице:

```
matrix = [  
    [1,2,3],  
    [4,5,-6],  
    [7,8,9],  
]  
  
found = False  
for row in matrix:  
    for x in row:  
        if x < 0:  
            print(x)  
            found = True  
            break  
    if found:  
        break
```

continue во вложенных циклах

Во вложенных циклах `continue` влияет только на тот цикл, где он расположен.

Пример:

```
for i in range(3):  
    for j in range(3):  
        if j == 1:  
            continue  
        print(i, j)
```

Результат:

значения с `j == 1` пропускаются;

внешний цикл продолжает работу.

Комбинированные алгоритмы позволяют:

- управлять логикой выполнения циклов;
- решать сложные задачи обработки данных;
- эффективно работать с таблицами и матрицами.

Грамотное сочетание циклов и условий — основа алгоритмического мышления и написания корректных программ.

4. Решение задач с вложенными циклами.

Таблица умножения

```
for i in range(1, 10):  
    for j in range(1, 10):  
        print(f"{i} * {j} = {i*j}", end="\t")  
    print() # переход на новую строку
```

Прямоугольник из символов

```
rows = 5
cols = 10

for i in range(rows):
    for j in range(cols):
        print("*", end="")
    print()
```

Поиск простых чисел

```
n = 100
print("Простые числа от 2 до", n, ":")

for num in range(2, n + 1):
    is_prime = True
    # Вложенный цикл проверяет делители
    for i in range(2, int(num**0.5) + 1):
        if num % i == 0:
            is_prime = False
            break
    if is_prime:
        print(num, end=" ")
```

Работа с двумерными списками (матрицами)

```
# Создание матрицы 3x4 и заполнение её значениями
matrix = []
for i in range(3):
    row = []
    for j in range(4):
        row.append(i * 4 + j)
    matrix.append(row)
```

```
# Вывод матрицы в читаемом виде
for row in matrix:
    for element in row:
        print(f"{element:3}", end=" ")
    print()
```

5. Основы отладки программ.

Отладка программы — это процесс поиска, анализа и исправления ошибок в коде.

Цель отладки:

- добиться корректной работы программы;
- получить ожидаемый результат выполнения;
- понять причину возникновения ошибки.

Виды ошибок в программах

1. Синтаксические ошибки

Возникают при нарушении правил языка Python.

Примеры:

- пропущена двоеточие `::`;
- неправильные отступы;
- опечатки в ключевых словах.

Такие ошибки выявляются до выполнения программы.

2. Ошибки времени выполнения (runtime errors)

Возникают во время работы программы.

Примеры:

- деление на ноль;
- выход за пределы индексов;
- обращение к несуществующему ключу.

3. Логические ошибки

Программа выполняется без ошибок, но результат неверный.

Примеры:

- неверное условие;
- ошибка в формуле;
- неправильная логика цикла.

Наиболее сложный вид ошибок.

Основные приёмы отладки

1. Использование `print()`

Самый простой и доступный способ отладки.

Позволяет:

- отслеживать значения переменных;
- контролировать порядок выполнения кода.

2. Пошаговый разбор алгоритма

- выполнение программы «вручную», когда программист симулирует работу программы на бумаге или в голове.;
- проверка каждой итерации цикла;
- анализ промежуточных результатов.

3. Использование режима отладки.

Что такое режим Debug в PyCharm?

Debug (отладка) — это режим выполнения программы, который позволяет пошагово отслеживать работу кода:

- видеть значения переменных на каждом шаге;
- контролировать порядок выполнения;
- проверять работу циклов и условий;
- находить логические и runtime ошибки.

Основные возможности режима Debug:

- Точки останова (breakpoints):
 - ставятся кликом слева от номера строки;
 - программа останавливается на этой строке во время выполнения.
- Пошаговое выполнение:
 - Step Over (F8) — выполнить строку и перейти к следующей;
 - Step Into (F7) — войти внутрь вызываемой функции;
 - Step Out (Shift+F8) — выйти из функции и продолжить внешний цикл.
- Просмотр значений переменных:
 - Значения переменных отображаются в окне Variables;
 - Можно изменять значения переменных вручную во время отладки.
- Выражения и Watches. Можно добавлять наблюдение за выражениями (Watches) для отслеживания конкретных переменных или формул.
- Консоль Debug. Позволяет выполнять команды Python прямо во время остановки программы.

Пример отладки программы с логической ошибкой.

```
# Программа считает сумму всех чётных чисел в списке
nums = [1, 2, 3, 4, 5, 6]
total = 0

for x in nums:
    if x % 2 == 1: # Ошибка: проверка на нечётное вместо чётного
        total += x

print("Сумма чётных чисел:", total)
```

Что происходит:

- Логическая ошибка: условие **$x \% 2 == 1$** добавляет нечётные числа, хотя программа должна суммировать чётные.
- Вывод будет: Сумма чётных чисел: 9 вместо правильного $2+4+6=12$.

Как отлаживать через Debug в PyCharm?

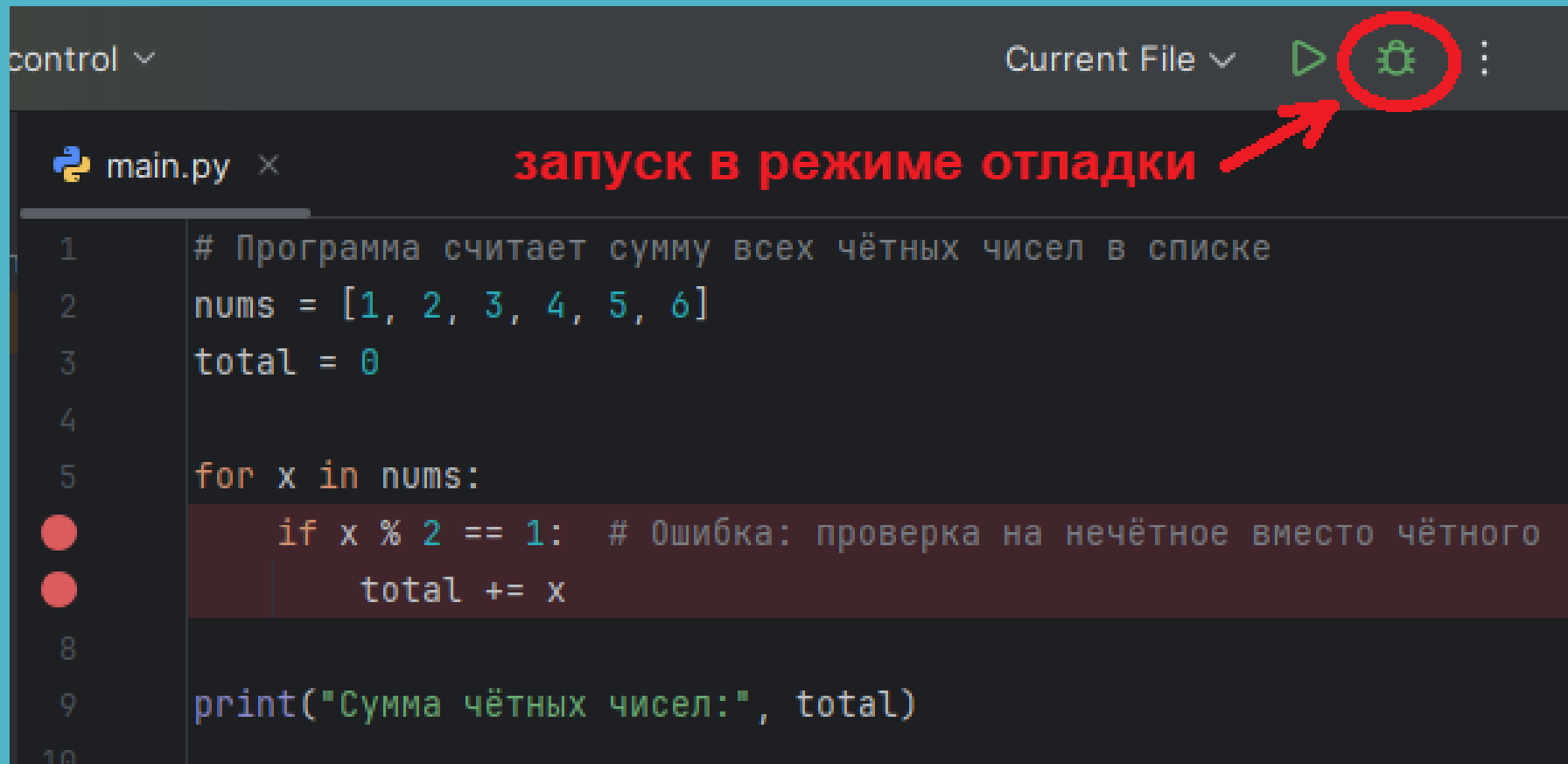
1. Установка точки останова

- Ставим **breakpoint** на строке `if x % 2 == 1:` или на `total += x`.

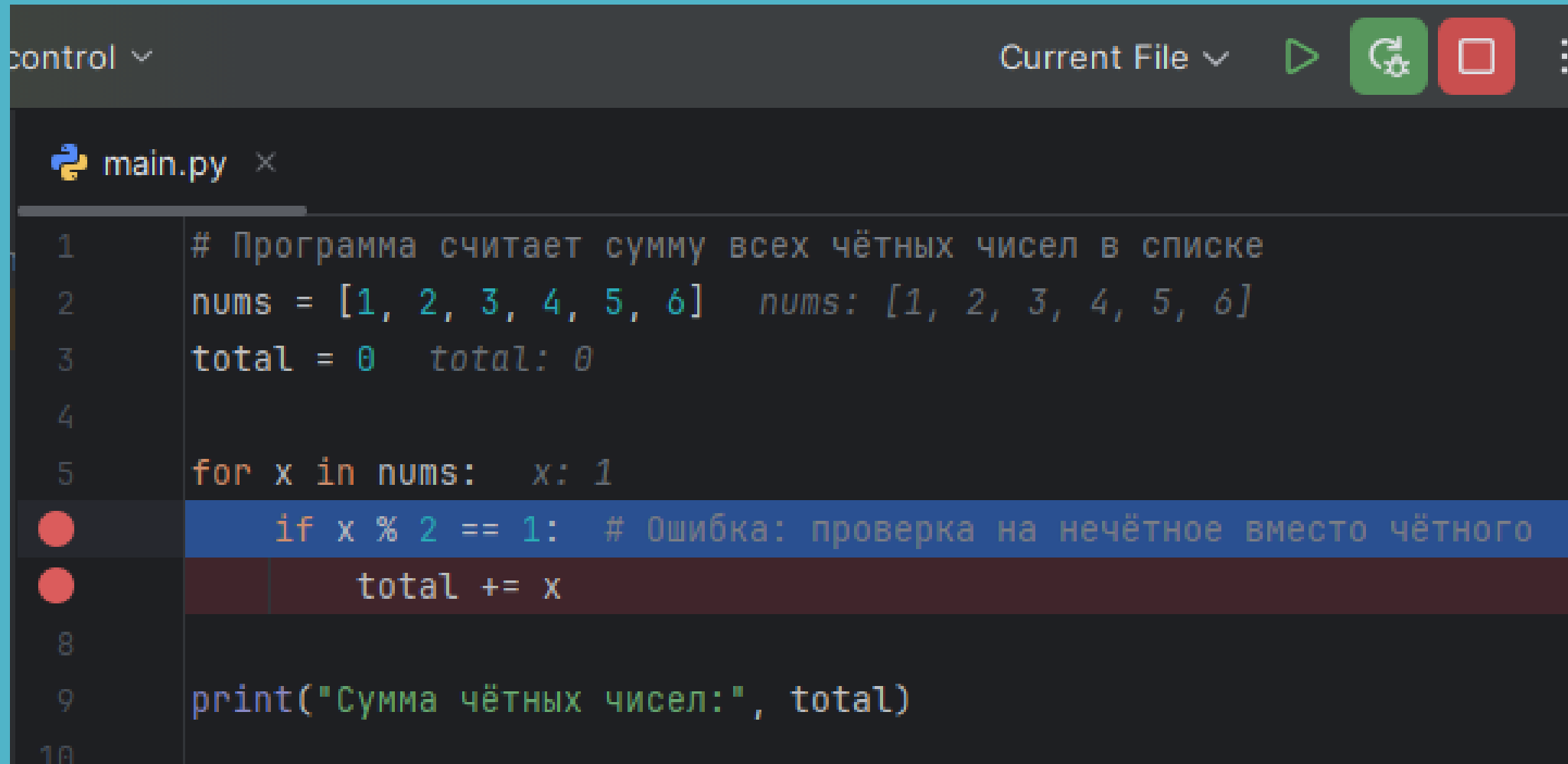
```
1  # Программа считает сумму всех чётных чисел в списке
2  nums = [1, 2, 3, 4, 5, 6]
3  total = 0
4
5  for x in nums:
6      if x % 2 == 1: # Ошибка: проверка на нечётное вместо чётного
7          total += x
8
9  print("Сумма чётных чисел:", total)
```

2. Запуск в режиме Debug

- Нажимаем Shift+F9 или кнопку жука.

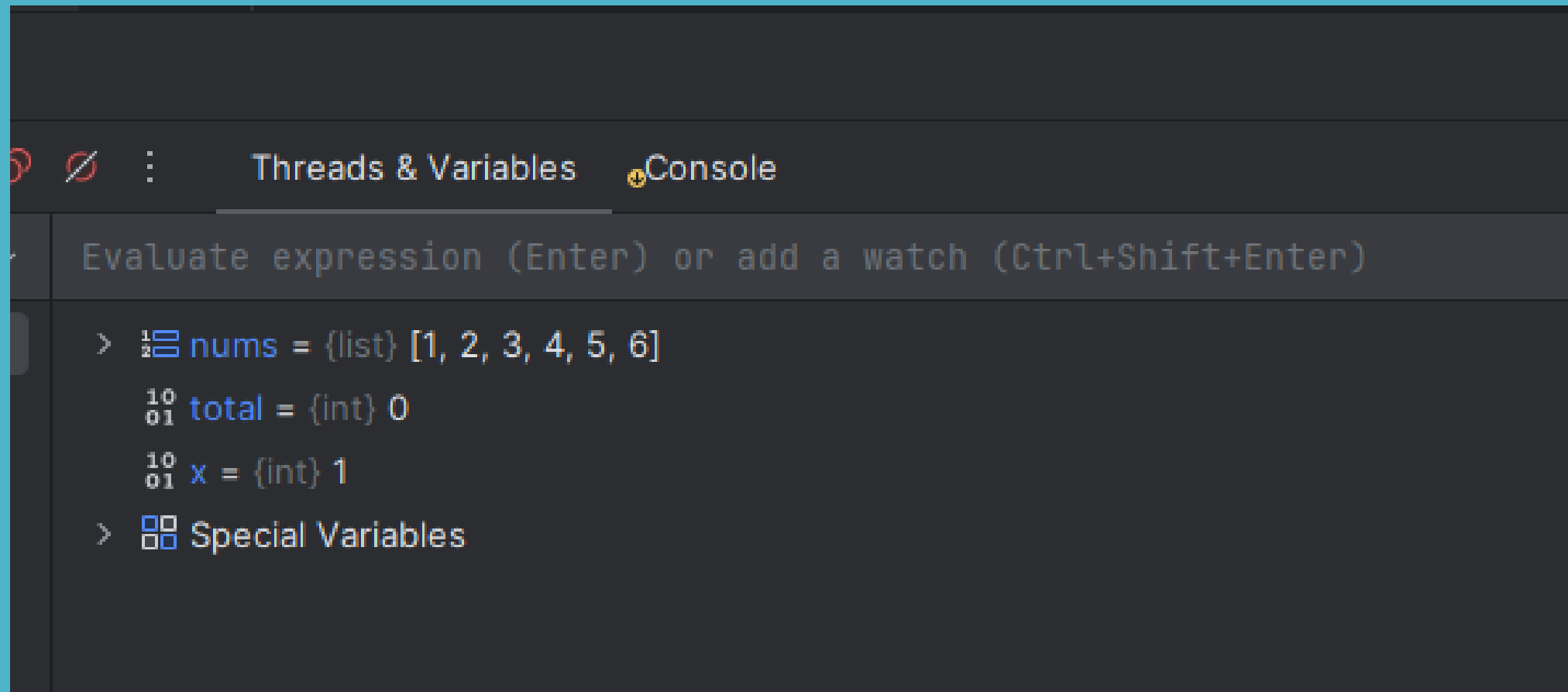


- Программа останавливается на первом breakpoint.



```
control ▾ Current File ▾ ▶ ⚙️ □ ⋮  
  
main.py ×  
1 # Программа считает сумму всех чётных чисел в списке  
2 nums = [1, 2, 3, 4, 5, 6]  nums: [1, 2, 3, 4, 5, 6]  
3 total = 0  total: 0  
4  
5 for x in nums:  x: 1  
●   if x % 2 == 1: # Ошибка: проверка на нечётное вместо чётного  
●   total += x  
8  
9 print("Сумма чётных чисел:", total)  
10
```

- Проверяем значение переменной **x** на каждой итерации и смотрим значение **total**.



The screenshot shows the 'Threads & Variables' panel in an IDE. The 'Console' tab is active, displaying the following content:

```
Evaluate expression (Enter) or add a watch (Ctrl+Shift+Enter)
```

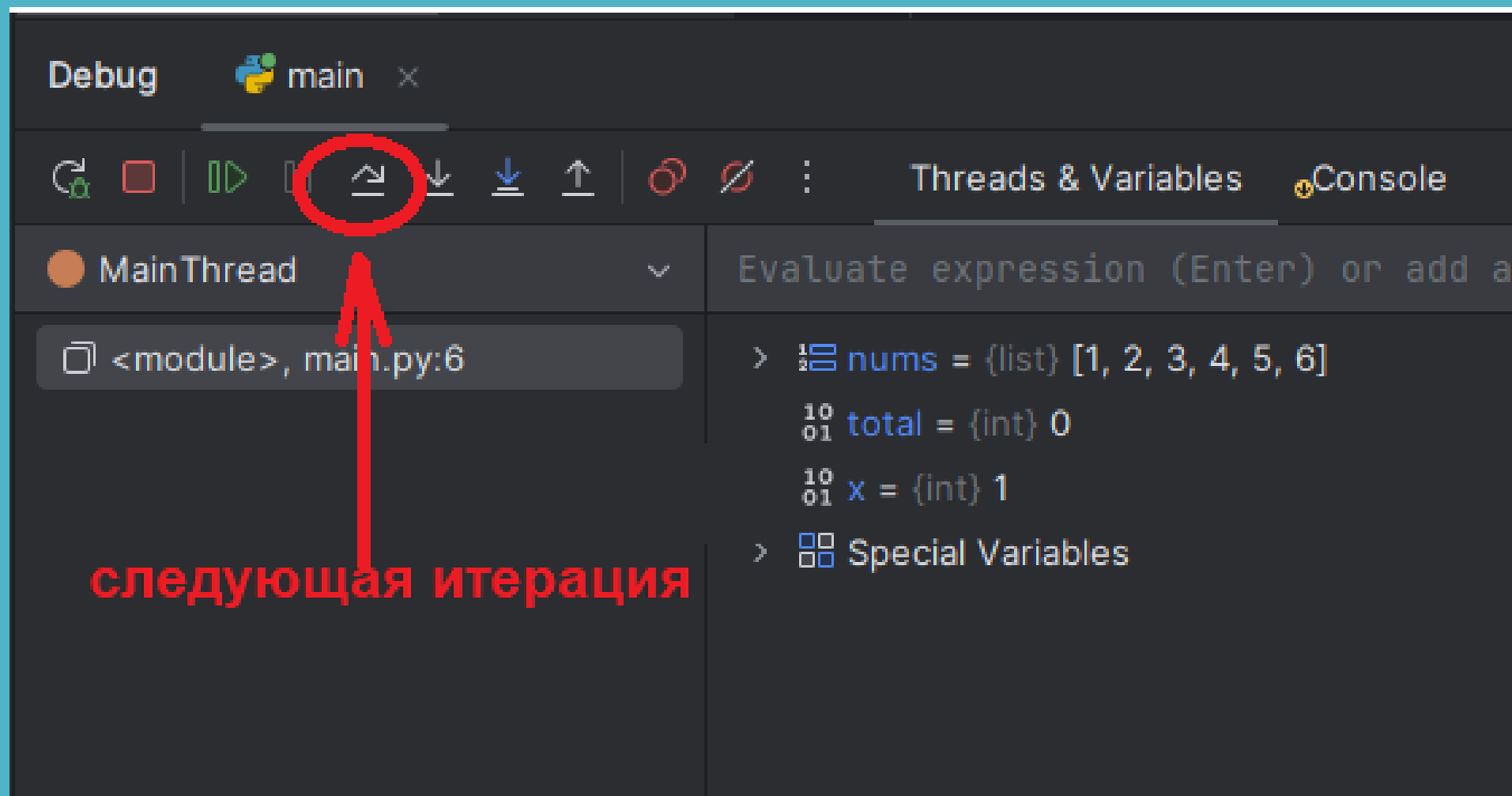
```
> nums = {list} [1, 2, 3, 4, 5, 6]
```

```
  total = {int} 0
```

```
  x = {int} 1
```

```
> Special Variables
```

- Используем Step Over (F8) для перехода к следующей итерации.





Threads & Variables



Console



Evaluate expression (Enter) or add a watch (Ctrl+Shi

>  nums = {list} [1, 2, 3, 4, 5, 6]

 total = {int} 1

 x = {int} 1

>  Special Variables



Threads & Variables



Console



Evaluate expression (Enter) or add a watch (Ctrl+Shift)

>  nums = {list} [1, 2, 3, 4, 5, 6]

01  total = {int} 4

10
01  x = {int} 3

>  Special Variables

Видим, что суммируются нечётные числа, логическая ошибка очевидна.

Исправление ошибки.

Меняем условие:

if x % 2 == 0: # теперь проверка на чётное

Запускаем снова — результат правильный: Сумма чётных чисел: 12.

Вывод:

- Использование debug помогает:
- пошагово пройти по циклу и условиям;
- увидеть, что именно работает неправильно;
- выявить логические ошибки без множества `print()`;
- быстро тестировать исправления.

Вывод по теме лекции:

- Вложенные циклы позволяют решать задачи, требующие перебора элементов в нескольких измерениях или комбинаций значений.
 - Важно понимать, что внутренний цикл полностью выполняется на каждой итерации внешнего.
 - Сложность алгоритма возрастает с увеличением числа уровней вложенности: один цикл — $O(n)$, два вложенных — $O(n^2)$ и т.д.
- Комбинированные алгоритмы — это сочетание циклов и условий (if, elif, else), которое позволяет управлять логикой выполнения и выполнять действия только при соблюдении определённых условий.
 - Использование break и continue помогает контролировать ход выполнения циклов и пропускать или завершать итерации по необходимости.
- Отладка программ — необходимый этап разработки, который помогает находить и исправлять ошибки:
 - Синтаксические ошибки выявляются сразу;
 - Runtime ошибки появляются во время работы программы;
 - Логические ошибки требуют внимательного анализа или использования debug.

Контрольные вопросы:

- Что называется вложенным циклом?
- Сколько раз выполняется внутренний цикл?
- В каком порядке выполняются вложенные циклы?
- Можно ли использовать break во внутреннем цикле? Как это влияет на внешний?
- Назовите типичные ошибки при работе с вложенными циклами.
- Чем логическая ошибка отличается от синтаксической?
- Какие способы отладки программ вы знаете?
- Почему важно пошагово анализировать выполнение программы?

Домашнее задание:

<https://ru.hexlet.io/courses/js-asynchronous-programming>

Материалы лекций:

<https://github.com/ShViktor72/Education>

Обратная связь:

colledge20education23@gmail.com