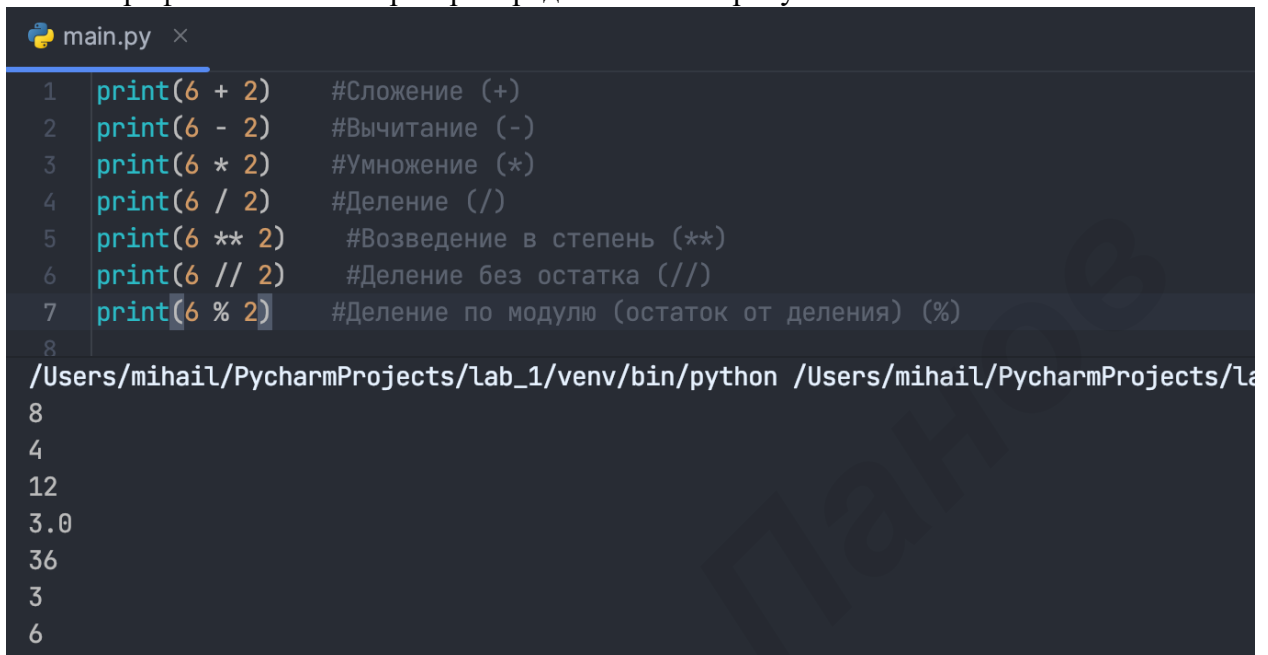


ТЕМА 3. ОПЕРАТОРЫ, УСЛОВИЯ, ЦИКЛЫ.

3.1. Основные операторы Python.

Операторы Python бывают 7 типов:

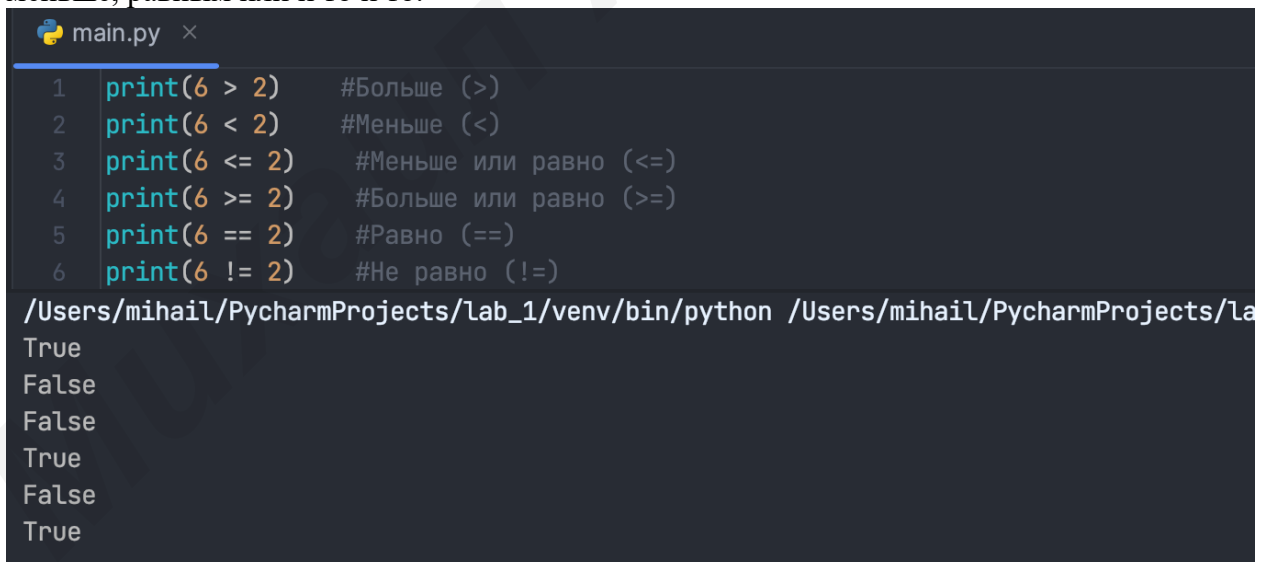
Арифметические операторы представлены на рисунке 3.1.



```
main.py x
1 print(6 + 2) #Сложение (+)
2 print(6 - 2) #Вычитание (-)
3 print(6 * 2) #Умножение (*)
4 print(6 / 2) #Деление (/)
5 print(6 ** 2) #Возведение в степень (**)
6 print(6 // 2) #Деление без остатка (//)
7 print(6 % 2) #Деление по модулю (остаток от деления) (%)
8
/Users/mihail/PycharmProjects/lab_1/venv/bin/python /Users/mihail/PycharmProjects/lab_1/main.py
8
4
12
3.0
36
3
6
```

Рис. 3.1. Арифметические операторы.

Операторы сравнения представлены на рисунке 3.2. Операторы сравнения в Python проводят сравнение операндов. Они сообщают, является ли один из них больше второго, меньше, равным или и то и то.



```
main.py x
1 print(6 > 2) #Больше (>)
2 print(6 < 2) #Меньше (<)
3 print(6 <= 2) #Меньше или равно (<=)
4 print(6 >= 2) #Больше или равно (>=)
5 print(6 == 2) #Равно (==)
6 print(6 != 2) #Не равно (!=)
/Users/mihail/PycharmProjects/lab_1/venv/bin/python /Users/mihail/PycharmProjects/lab_1/main.py
True
False
False
True
False
True
```

Рис. 3.2. Операторы сравнения.

Операторы присваивания представлены на рисунке 3.3. Данный оператор принимает (присваивает) значение переменной.

```
main.py ×
1  a = 7 #Присваивание (=)
2  print(a)
3
4  a += 2 #Сложение и присваивание (+=)
5  print(a)
6
7  a -= 3 #Вычитание и присваивание (-=)
8  print(a)
9
10 a /= 7 #Деление и присваивание (/=)
11 print(a)
12
13 a *= 8 #Умножение и присваивание (*=)
14 print(a)
15
16 a %= 3 #Деление по модулю и присваивание (%=)
17 print(a)
18
19 a **= 5 #Возведение в степень и присваивание (**=)
20 print(a)
21
22 a //= 3 #Деление с остатком и присваивание (//=)
23 print(a)
/Users/mihail/PycharmProjects/lab_1/venv/bin/python /Users/mihail/PycharmProjects/lab_1/main.py
7
9
6
0.8571428571428571
6.857142857142857
0.8571428571428568
0.46266436603795935
0.0
```

Рис. 3.3. Операторы присваивания.

Логические операторы представлены на рисунке 3.4. Это союзы, которые позволяют объединять по несколько условий. В Python есть всего три оператора: *and* (и), *or* (или) и *not* (не).

```
main.py x
1 a = 7 > 7 and 2 > -1 # И (and)
2 print(a)
3
4 a = 7 > 7 or 2 > -1 # ИЛИ (or)
5 print(a)
6
7 a = not 0 # НЕ (not)
8 print(a)

/Users/mihail/PycharmProjects/lab_1/venv/bin/python /Users/mihail/PycharmProjects/La
False
True
True

Process finished with exit code 0
```

Рис. 3.4. Логические операторы.

Операторы принадлежности. Эти операторы проверяют, является ли значение частью последовательности. Есть всего два таких оператора: *in* и *not in*.

in: используется для проверки, присутствует ли элемент в последовательности.

not in: используется для проверки, отсутствует ли элемент в последовательности.

Вот несколько примеров использования операторов принадлежности в Python:

Пример 1: Проверка наличия элемента в списке (рисунок 3.5.)

```
main.py x
1 fruits = ['apple', 'banana', 'orange']
2 if 'apple' in fruits:
3     print("Я люблю яблоки!")

/Users/mihail/PycharmProjects/lab_1/venv/bin/python /Users/mihail/PycharmProjects/La
Я люблю яблоки!

Process finished with exit code 0
```

Рис. 3.5. Проверка наличия элемента в списке.

Пример 2: Проверка наличия символа в строке (рисунок 3.6.)

```
main.py x
1 name = 'John Doe'
2 if 'D' in name:
3     print("Буква 'D' присутствует в имени.")

/Users/mihail/PycharmProjects/lab_1/venv/bin/python /Users/mihail/PycharmProjects/La
Буква 'D' присутствует в имени.

Process finished with exit code 0
```

Рис. 3.6 Проверка наличия символа в строке.

Пример 3: Проверка наличия ключа в словаре (рисунок 3.7.)

```
main.py x
1 student = {'name': 'John', 'age': 20, 'grade': 'A'}
2 if 'age' in student:
3     print("Ключ 'age' присутствует в словаре.")
/Users/mihail/PycharmProjects/lab_1/venv/bin/python /Users/mihail/PycharmProjects/lab_1/main.py
Ключ 'age' присутствует в словаре.
Process finished with exit code 0
```

Рис. 3.7 Проверка наличия ключа в словаре.

Пример 4: Проверка наличия элемента в множестве (рисунок 3.8.)

```
main.py x
1 numbers = {1, 2, 3, 4, 5}
2 if 3 in numbers:
3     print("Число 3 содержится в множестве.")
/Users/mihail/PycharmProjects/lab_1/venv/bin/python /Users/mihail/PycharmProjects/lab_1/main.py
Число 3 содержится в множестве.
Process finished with exit code 0
```

Рис. 3.8 Проверка наличия элемента в множестве.

Пример 5: Проверка наличия элемента в кортеже (рисунок 3.9.)

```
main.py x
1 point = (2, 4)
2 if 2 in point:
3     print("Кортеж содержит число 2.")
/Users/mihail/PycharmProjects/lab_1/venv/bin/python /Users/mihail/PycharmProjects/lab_1/main.py
Кортеж содержит число 2.
Process finished with exit code 0
```

Рис. 3.9 Проверка наличия элемента в кортеже.

Во всех этих примерах *in* и *not in* используется для проверки наличия/отсутствия элемента в указанной последовательности, такой как список, строка, словарь, множество или кортеж. Если элемент присутствует/отсутствует, условие истинно/ложно и соответствующий код выполняется/не выполняется.

Операторы тождественности. Операторы тождественности в Python используются для сравнения двух объектов и проверки их идентичности. В Python есть два оператора тождественности:

is: проверяет, являются ли два объекта одним и тем же объектом.

is not: проверяет, являются ли два объекта разными объектами.

Примеры использования операторов тождественности:

Пример 1: Использование оператора *is* (рисунок 3.10.)

```
main.py x
1 x = [1, 2, 3]
2 y = x
3
4 if x is y:
5     print("x и y указывают на один и тот же объект.")
6
7 z = [1, 2, 3]
8 if x is z:
9     print("x и z указывают на один и тот же объект.")
10 else:
11     print("x и z указывают на разные объекты.")
12

/Users/mihail/PycharmProjects/lab_1/venv/bin/python /Users/mihail/PycharmProjects/lab_1/main.py
x и y указывают на один и тот же объект.
x и z указывают на разные объекты.
```

Рис. 3.10. Использование оператора *is*.

Пример 2: Использование оператора *is not* (рисунок 3.11.)

```
main.py x
1 a = "Hello"
2 b = "Hello"
3
4 if a is not b:
5     print("a и b указывают на разные объекты.")
6
7 c = [1, 2, 3]
8 if a is not c:
9     print("a и c указывают на разные объекты.")
10

/Users/mihail/PycharmProjects/lab_1/venv/bin/python /Users/mihail/PycharmProjects/lab_1/main.py
a и c указывают на разные объекты.
```

Рис. 3.11. Использование оператора *is not*.

Битовые операторы.

Битовые операторы в Python позволяют выполнять операции на уровне отдельных битов чисел. Они работают с двоичным представлением чисел и манипулируют отдельными битами для выполнения различных операций. Список битовых операторов в Python:

Побитовое И (&): возвращает единицу в битовой позиции, если оба бита в этой позиции равны единице.

Побитовое ИЛИ (|): возвращает единицу в битовой позиции, если хотя бы один из битов в этой позиции равен единице.

Побитовое Исключающее ИЛИ (^): возвращает единицу в битовой позиции, если только один из битов в этой позиции равен единице.

Побитовый сдвиг влево (<<): сдвигает биты числа влево на указанное количество позиций. Значение сдвинутых битов зависит от типа данных.

Побитовый сдвиг вправо (>>): сдвигает биты числа вправо на указанное количество позиций. Значение сдвинутых битов зависит от типа данных.

Побитовое отрицание (~): инвертирует все биты числа, заменяя 0 на 1 и 1 на 0.

Примеры использования битовых операторов в Python (рисунок 3.12.)

```
main.py ×
1 a = 0b10101010 # Двоичное число 170
2 b = 0b11001100 # Двоичное число 204
3
4 # Побитовое И
5 result_and = a & b
6 print(bin(result_and)) # Вывод: 0b10001000 (Двоичное число 136)
7
8 # Побитовое ИЛИ
9 result_or = a | b
10 print(bin(result_or)) # Вывод: 0b11101110 (Двоичное число 238)
11
12 # Побитовое Исключающее ИЛИ
13 result_xor = a ^ b
14 print(bin(result_xor)) # Вывод: 0b01100110 (Двоичное число 102)
15
16 # Побитовый сдвиг влево
17 result_left_shift = a << 2
18 print(bin(result_left_shift)) # Вывод: 0b1010101000 (Двоичное число 680)
19
20 # Побитовый сдвиг вправо
21 result_right_shift = b >> 3
22 print(bin(result_right_shift)) # Вывод: 0b110011 (Двоичное число 51)
23
24 # Побитовое отрицание
25 result_complement = ~a
26 print(bin(result_complement)) # Вывод: -0b10101011 (Двоичное число -171)
27

/Users/mihail/PycharmProjects/lab_1/venv/bin/python /Users/mihail/PycharmProjects/lab_1
0b10001000
0b11101110
0b1100110
0b1010101000
0b11001
-0b10101011
```

Рис. 3.12. Примеры использования битовых операторов в Python.

Обратите внимание, что числа, используемые в примерах, представлены в двоичной системе счисления с префиксом `0b`. Функция `bin()` используется для преобразования числа в его двоичное представление.

3.2. Условные конструкции

Условные конструкции в Python позволяют программе выполнять различные действия в зависимости от условия. Они основаны на ключевых словах `if`, `else` и `elif` (сокращение от "else if").

Основная форма условной конструкции выглядит следующим образом:

if условие1:

 # блок кода, который будет выполнен, если условие1 истинно

elif условие2:

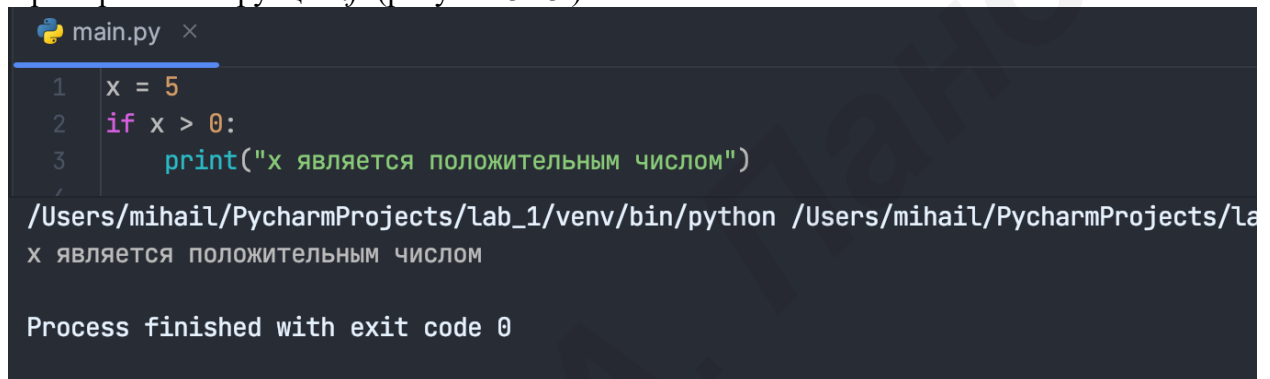
```
# блок кода, который будет выполнен, если условие1 ложно, а условие2 истинно
else:
    # блок кода, который будет выполнен, если оба условия ложны
```

Здесь сначала проверяется условие1, и если оно истинно, то выполнится первый блок кода. Если оно ложно, то будет проверено условие2, и если оно истинно, то выполнится второй блок кода. Если оба условия ложны, то выполнится третий блок кода.

В Python тело конструкции *if* обозначается отступами. Тело начинается с первой строки с отступом и заканчивается первой строкой без отступа.

Ключевые слова *if*, *else* и *elif* могут использоваться в различных комбинациях, в зависимости от конкретной задачи. Разберем это на примерах.

Пример 1. Конструкция *if*: (рисунок 3.13.)

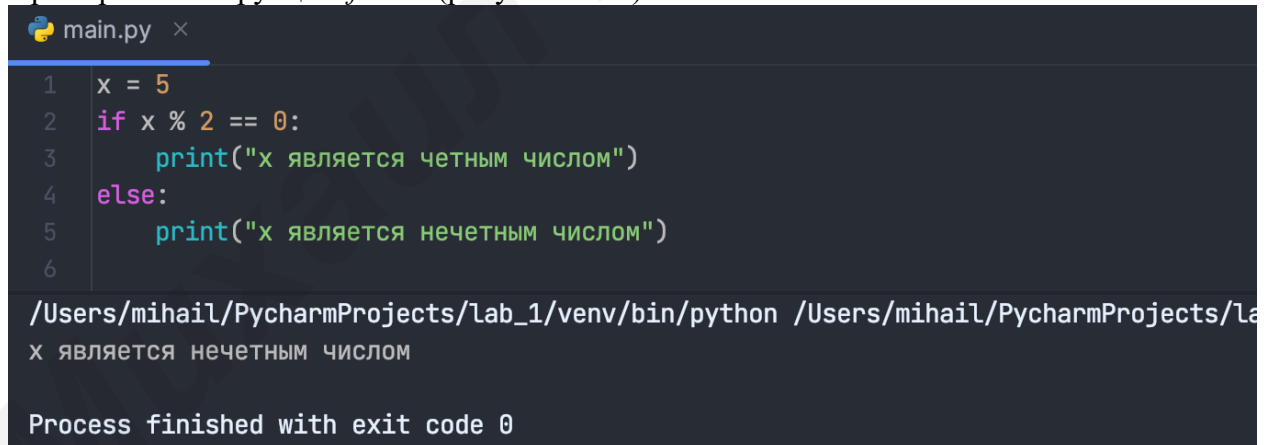


```
main.py x
1 x = 5
2 if x > 0:
3     print("x является положительным числом")
/
/Users/mihail/PycharmProjects/Lab_1/venv/bin/python /Users/mihail/PycharmProjects/La
x является положительным числом

Process finished with exit code 0
```

Рис. 3.13. Конструкция *if*.

Пример 2. Конструкция *if-else*: (рисунок 3.14.)



```
main.py x
1 x = 5
2 if x % 2 == 0:
3     print("x является четным числом")
4 else:
5     print("x является нечетным числом")
6
/Users/mihail/PycharmProjects/Lab_1/venv/bin/python /Users/mihail/PycharmProjects/La
x является нечетным числом

Process finished with exit code 0
```

Рис. 3.14. Конструкция *if-else*.

Пример 3. Конструкция *if-elif-else*: (рисунок 3.15.)

```
main.py x
1 x = 5
2 if x > 0:
3     print("x является положительным числом")
4 elif x < 0:
5     print("x является отрицательным числом")
6 else:
7     print("x равен нулю")
8
/Users/mihail/PycharmProjects/lab_1/venv/bin/python /Users/mihail/PycharmProjects/lab_1/main.py
x является положительным числом

Process finished with exit code 0
```

Рис. 3.15. Конструкция *if-elif-else*.

Пример 4. Тернарный оператор *if-else*: (рисунок 3.16.)

```
main.py x
1 x = 5
2 result = "x больше нуля" if x > 0 else "x меньше или равен нулю"
3 print(result)
4
/Users/mihail/PycharmProjects/lab_1/venv/bin/python /Users/mihail/PycharmProjects/lab_1/main.py
x больше нуля

Process finished with exit code 0
```

Рис. 3.16. Тернарный оператор *if-else*.

В условиях можно использовать операторы сравнения (например, `>`, `<`, `==`, `!=`) и логические операторы (например, *and*, *or*, *not*) для определения условий.

В Python флаги - это булевы переменные, которые используются для контроля выполнения или невыполнения определенных частей программы на основе определенного условия или состояния. Флаги могут быть установлены или сброшены в зависимости от определенных событий или условий, и код программы может реагировать на эти флаги, чтобы принять соответствующие решения (рисунок 3.17.)


```
main.py x
1 import condition as condition
2
3 flag = False
4
5 # Установка флага в зависимости от условия
6 if condition:
7     flag = True
8
9 # Использование флага для выполнения или невыполнения кода
10 if flag:
11     # Код, который должен выполняться, если флаг установлен
12     # ...
13
14 # Другой код программы
15 # ...
16
```

Рис. 3.17. Пример использования флага

В этом примере флаг устанавливается в значение *True* в зависимости от определенного условия *condition*. Затем флаг проверяется в последующем коде программы. Если флаг установлен (*True*), соответствующий блок кода выполняется. Если флаг не установлен (*False*), блок кода пропускается.

Флаги в программировании могут использоваться для различных целей, например:

- Контроль выполнения определенного блока кода в зависимости от условий или состояния программы.
- Отслеживание наличия или отсутствия определенного события или действия.
- Управление выполнением циклов или других конструкций в программе.

Флаги являются одним из способов реализации логики и контроля потока выполнения программы и позволяют гибко управлять ее поведением в зависимости от различных условий или событий.

3.3. Циклы

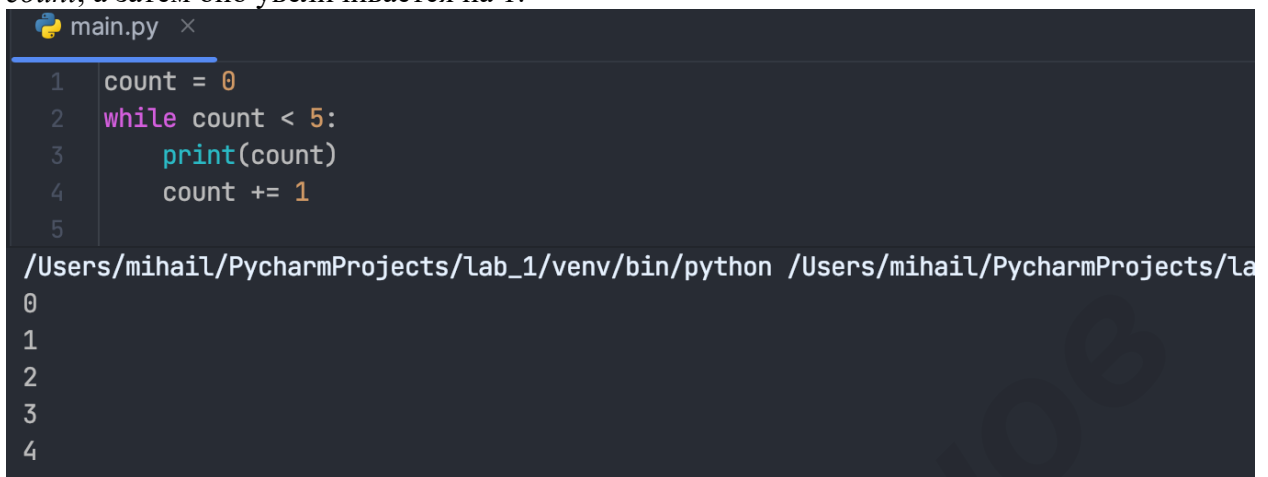
В Python существуют различные типы циклов, позволяющие выполнять повторяющиеся операции.

Пример 1. Цикл *for*: (рисунок 3.18.) Этот цикл позволяет выполнить блок кода для каждого элемента из списка *fruits*.

```
main.py x
1 fruits = ['apple', 'banana', 'orange']
2 for fruit in fruits:
3     print(fruit)
4
/Users/mihail/PycharmProjects/lab_1/venv/bin/python /Users/mihail/PycharmProjects/La
apple
banana
orange
```

Рис. 3.18. Цикл *for*.

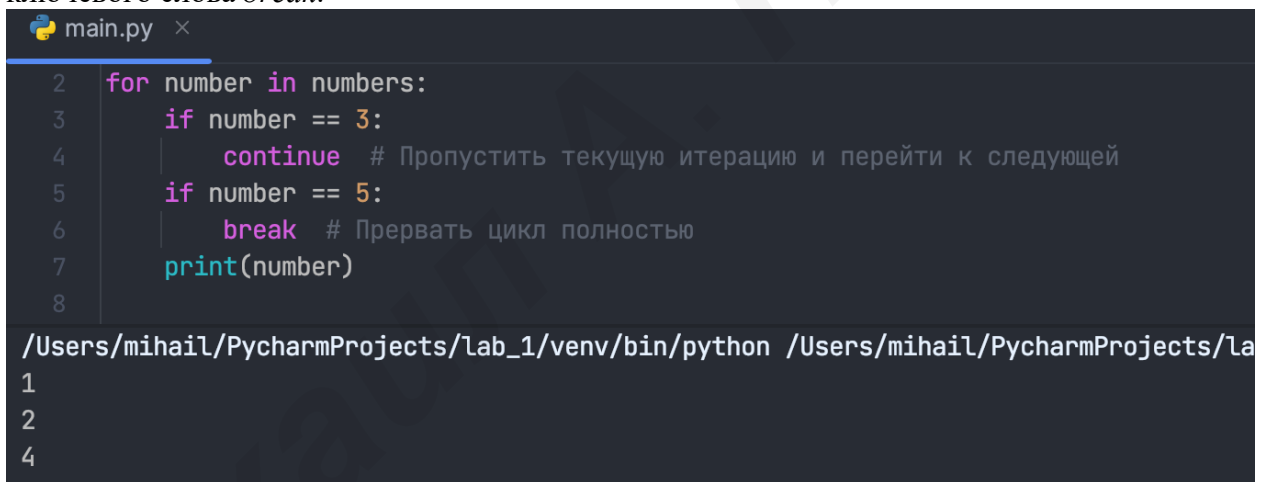
Пример 2. Цикл *while*: (рисунок 3.19.) В этом примере цикл *while* продолжается, пока условие *count < 5* истинно. Каждый раз внутри цикла выводится значение переменной *count*, а затем оно увеличивается на 1.



```
main.py x
1 count = 0
2 while count < 5:
3     print(count)
4     count += 1
5
/Users/mihail/PycharmProjects/lab_1/venv/bin/python /Users/mihail/PycharmProjects/lab_1/main.py
0
1
2
3
4
```

Рис. 3.19. Цикл *while*.

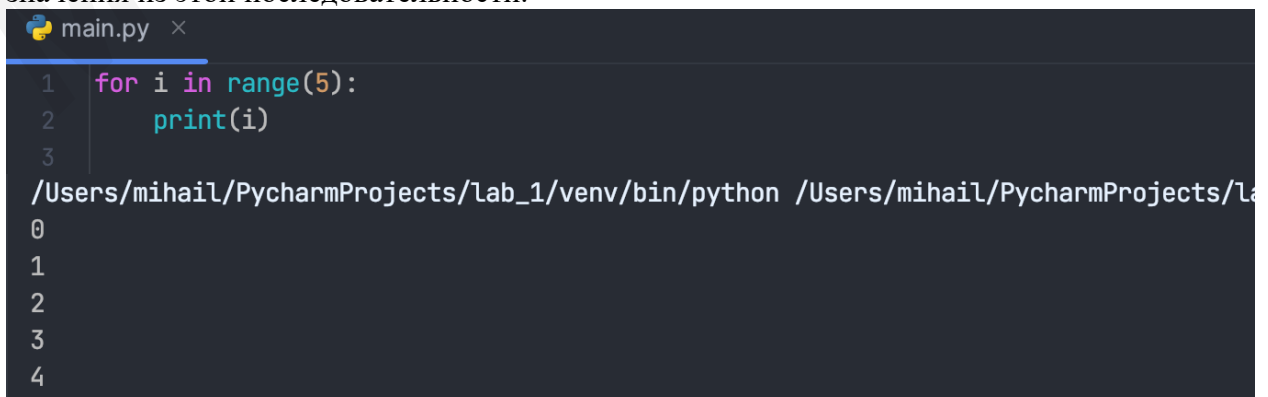
Пример 3. Использование ключевых слов *break* и *continue*: (рисунок 3.20.) В этом примере, при достижении значения 3 цикл пропускает текущую итерацию и переходит к следующей, а при достижении значения 5 он полностью прерывается с помощью ключевого слова *break*.



```
main.py x
2 for number in numbers:
3     if number == 3:
4         continue # Пропустить текущую итерацию и перейти к следующей
5     if number == 5:
6         break # Прервать цикл полностью
7     print(number)
8
/Users/mihail/PycharmProjects/lab_1/venv/bin/python /Users/mihail/PycharmProjects/lab_1/main.py
1
2
4
```

Рис. 3.20. Использование ключевых слов *break* и *continue*.

Пример 4. Использование функции *range()* в цикле *for*: (рисунок 3.21.) Функция *range(5)* генерирует последовательность чисел от 0 до 4, и цикл *for* выполняется для каждого значения из этой последовательности.



```
main.py x
1 for i in range(5):
2     print(i)
3
/Users/mihail/PycharmProjects/lab_1/venv/bin/python /Users/mihail/PycharmProjects/lab_1/main.py
0
1
2
3
4
```

Рис. 3.21. Использование функции *range()* в цикле *for*.

В контексте использования функции `range()` в Python выражения `range(0, 10)` и `range(10)` эквивалентны и создают одинаковую последовательность чисел от 0 до 9.

Функция `range()` имеет несколько вариантов использования:

1. `range(stop)`: создает последовательность чисел от 0 до `stop - 1`.

Пример: `range(10)` создаст последовательность чисел от 0 до 9.

2. `range(start, stop)`: создаёт последовательность чисел от `start` до `stop - 1`.

Пример: `range(0, 10)` создаст последовательность чисел от 0 до 9.

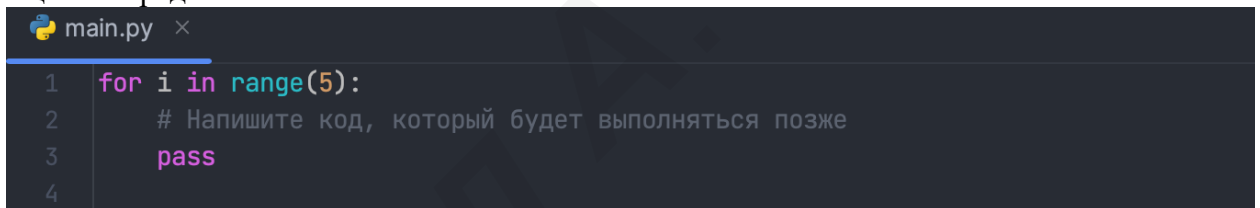
3. `range(start, stop, step)`: Создает последовательность чисел от `start` до `stop - 1` с указанным шагом `step`.

Пример: `range(0, 10, 2)` создаст последовательность четных чисел от 0 до 8 (0, 2, 4, 6, 8).

Во втором варианте, когда не указывается параметр `start`, он считается равным 0. Поэтому `range(0, 10)` и `range(10)` эквивалентны и создают последовательность чисел от 0 до 9.

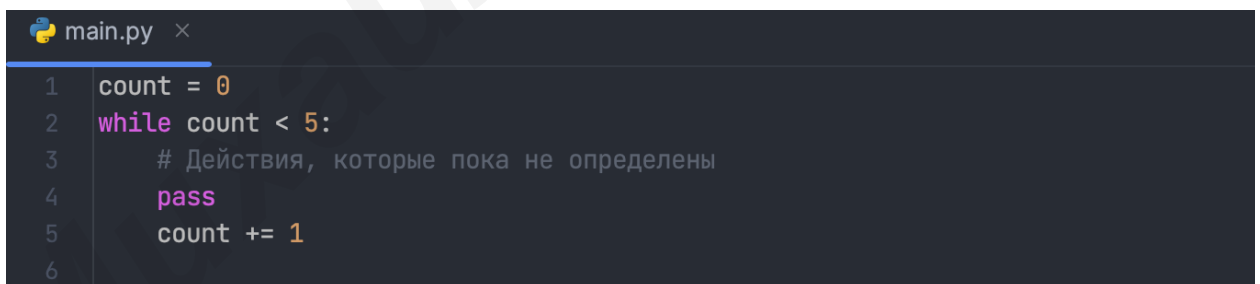
Таким образом, оба выражения `range(0, 10)` и `range(10)` создают одинаковую последовательность чисел и могут использоваться в коде взаимозаменяемо в зависимости от вашего предпочтения или стиля программирования.

Пример 5. Ключевое слово `pass` (рисунок 3.22., рисунок 3.23.) в Python используется для создания пустого блока кода. Оно не выполняет никаких операций и не влияет на ход выполнения программы. Конструкция `pass` часто используется вместо блока кода, когда синтаксически требуется наличие инструкции, но логика или реализация этой инструкции еще не определены.



```
main.py x
1 for i in range(5):
2     # Напишите код, который будет выполняться позже
3     pass
4
```

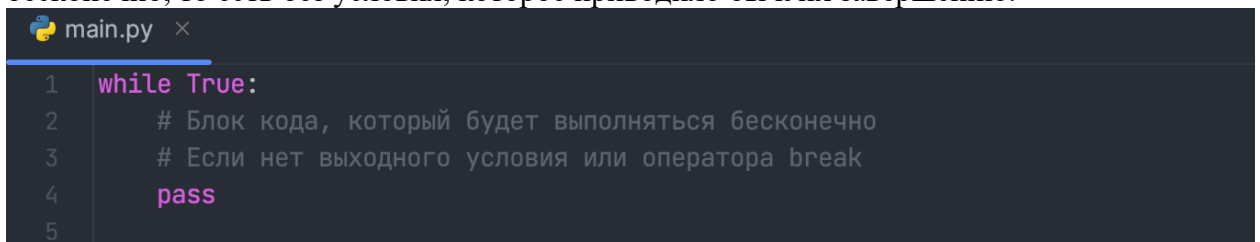
Рис. 3.22. Ключевое слово `pass` в цикле `for`.



```
main.py x
1 count = 0
2 while count < 5:
3     # Действия, которые пока не определены
4     pass
5     count += 1
6
```

Рис. 3.23. Ключевое слово `pass` в цикле `while`.

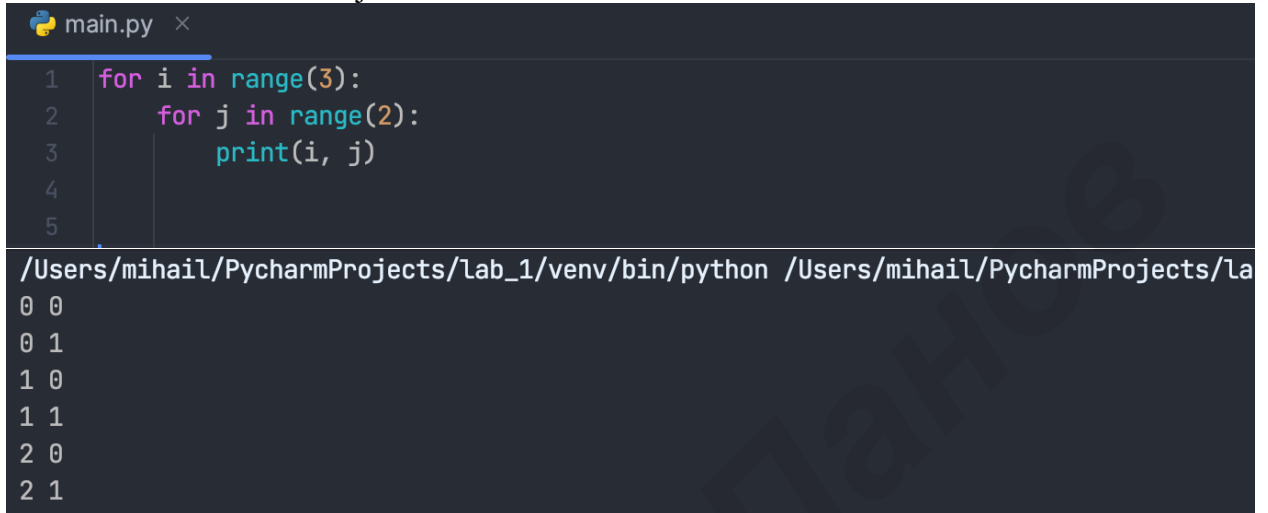
Пример 6. Бесконечные циклы (рисунок 3.24.) Это циклы, которые выполняются бесконечно, то есть без условия, которое приводило бы к их завершению.



```
main.py x
1 while True:
2     # Блок кода, который будет выполняться бесконечно
3     # Если нет выходного условия или оператора break
4     pass
5
```

Рис. 3.24. Бесконечные циклы.

Пример 7. Вложенные циклы (рисунок 3.25.) В Python можно создавать вложенные циклы, то есть циклы, которые находятся внутри других циклов. Вложенные циклы полезны для обработки многомерных структур данных, таких как списки списков или матрицы. В этом примере внешний цикл *for* выполняется три раза, а вложенный цикл *for* выполняется два раза для каждой итерации внешнего цикла. В результате выводятся все возможные комбинации значений *i* и *j*.



```
main.py x
1 for i in range(3):
2     for j in range(2):
3         print(i, j)
4
5
/Users/mihail/PycharmProjects/lab_1/venv/bin/python /Users/mihail/PycharmProjects/lab_1/main.py
0 0
0 1
1 0
1 1
2 0
2 1
```

Рис. 3.25. Вложенные циклы с *for*.

Еще один пример — это вложенные циклы с использованием цикла *while* (рисунок 3.26.) В этом примере у нас есть внешний цикл *while*, который выполняется, пока *i* меньше 3, и внутренний цикл *while*, который выполняется, пока *j* меньше 2. При каждой итерации внутреннего цикла будет выводиться комбинация значений *i* и *j*.



```
main.py x
3 j = 0
4 while j < 2:
5     print(f"i: {i}, j: {j}")
6     j += 1
7 i += 1
/Users/mihail/PycharmProjects/lab_1/venv/bin/python /Users/mihail/PycharmProjects/lab_1/main.py
i: 0, j: 0
i: 0, j: 1
i: 1, j: 0
i: 1, j: 1
i: 2, j: 0
i: 2, j: 1
```

Рис. 3.26. Вложенные циклы с *while*.

Вложенные циклы полезны, когда вам нужно выполнить повторяющиеся операции, которые зависят от нескольких переменных или требуют итерации по многомерной структуре данных, такой как двумерный список или матрица.

Обратите внимание, что при использовании вложенных циклов важно правильно управлять переменными счетчика и обеспечивать условия для их изменения и завершения. Неправильное управление может привести к заикливанию программы или нежелательным результатам.

Использование вложенных циклов может иметь некоторые негативные аспекты, особенно с точки зрения временной сложности выполнения программы. Вот некоторые из минусов, связанных с использованием вложенных циклов:

1. Временная сложность: Вложенные циклы могут привести к значительному увеличению временной сложности выполнения программы. Если внутренний цикл выполняется n раз для каждой итерации внешнего цикла, общее количество итераций становится равным n^2 умножить на количество итераций внешнего цикла. Это может привести к значительному увеличению времени выполнения программы, особенно при больших значениях n .

2. Ресурсоемкость: Использование вложенных циклов требует большего объема ресурсов, таких как память и процессорное время. Если вложенные циклы выполняются на больших объемах данных, это может привести к использованию большого количества ресурсов, что может сказаться на производительности программы.

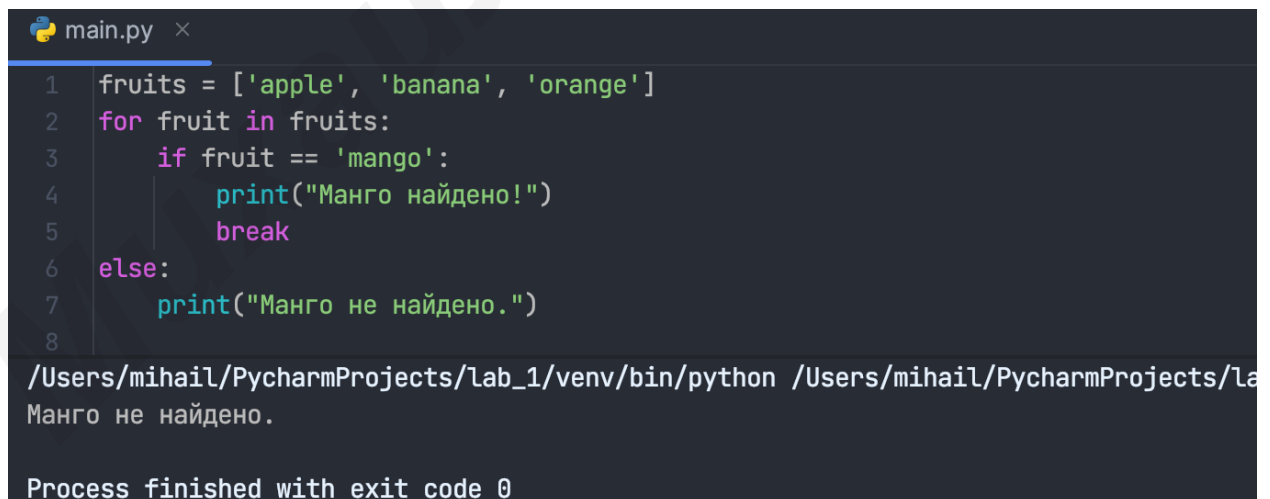
3. Читаемость кода: Вложенные циклы могут усложнить понимание и чтение кода. Чем больше уровней вложенности, тем сложнее разобраться в логике программы. Это может затруднить отладку, поддержку и сопровождение кода в будущем.

4. Возможность ошибок. При использовании вложенных циклов увеличивается вероятность возникновения ошибок. При неправильном управлении переменными счетчика, неправильных условиях или некорректных завершениях циклов может возникнуть заикливание или нежелательные результаты.

5. Альтернативные решения. В некоторых случаях можно найти альтернативные способы выполнения задачи, которые не требуют использования вложенных циклов. Например, использование более эффективных алгоритмов, оптимизация кода или использование специализированных структур данных.

Необходимо тщательно оценить использование вложенных циклов и учитывать возможные негативные последствия, особенно при работе с большими объемами данных или в случаях, когда есть альтернативные решения. В некоторых ситуациях можно избежать использования вложенных циклов, что может привести к более эффективному

Пример 8. Else для циклов (рисунок 3.27.) В Python циклы (как цикл *for*, так и цикл *while*) могут содержать блок *else*, который выполняется, когда цикл завершается естественным образом (то есть без использования оператора *break*). Блок *else* в циклах предоставляет возможность выполнить определенные действия после завершения цикла.



```
main.py x
1 fruits = ['apple', 'banana', 'orange']
2 for fruit in fruits:
3     if fruit == 'mango':
4         print("Манго найдено!")
5         break
6 else:
7     print("Манго не найдено.")
8

/Users/mihail/PycharmProjects/lab_1/venv/bin/python /Users/mihail/PycharmProjects/lab_1/main.py
Манго не найдено.

Process finished with exit code 0
```

Рис. 3.27. Else для циклов.

Здесь блок *else* относится к циклу, а не к условию *if*. Блок кода внутри *else* выполнится только в том случае, если цикл завершился естественным образом, без выполнения оператора *break*. Если цикл был прерван с помощью *break*, блок *else* не будет выполнен.

В этом примере цикл *for* перебирает элементы списка *fruits*, и если встречается элемент 'mango', выводится сообщение "Манго найдено!". Если после завершения цикла

'mango' не было найдено, то блок *else* выполняется и выводится сообщение "Манго не найдено."

Михаил А. Панов