

ТЕМА 4. ФУНКЦИИ И СТАНДАРТНЫЕ МОДУЛИ/БИБЛИОТЕКИ.

4.1. Функции в Python.

В Python функция - это блок кода, который выполняет определенную задачу. Она позволяет группировать повторяющийся код и делает программу более организованной, читаемой и легко поддерживаемой. Функции в Python могут иметь входные параметры (аргументы) и возвращать результаты.

Для создания функции в Python используется ключевое слово `"def"`, после которого следует имя функции и круглые скобки, в которых указываются аргументы функции (если они есть). Затем идет двоеточие, а после него блок кода функции, который выполняется при вызове функции. (рисунок 4.1.)



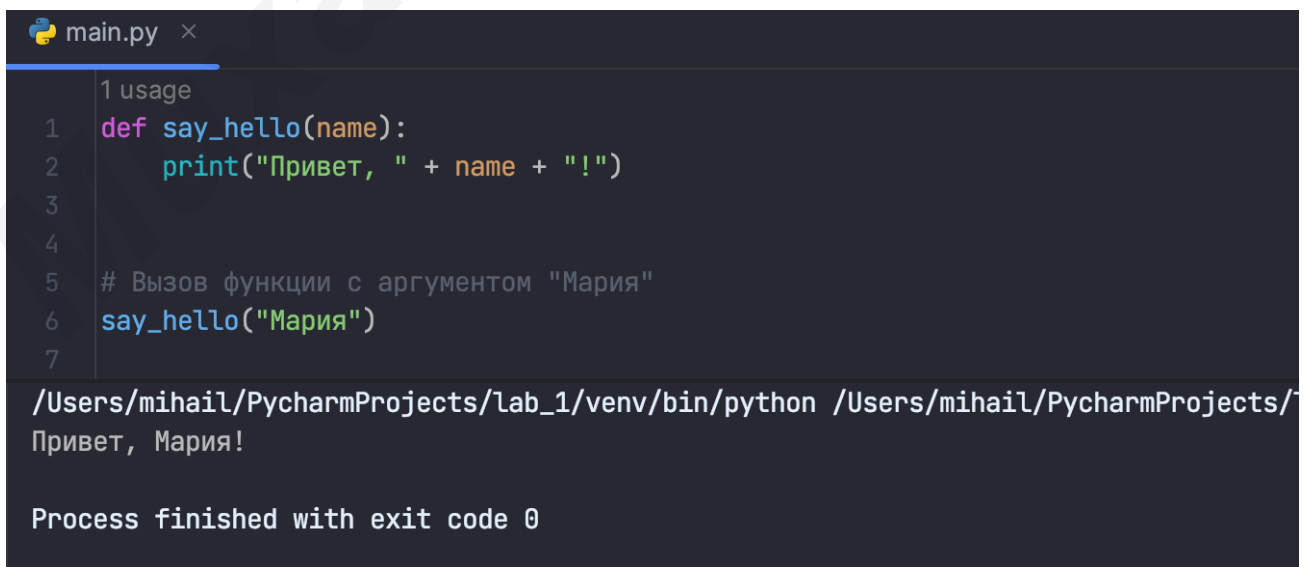
```
main.py x
1 usage
2 def say_hello():
3     print("Привет!")
4
5 # Вызов функции
6 say_hello()
7

/Users/mihail/PycharmProjects/lab_1/venv/bin/python /Users/mihail/PycharmProjects/l
Привет!

Process finished with exit code 0
```

Рис. 4.1. Вызов функций без аргументов

Функции могут принимать аргументы, которые передаются в скобках при их объявлении. Вот пример функции, которая принимает имя в качестве аргумента и выводит приветствие с использованием этого имени (рисунок 4.2.)



```
main.py x
1 usage
2 def say_hello(name):
3     print("Привет, " + name + "!")
4
5 # Вызов функции с аргументом "Мария"
6 say_hello("Мария")
7

/Users/mihail/PycharmProjects/lab_1/venv/bin/python /Users/mihail/PycharmProjects/
Привет, Мария!

Process finished with exit code 0
```

Рис. 4.2. Пример простой функции, которая принимает имя в качестве аргумента

В следующем примере мы определяем функцию с именем *"multiply"*. Она принимает два аргумента - *"a"* и *"b"*, и возвращает их произведение с помощью оператора умножения *"*"*.

Чтобы вызвать именную функцию, используйте ее имя, за которым следуют круглые скобки, содержащие значения аргументов. Вот пример вызова функции *"multiply"* (рисунок 4.3.)



```
main.py x
1 usage
2 def multiply(a, b):
3     return a * b
4
5 result = multiply(3, 4)
6 print(result)
7
/Users/mihail/PycharmProjects/lab_1/venv/bin/python /Users/mihail/PycharmProjects/
12


Process finished with exit code 0
```

Рис. 4.3. Вызов функций с аргументами

В данном примере функция *"multiply"* вызывается с аргументами 3 и 4, и возвращаемое ею значение (произведение) сохраняется в переменную *"result"*. Затем это значение выводится на экран с помощью функции *"print"*.

Конструкция *«return»* в Python используется внутри функций для возврата значения из функции. Когда *«return»* выполняется, функция прекращает свое выполнение и возвращает указанное значение в вызывающую ее часть программы. Это позволяет использовать результаты работы функции в дальнейшем коде.

Вызов функций с именованными аргументами представлен на рисунке 4.4. В этом случае аргументы передаются по имени, что позволяет явно указать, какое значение относится к какому аргументу.:



```
main.py x
1 usage
2 def greet(name, message):
3     print("Привет, " + name + "! " + message)
4
5 greet(name="Мария", message="Как дела?")
6
/Users/mihail/PycharmProjects/lab_1/venv/bin/python /Users/mihail/PycharmProjects/
Привет, Мария! Как дела?

Process finished with exit code 0
```

Рис. 4.4. Вызов функций с именованными аргументами.

Вызов функций с переменным числом аргументов представлен на рисунке 4.5. Здесь функция "add_numbers" может принимать любое количество аргументов, переданных через запятую. Аргументы собираются в кортеж с помощью оператора "*args", и мы можем выполнять операции над ними.

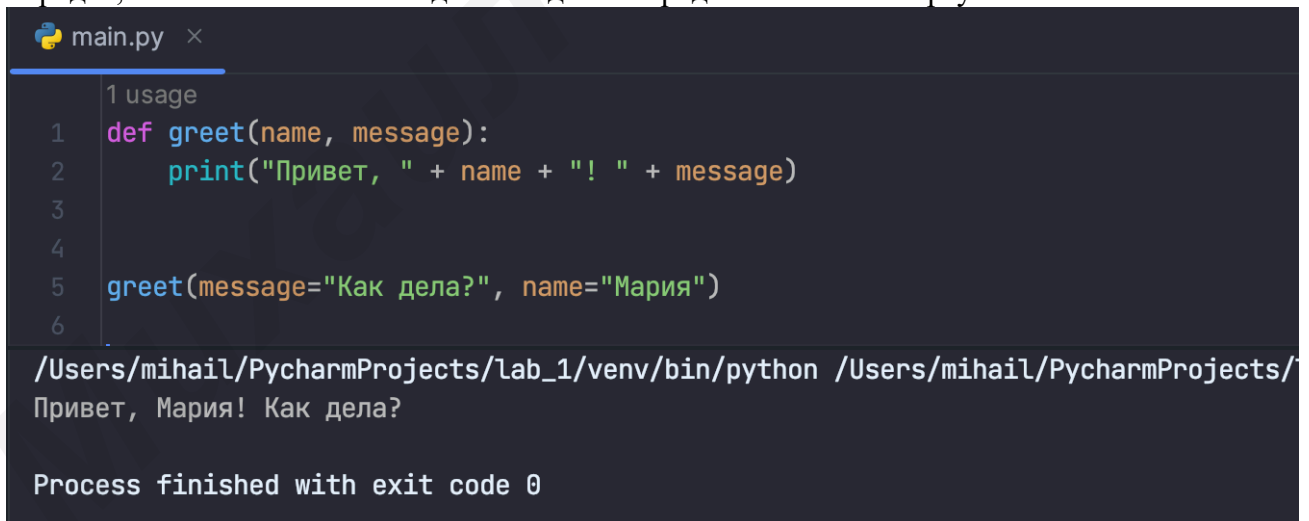


```
main.py x
1 usage
2 def add_numbers(*args):
3     sum = 0
4     for num in args:
5         sum += num
6     return sum
7
8 result = add_numbers(1, 2, 3, 4, 5)
9 print(result)
10
/Users/mihail/PycharmProjects/lab_1/venv/bin/python /Users/mihail/PycharmProjects/
15

Process finished with exit code 0
```

Рис. 4.5. Вызов функций с переменным числом аргументов.

Вызов функций с ключевыми аргументами представлен на рисунке 4.6. В этом случае аргументы передаются по ключу, что позволяет указывать значения аргументов в любом порядке, не обязательно соблюдая исходный порядок объявления аргументов.



```
main.py x
1 usage
2 def greet(name, message):
3     print("Привет, " + name + "! " + message)
4
5 greet(message="Как дела?", name="Мария")
6
/Users/mihail/PycharmProjects/lab_1/venv/bin/python /Users/mihail/PycharmProjects/
Привет, Мария! Как дела?

Process finished with exit code 0
```

Рис. 4.6. Вызов функций с ключевыми аргументами.

4.2. Аргументы функции и параметры.

Аргументы функции и параметры - это основные компоненты функции в Python. Аргументы представляют значения, которые передаются функции при ее вызове, а параметры - это именованные переменные, которые принимают значения аргументов внутри функции.

Когда функция вызывается, значения аргументов передаются в параметры функции. Параметры определяются в определении функции внутри круглых скобок после имени функции (Рисунок 4.7.)

A screenshot of a Python IDE window titled 'main.py'. The code defines a function 'multiply' with two parameters 'a' and 'b'. The function body consists of a single line 'return a * b'. The lines are numbered 1, 2, and 3 on the left margin.

```
1 def multiply(a, b):
2     return a * b
3
```

Рис. 4.7. Определении функции внутри круглых скобок.

В этом примере *a* и *b* - это параметры функции *multiply*. Они будут принимать значения аргументов, переданных при вызове функции.

При вызове функции значения аргументов передаются в параметры в том же порядке, в котором они указаны. На рисунке 4.8. представлен пример вызова функции *multiply*:

A screenshot of a Python IDE window titled 'main.py'. The code defines the 'multiply' function and then calls it with arguments 3 and 4. The result is printed. The lines are numbered 1 through 7. Below the code, the terminal output shows the file path and the result '12'. At the bottom, it says 'Process finished with exit code 0'.

```
1 usage
2 def multiply(a, b):
3     return a * b
4
5 result = multiply(3, 4)
6 print(result) # Выведет 12
7
```

/Users/mihail/PycharmProjects/lab_1/venv/bin/python /Users/mihail/PycharmProjects/
12

Process finished with exit code 0

Рис. 4.8. пример вызова функции *multiply*.

Здесь аргументы 3 и 4 передаются в параметры *a* и *b* соответственно. Функция выполняет умножение и возвращает результат 12, который сохраняется в переменной *result*.

Функции могут иметь различные типы параметров:

Позиционные параметры: это параметры, значения которых передаются в том же порядке, в котором они объявлены. Примером является функция *multiply* из предыдущего примера, где *a* и *b* являются позиционными параметрами.

Именованные параметры: здесь значения передаются с указанием имени параметра. Это позволяет передавать аргументы в любом порядке и явно указывать, какое значение относится к какому параметру (рисунок 4.9.) Здесь *name* и *message* являются именованными параметрами. При вызове функции мы передаем значения с указанием имени параметра.

A screenshot of a Python IDE window titled 'main.py'. The code defines a function 'greet' with two parameters 'name' and 'message'. The function body consists of a single line 'print("Привет, " + name + "! " + message)'. The function is then called with 'name="Мария"' and 'message="Как дела?"'. The lines are numbered 1 through 6.

```
1 usage
2 def greet(name, message):
3     print("Привет, " + name + "! " + message)
4
5 greet(name="Мария", message="Как дела?")
6
```

```
/Users/mihail/PycharmProjects/lab_1/venv/bin/python /Users/mihail/PycharmProjects/
Привет, Мария! Как дела?

Process finished with exit code 0
```

Рис. 4.9. Пример использования именованных параметров.

Параметры со значением по умолчанию: это параметры, которые имеют значение по умолчанию, которое будет использовано, если аргумент не будет передан при вызове функции (рисунок 4.10.)

Здесь *message* имеет значение по умолчанию "Привет". Если мы не передаем аргумент для *message*, будет использовано значение по умолчанию. Мы также можем передать аргумент для *message*, чтобы переопределить значение по умолчанию.

```
main.py x
2 usages
1 def greet(name, message="Привет"):
2     print(message + ", " + name + "!")
3
4
5 greet("Мария") # Выведет "Привет, Мария!"
6 greet("Мария", "Как дела?") # Выведет "Как дела?, Мария!"
7

/Users/mihail/PycharmProjects/lab_1/venv/bin/python /Users/mihail/PycharmProjects/
Привет, Мария!
Как дела?, Мария!

Process finished with exit code 0
```

Рис. 4.10. Пример использования параметров со значением по умолчанию.

Подведем итоги: разница между аргументами и параметрами в функциях в Python заключается в следующем:

Параметры - это именованные переменные, объявляемые в определении функции внутри круглых скобок после имени функции. Они служат для принятия и обработки значений, переданных функции при ее вызове. Параметры определяют, какие данные функция ожидает получить для своей работы.

Аргументы – это значения, передаваемые в функцию при ее вызове. Они передаются в соответствующие параметры функции, чтобы использоваться внутри функции. Аргументы представляют фактические данные, которые передаются функции во время ее выполнения.

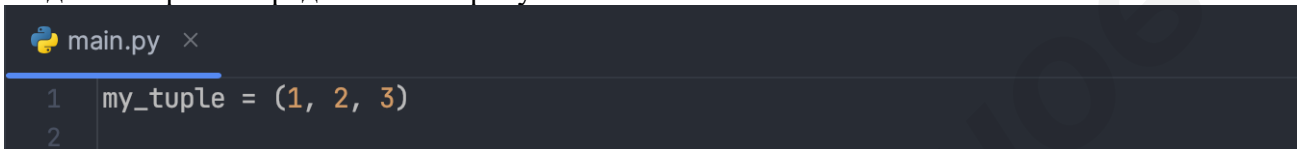
Обязательные аргументы - это аргументы, которые должны быть переданы функции при ее вызове. Если обязательный аргумент не будет передан, это может привести к ошибке. В примере на рисунке 4.7. *a* и *b* являются обязательными аргументами функции *multiply*.

Необязательные аргументы - это аргументы, для которых в определении функции устанавливается значение по умолчанию. Если необязательный аргумент не будет передан при вызове функции, будет использовано значение по умолчанию. В Python необязательные аргументы объявляются после обязательных аргументов и указываются с помощью `имя_параметра=значение_по_умолчанию`.

В примере на рисунке 4.10. *message* является необязательным аргументом со значением по умолчанию "Привет". Если аргумент *message* не будет передан, будет использовано значение по умолчанию. Если передать аргумент *message*, то значение по умолчанию будет заменено на переданное значение.

4.3. Кортеж, как аргумент функции.

В Python кортеж (*tuple*) представляет неизменяемую последовательность элементов, разделенных запятыми и заключенных в круглые скобки. Кортежи ведут себя подобно спискам, но не могут быть изменены после создания, что делает их неизменяемыми. Пример создания кортежа представлен на рисунке 4.11.

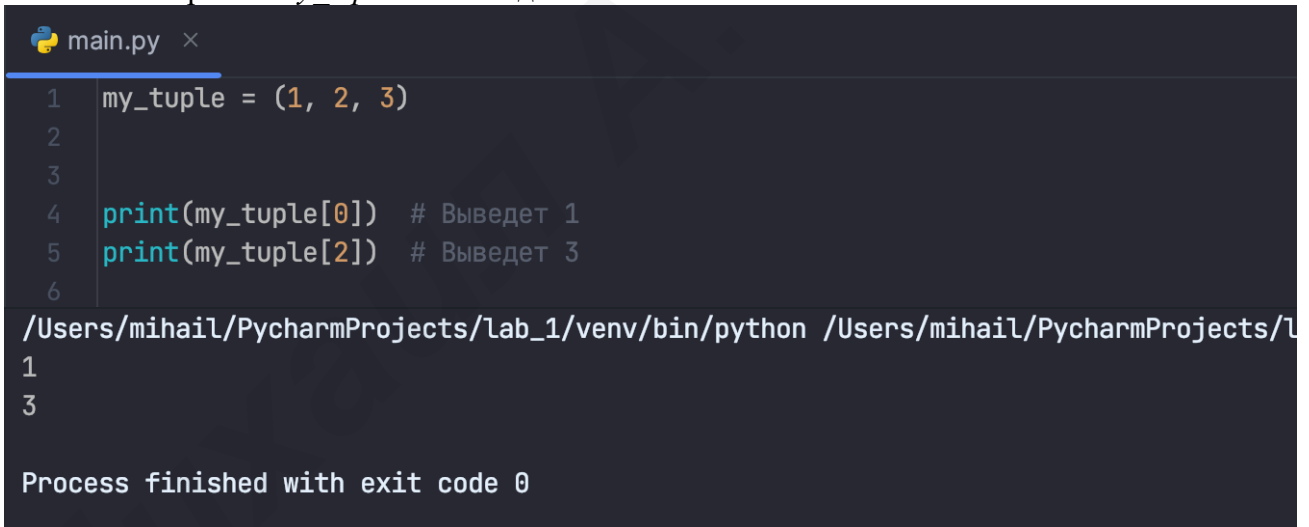


```
main.py x
1 my_tuple = (1, 2, 3)
2
```

Рис. 4.11. Пример создания кортежа

В этом примере мы создали кортеж *my_tuple*, содержащий три элемента: 1, 2 и 3. Кортежи могут содержать элементы разных типов данных, включая числа, строки, списки и другие кортежи.

Кортежи поддерживают индексацию, что позволяет обратиться к элементам по их позиции. Индексация начинается с нуля (рисунок 4.12.) В этом примере мы обращаемся к элементам кортежа *my_tuple* по их индексам.

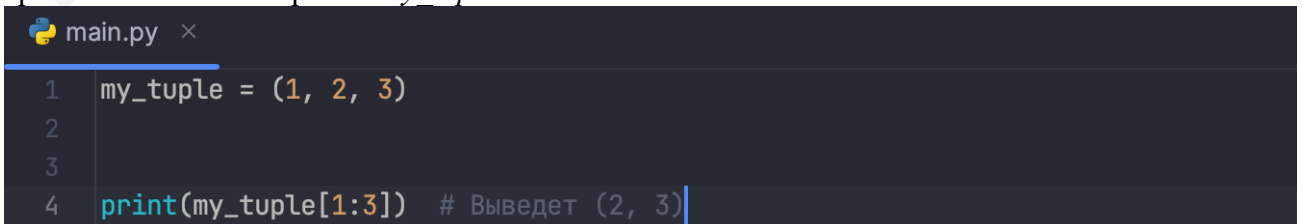


```
main.py x
1 my_tuple = (1, 2, 3)
2
3
4 print(my_tuple[0]) # Выведет 1
5 print(my_tuple[2]) # Выведет 3
6
/Users/mihail/PycharmProjects/lab_1/venv/bin/python /Users/mihail/PycharmProjects/l
1
3

Process finished with exit code 0
```

Рис. 4.12. Пример создания кортежа

Кортежи также поддерживают срезы (*slicing*), которые позволяют получить подмножество элементов из кортежа (рисунок 4.13.) Здесь мы получаем срез от второго до третьего элемента кортежа *my_tuple*.



```
main.py x
1 my_tuple = (1, 2, 3)
2
3
4 print(my_tuple[1:3]) # Выведет (2, 3)
```

```
/Users/mihail/PycharmProjects/lab_1/venv/bin/python /Users/mihail/PycharmProjects/
(2, 3)

Process finished with exit code 0
```

Рис. 4.13. Пример среза кортежа

Одной из важных особенностей кортежей является их неизменяемость. После создания кортежа нельзя изменить его элементы. Попытка присвоить новое значение элементу кортежа вызовет ошибку (рисунок 4.14.)

```
main.py x
1 my_tuple = (1, 2, 3)
2
3
4 my_tuple[0] = 5 # Ошибка: кортежи не поддерживают присваивание элементам
5
6
```

Рис. 4.14. Пример попытки присвоить новое значение элементу кортежа

Однако, если кортеж содержит изменяемые объекты, например, списки, то эти объекты могут быть изменены внутри кортежа. Но сам кортеж остается неизменным (рисунок 4.15.) В этом примере мы изменяем список, который является элементом кортежа *my_tuple*. Однако, сам кортеж *my_tuple* остается неизменным.

```
main.py x
1 my_tuple = ([1, 2, 3], 'Hello')
2 my_tuple[0].append(4)
3 print(my_tuple) # Выведет ([1, 2, 3, 4], 'Hello')
4

/Users/mihail/PycharmProjects/lab_1/venv/bin/python /Users/mihail/PycharmProjects/
([1, 2, 3, 4], 'Hello')

Process finished with exit code 0
```

Рис. 4.15. Пример изменения списка в кортеже

Кортежи широко используются в Python, особенно в ситуациях, когда требуется сохранить набор значений, которые не должны быть изменены. Кортежи могут быть возвращаемыми значениями функций, использоваться в качестве ключей словарей и использоваться в других контекстах, где неизменяемость данных является важным требованием.

В Python кортежи могут быть переданы в качестве аргументов функции точно так же, как и другие типы данных. Кортеж может быть передан в функцию как отдельный аргумент или вместе с другими аргументами (рисунок 4.16.)

В этом примере функция *process_data* принимает аргумент *data*, который предполагается быть кортежем. Внутри функции мы можем обращаться к элементам кортежа

data и выполнять необходимую обработку. В данном случае мы просто выводим каждый элемент кортежа.

```
main.py x
1 usage
2 def process_data(data):
3     for item in data:
4         print(item)
5
6 my_tuple = (1, 2, 3, 4)
7 process_data(my_tuple)
8
/Users/mihail/PycharmProjects/lab_1/venv/bin/python /Users/mihail/PycharmProjects/l
1
2
3
4
Process finished with exit code 0
```

Рис. 4.16. Пример функции, принимающей кортеж в качестве аргумента.

Мы также можем передать кортеж вместе с другими аргументами функции (рисунок 4.17.) В этом примере функция *process_data* принимает два аргумента: *name* и *data*, где *data* представляет собой кортеж. Мы передаем строку "John" в качестве значения аргумента *name* и кортеж *my_tuple* в качестве значения аргумента *data*.

```
main.py x
1 usage
2 def process_data(name, data):
3     print("Name:", name)
4     for item in data:
5         print(item)
6
7 my_tuple = (1, 2, 3, 4)
8 process_data("John", my_tuple)
9
/Users/mihail/PycharmProjects/lab_1/venv/bin/python /Users/mihail/PycharmProjects/
Name: John
1
2
3
4
Process finished with exit code 0
```

Рис. 4.17. Пример передачи кортеж вместе с другими аргументами

Функции в Python могут принимать неограниченное количество аргументов, в том числе и кортежи. Для этого используется специальный синтаксис с оператором `*` (рисунок 4.18.) В этом примере мы определили функцию `process_data`, которая принимает неограниченное количество аргументов с помощью оператора `*args`. Затем мы передаем кортеж `my_tuple` в качестве аргументов с помощью оператора `*` при вызове функции. Это позволяет передать каждый элемент кортежа в качестве отдельного аргумента функции.



```
main.py x
1 usage
2 def process_data(*args):
3     for item in args:
4         print(item)
5
6 my_tuple = (1, 2, 3, 4)
7 process_data(*my_tuple)
8

/Users/mihail/PycharmProjects/lab_1/venv/bin/python /Users/mihail/PycharmProjects/1
1
2
3
4

Process finished with exit code 0
```

Рис. 4.18. Пример передачи каждого элемент кортежа в качестве отдельного аргумента функции

Таким образом, кортежи могут быть использованы в качестве аргументов функций в Python, их элементы могут быть обработаны внутри функции или переданы в другие функции для дальнейшей обработки.

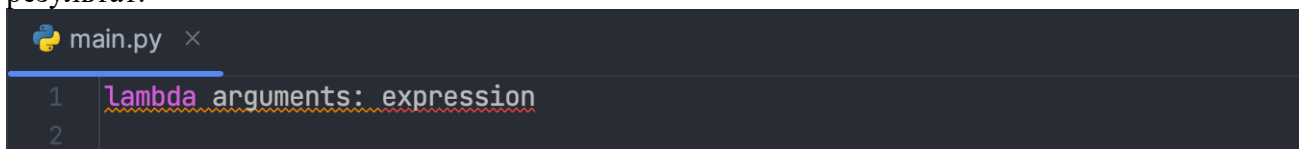
4.4. Анонимные функции `lambda`.

В Python есть возможность создавать анонимные функции с помощью ключевого слова `lambda`. Анонимные функции, также известные как `lambda`-функции, представляют собой компактный способ определения функции без необходимости использования стандартного синтаксиса определения функций с помощью `def`.

Общий синтаксис для создания анонимных функций выглядит следующим образом (рисунок 4.19.)

arguments - список аргументов, которые принимает анонимная функция.

expression - выражение, которое выполняется внутри анонимной функции и возвращает результат.



```
main.py x
1 lambda arguments: expression
2
```

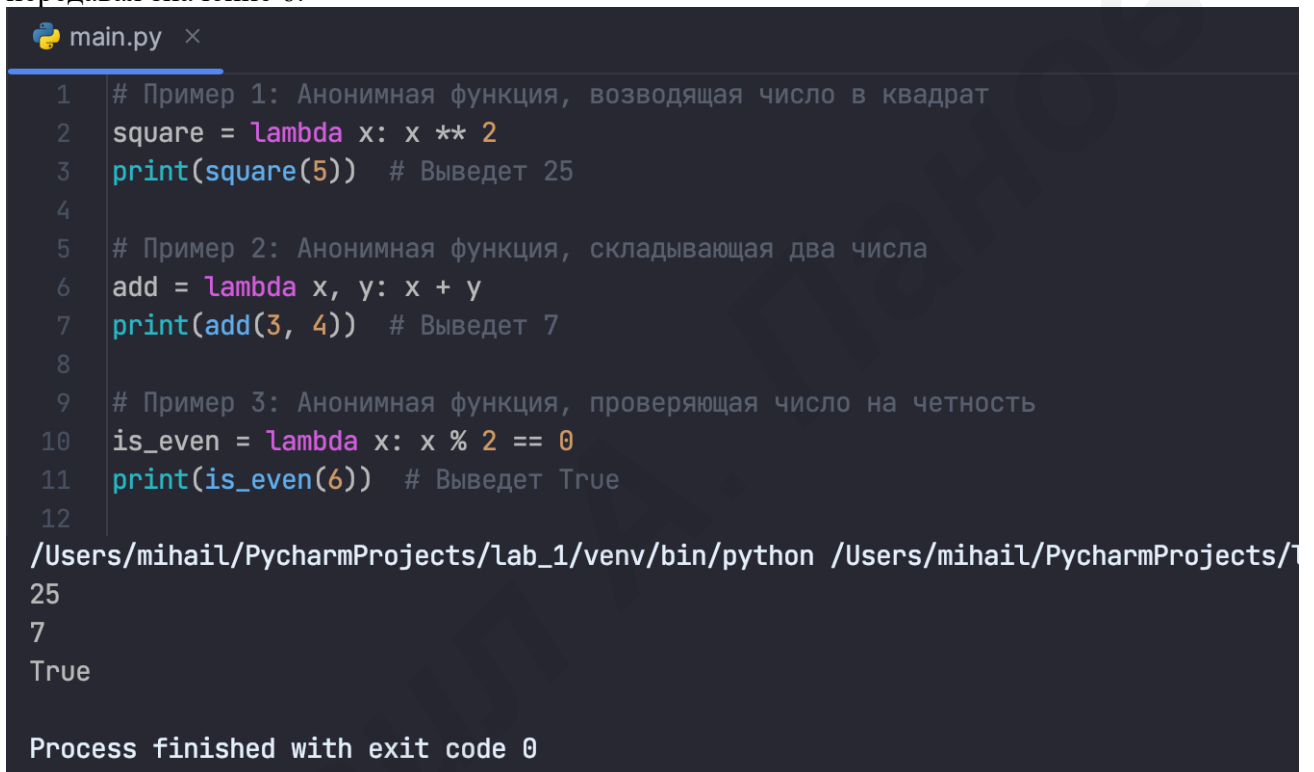
Рис. 4.19. Общий синтаксис для создания анонимных функций

Примеры использования анонимных функций представлены на рисунке 4.20.

В первом примере создается анонимная функция *square*, которая принимает аргумент *x* и возвращает квадрат этого числа. Мы присваиваем эту функцию переменной *square* и вызываем ее, передавая значение 5.

Во втором примере создается анонимная функция *add*, которая принимает два аргумента *x* и *y* и возвращает их сумму. Мы присваиваем эту функцию переменной *add* и вызываем ее, передавая значения 3 и 4.

В третьем примере создается анонимная функция *is_even*, которая принимает аргумент *x* и проверяет, является ли число четным. Она возвращает *True*, если число четное, и *False* в противном случае. Мы присваиваем эту функцию переменной *is_even* и вызываем ее, передавая значение 6.



```
main.py x
1 # Пример 1: Анонимная функция, возводящая число в квадрат
2 square = lambda x: x ** 2
3 print(square(5)) # Выведет 25
4
5 # Пример 2: Анонимная функция, складывающая два числа
6 add = lambda x, y: x + y
7 print(add(3, 4)) # Выведет 7
8
9 # Пример 3: Анонимная функция, проверяющая число на четность
10 is_even = lambda x: x % 2 == 0
11 print(is_even(6)) # Выведет True
12
/Users/mihail/PycharmProjects/lab_1/venv/bin/python /Users/mihail/PycharmProjects/
25
7
True

Process finished with exit code 0
```

Рис. 4.20. Примеры использования анонимных функций

Анонимные функции особенно полезны, когда требуется определить небольшие функции на лету или передать их в качестве аргументов другим функциям, таким как *map()*, *filter()*, *reduce()* и другим. Они обычно используются в ситуациях, где необходимо выполнить некоторые простые операции без необходимости определения именованной функции с помощью *def*.

4.5. Рекурсивные функции.

Рекурсивные функции в программировании - это функции, которые вызывают сами себя в своем теле. Это особый подход к решению задач, когда задача разбивается на более простые подзадачи, и каждая подзадача решается путем вызова той же функции. Рекурсия является мощным инструментом для решения определенных типов задач и часто используется в алгоритмах и структурах данных.

Основные характеристики рекурсивных функций:

Базовый случай: каждая рекурсивная функция должна иметь базовый случай, когда вызов функции прекращается и она возвращает результат напрямую, без дополнительных рекурсивных вызовов. Это предотвращает бесконечное повторение вызовов и обеспечивает завершение рекурсии.

Рекурсивный случай: когда условие базового случая не выполняется, функция вызывает себя снова с измененными аргументами. Это позволяет сократить сложную задачу до более простых подзадач, которые решаются рекурсивными вызовами.

Пример простой рекурсивной функции для вычисления факториала числа представлен на рисунке 4.21. В этой функции, если n равно 0, функция возвращает 1 (базовый случай). В противном случае, функция вызывает себя с аргументом $n-1$ и умножает результат на n (рекурсивный случай). Факториал числа 5 вычисляется как $5 * 4 * 3 * 2 * 1 = 120$.



```
main.py x
2 usages
1 def factorial(n):
2     if n == 0:
3         return 1 # Базовый случай: факториал 0 равен 1
4     else:
5         return n * factorial(n-1) # Рекурсивный случай: n! = n * (n-1)!
6
7
8 result = factorial(5)
9 print(result) # Выведет 120
10

/Users/mihail/PycharmProjects/lab_1/venv/bin/python /Users/mihail/PycharmProjects/
120

Process finished with exit code 0
```

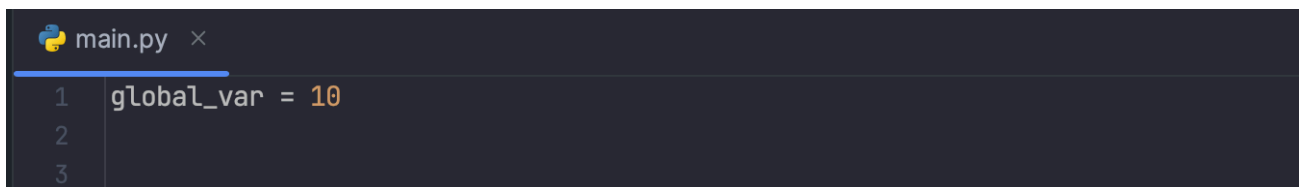
Рис. 4.21. Пример простой рекурсивной функции для вычисления факториала числа

Рекурсивные функции могут быть мощным инструментом для решения определенных задач, но их следует использовать с осторожностью, так как неправильная реализация может привести к переполнению стека вызовов (stack overflow) или бесконечной рекурсии. Важно правильно определить базовый случай и убедиться, что рекурсивные вызовы приводят к сокращению задачи до базового случая.

4.6. Глобальные переменные.

Глобальные переменные в Python - это переменные, которые определены за пределами всех функций и доступны из любого места в программе. Они могут быть использованы и изменены внутри любой функции или блока кода.

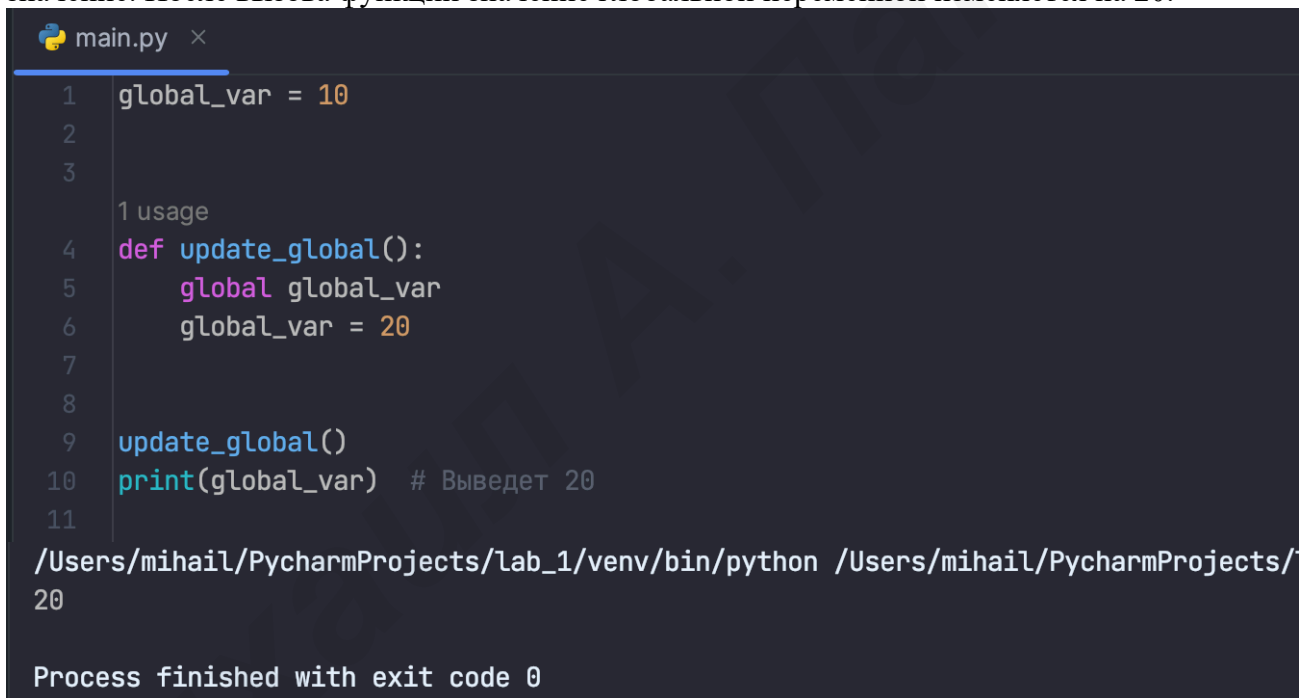
Объявление глобальной переменной происходит путем присваивания значения переменной вне функций или блоков кода. Пример представлен на рисунке 4.22. В этом примере `global_var` - глобальная переменная, доступная из любого места в программе.



```
main.py x
1 global_var = 10
2
3
```

Рис. 4.22. Пример объявления глобальной переменной

Внутри функций можно использовать глобальные переменные без необходимости объявления их как аргументы функции или локальные переменные. Однако, если вы попытаетесь изменить значение глобальной переменной внутри функции, не указав ее как глобальную, будет создана новая локальная переменная с тем же именем, и глобальная переменная останется неизменной. Для того чтобы явно указать, что вы хотите изменить значение глобальной переменной, внутри функции используйте ключевое слово *global* (рисунок 4.22.) В этом примере функция *update_global* использует ключевое слово *global* перед обращением к глобальной переменной *global_var*, чтобы указать, что мы хотим изменить ее значение. После вызова функции значение глобальной переменной изменяется на 20.



```
main.py x
1 global_var = 10
2
3
4 1 usage
5 def update_global():
6     global global_var
7     global_var = 20
8
9 update_global()
10 print(global_var) # Выведет 20
11

/Users/mihail/PycharmProjects/lab_1/venv/bin/python /Users/mihail/PycharmProjects/
20

Process finished with exit code 0
```

Рис. 4.22. Пример использования ключевого слова *global*.

Хотя использование глобальных переменных может быть удобным в некоторых случаях, их следует использовать с осторожностью, так как они могут усложнить отслеживание и понимание программы. Модификация глобальных переменных внутри функций может привести к неожиданному поведению и усложнить отладку программы. Поэтому рекомендуется использовать глобальные переменные только в случаях, когда они действительно необходимы и оправданы. В большинстве случаев предпочтительнее использовать локальные переменные и передавать значения через аргументы функций.

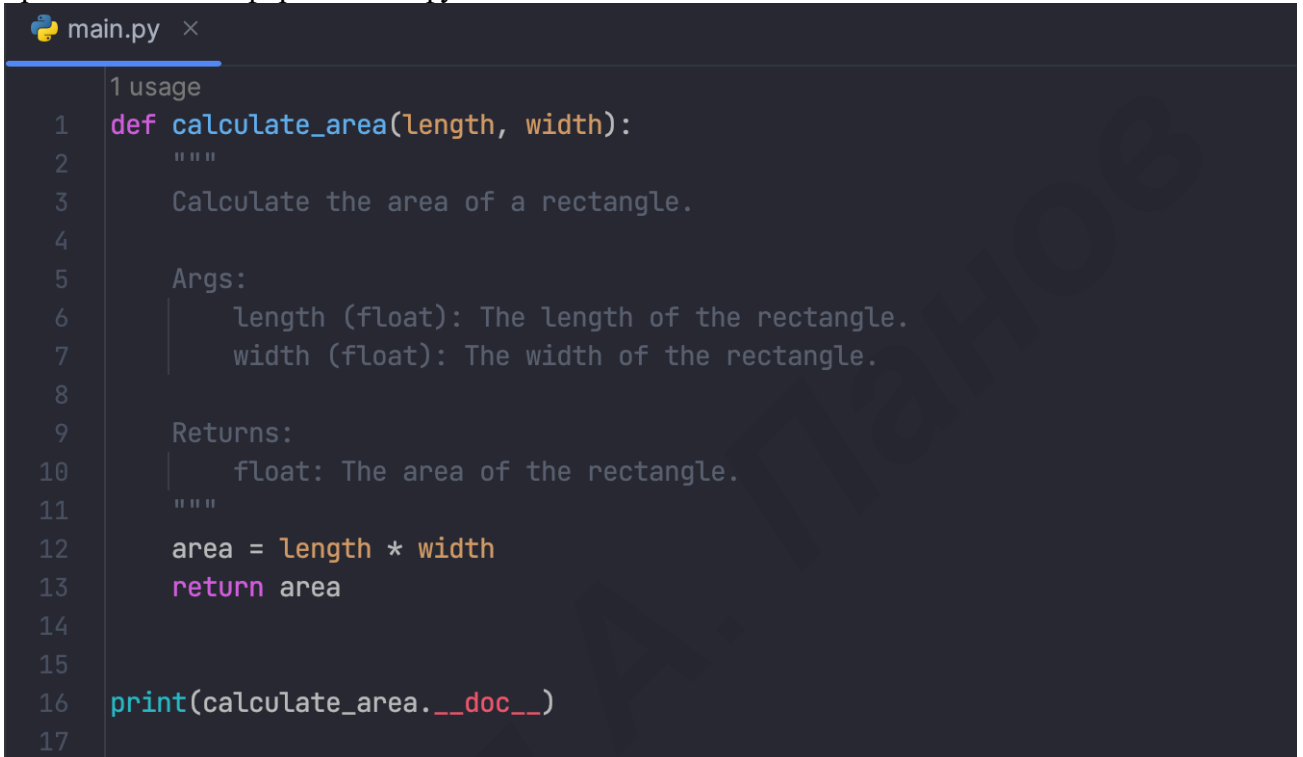
4.7. Документирование функций.

В Python существует соглашение о документировании функций с помощью строки документации, которая помещается внутри тройных кавычек (`"""`). Эта строка документации

предоставляет описание функции, ее параметры, возвращаемое значение и другую полезную информацию (рисунок 4.23.)

Вот пример, как можно задокументировать функцию с помощью строки документации:

В этом примере функция `calculate_area` вычисляет площадь прямоугольника. Строка документации располагается внутри тройных кавычек сразу после определения функции и предоставляет информацию о функции.

A screenshot of a code editor window titled 'main.py'. It shows a Python function definition for 'calculate_area'. The function takes 'length' and 'width' as arguments. The docstring is a multi-line string starting with 'usage', followed by a description, arguments, and returns. The function body calculates the area and returns it. Finally, the docstring is printed out.

```
1 usage
2
3 Calculate the area of a rectangle.
4
5 Args:
6     length (float): The length of the rectangle.
7     width (float): The width of the rectangle.
8
9 Returns:
10    float: The area of the rectangle.
11
12 area = length * width
13 return area
14
15
16 print(calculate_area.__doc__)
17
```

Рис. 4.23. Пример, как можно задокументировать функцию с помощью строки документации.

Строка документации может содержать следующие элементы:

Описание функции: Общее описание того, что делает функция и ее назначение.

Аргументы (параметры): Описание каждого аргумента функции, указывающее тип данных (если необходимо) и его роль.

Возвращаемое значение: Описание значения, которое возвращает функция.

Примеры использования: Примеры кода, которые демонстрируют, как использовать функцию.

Другая информация: Любая другая полезная информация, связанная с функцией.

Документируя функции с помощью строки документации, вы облегчаете понимание и использование функции другим разработчикам, а также помогаете себе в будущем, когда вам нужно будет вернуться к функции и вспомнить ее основные детали.

Доступ к строке документации функции осуществляется с помощью атрибута `__doc__`. В данном примере это выведет содержимое строки документации функции `calculate_area` (рисунок 4.24.)

```
/Users/mihail/PycharmProjects/lab_1/venv/bin/python /Users/mihail/PycharmProjects/

Calculate the area of a rectangle.

Args:
    length (float): The length of the rectangle.
    width (float): The width of the rectangle.

Returns:
    float: The area of the rectangle.

Process finished with exit code 0
```

Рис. 4.24. Вывод содержимого строки документации функции.

Обратите внимание, что документация функции является хорошей практикой программирования, но она не является обязательной. Однако, ее использование считается хорошим стилем кодирования и помогает создавать более читаемый и поддерживаемый код.

4.8. Присвоение функции переменной.

В Python функции можно присваивать переменным так же, как и любые другие значения. Это означает, что вы можете создать функцию и присвоить ее переменной, а затем использовать эту переменную для вызова функции (рисунок 4.25.)

В этом примере мы создали функцию *greet*, которая принимает аргумент *name* и выводит приветствие. Затем мы присвоили эту функцию переменной *my_function*. Теперь *my_function* ссылается на функцию *greet*. Мы можем вызвать функцию, используя переменную *my_function*, передавая имя в качестве аргумента.

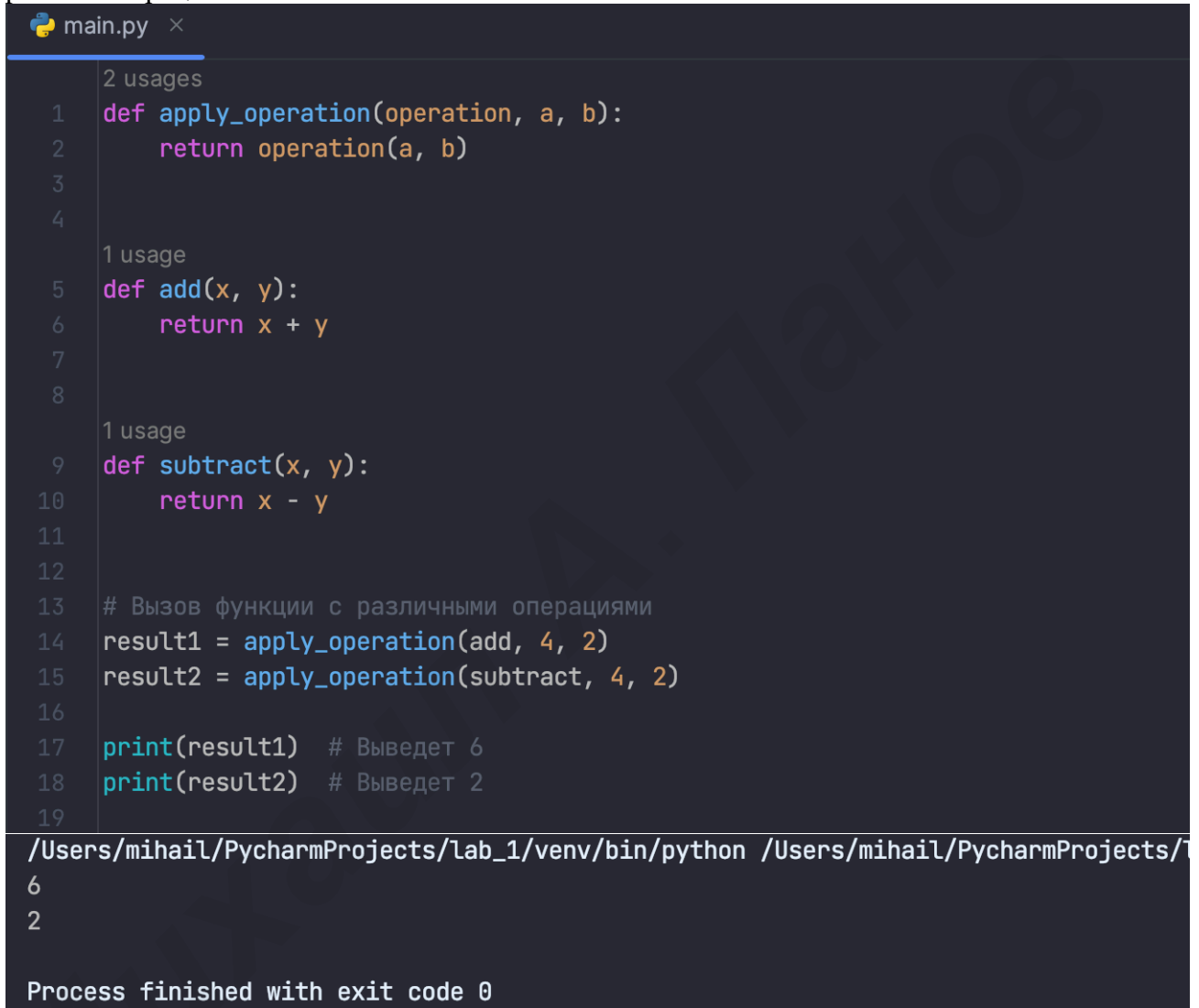
```
main.py x
1 usage
2 def greet(name):
3     print("Hello, " + name + "!")
4
5 # Присваивание функции переменной
6 my_function = greet
7
8 # Вызов функции через переменную
9 my_function("Alice") # Выведет "Hello, Alice!"
10

/Users/mihail/PycharmProjects/lab_1/venv/bin/python /Users/mihail/PycharmProjects/
Hello, Alice!

Process finished with exit code 0
```

Рис. 4.25. Вывод содержимого строки документации функции.

Присваивание функции переменной полезно, когда вам нужно передать функцию как аргумент в другую функцию или сохранить функцию для последующего вызова (рисунок 4.26.) В этом примере у нас есть функция *apply_operation*, которая принимает операцию в качестве аргумента и применяет ее к двум числам. Затем у нас есть функции *add* и *subtract*, которые выполняют сложение и вычитание соответственно. Мы передаем эти функции в *apply_operation* как аргументы, используя их переменные, и получаем результаты выполнения разных операций.



```
main.py x
2 usages
1 def apply_operation(operation, a, b):
2     return operation(a, b)
3
4
1 usage
5 def add(x, y):
6     return x + y
7
8
1 usage
9 def subtract(x, y):
10    return x - y
11
12
13 # Вызов функции с различными операциями
14 result1 = apply_operation(add, 4, 2)
15 result2 = apply_operation(subtract, 4, 2)
16
17 print(result1) # Выведет 6
18 print(result2) # Выведет 2
19

/Users/mihail/PycharmProjects/lab_1/venv/bin/python /Users/mihail/PycharmProjects/
6
2

Process finished with exit code 0
```

Рис. 4.26. Передача функции через аргументы.

Присваивание функции переменной является мощным инструментом в Python, который позволяет гибко работать с функциями и использовать их в различных сценариях программирования.

4.9. Модули

В Python модуль — это файл, содержащий код (функции, классы, переменные) и предназначенный для организации и управления кодом в более крупных программных проектах. Модули позволяют разделить код на логически связанные блоки и обеспечить повторное использование кода.

Преимущества использования модулей в Python:

Организация кода: Модули позволяют логически разделить функциональность программы на отдельные файлы, что облегчает его понимание, обслуживание и развитие.

Повторное использование кода: Модули позволяют создавать функции, классы и переменные, которые можно повторно использовать в разных частях программы или в других проектах.

Разделение обязанностей: Модули позволяют разделить код между несколькими разработчиками или командами, так что каждый модуль может быть независимо разрабатываемым и поддерживаемым.

Для использования модуля в Python вы должны выполнить следующие шаги:

- Создайте файл с расширением `.py`, в котором будет содержаться код модуля.
- Определите функции, классы и переменные внутри файла модуля.
- В другом файле или интерактивной оболочке Python используйте оператор `import` для импорта модуля и доступа к его содержимому.

Обратите внимание, что файл с расширением `.py`, в котором будет содержаться код модуля должен находиться в той же папке что и основной файл (рисунок 4.27.)

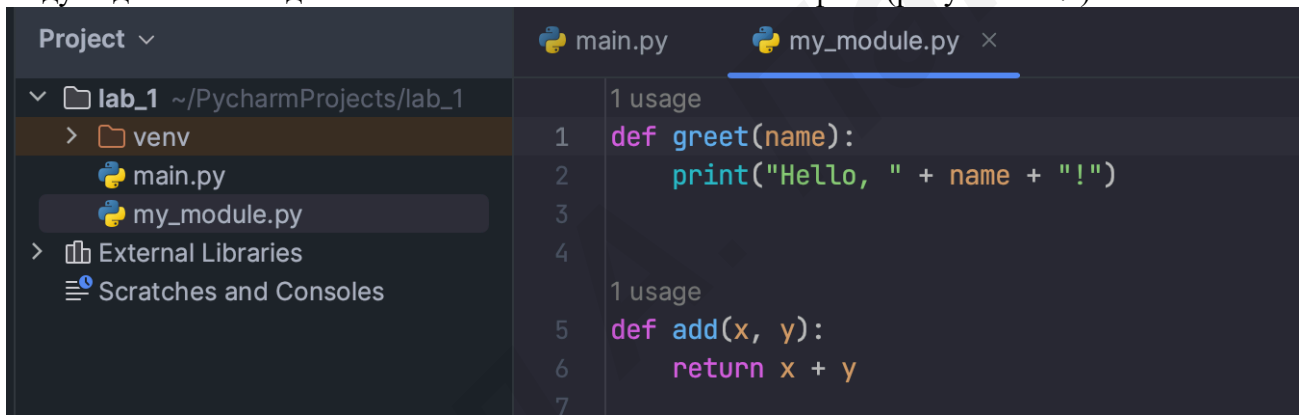


Рис. 4.27. Дерево проекта

Пример использования модуля представлен на рисунке 4.28 и 4.29.

В этом примере у нас есть модуль `my_module`, содержащий функции `greet` и `add`. Мы импортируем модуль `my_module` в файле `main.py` с помощью оператора `import`. Затем мы можем использовать функции из модуля, добавляя имя модуля перед именем функции.

Кроме простого импорта с помощью `import`, существуют и другие способы импорта модулей, например, `from module import item`. Этот способ позволяет импортировать только определенные элементы из модуля, а не весь модуль целиком.

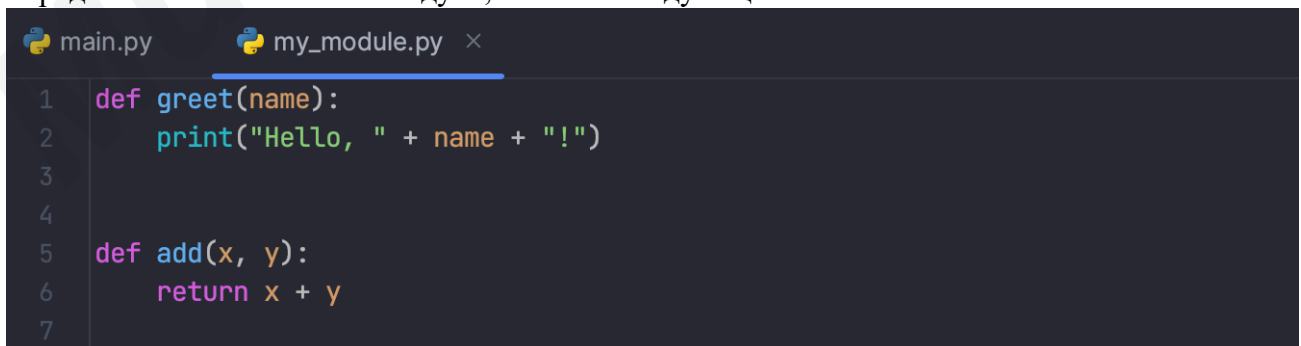


Рис. 4.28. Файл `my_module.py`


```
main.py × my_module.py
1 import my_module
2
3 my_module.greet("Alice")
4 result = my_module.add(4, 2)
5
6 print(result) # Выведет 6
7
```

Рис. 4.29. Файл main.py

Результатом выполнения будет значение консоли на рисунке 4.30

```
/Users/mihail/PycharmProjects/lab_1/venv/bin/python /Users/mihail/PycharmProjects/l
Hello, Alice!
6

Process finished with exit code 0
```

Рис. 4.30. Результат выполнения программы

Модули являются важной частью организации кода в Python, позволяя создавать масштабируемые и структурированные программные проекты. Они предоставляют возможность разделить код на логические блоки, повторно использовать код и упростить совместную работу между разработчиками.

4.10. Глобальное использование функций модуля

Чтобы сделать функции модуля доступными для глобального использования в других модулях или в основной программе, вы можете использовать ключевое слово *from* в операторе импорта или импортировать модуль целиком.

Вот несколько способов использования функций модуля глобально:

Импортирование модуля целиком (рисунок 4.31.) В этом случае вы импортируете весь модуль `my_module`. Вы можете обращаться к функциям модуля, используя синтаксис `module name.function name()` или `module name.variable name`.

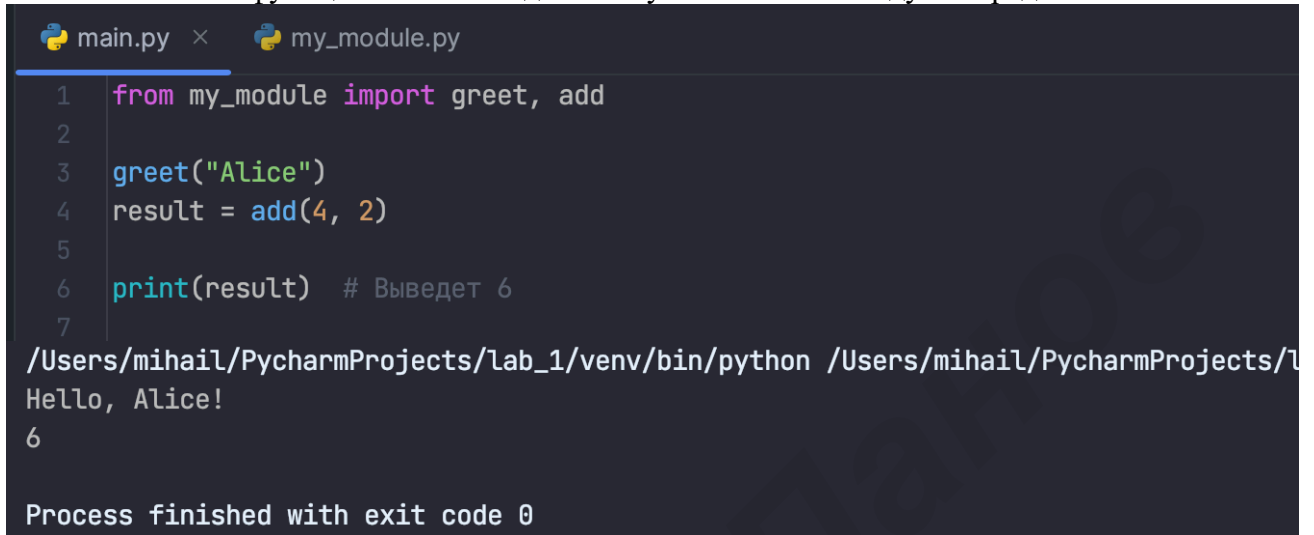
```
main.py × my_module.py
1 import my_module
2
3 my_module.greet("Alice")
4 result = my_module.add(4, 2)
5
6 print(result) # Выведет 6
7

/Users/mihail/PycharmProjects/lab_1/venv/bin/python /Users/mihail/PycharmProjects/l
Hello, Alice!
6

Process finished with exit code 0
```

Рис. 4.31. Импортирование модуля целиком

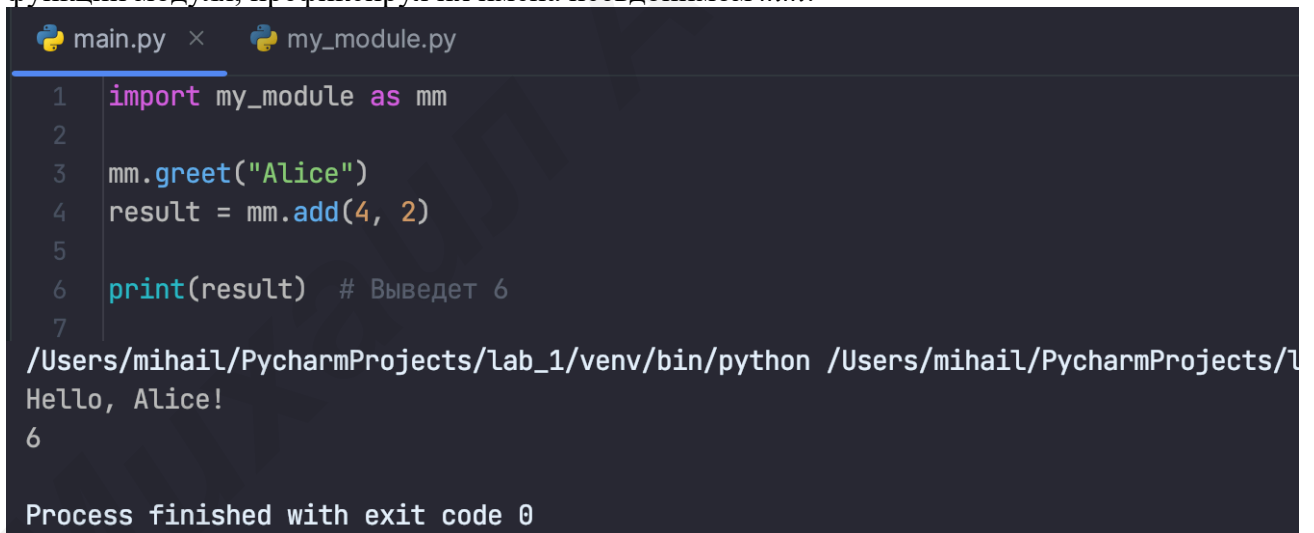
Импортирование конкретных функций из модуля (рисунок 4.32.) В этом случае вы импортируете только функции *greet* и *add* из модуля *my_module*. Теперь вы можете использовать эти функции без необходимости указывать имя модуля перед их именем.



```
main.py × my_module.py
1 from my_module import greet, add
2
3 greet("Alice")
4 result = add(4, 2)
5
6 print(result) # Выведет 6
7
/Users/mihail/PycharmProjects/lab_1/venv/bin/python /Users/mihail/PycharmProjects/l
Hello, Alice!
6
Process finished with exit code 0
```

Рис. 4.32. Импортирование конкретных функций из модуля

Импортирование всего модуля с псевдонимом (рисунок 4.33.) В этом случае вы импортируете модуль *my_module* с псевдонимом *mm*. Теперь вы можете использовать функции модуля, префиксируя их имена псевдонимом *mm*.



```
main.py × my_module.py
1 import my_module as mm
2
3 mm.greet("Alice")
4 result = mm.add(4, 2)
5
6 print(result) # Выведет 6
7
/Users/mihail/PycharmProjects/lab_1/venv/bin/python /Users/mihail/PycharmProjects/l
Hello, Alice!
6
Process finished with exit code 0
```

Рис. 4.33. Импортирование всего модуля с псевдонимом

Это лишь несколько способов использования функций модуля глобально. Важно помнить, что глобальное использование функций модуля может упростить код, но также может создать конфликты и неоднозначности, если имена функций перекрываются с другими именами в вашей программе. Поэтому важно выбирать имена функций и модулей таким образом, чтобы избежать возможных конфликтов и создать читаемый и понятный код.

В Python вы можете импортировать один или несколько модулей в одной строке, разделяя их запятыми. Например:

- `import my_module;`

- `import module1, module2, module3;`
- `import module1 as m1, module2 as m2, module3 as m3;`
- `import math`
`import random, datetime, os`
`import numpy as np, pandas as pd.`

Также в Python существует ключевое слово *import **, оно используется для импорта всех функций, классов и переменных из модуля в текущее пространство имен. Это позволяет обращаться к импортированным элементам напрямую, без необходимости указывать имя модуля перед ними.

Однако, использование *import ** не рекомендуется в большинстве случаев по нескольким причинам:

Неясность имен: Использование *import ** может привести к неоднозначности имен в вашем коде. Если модуль, из которого вы импортируете, содержит функции или переменные с именами, которые уже используются в вашей программе, это может привести к конфликтам и трудностям в понимании, откуда именно эти имена происходят.

Загромождение пространства имен: Импортирование всех элементов из модуля с помощью *import ** может привести к загромождению вашего пространства имен. Если модуль содержит много функций и переменных, это может сделать ваш код менее читабельным и усложнить отслеживание происхождения и использования определенных имен.

Непредсказуемость: Использование *import ** делает ваш код менее предсказуемым и может затруднить отслеживание и анализ зависимостей между модулями. При чтении кода другим разработчикам может быть сложно определить, откуда именно происходят импортированные элементы, и какие модули необходимы для правильной работы программы.

Использование *import ** не рекомендуется, поскольку это может привести к неясности имен, загромождению пространства имен и непредсказуемости кода.

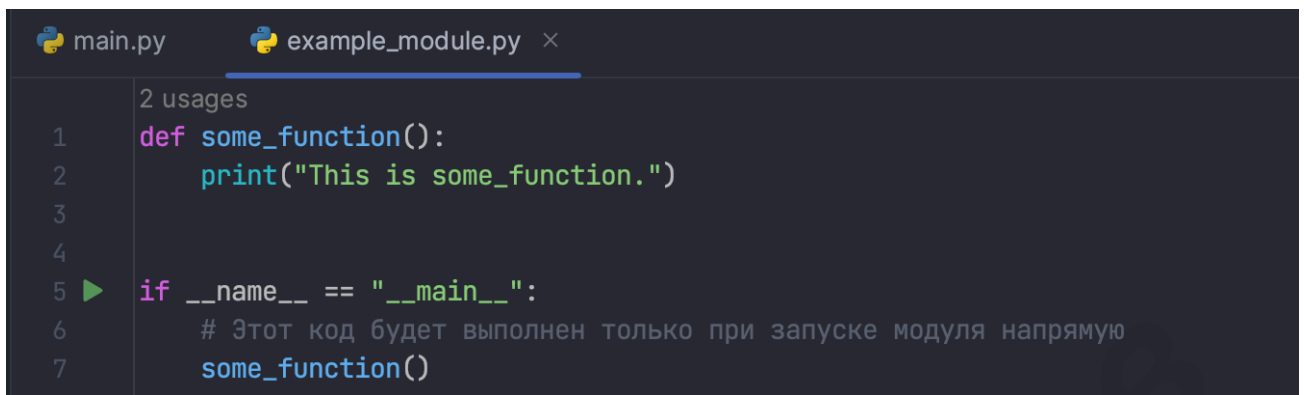
4.11. Точка входа в Python

Точка входа в Python - это основная часть программы, с которой начинается ее выполнение. Она определяет, какой код будет запущен при запуске программы.

В Python точка входа обычно определена в виде вызова функции *main()*. Это позволяет разделить логику программы на отдельные функции и выполнить основную логику в функции *main()*.

В Python переменная `__name__` представляет специальное внутреннее имя, которое предоставляется интерпретатором Python для каждого модуля. Значение переменной `__name__` зависит от контекста, в котором используется.

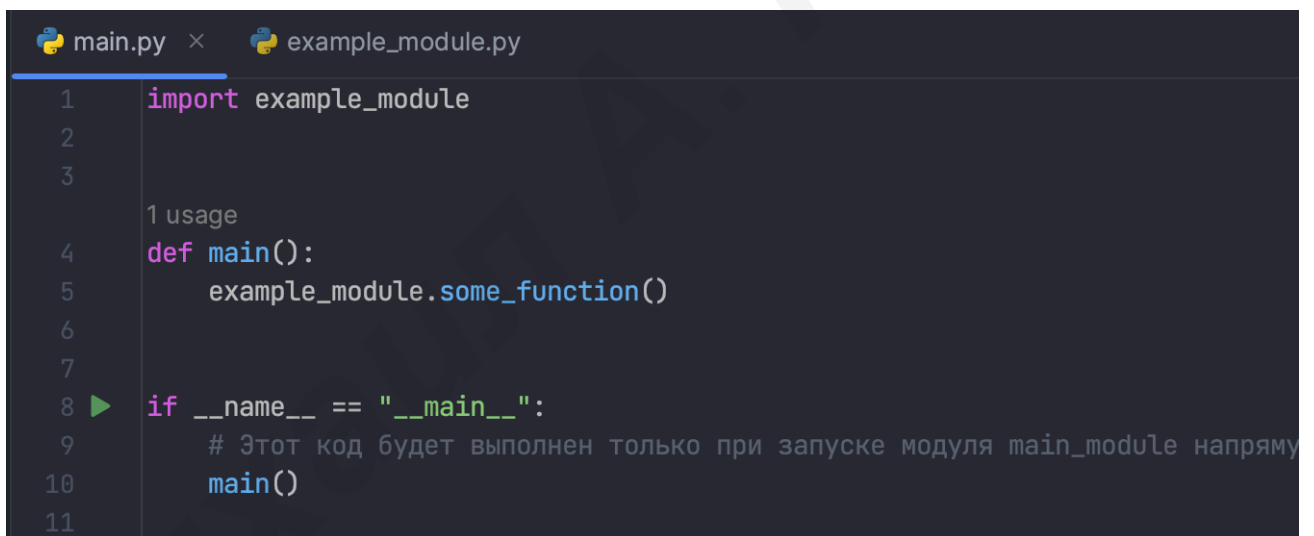
Когда модуль выполняется как точка входа (т.е. запускается напрямую), значение переменной `__name__` будет равно строке `"__main__"`. Это позволяет определить, является ли текущий модуль точкой входа в программу. Например, вы можете использовать условие `if __name__ == "__main__":` для выполнения определенного кода только при запуске модуля напрямую, а не при его импортировании в другой модуль (рисунок 4.34.)



```
main.py example_module.py x
2 usages
1 def some_function():
2     print("This is some_function.")
3
4
5 ► if __name__ == "__main__":
6     # Этот код будет выполнен только при запуске модуля напрямую
7     some_function()
```

Рис. 4.34. Пример выполнения определенного кода только при запуске модуля напрямую

Когда модуль импортируется в другой модуль, значение переменной `__name__` будет равно имени самого модуля. Это позволяет отличить запуск модуля от его импорта и, например, вызвать определенные функции или классы из импортированного модуля (рисунок 4.35.) В этом примере модуль `main_module` импортирует модуль `example_module` и вызывает функцию `some_function()` из него только при запуске `main_module` напрямую. Если `main_module` импортируется в другой модуль, код в блоке `if __name__ == "__main__":` не будет выполняться.



```
main.py x example_module.py
1 import example_module
2
3
4 1 usage
5 def main():
6     example_module.some_function()
7
8 ► if __name__ == "__main__":
9     # Этот код будет выполнен только при запуске модуля main_module напрямую
10    main()
11
```

Рис. 4.34. Пример выполнения определенного кода при импорте модуля

Таким образом, переменная `__name__` предоставляет информацию о том, в каком контексте выполняется модуль, и позволяет контролировать выполнение определенного кода в зависимости от этого контекста.

4.12. Рекурсивный импорт

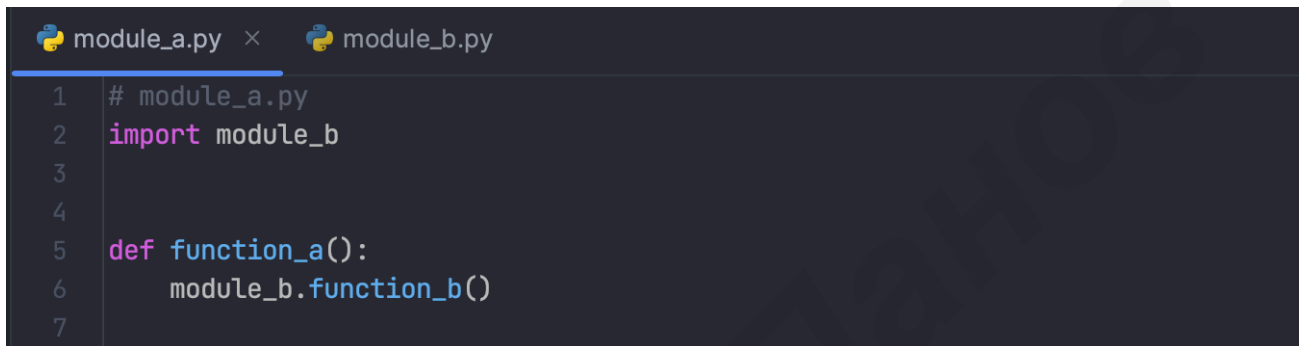
Рекурсивный импорт - это ситуация, когда два или более модуля взаимно импортируют друг друга непосредственно или через цепочку импортов. Такой сценарий может возникнуть, если в модуле А импортируется модуль В, а в модуле В импортируется модуль А.

Рекурсивный импорт может привести к проблемам и ошибкам, таким как "ImportError: cannot import name" или "RuntimeError: maximum recursion depth exceeded". Это связано с тем, что Python пытается разрешить импорт внутри циклической зависимости, но не может это

сделать, так как один модуль еще не полностью загружен при попытке импорта другого модуля.

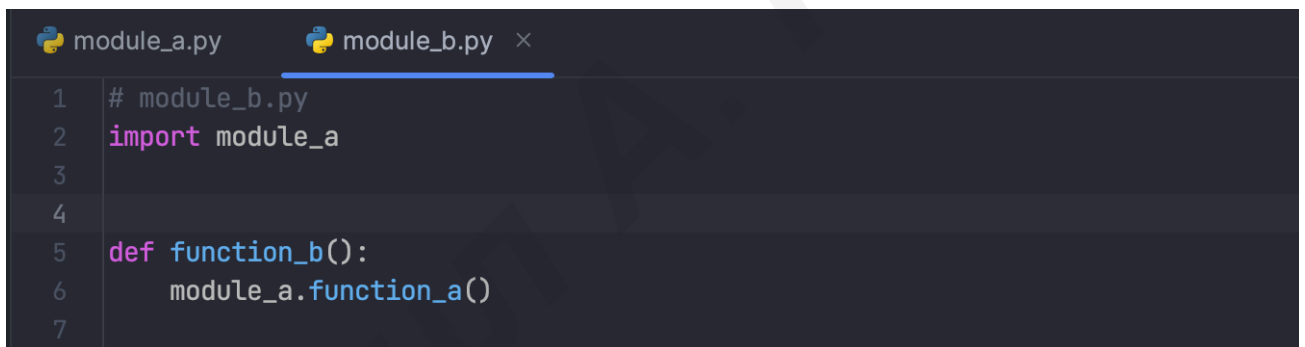
Для избежания рекурсивного импорта можно пересмотреть структуру кода и выделить общую функциональность в третий модуль или переорганизовать зависимости между модулями. В некоторых случаях также может быть полезно использовать отложенный импорт (lazy import) или импортировать модули внутри функций, чтобы избежать рекурсивных проблем.

Вот пример ситуации рекурсивного импорта (Рисунок 4.35 и 4.36)



```
1 # module_a.py
2 import module_b
3
4
5 def function_a():
6     module_b.function_b()
7
```

Рис. 4.35. Модуль «module_a.py»



```
1 # module_b.py
2 import module_a
3
4
5 def function_b():
6     module_a.function_a()
7
```

Рис. 4.36. Модуль «module_b.py»

При попытке импортировать модуль А или модуль В произойдет ошибка рекурсивного импорта. Решение проблемы рекурсивного импорта требует пересмотра архитектуры программы и избегания прямых или косвенных циклических зависимостей между модулями.

4.13. Встроенные модули Python

Python поставляется со множеством встроенных модулей, которые предоставляют различные функциональные возможности. Вот несколько примеров встроенных модулей Python:

1. ``math``: Предоставляет математические функции и константы, такие как тригонометрические функции, логарифмы, возведение в степень и другие операции.
2. ``random``: Позволяет генерировать случайные числа, выбирать случайные элементы из списков и выполнять другие операции, связанные с случайными значениями.
3. ``datetime``: Предоставляет классы и функции для работы с датами и временем. Этот модуль позволяет выполнять операции с датами, вычислять разницу между датами, форматировать даты и времена и многое другое.

4. ``os``: Предоставляет функции для взаимодействия с операционной системой. Этот модуль позволяет получать информацию о файловой системе, выполнять операции с файлами и папками, работать с переменными окружения и выполнять другие операции, связанные с операционной системой.

5. ``sys``: Предоставляет функции и переменные, связанные с интерпретатором Python. Этот модуль позволяет получать информацию о версии Python, аргументах командной строки, путях поиска модулей и других системных параметрах.

6. ``json``: Предоставляет функции для работы с данными в формате JSON. Этот модуль позволяет сериализовывать объекты Python в формат JSON и десериализовывать JSON-данные в объекты Python.

7. ``re``: Предоставляет поддержку регулярных выражений в Python. Этот модуль позволяет выполнять поиск, сопоставление и манипуляции текстом с использованием регулярных выражений.

8. ``csv``: Предоставляет функции для работы с данными в формате CSV (Comma-Separated Values). Этот модуль позволяет читать данные из CSV-файлов, записывать данные в CSV-файлы и выполнять другие операции, связанные с CSV-данными.

Это лишь некоторые примеры встроенных модулей Python. Полный список встроенных библиотек Python можно найти в официальной документации Python. Вот ссылка на документацию Python 3.9, где можно найти полный список встроенных модулей:

<https://docs.python.org/3/library/index.html>

На этой странице вы найдете список модулей с описанием их функциональности. Вы можете выбрать интересующий вас модуль и изучить его подробное описание, включая доступные классы, функции и методы.

Кроме того, в самой среде разработки Python, такой как IDLE, PyCharm или Jupyter Notebook, вы также можете получить список доступных модулей и их документацию с помощью функций автодополнения и подсказок.

Например, в среде IDLE вы можете воспользоваться командой ``help()`` для получения информации о модуле или функции. Например, ``help(math)`` покажет документацию модуля ``math``. Также можно использовать функцию ``dir()`` для получения списка доступных атрибутов модуля или объекта. Например, ``dir(math)`` покажет список доступных функций и констант в модуле ``math``.

Исследование официальной документации Python и использование встроенных сред разработки помогут вам полнее понять функциональность и использование встроенных библиотек Python.