

Теоретический материал

Москва 2023

## Оглавление

Введение .....	3
1. Уровень Junior: .....	4
Основные типы данных: .....	4
Условные выражения: .....	5
Циклы (for и while): .....	5
Функции и аргументы функций: .....	6
Работа с файлами: .....	6
Ошибки и исключения: .....	6
Импорт модулей и использование сторонних библиотек: .....	7
Объектно-ориентированное программирование (ООП): .....	8
2. Уровень Middle: .....	10
Что такое модель памяти Python? .....	10
Thread locals .....	10
Что такое <code>_slots_</code> ? .....	11
Для чего используются нижние подчеркивания в именах классов? .....	12
Работа с регулярными выражениями: .....	13
Итераторы и генераторы: .....	14
3. Уровень Senior: .....	16
Декораторы: .....	16
Многопоточность и асинхронное программирование: .....	16
Работа с базами данных: .....	17
Тестирование кода: .....	19
Профилирование и оптимизация: .....	19

## **Введение**

Главные отличия между уровнями Junior, Middle и Senior связаны с опытом, навыками, ответственностью и способностью решать сложные задачи.

Опыт работы:

Junior-уровень обычно предполагает отсутствие или небольшой опыт работы в области тестирования.

Middle-уровень предполагает наличие нескольких лет опыта работы в области тестирования программного обеспечения.

Senior-уровень предполагает значительный опыт работы, часто связанный с руководством проектов и командой тестирования.

Навыки и знания:

На уровне Junior ожидается, что специалист обладает базовыми навыками тестирования, знанием основных методов и инструментов. Он постоянно учится и готов выслушивать критику со стороны более опытных разработчиков.

На Middle-уровне ожидается более глубокое понимание процесса тестирования, опыт работы с различными методами тестирования и знание автоматизации тестирования. Он способен самостоятельно браться за задачи и решать их в поставленные сроки.

Senior-уровень предполагает широкие знания и опыт в области тестирования, глубокое понимание тестовых стратегий, методов тестирования и лидерских навыков. Он уже руководит подразделением и является самым опытным разработчиком.

Для дальнейшего развития своих навыков необходимо всегда стремиться узнавать новое и расширять свои знания: читать статьи, просматривать онлайн-курсы или видеоуроки. Изучать новые языки программирования, фреймворки и инструменты.

## 1. Уровень Junior:

### Основные типы данных:

- Числа: целые числа (int), числа с плавающей запятой (float), комплексные числа (complex).

```
python +
1 x = 10
2 y = 3.14
3 z = complex(2, 3)
```

- Строки: последовательности символов, можно использовать одинарные (') или двойные (") кавычки.

```
python +
1 name = "John"
2 message = 'Hello, ' + name
```

- Списки: упорядоченные изменяемые коллекции элементов, записываются в квадратных скобках ([]).

```
python +
1 numbers = [1, 2, 3, 4, 5]
2 fruits = ['apple', 'banana', 'orange']
3
```

- Кортежи: упорядоченные неизменяемые коллекции элементов, записываются в круглых скобках ().

```
python +
1 point = (3, 5)
2 colors = ('red', 'green', 'blue')
3
```

- Словари: неупорядоченные коллекции пар "ключ-значение", записываются в фигурных скобках ({}), с использованием двоеточия (:).

```
python +
1 person = {'name': 'John', 'age': 25, 'city': 'New York'}
2
```

- Множества: неупорядоченные коллекции уникальных элементов, записываются в фигурных скобках (`{}`).

```
python +
1 set1 = {1, 2, 3, 4, 5}
2 set2 = {3, 4, 5, 6, 7}
3
```

### Условные выражения:

- `if`: позволяет выполнять блок кода, если условие истинно.
- `elif (else if)`: используется для проверки дополнительных условий, если предыдущие условия ложные.
- `else`: выполняется, если ни одно из предыдущих условий не является истинным.

```
python +
1 age = 18
2 if age < 18:
3     print("You are underage")
4 elif age >= 18 and age < 65:
5     print("You are an adult")
6 else:
7     print("You are a senior citizen")
8
```

- Операторы сравнения: `==`, `!=`, `>`, `<`, `>=`, `<=`.
- Логические операторы: `and`, `or`, `not`.

### Циклы (`for` и `while`):

- `for`: используется для итерации по элементам коллекции или выполнения блока кода определенное количество раз.
- `while`: выполняет блок кода, пока условие истинно.

```
python +
1 numbers = [1, 2, 3, 4, 5]
2 for num in numbers:
3     print(num)
4
5 count = 0
6 while count < 5:
7     print(count)
8     count += 1
```

### Функции и аргументы функций:

- Функции позволяют упорядочить и повторно использовать код.
- Аргументы функций - значения, передаваемые в функцию для выполнения операций.

```
python +
1 def add_numbers(a, b):
2     return a + b
3
4 result = add_numbers(5, 3)
5 print(result)
```

- Лямбда-функции: создание анонимных функций.

### Работа с файлами:

- Открытие файлов: функция **open()** используется для открытия файла с указанием пути и режима доступа (чтение, запись и т. д.).
- Чтение файлов: методы **read()**, **readline()** и **readlines()** используются для чтения содержимого файла.

```
python +
1 with open("example.txt", "r") as file:
2     content = file.read()
3     print(content)
```

- Запись файлов: методы **write()** и **writelines()** используются для записи данных в файл.

```
python +
1 with open("example.txt", "w") as file:
2     file.write("Hello, world!")
3
```

### Ошибки и исключения:

- Ошибки возникают при выполнении программы, исключения позволяют обрабатывать их.
- Конструкция **try-except** используется для обработки исключений: код в блоке **try** выполняется, и, если возникает исключение, управление передается блоку **except**, где можно обработать исключение или продолжить выполнение программы после обработки исключения.

```
python +
1 try:
2     result = 10 / 0
3 except ZeroDivisionError:
4     print("Error: division by zero")
5
```

### Импорт модулей и использование сторонних библиотек:

- Модули — это файлы, содержащие определения и инструкции Python, которые можно импортировать и использовать в других программах.
- Импортирование модулей: с помощью ключевого слова **import** можно импортировать модуль целиком или определенные элементы модуля.
- Сторонние библиотеки — это модули, разработанные сторонними разработчиками, предоставляющие дополнительные функциональные возможности для Python.
- Установка сторонних библиотек: с помощью инструментов установки пакетов, таких как **pip**, можно установить сторонние библиотеки и использовать их в своей программе.

```
python +
1 import math
2
3 radius = 5
4 area = math.pi * radius ** 2
5 print(area)
```

## Объектно-ориентированное программирование (ООП):

- Классы:

Классы являются основным инструментом ООП. Они определяют общую структуру и поведение объектов. Вот пример определения класса **Person**:

```
python +
1 class Person:
2     def __init__(self, name, age):
3         self.name = name
4         self.age = age
5
6     def greet(self):
7         print(f"Hello, my name is {self.name} and I'm {self.age} years old.")
8
9 # Создание экземпляра класса
10 person = Person("Alice", 25)
11 person.greet() # Output: Hello, my name is Alice and I'm 25 years old.
```

- Наследование:

Наследование позволяет создавать новые классы на основе уже существующих. Новый класс наследует атрибуты и методы родительского класса. Вот пример наследования класса **Employee** от класса **Person**:

```
python +
1 class Employee(Person):
2     def __init__(self, name, age, salary):
3         super().__init__(name, age)
4         self.salary = salary
5
6     def display_salary(self):
7         print(f"My salary is {self.salary} dollars.")
8
9 # Создание экземпляра класса Employee
10 employee = Employee("Bob", 30, 5000)
11 employee.greet() # Output: Hello, my name is Bob and I'm 30 years old.
12 employee.display_salary() # Output: My salary is 5000 dollars.
```

- Инкапсуляция:

Инкапсуляция скрывает детали реализации класса и предоставляет публичный интерфейс для работы с объектами. Методы и атрибуты могут быть объявлены как публичные, защищенные или приватные. Вот пример использования защищенного атрибута `_age`:



```
python +
1 class Person:
2     def __init__(self, name, age):
3         self.name = name
4         self._age = age # Защищенный атрибут
5
6     def greet(self):
7         print(f"Hello, my name is {self.name} and I'm {self._age} years old.")
8
9 person = Person("Alice", 25)
10 print(person._age) # Output: 25 (можно обратиться, но считается соглашением не обращаться напрямую)
```

- **Полиморфизм:**

Полиморфизм позволяет использовать один и тот же интерфейс для различных классов. Различные классы могут предоставлять свою собственную реализацию методов с одинаковыми именами. Вот пример полиморфизма с методом `greet()`:

```
python +
1 class Dog:
2     def greet(self):
3         print("Woof!")
4
5 class Cat:
6     def greet(self):
7         print("Meow!")
8
9 class Human:
10    def greet(self):
11        print("Hello!")
12
13 def greet_all(animals):
14     for animal in animals:
15         animal.greet()
16
17 dog = Dog()
18 cat = Cat()
19 human = Human()
20
21 greet_all([dog, cat, human])
22
```

## 2. Уровень Middle:

### Что такое модель памяти Python?

Модель памяти Python описывает, как объекты создаются, хранятся и взаимодействуют друг с другом в памяти во время выполнения программы. Основные принципы модели памяти Python включают:

- **Объекты:** все данные в Python являются объектами или коллекциями объектов. Каждый объект имеет свой тип, атрибуты и значение.
- **Ссылки:** в Python объекты манипулируются через ссылки на них. Ссылка — это имя, которое связывается с объектом и позволяет получать доступ к его содержимому.
- **Присваивание:** оператор присваивания создает ссылку на объект. Например, при выполнении выражения `a = 10`, создается объект целого числа со значением 10, и ссылка `a` указывает на этот объект.
- **Сборка мусора:** Python автоматически управляет памятью и освобождает объекты, которые больше не используются. Сборка мусора происходит благодаря механизму подсчета ссылок и использованию алгоритма сборки мусора с отслеживанием поколений.
- **Мутабельность и неизменяемость:** в Python существуют как мутабельные (изменяемые) объекты, так и неизменяемые объекты. Неизменяемые объекты не могут быть изменены после создания, в то время как мутабельные объекты могут изменять свое состояние.

Например, числа, строки и кортежи являются неизменяемыми объектами, а списки и словари являются мутабельными объектами. При изменении неизменяемого объекта создается новый объект с обновленным значением, а ссылка на предыдущий объект остается неизменной.

### Thread locals

Thread locals (локальные переменные потока) — это механизм в Python, который позволяет каждому потоку иметь свою собственную версию глобальных переменных. Это полезно в ситуациях, когда необходимо иметь отдельные значения переменных для каждого потока, чтобы избежать состояния гонки (race condition) и непредсказуемого поведения.

Механизм thread locals обеспечивает глобальный доступ к данным, которые видны только в пределах одного потока. Это достигается путем создания объекта `threading.local()`, который предоставляет контейнер для хранения данных, специфичных для каждого потока.

## Пример использования thread locals в Python:

```
python +
1 import threading
2 # Создание объекта thread local
3 my_data = threading.local()
4
5 # Установка значения для текущего потока
6 my_data.x = 10
7
8 # Получение значения из текущего потока
9 print(my_data.x) # Вывод: 10
10
11 # Каждый поток имеет свое собственное значение
12 def my_thread_func():
13     my_data.x = 20
14     print(my_data.x) # Вывод: 20
15
16 # Создание и запуск потока
17 my_thread = threading.Thread(target=my_thread_func)
18 my_thread.start()
19 my_thread.join()
20
21 # Значение в основном потоке остается неизменным
22 print(my_data.x) # Вывод: 10
```

Каждый поток имеет свою собственную копию объекта **my\_data**, и изменения, внесенные в одном потоке, не влияют на значения в других потоках. Это позволяет безопасно использовать глобальные переменные в многопоточных сценариях.

## Что такое `__slots__`?

`__slots__` — это специальный атрибут класса в Python, который позволяет явно определить набор атрибутов (полей) объекта, которые будут созданы и управляются во время выполнения программы. Объявление `__slots__` может быть полезным, особенно когда необходимо эффективно использовать память или иметь строгий контроль над атрибутами объекта.

При использовании `__slots__` определяется ограниченный набор атрибутов, которые могут быть присутствующими в объекте, и Python оптимизирует память, зарезервированную для каждого объекта. Вместо использования словаря для хранения атрибутов, Python выделяет фиксированное количество места в памяти для каждого объекта, и доступ к атрибутам осуществляется через смещение в этом блок памяти.

## Преимущества использования `__slots__`:

- Экономия памяти: поскольку для каждого объекта выделяется фиксированное количество памяти, использование `__slots__` может значительно снизить потребление памяти при создании большого числа объектов.

- Ускорение доступа к атрибутам: доступ к атрибутам объекта с использованием `__slots__` выполняется непосредственно через смещение в памяти, что может ускорить операции чтения и записи значений атрибутов.
- Защита от ошибок в именах атрибутов: при использовании `__slots__` можно явно указать список разрешенных атрибутов для объекта, что предотвращает случайное создание новых атрибутов с опечатками в именах.

Например, рассмотрим класс **Person** с использованием `__slots__`:

```
python +
1 class Person:
2     __slots__ = ('name', 'age')
3
4     def __init__(self, name, age):
5         self.name = name
6         self.age = age
7
```

В этом примере атрибуты **name** и **age** явно определены в `__slots__`. Это означает, что объекты класса **Person** будут иметь только эти атрибуты, и другие атрибуты не могут быть добавлены динамически.

### Для чего используются нижние подчеркивания в именах классов?

В Python нижние подчеркивания в именах классов имеют специальное значение и используются для обозначения определенных соглашений и семантики.

1. Префикс одного нижнего подчеркивания (например, `_classname`) указывает на "слабую" конвенцию, что имя класса является внутренним или частным для модуля. Это не является строгим ограничением, но считается соглашением между разработчиками, чтобы не использовать такие классы извне модуля.

```
python +
1 class _InternalClass:
2     pass
3
```

2. Префикс двух нижних подчеркиваний (например, `__classname`) используется для механизма именования под названием "name mangling" (сокрытие имен). Имена классов, начинающиеся с двух подчеркиваний, автоматически изменяются, чтобы избежать конфликтов имен с классами, унаследованными или содержащими атрибуты с такими же именами.

```
python +
1 class __MangledClass:
2     pass
3
4 # Имя класса будет изменено:
5 print(__MangledClass.__name__) # Вывод: __MangledClass__MangledClass
```

1. Префикс и суффикс двух нижних подчеркиваний (например, `__classname__`) обычно используется для специальных методов, известных как "специальные методы". Эти методы имеют определенное значение в языке Python и автоматически вызываются в определенных ситуациях, таких как создание экземпляра класса, доступ к атрибутам, перегрузка операторов и т. д.

```
python +
1 class SpecialClass:
2     def __init__(self):
3         pass
4
5     def __str__(self):
6         return "This is a special class"
7
8 obj = SpecialClass()
9 print(obj) # Вывод: This is a special class
```

Использование нижних подчеркиваний в именах классов помогает обеспечить ясность и предотвращает конфликты имен между классами, а также указывает на особые свойства или ограничения для разработчиков, которые используют классы.

### **Работа с регулярными выражениями:**

Регулярные выражения (Regular Expressions) - это мощный инструмент для работы с текстовыми данными. Они позволяют осуществлять поиск, сопоставление и замену подстрок в строках на основе определенных шаблонов.

Синтаксис регулярных выражений: Регулярные выражения состоят из метасимволов, которые представляют особые символы или шаблоны, и квантификаторов, которые определяют количество повторений символов.

Некоторые распространенные метасимволы включают:

"." - соответствует любому символу, кроме символа новой строки.

"^" - соответствует началу строки.

"\$" - соответствует концу строки.

"\d" - соответствует любой цифре.

"\w" - соответствует любой букве или цифре.

"[ ]" - соответствует любому символу в указанном наборе.

"|" - используется для указания альтернативных вариантов.

Квантификаторы позволяют указывать количество повторений символов, например:

"\*" - ноль или более повторений.

"+" - одно или более повторений.

"?" - ноль или одно повторение.

"{n}" - ровно n повторений.

"{n, m}" - от n до m повторений.

Сочетание метасимволов и квантификаторов позволяет строить сложные шаблоны для поиска и сопоставления текста.

Поиск и замена текста: Регулярные выражения могут быть использованы для поиска подстрок в тексте и их замены. Например, можно найти все числа в строке или заменить все вхождения определенного шаблона на другую строку. Для этого используются функции и методы модуля re в Python, такие как search(), match(), findall() и sub().

### **Итераторы и генераторы:**

- Создание и использование итераторов:

Итераторы позволяют проходить по элементам коллекции последовательно и эффективно. Для создания итератора в Python необходимо определить класс, который содержит методы `__iter__()` и `__next__()`. Метод `__iter__()` возвращает сам объект итератора, а метод `__next__()` возвращает следующий элемент в последовательности или вызывает исключение `StopIteration`, если достигнут конец последовательности. После определения класса итератора, можно использовать его для обхода элементов коллекции. Для этого используется цикл `for` или функция `next()`. Пример создания и использования итератора:

```
python +
1 ~ class MyIterator:
2 ~     def __init__(self, data):
3 ~         self.data = data
4 ~         self.index = 0
5
6 ~     def __iter__(self):
7 ~         return self
8
9 ~     def __next__(self):
10 ~         if self.index >= len(self.data):
11 ~             raise StopIteration
12 ~         value = self.data[self.index]
13 ~         self.index += 1
14 ~         return value
15
16 # Использование итератора
17 my_list = [1, 2, 3, 4, 5]
18 my_iterator = MyIterator(my_list)
19 ~ for item in my_iterator:
20 ~     print(item)
```

- Генераторы:

Генераторы представляют собой удобный способ создания итерируемых объектов в Python. Вместо определения класса итератора, можно использовать функцию-генератор. Функция-генератор содержит оператор `yield`, который возвращает значение и приостанавливает выполнение функции, сохраняя свое состояние. При следующем вызове функции, выполнение продолжается с точки, где оно было остановлено.

Пример генератора, возвращающего числа от 1 до n:

```
python +
1 ~ def number_generator(n):
2 ~     for i in range(1, n+1):
3 ~         yield i
4
5 # Использование генератора
6 my_generator = number_generator(5)
7 ~ for num in my_generator:
8 ~     print(num)
```

Генераторы обладают преимуществами перед итераторами, так как они не требуют явного определения класса итератора и автоматически поддерживают итерацию через цикл `for`.

Использование итераторов и генераторов позволяет эффективно обрабатывать большие объемы данных и экономить ресурсы памяти, так как элементы генерируются по мере необходимости, а не заранее сохраняются в памяти.

### 3. Уровень Senior:

#### Декораторы:

В Python декораторы — это функции, которые принимают другую функцию в качестве аргумента и возвращают новую функцию или класс. Декораторы позволяют добавлять дополнительное поведение к функциям или классам без изменения их исходного кода.

Декораторы могут быть применены к функциям или классам, изменяя их поведение или добавляя дополнительную функциональность.

Пример создания декоратора:

```
python +
1 def my_decorator(func):
2     def wrapper():
3         print("Before function execution")
4         func()
5         print("After function execution")
6     return wrapper
7
8 @my_decorator
9 def my_function():
10     print("Inside the function")
11
12 my_function()
```

#### Многопоточность и асинхронное программирование:

Использование потоков: потоки позволяют выполнять несколько частей кода параллельно. В Python для работы с потоками можно использовать модуль **threading**.

Пример использования потоков:

```
python +
1 import threading
2
3 def my_function():
4     print("Inside the thread")
5
6 thread = threading.Thread(target=my_function)
7 thread.start()
8 thread.join()
```

Использование мьютексов: мьютексы (mutex) предоставляют механизм синхронизации для потоков. Они используются для предотвращения одновременного доступа нескольких потоков к общим ресурсам. В Python



для работы с мьютексами можно использовать модуль **threading** и класс **Lock**.

Пример использования мьютекса:

```
python +
1 import threading
2
3 counter = 0
4 lock = threading.Lock()
5
6 def increment():
7     global counter
8     lock.acquire()
9     counter += 1
10    lock.release()
11
12 threads = []
13 for _ in range(10):
14     thread = threading.Thread(target=increment)
15     threads.append(thread)
16     thread.start()
17
18 for thread in threads:
19     thread.join()
20
21 print("Counter value:", counter)
```

### Работа с базами данных:

Подключение к базе данных: для подключения к базе данных в Python можно использовать различные библиотеки, такие как **sqlite3**, **psycopg2**, **MySQLdb**. Необходимо указать параметры подключения, такие как хост, порт, имя пользователя, пароль и название базы данных.

Пример подключения к базе данных SQLite:

```
python +
1 import sqlite3
2
3 # Устанавливаем соединение с базой данных
4 connection = sqlite3.connect("example.db")
5
6 # Создаем курсор для выполнения запросов
7 cursor = connection.cursor()
8
9 # Выполняем SQL-запрос
10 cursor.execute("SELECT * FROM users")
11
12 # Получаем результаты запроса
13 results = cursor.fetchall()
14
15 # Закрываем соединение с базой данных
16 connection.close()
```

Выполнение запросов: после установления соединения с базой данных можно выполнять SQL-запросы для извлечения, изменения или удаления данных. Для выполнения запросов используется курсор, который создается на основе соединения.

Пример выполнения SELECT-запроса:

```
python +
1 import sqlite3
2
3 connection = sqlite3.connect("example.db")
4 cursor = connection.cursor()
5
6 cursor.execute("SELECT * FROM users")
7 results = cursor.fetchall()
8
9 for row in results:
10     print(row)
11
12 connection.close()
```

Работа с транзакциями: транзакции позволяют выполнять группу SQL-операций как единое целое. В случае успешного выполнения всех операций, изменения сохраняются в базе данных. В противном случае, при возникновении ошибки, все изменения откатываются.

Пример использования транзакций:

```
python +
1 import sqlite3
2
3 connection = sqlite3.connect("example.db")
4 cursor = connection.cursor()
5
6 # Начало транзакции
7 cursor.execute("BEGIN TRANSACTION")
8
9 try:
10     # Выполнение SQL-операций
11     cursor.execute("INSERT INTO users (name, email) VALUES ('John', 'john@example.com')")
12     cursor.execute("UPDATE users SET age = 30 WHERE id = 1")
13
14     # Подтверждение транзакции
15     connection.commit()
16 except:
17     # Откат транзакции при ошибке
18     connection.rollback()
19
20 connection.close()
```

## Тестирование кода:

Модульное тестирование: модульное тестирование позволяет проверить отдельные модули или функции на правильность работы. В Python для модульного тестирования используется стандартный модуль **unittest**.

Пример написания модульного теста:

```
python +
1 import unittest
2
3 def add_numbers(a, b):
4     return a + b
5
6 class MyTest(unittest.TestCase):
7     def test_add_numbers(self):
8         result = add_numbers(2, 3)
9         self.assertEqual(result, 5)
10
11 if __name__ == '__main__':
12     unittest.main()
```

В данном примере мы импортируем модуль `unittest` и определяем функцию `add_numbers`, которая складывает два числа. Затем мы создаем класс `MyTest`, унаследованный от `unittest.TestCase`, и определяем в нем метод `test_add_numbers`, который проверяет правильность работы функции `add_numbers` с помощью метода `assertEqual`.

Затем, в блоке `if __name__ == '__main__':` мы вызываем метод `unittest.main()`, который запускает все тестовые методы в классе `MyTest`.

При запуске этого скрипта, `unittest` выполнит тестовый метод `test_add_numbers` и проверит, что результат сложения 2 и 3 равен 5. Если результат будет верным, тест будет успешным, в противном случае будет выдано сообщение об ошибке.

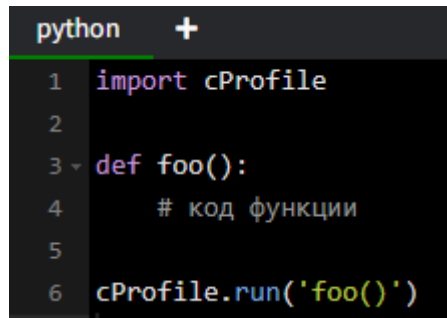
Модульное тестирование помогает обнаружить ошибки и проверить правильность работы отдельных частей кода, что способствует более надежной и устойчивой разработке программного обеспечения.

## Профилирование и оптимизация:

Профилирование — это процесс измерения производительности программного обеспечения с целью определения его узких мест и нахождения путей улучшения производительности.

В Python для профилирования можно использовать модуль `cProfile`. Он предоставляет подробную информацию о времени выполнения каждой функции в программе, количество вызовов функции и время, затраченное на каждый вызов.

Например, чтобы профилировать функцию `foo()`:

A screenshot of a code editor with a dark background. At the top, there is a tab labeled 'python' with a '+' icon to its right. The code is as follows:

```
1 import cProfile
2
3 def foo():
4     # код функции
5
6 cProfile.run('foo()')
```

Также существуют инструменты для визуализации результатов профилирования, такие как **pstats** и **snakeviz**.

Оптимизация — это процесс улучшения производительности программного обеспечения. Она может происходить на разных уровнях, начиная от алгоритмов и структур данных и заканчивая оптимизацией кода на низком уровне.

В Python можно использовать различные методы оптимизации, такие как:

- Использование более эффективных алгоритмов и структур данных
- Использование встроенных функций и методов вместо написания своих
- Использование генераторов и итераторов вместо списков, если это возможно
- Оптимизация операций с числами, например, использование операций битового сдвига вместо умножения и деления на 2
- Использование статической типизации (`type hints`) и компилятора Cython для ускорения выполнения кода

Например, рассмотрим оптимизацию кода, который суммирует все элементы в списке:

```
python +
1 # неоптимизированный код
2 def sum_list(lst):
3     result = 0
4     for i in lst:
5         result += i
6     return result
7
8 # оптимизированный код
9 def sum_list(lst):
10     return sum(lst)
```

Также можно использовать модуль **timeit** для сравнения производительности разных вариантов кода:

```
python +
1 import timeit
2
3 # замер времени выполнения неоптимизированного кода
4 t1 = timeit.timeit('sum_list([1, 2, 3, 4, 5])', setup='from __main__ import sum_list', number=100000)
5
6 # замер времени выполнения оптимизированного кода
7 t2 = timeit.timeit('sum([1, 2, 3, 4, 5])', number=100000)
8
9 print(t1, t2)
```