Функции и процедуры. Рекурсия

01 Функции

Функция | Блок кода, который можно многократно использовать в программе.

Синтаксис функции

Для определения функции в Python используется ключевое слово def - оператор, после которого должно следовать название функции, затем круглые скобки, в которых может стоять произвольное число аргументов. После скобок ставится двоеточие и следует блок кода функции, аналогично блокам If, while и так далее

Синтаксис

```
def function_name(parameter1, parameter2, ...):
    # Тело функции
    # Выполняемые операции
    # Возвращение значения (при необходимости)

# Вызов функции
function_name(argument1, argument2, ...)
```

Где

- **def** - ключевое слово, используемое для определения функции
- **function_name** - идентификатор, имя функции, которое должно быть уникальным в пределах программы
- **parameter1, parameter2, ...** - аргументы (параметры), которые могут быть переданы в функцию. Они являются переменными, которые используются внутри функции для обработки данных
- **:** - двоеточие, обозначающее начало блока кода функции.
- **# Тело функции** - комментарий, который поясняет, что блок кода ниже является телом функции
- **Выполняемые операции** - код, который выполняется при вызове функции

- **Возвращение значения (при необходимости)** - с помощью ключевого слова return можно возвратить значение из функции

Примеры

```
def add_numbers(a, b):
    sum = a + b
    return sum

result = add_numbers(3, 4)
print(result)  # Вывод: 7
```

В данном примере функция add_numbers принимает два параметра a и b, складывает их и возвращает результат с помощью return. При вызове функции с аргументами 3 и 4, будет возвращено значение 7, которое затем выводится на экран

Аргументы (параметры)

Аргументы в функции — это значения, передаваемые в функцию в момент ее вызова. Аргументы могут быть использованы внутри функции для обработки данных или выполнения операций

01 Позиционные аргументы

Позиционные аргументы - это аргументы, передаваемые в функцию в определенном порядке. Значения передаются в соответствующих позициях параметров

Примеры

```
def greet(name, age):
        print(f"Привет, {name}! Тебе {age} лет.")

    greet("Алиса", 25)
```

В данном примере, значение "Алиса" передается в name, а значение 25 - в age

02 Именованные аргументы

Именованные аргументы - это аргументы, передаваемые в функцию с указанием имени параметра. Чтение и понимание кода становится проще, так как ясно, какое значение предназначено для какого параметра

Пример

```
def greet(name, age):
      print(f"Привет, {name}! Тебе {age} лет.")

 greet(age=25, name="Алиса")
```

В данном примере аргументы передаются по имени, что позволяет явно указать, какое значение относится к конкретному параметру

03 Аргументы со значениями по умолчанию

Параметры функции могут иметь значения по умолчанию, которые используются, если аргумент не был предоставлен при вызове функции

Пример

```
def greet(name, age=18):
  print(f"Привет, {name}! Тебе {age} лет.")

  greet("Алиса")  # age по умолчанию 18
  greet("Боб", 25)  # передается значение 25 для age
```

В этом примере, если второй аргумент не указан при вызове функции, age будет иметь значение по умолчанию, равное 18

Аргументы являются гибким и мощным инструментом, который позволяет использовать функции с различными параметрами и значениями. Они упрощают передачу данных в функции и облегчают повторное использование кода

Класс function

В Python, функции являются объектами первого класса, что означает, что они могут быть присвоены переменным, передаваться в качестве аргументов другим функциям и использоваться в качестве атрибутов классов или других объектов. Также функции могут быть созданы и возвращены другими функциями

Класс function является встроенным классом, который представляет функцию в Python. Объекты этого класса могут быть созданы для любой пользовательской или встроенной функции

Пример

```
def add(a, b):
    return a + b

func_obj = add  # Присваиваем функцию add переменной func_obj
result = func_obj(2, 3)  # Вызываем функцию через переменную
print(result)  # Вывод: 5
print(isinstance(add, function))  # Проверяем тип функции add
```

В данном примере, функция add присваивается переменной func_obj. Затем мы можем вызвать функцию через переменную func_obj, таким образом, функция становится объектом класса function

Также мы можем использовать функцию isinstance() для проверки типа объекта. В данном случае, мы проверяем, является ли add экземпляром класса function

Важно отметить, что объекты класса function могут быть переданы другим функциям в качестве аргументов или возвращены из других функций, что открывает возможность создания более гибких и динамичных программных конструкций в Python

Алгоритм Евклида

алгоритм нахождения наибольшего общего делителя (НОД) двух чисел

Пример

```
def gcd(a, b):
    while b != 0:
        a, b = b, a % b
    return a

# Пример использования
num1 = 30
num2 = 18
result = gcd(num1, num2)
print(f"НОД чисел {num1} и {num2} равен {result}")
```

В данном примере функция gcd имеет два параметра a и b, которые представляют два числа, для которых мы хотим найти НОД. Алгоритм Евклида выполняется путем последовательного деления большего числа на меньшее до тех пор, пока остаток не станет равным нулю. Когда это происходит, НОД равен последнему ненулевому остатку

В данной реализации алгоритма, мы используем цикл while, чтобы продолжать деление, пока остаток не станет равным нулю. В каждой итерации значения a и b обновляются для выполнения следующего деления. В конце цикла, функция возвращает значение a, которое представляет НОД заданных чисел

В данном примере находится НОД между числами 30 и 18, и результат равен 6

02 Глобальные и локальные переменные

В Python есть глобальные и локальные переменные, которые различаются по своей области видимости и доступности в разных частях программы

Глобальные переменные

- Объявляются вне всех функций или в модуле на верхнем уровне
- Могут быть доступны и видимы в любой части программы, включая внутренние функции
- Могут быть изменены и перезаписаны в любой части программы
- Чтобы изменить значение глобальной переменной внутри функции, необходимо использовать ключевое слово global

Пример

```
global_var = 10  # Глобальная переменная

   def my_function():
       print(global_var)  # Можно получить доступ к глобальной переменной

   my_function()  # Выводит: 10

   def another_function():
       global global_var  # Используем ключевое слово global для изменения
                          глобальной переменной
       global_var = 20
   another_function()

   print(global_var)  # Выводит: 20
```

Локальные переменные

- Объявляются внутри функций или внутри блока кода
- Могут быть обращены только внутри той функции или блока кода, в котором они были объявлены
- Существуют только во время выполнения функции или блока кода и уничтожаются после завершения

Пример

```
def my_function():
       local_var = 5  # Локальная переменная
       print(local_var)

   my_function()  # Выводит: 5

   def another_function():
       for i in range(3):
           inner_var = i  # Локальная переменная внутри цикла
           print(inner_var)

   another_function()
```

Важно понимать разницу между глобальными и локальными переменными в Python. Глобальные переменные могут быть использованы и изменены в любом месте программы, но их использование должно быть осторожным и осознанным. Локальные переменные имеют меньшую область видимости и могут быть использованы только внутри определенного контекста

03 Рекурсия

Рекурсия	процесс, в котором функция вызывает саму себя во время своего выполнения
Факториал числа	это произведение всех положительных целых чисел от 1 до заданного числа. Факториал обозначается символом "!" и вычисляется с использованием рекурсии или цикла

Пример

```
def factorial(n):
    if n == 0 or n == 1:
        return 1
    else:
        return n * factorial(n-1)

# Пример использования
result = factorial(5)
print(result)  # Вывод: 120
```

В данном примере функция factorial рекурсивно вызывает саму себя с уменьшением значения n на 1 до тех пор, пока не достигнет базового условия (в данном случае n == 0 или n == 1). Когда базовое условие достигнуто, рекурсия останавливается и функция возвращает результат вычисления факториала

Классическим примером использования рекурсии является вычисление чисел Фибоначчи

Числа Фибоначчи - это последовательность чисел, в которой каждое число является суммой двух предыдущих чисел, начиная с 0 и 1

Пример

```
def fibonacci(n):
    if n <= 0:
        return []

    if n == 1:
        return [0]

    if n == 2:
        return [0, 1]

    sequence = fibonacci(n - 1)
    next_number = sequence[-1] + sequence[-2]
    sequence.append(next_number)
    return sequence

# Пример использования
result = fibonacci(10)
print(result)  # Вывод: [0, 1, 1, 2, 3, 5, 8, 13, 21, 34]
```

В данном примере функция fibonacci рекурсивно вызывает саму себя для вычисления предыдущих чисел Фибоначчи в последовательности. Функция базируется на том факте, что число Фибоначчи n является суммой двух предыдущих чисел Фибоначчи. Базовые случаи в этой рекурсивной функции - это числа 0 и 1, которые просто возвращаются в виде списков

Использование рекурсии для вычисления чисел Фибоначчи может быть неэффективным на больших значениях n из-за повторных вычислений. Поэтому для оптимизации рекурсивного решения часто используются дополнительные методы, такие как мемоизация или динамическое программирование

Рекурсия | это мощный инструмент в программировании, но она также имеет свои плюсы и минусы

Плюсы рекурсии

+ **Простота и ясность кода**
  Рекурсивные решения часто более лаконичные и понятные, поскольку они позволяют сфокусироваться на логике задачи и избавиться от сложных итераций

+ **Решение сложных задач**
  Рекурсия позволяет разделить сложную задачу на более простые подзадачи, что может упростить программирование сложных алгоритмов

+ **Универсальность**
  Рекурсия может использоваться для решения различных типов задач, таких как обработка деревьев, графов и других рекурсивных структур данных

+ **Более наглядное представление некоторых задач**
  В некоторых случаях рекурсивное решение может представлять более естественный и интуитивный способ решения проблемы, особенно когда задача формулируется в рекурсивной форме

Минусы рекурсии

— **Затраты на память**
  Рекурсивные вызовы требуют дополнительного использования памяти для хранения каждого вызова функции, что может вызвать переполнение стека вызовов при работе с большими и глубокими рекурсивными структурами данных

— **Производительность**
  Использование рекурсии может создавать больше накладных расходов и медленее работать, чем эквивалентные итеративные решения для некоторых задач

— **Сложность отладки**
  Рекурсия может быть сложной для отладки из-за повторных вызовов функций и сложности отслеживания состояния при каждом вызове

— **Возможность зацикливания**
  Неправильно написанная рекурсия может вызвать бесконечную циклическую последовательность вызовов и вызвать ошибку переполнения стека вызовов

При использовании рекурсии важно скрупулезно продумывать логику и базовые случаи, чтобы избежать потенциальных проблем с производительностью и возможные ошибки. Определенные задачи, такие как обход деревьев или графов, естественно подходят для рекурсивного решения, но в других случаях, использование циклов и итераций может быть более эффективным и простым подходом. Используйте рекурсию с умом и осознанно, и она станет полезным инструментом в вашем арсенале разработки программного обеспечения

04 Замыкания

Замыкание
Closure

функция, которая запоминает и имеет доступ к переменным из своего окружения, где она была определена, даже после того, как эта функция вышла из этого окружения. Он создает "замкнутое" окружение, которое сохраняет значения переменных, тем самым позволяя функции обращаться к ним в будущем

В Python замыкания создаются при определении вложенных функций и их возвращении или передаче в другие функции. Вложенная функция имеет доступ к переменным внешней функции и может сохранять и использовать их значения

Пример

```
def outer_function(x):
    def inner_function(y):
        return x + y
    return inner_function

closure = outer_function(10)
result = closure(5)
print(result)  # Вывод: 15
```

В данном примере, inner_function является вложенной функцией внутри outer_function. Она имеет доступ к переменной x из внешней функции outer_function, даже после завершения работы outer_function. Затем inner_function возвращается и сохраняется в переменную closure. Мы можем вызвать closure с аргументом 5, и это приведет к возвращению результата 10 + 5 = 15

Замыкания полезны в ситуациях, когда требуется сохранить и использовать изменяемые значения между вызовами функций, либо когда нужно создать функции с фиксированными параметрами на основе общей логики. Они также помогают в создании декораторов и анонимных функций (lambda-функций)

05 Декораторы

В Python декораторы позволяют модифицировать или расширять поведение функций, классов или методов, добавляя к ним дополнительную функциональность. Декораторы представляют собой особый вид функций, которые принимают другую функцию в качестве аргумента, расширяют ее функциональность и возвращают новую функцию, сохраняя ее оригинальное имя и свойства

Декораторы могут использоваться для различных задач, таких как добавление логирования, проверки аутентификации, измерения времени выполнения и многое другое. Они предоставляют гибкость и повторное использование кода

Пример

```
def decorator_function(function):
    def wrapper(*args, **kwargs):
        # Код, который выполняется перед вызовом функции
        print("Декорируем функцию")
        result = function(*args, **kwargs)
        # Код, который выполняется после вызова функции
        print("Функция выполнена")
        return result
    return wrapper

@decorator_function
def my_function():
    print("Оригинальная функция")

my_function()
```

Декораторы могут использоваться для различных задач, таких как добавление логирования, проверки аутентификации, измерения времени выполнения и многое другое. Они предоставляют гибкость и повторное использование кода

Затем мы применяем декоратор к функции my_function с помощью символа @. В результате, при вызове my_function, будет выполнена логика из decorator_function, а затем вызвана оригинальная функция

Декораторы позволяют добавлять функциональность без изменения исходного кода функций, повышая модульность и чистоту кода. Они помогают избежать дублирования кода и упрощают расширение функциональности программы

Итог

В этой теме мы разобрали понятие функции и ее синтаксис, рассмотрели процесс возвращения и передачи аргументов, изучили глобальные и локальные переменные, познакомились с понятием рекурсии, а также поговорили о замыканиях и и декораторах