Рекурсия

Теория

Рекурсия в программировании — это техника, когда функция вызывает саму себя в своем теле. Это позволяет решать задачи, которые могут быть разделены на более мелкие подзадачи того же типа.

В Python *рекурсия* реализуется путем определения функции, которая содержит хотя бы одно выражение, вызывающее эту же функцию. Для предотвращения бесконечного цикла рекурсивных вызовов необходимо предусмотреть условие выхода из *рекурсии*.

Аналогия с реальной жизнью:

- Рассмотрим процесс подсчета количества предметов в комнате. Вы можете посмотреть на каждый предмет и увеличить счетчик на 1. Но если в комнате есть шкафы или ящики, то вы можете повторить этот процесс для каждого шкафа или ящика, пока не переберете все предметы. Это рекурсивный процесс.

Отличия от цикла:

- *Рекурсия* и циклы используются для повторения кода, но есть несколько ключевых отличий:
  1. *Рекурсия* использует стек вызовов, в то время как циклы используют переменные счетчика.
  2. *Рекурсия* может быть менее эффективна, чем циклы, из-за накладных расходов на вызов функции.
  3. *Рекурсия* может быть более наглядной для некоторых задач, таких как обход деревьев или разбиение списков.

Когда использовать:

- *Рекурсию* следует использовать, когда задача может быть разделена на более мелкие подзадачи того же типа. Например, обход деревьев, поиск факториала, вычисление чисел Фибоначчи и т.д. Циклы следует использовать, когда нужно выполнить повторяющиеся действия определенное количество раз или пока выполняется определенное условие.

Стек вызовов — это область памяти, используемая для хранения информации о вызовах функций в программе. Когда функция вызывается, ее аргументы, локальные переменные и адрес возврата (то есть адрес инструкции, следующей за вызовом функции) сохраняются в *стеке вызовов*. Когда функция завершает работу, ее информация удаляется из *стека вызовов*.

Аналогии с реальной жизнью:

- Стопка тарелок в столовой. Когда тарелка берется, она кладется на верхнюю тарелку в стопке. Когда тарелка возвращается, она снимается с верхней части стопки.
- Стопка книг на столе. Когда книга берется, она кладется на верхнюю книгу в стопке. Когда книга возвращается, она снимается с верхней части стопки.

Рекурсия и *стек вызовов* тесно связаны друг с другом. Когда функция вызывает саму себя рекурсивно, каждый вызов функции добавляется в *стек вызовов*. В стеке вызовов хранятся локальные переменные и аргументы каждого вызова функции, а также адрес возврата, указывающий на точку в коде, где функция была вызвана.

Когда функция завершает свою работу, она удаляется из *стека вызовов*, и управление передается функции, которая ее вызвала. Это происходит путем возврата адреса возврата из стека вызовов и перехода к этому адресу в коде.

В рекурсивных функциях *стек вызовов* используется для хранения информации о каждом вызове функции, включая локальные переменные и аргументы. Это позволяет каждому вызову функции работать со своими собственными данными, не влияя на данные других вызовов функции.

Когда рекурсивная функция достигает условия выхода из *рекурсии*, она начинает возвращать результаты вычислений в обратном порядке, пока *стек вызовов* не станет пустым. Это происходит путем последовательного удаления каждого вызова функции из *стека вызовов* и возврата результатов вычислений из каждого вызова функции.

Таким образом, *стек вызовов* играет важную роль в *рекурсии*, позволяя реализовывать рекурсивные алгоритмы и хранить информацию о каждом вызове функции. Без стека вызовов рекурсия была бы невозможна, так как не было бы

способа хранить информацию о каждом вызове функции и возвращать результаты вычислений в обратном порядке.

Примеры рекурсии на Python:

1. Вычисление факториала с помощью *рекурсии*:

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

Здесь функция `factorial` вызывает саму себя, уменьшая значение `n` на 1, пока не достигнет 0. Когда `n` равно 0, функция возвращает 1, что является условием выхода из *рекурсии*.

Когда мы вызываем `factorial(5)`, функция `factorial` начинает выполняться с аргументом `n=5`. Затем она вызывает саму себя с аргументом `n-1`, то есть `factorial(4)`. Эта функция в свою очередь вызывает `factorial(3)`, затем `factorial(2)`, затем `factorial(1)`, и наконец `factorial(0)`. Когда `factorial(0)` возвращает 1, это значение возвращается в `factorial(1)`, которое умножает его на 1 и возвращает 1. Затем это значение возвращается в `factorial(2)`, которое умножает его на 2 и возвращает 2. Это продолжается до тех пор, пока `factorial(5)` не вернет 120, что является факториалом 5.

2. Вычисление чисел Фибоначчи с помощью *рекурсии*:

```
def fibonacci(n):
    if n <= 1:
        return n
    else:
        return fibonacci(n-1) + fibonacci(n-2)
```

Здесь функция `fibonacci` вызывает саму себя два раза, с аргументами `n-1` и `n-2`, пока не достигнет 0 или 1. Когда `n` равно 0 или 1, функция возвращает `n`, что является условием выхода из *рекурсии*.

Когда мы вызываем fibonacci(5), функция fibonacci начинает выполняться с аргументом n=5. Затем она вызывает саму себя с аргументами n-1 и n-2, то есть fibonacci(4) и fibonacci(3). Эти функции в свою очередь вызывают fibonacci(2) и fibonacci(1). Когда fibonacci(1) и fibonacci(0) возвращают 1 и 0 соответственно, эти значения используются для вычисления значения fibonacci(2), которое равно 1. Затем это значение используется для вычисления значения fibonacci(3), которое равно 2. Это продолжается до тех пор, пока fibonacci(5) не вернет 5, что является пятым числом Фибоначчи.

3. Рекурсивная функция для нахождения суммы элементов списка:

```
def sum_list(lst):
    if not lst:
        return 0
    else:
        return lst[0] + sum_list(lst[1:])
```

Здесь функция `sum_list` вызывает саму себя, уменьшая список `lst` на один элемент за раз, пока он не станет пустым. Когда список пуст, функция возвращает 0, что является условием выхода из *рекурсии*.

Когда мы вызываем `sum_list([1, 2, 3, 4, 5])`, функция `sum_list` начинает выполняться с аргументом `lst=[1, 2, 3, 4, 5]`. Затем она вызывает саму себя с аргументом `lst[1:]`, то есть `[2, 3, 4, 5]`. Эта функция в свою очередь вызывает `sum_list([3, 4, 5])`, затем `sum_list([4, 5])`, затем `sum_list([5])`, и наконец `sum_list([])`. Когда `sum_list([])` возвращает 0, это значение возвращается в `sum_list([5])`, которое прибавляет к нему первый элемент списка (5) и возвращает 5. Затем это значение возвращается в `sum_list([4, 5])`, которое прибавляет к нему первый элемент списка (4) и возвращает 9. Это продолжается до тех пор, пока `sum_list([1, 2, 3, 4, 5])` не вернет 15, что является суммой элементов списка.

Обратите внимание, что при каждом вызове функции в стеке вызовов создается новый экземпляр локальных переменных и аргументов функции. Это позволяет каждому вызову функции работать со своими собственными данными, не влияя на данные других вызовов функции. Когда функция завершает работу, ее экземпляр удаляется из *стека вызовов*, освобождая память.

Еще одна важная деталь про *стек вызовов* — это то, что он имеет ограничение на максимальную глубину *рекурсии*. Это означает, что если функция вызывает саму себя слишком много раз, то стек вызовов может переполниться, что приведет к ошибке "RecursionError: maximum recursion depth exceeded". Это происходит из-за того, что каждый вызов функции требует некоторого количества памяти для хранения локальных переменных и аргументов, и если функция вызывает саму себя слишком много раз, то *стек вызовов* может стать слишком большим, что приведет к переполнению памяти.

Поэтому, при использовании *рекурсии*, необходимо всегда предусматривать условие выхода из *рекурсии*, чтобы избежать бесконечного цикла рекурсивных вызовов и переполнения *стека вызовов*.

Задачи

1. Найти длину строки с помощью *рекурсии*.

Подсказка: Вы можете разделить строку на первый символ и оставшуюся часть. Длина строки равна сумме длины первого символа (который равен 1) и длины оставшейся части.

2. Определить, является ли строка палиндромом с помощью *рекурсии*.

Подсказка: Вы можете разделить строку на первый символ, последний символ и оставшуюся часть. Если первый и последний символы равны, то строка является палиндромом, если только оставшаяся часть также является палиндромом.

3. Найти наибольший общий делитель (НОД) двух чисел с помощью *рекурсии*.

Подсказка: Вы можете использовать алгоритм Евклида, который основан на том факте, что НОД двух чисел равен НОД второго числа и остатка от деления первого числа на второе.

Решения

1. Найти длину строки с помощью *рекурсии*.
   Решение:

```
def string_length(s):
    if s == "":
        return 0
    else:
        return 1 + string_length(s[1:])

s = "Hello, world!"
print(string_length(s))  # Вывод: 13
```

Разбор кода:

- Определяем функцию `string_length`, которая принимает один аргумент `s` — строку.
- Проверяем базовый случай: если строка пустая, то возвращаем 0 (длина пустой строки равна 0).
- В противном случае, вызываем функцию `string_length` рекурсивно с аргументом `s[1:]` (подстрока, начиная со второго символа) и прибавляем 1 к результату.
- Вызываем функцию `string_length` с аргументом `s="Hello, world!"` и выводим результат.

2. Определить, является ли строка палиндромом с помощью *рекурсии*.
   Решение:

```
def is_palindrome(s):
    if len(s) ⩽ 1:
        return True
    else:
        if s[0] ≠ s[-1]:
            return False
        else:
            return is_palindrome(s[1:-1])

s = "racecar"
print(is_palindrome(s))  # Вывод: True
```

Разбор кода:

- Определяем функцию `is_palindrome`, которая принимает один аргумент `s` — строку.
- Проверяем базовый случай: если длина строки меньше или равна 1, то возвращаем `True` (строка является палиндромом).
- В противном случае, проверяем, равны ли первый и последний символы строки. Если не равны, то возвращаем `False` (строка не является палиндромом).
- Если первый и последний символы равны, то вызываем функцию `is_palindrome` рекурсивно с аргументом `s[1:-1]` (подстрока, без первого и последнего символов).
- Вызываем функцию `is_palindrome` с аргументом `s="racecar"` и выводим результат.

3. Найти наибольший общий делитель (НОД) двух чисел с помощью *рекурсии*.
   Решение:

```
def gcd(a, b):
    if b == 0:
        return a
    else:
        return gcd(b, a % b)

a = 56
b = 98
print(gcd(a, b))  # Вывод: 14
```

Разбор кода:

- Определяем функцию `gcd`, которая принимает два аргумента `a` и `b` — два целых числа.
- Проверяем базовый случай: если `b` равно 0, то возвращаем `a` (наибольший общий делитель двух чисел равен первому числу, если второе число равно 0).
- В противном случае, вызываем функцию `gcd` рекурсивно с аргументами `b` и `a % b` (остаток от деления `a` на `b`).
- Вызываем функцию `gcd` с аргументами `a=56` и `b=98` и выводим результат.