IT CAREER HUB

Урок 24.

Рекурсия

IT CAREER HUB

Примеры рекурсивных функций

1. Факториал числа

Факториал числа n!n! – это произведение всех чисел от 1 до nn.

Рекурсивная реализация:

Python

```python
def factorial(n: int) -> int:

    if n == 0 or n == 1:  # Базовый случай

        return 1

    return n * factorial(n - 1)  # Рекурсивный случай



print(factorial(5))
```

2. Бинарный поиск (поиск в отсортированном списке)

Бинарный поиск – это эффективный алгоритм поиска элемента в отсортированном списке.

Рекурсивная реализация:

Python

```python
def binary_search(arr: list[int], target: int, left: int, right: int) -> int:

    if left > right:  # Базовый случай: элемент не найден

        return -1

    mid = (left + right) // 2

    if arr[mid] == target:

        return mid
```

```
    elif arr[mid] < target:
            return binary_search(arr, target, mid + 1, right)  # Поиск в правой части
    else:
        return binary_search(arr, target, left, mid - 1)  # Поиск в левой части


array = [1, 3, 5, 7, 9, 11, 13]
print(binary_search(array, 5, 0, len(array) - 1))
print(binary_search(array, 13, 0, len(array) - 1))
print(binary_search(array, 8, 0, len(array) - 1))
```

Хвостовая рекурсия

Хвостовая рекурсия – это особый вид рекурсии, в котором рекурсивный вызов является последней операцией функции перед возвратом результата.

Разница между обычной и хвостовой рекурсией

Обычная рекурсия

Python

```python
def factorial(n: int) -> int:
    if n == 0 or n == 1:
        return 1
    return n * factorial(n - 1)


print(factorial(5))
```

Хвостовая рекурсия

Python

```python
def factorial_tail(n: int, accumulator: int = 1) -> int:
    if n == 0 or n == 1:
        return accumulator
    return factorial_tail(n - 1, n * accumulator)


print(factorial_tail(5))
```

В Python нет встроенной оптимизации хвостовой рекурсии, которая заменяет хвостовой рекурсивный вызов на повторное использование текущего кадра стека, что позволяет избежать переполнения стека. Поэтому рекомендуется заменять

хвостовую рекурсию на итерацию, чтобы избежать переполнения стека при большом количестве рекурсивных вызовов.

Итеративный вариант факториала

Python

```python
def factorial_iterative(n: int) -> int:

    accumulator = 1

    while n > 1:

        accumulator *= n

        n -= 1

    return accumulator



print(factorial_iterative(5))
```

IT CAREER HUB

Задания для закрепления

1. Что произойдёт при выполнении следующего кода?

```python
def infinite():

    return infinite()



infinite()
```

a) Бесконечный цикл
b) Ошибка RecursionError
c) Программа завершится без ошибок
d) Ошибка SyntaxError

2. У вас есть две функции. Чем они отличаются и что будет выведено в результате выполнения?

```python
def print_numbers(n):

    if n == 0:

        return

    print(n)

    print_numbers(n - 1)



def print_nums(n):

    if n == 0:

        return
```

```
    print_nums(n - 1)

    print(n)
```

Ответ: обычная и хвостовая рекурсия. Разный порядок вывода чисел.

IT CAREER HUB

Рекурсия или итерация?

Рекурсия и итерация – два подхода к решению задач, которые могут быть взаимозаменяемыми в некоторых задачах. Однако выбор между ними зависит от производительности, читабельности кода и ограничений памяти.

Преимущества и недостатки рекурсии

Плюсы:

- Позволяет разделить сложную задачу на подзадачи.
- Упрощает чтение кода (например, обход деревьев и графов).
- Некоторые алгоритмы легче выразить рекурсивно (например, поиск пути в лабиринте).

Минусы:

- Может привести к переполнению стека (RecursionError).
- Медленнее итеративных решений из-за затрат на вызовы функций.
- Использует дополнительную память (каждый вызов создаёт новый стековый фрейм).

Когда использовать рекурсию?

- Когда задача делится на подзадачи (разбиение на более мелкие части).
- Для работы с деревьями, графами и вложенными структурами.
- Когда важна выразительность кода (код читается легче, чем итеративная версия).

Когда использовать итерацию?

- Когда рекурсия создаёт избыточную нагрузку на стек вызовов.
- Когда можно решить задачу без хранения промежуточных вызовов.
- Когда важна эффективность по памяти и скорости.

Задачи, решаемые рекурсией

Некоторые задачи сложно или невозможно эффективно решить без использования рекурсии. Это связано с необходимостью разбиения задачи на более мелкие подзадачи, а также с работой со сложными вложенными структурами.

1. **Обход дерева (DFS - поиск в глубину)**
   Дерево – это иерархическая структура данных, где каждый узел может иметь несколько дочерних узлов. Рекурсивный подход позволяет обойти все узлы, последовательно углубляясь в каждый из них.

2. **Обход графа (DFS, рекурсивный поиск путей)**
   Граф – это структура, состоящая из узлов и рёбер, соединяющих их. Использование рекурсии в глубинном поиске позволяет эффективно находить пути между вершинами.

3. **Разбор выражений и синтаксический анализ**
   При обработке вложенных математических выражений или языковых конструкций удобно использовать рекурсию, так как каждое подвыражение может рассматриваться как самостоятельная подзадача.

4. **Ханойские башни**
   Задача, требующая последовательного перемещения дисков между тремя стержнями с сохранением их порядка. Разбивается на две меньшие аналогичные задачи, что делает рекурсию естественным решением.

5. **Разбиение на подмножества и комбинаторика**
   Задачи, связанные с генерацией всех возможных комбинаций, перестановок и подмножеств множества, решаются с разбиением на подзадачи: выбор одного элемента и рекурсивное построение оставшейся структуры.

Функция deepcopy

Функция `deepcopy()` из модуля `copy` используется для создания глубокой копии объекта. В отличие от обычного `copy()`, который выполняет поверхностное копирование, `deepcopy()` рекурсивно копирует все вложенные объекты, создавая полностью независимую копию.

Когда использовать `deepcopy()`?

- Если объект содержит вложенные изменяемые структуры (`list`, `dict`, `set`).
- Если необходимо избежать изменения оригинальных данных при работе с копией.

Разница между `copy()` и `deepcopy()`

Операция	Что копируется?	Вложенные изменяемые объекты
`copy()`	Поверхностный уровень	Сохраняют ссылки на оригинальные объекты
`deepcopy()`	Полная независимая копия	Создаются новые копии вложенных объектов

Пример с `copy()` (поверхностная копия):

```python
original_list = [[1, 2], [3, 4]]


# Поверхностная копия, вложенные коллекции скопированы как ссылки
copy_lst = original_list.copy()
# Добавляет в копию, не затрагивает вложенные элементы.
copy_lst.append(99)
# Изменяет копию списка, но затрагивает вложенные элементы.
copy_lst[0][0] = "X"  # Влияет на оригинал!


print("Оригинал:", original_list)
print("Копия:", copy_lst)
```

Пример с `deepcopy()` (глубокая копия):

Python

```python
from copy import deepcopy


original_list = [[1, 2], [3, 4]]


# Глубокая копия, для вложенных коллекций созданы дубликаты объектов
copy_lst = deepcopy(original_list)
# Добавляет в копию, не затрагивает вложенные элементы.
copy_lst.append(99)
# Изменяет вложенные элементы, которые не связаны с изначальным списком.
copy_lst[0][0] = "X"  # Не влияет на оригинал!


print("Оригинал:", original_list)
print("Копия:", copy_lst)
```

Когда `deepcopy()` не нужен?

- Если объект не содержит изменяемых вложенных объектов.
- Если производительность критична — `deepcopy()` медленнее, чем `copy()`, а копируемые данные не требуют полного дублирования.

IT CAREER HUB

Функция isinstance

`isinstance(obj, class)` — это встроенная функция Python, которая проверяет, является ли объект `obj` экземпляром указанного класса или его подкласса.

Синтаксис:

```python
isinstance(obj, classinfo)
```

- `obj` – объект, который проверяется.
- `classinfo` – класс или кортеж классов, с которыми выполняется проверка.

Возвращает `True`, если объект принадлежит указанному классу или его подклассу, иначе `False`.

Примеры использования `isinstance`

Пример 1: Проверка типа переменной

```python
x = 10

y = "Hello"



print(isinstance(x, int))  # True

print(isinstance(y, str))  # True

print(isinstance(y, int))  # False
```

Пример 2: Проверка нескольких типов

Можно передать кортеж классов, чтобы проверить принадлежность к нескольким типам.

```python
value = 3.14

# Проверяем, является ли число целым или вещественным

if isinstance(value, (int, float)):

    print("Число")

else:

    print("Не число")
```

Пример 3: Фильтрация данных по типу

С помощью `isinstance()` можно отбирать элементы нужного типа из списка.

```python
data = [1, "hello", 2.5, True, "world", 42]

numbers = [x for x in data if isinstance(x, (int, float))]

print(numbers)  # [1, 2.5, 42]
```

IT CAREER HUB

Задания для закрепления

Чем `deepcopy()` отличается от `copy()`?

a. `copy()` создаёт полную независимую копию, включая вложенные объекты
b. `copy()` всегда создаёт новый объект без ссылок на оригинал
c. `deepcopy()` создаёт полную независимую копию, включая вложенные объекты
d. `deepcopy()` выполняет копирование быстрее, чем `copy()`

IT CAREER HUB

Практические задания

Функция `deepcopy`

Реализуйте аналог `deepcopy()` с помощью рекурсии. Не забудьте проверить, чтобы изменения в копии не затронули оригинал.

Данные:

```python
original_data = [
    [1, 2, 3],                  # Вложенный список
    (4, [5, 6], {7, 8}),        # Кортеж с вложенными структурами
    {"a": 9, "b": [10, 11]},    # Словарь со списком
    "Hello",                    # Строка
    [12, (13, 14)],             # Список с кортежем
    15.5,                       # Число с плавающей точкой
    5                           # Целое число
]
```

IT CAREER HUB

Проверяем, что копия независима от оригинала

Python
```
original_data[1][1][0] = 0
```

Пример вывода:

Unset
```
Исходный: [[1, 2, 3], (4, [0, 6], {8, 7}), {'a': 9, 'b': [10, 11]}, 'Hello',
[12, (13, 14)], 15.5, 5]

Копия:    [[1, 2, 3], (4, [5, 6], {8, 7}), {'a': 9, 'b': [10, 11]}, 'Hello',
[12, (13, 14)], 15.5, 5]
```

Решение:

Python
```
def deep_copy(data):

    if isinstance(data, list):

        return [deep_copy(item) for item in data]

    elif isinstance(data, dict):

        return {key: deep_copy(value) for key, value in data.items()}

    elif isinstance(data, set):

        return {deep_copy(item) for item in data}

    elif isinstance(data, tuple):

        return tuple([deep_copy(item) for item in data])

    else:

        return data
```

```
original_data = [
    [1, 2, 3],                  # Вложенный список
    (4, [5, 6], {7, 8}),        # Кортеж с вложенными структурами
    {"a": 9, "b": [10, 11]},    # Словарь со списком
    "Hello",                    # Строка
    [12, (13, 14)],             # Список с кортежем
    15.5,                       # Число с плавающей точкой
    5                           # Целое число
]
copied_data = deep_copy(original_data)

# Проверяем, что копия независима от оригинала
original_data[1][1][0] = 0

print(f"Исходный: {original_data}")
print(f"Копия:    {copied_data}")
```

Обратный вывод строки

Напишите рекурсивную функцию, которая выводит строку в обратном порядке.

Данные:

Python
```
text = "hello"
```

Пример вывода:

```
Unset
olleh
```

Решение:

```python
def reverse_string(s):

    if not s:

        return ""

    return s[-1] + reverse_string(s[:-1])



text = "hello"

print(reverse_string(text))
```

Подсчёт определённых слов

Напишите рекурсивную функцию, которая подсчитывает количество вхождений определённого слова во вложенной структуре (список списков строк).

Данные:

```python
nested_sentences = [

    ["Python is great", "I love Python"],

    ["Python is powerful", ["Python is everywhere", "Learn Python"]],

    "Coding in Python is fun"

]

word = "Python"
```

Пример вывода:

```
Количество вхождений: 6
```

Решение:

```python
def count_word(nested_sentences, word):
    if isinstance(nested_sentences, str):
        return nested_sentences.split().count(word)
    if isinstance(nested_sentences, list):
        return sum(count_word(sublist, word) for sublist in nested_sentences)
    return 0


nested_sentences = [
    ["Python is great", "I love Python"],
    ["Python is powerful", ["Python is everywhere", "Learn Python"]],
    "Coding in Python is fun"
]
word = "Python"


print("Количество вхождений:", count_word(nested_sentences, word))
```