GeekBrains

Функции, рекурсия, алгоритмы.

Курс

Оглавление

Термины, используемые в лекции

Функция — это фрагмент программы, используемый многократно:

Alias (псевдоним) — альтернативное имя, которое даётся функции при её импорте из файла.

Рекурсия — это функция, вызывающая сама себя.

Функции и модули

💡 Функция — это фрагмент программы, используемый многократно

Мы знакомы уже с функциями с C#, давайте теперь посмотрим, как создаются и используются функции внутри Python.

```
def function_name(x):
     # body line 1
              # ...
     # body line n
     # optional return
```

Задание: Необходимо создать функцию **sumNumbers(n)**, которая будет считать сумму всех элементов от 1 до n.

Решение:

1. Необходимо создать функцию:

```
def sumNumbers(n):
```

Очень важно понимать одну вещь, сколько аргументов мы передаем, столько и принимаем. Или наоборот сколько аргументов мы принимаем, столько и передаем. В нашем случае функция **sumNumbers** принимает 1 аргумент(n).

2. Реализовать решение задачи внутри функции

```
def sumNumbers(n):
    summa = 0
    for i in range(1, n + 1):
        summa += i
    print(summa)
```

Так как нам нужны все значения из промежутка [1, n], мы вызываем функцию **range**, от 1 до n + 1, так как range **не** включает последний элемент. **НО** возникает вопрос, почему мы здесь используем **print()** ? Где же всем известный **return**, которым мы пользовались на C#?

На самом деле, нам не нужно теперь писать **void**, для того чтобы выводить данные, а для того чтобы возвращать значения ставить их тип, все намного проще. В Python нет такого понятия как процедура(void). Здесь существует только **def.** Вернемся к задаче

3. Спросим у пользователя число

```
def sumNumbers(n):
    summa = 0
    for i in range(1, n + 1):
        summa += i
    print(summa)

n = int(input()) # 5
sumNumbers(n) # 15
```

Программный код, который мы написали прекрасно справляется с поставленной задачей. Давайте изменим наш код и добавим в него **return. НО** перед этим давайте вспомним, что делает **return:**

1. Завершает работу функции
2. Возвращает значение

```
def sumNumbers(n):
    summa = 0
    for i in range(1, n + 1):
        summa += i
    return summa

n = int(input()) # 5
print(sumNumbers(n)) # 15
```

Модульность

Вы когда-нибудь задавались вопрос, как например работает функция **.append**
Это же точно такая же функция, как и **sumNumbers(n)**, но мы ее нигде не создаем, все дело в том что, это функция автоматически срабатывает и чтобы ей пользоваться ничего дополнительно писать не надо.
Представьте себе такую ситуацию, что Вы создаете огромный проект и у Вас имеется большое количество функций, к примеру 5 функций работают со словарями, 18 со списками и тд. и у каждой функции свой алгоритм, но их объединяет работа с одной коллекцией данных. Согласитесь неудобно работать в таком большом файле, где около 80 функций, очень легко потеряться и на перемотку кода Вы будете терять драгоценное время. Решение данной проблемы есть. Давай будем создавать отдельные файлы, где будут находиться только функции, и эти функции при необходимости вызывать из главного файла.

1. **function_file.py (**Новый Python файл, в котором находятся функция f(x)**)**

```
def f(x):
    if x == 1:
        return 'Целое'
    elif x == 2.3:
        return 23
    return # выход из функции
```

2. **working_file.py**

Чтобы начать взаимодействовать с функцией в файле **function_file.py** необходимо добавить эту возможность к себе в программный код. Сначала мы обращаемся к файлу(без расширения)

С помощью **import** мы можем вызвать эту функцию в другом скрипте и дальше использовать её в новом файле. Можно сократить название функции в рабочем файле с помощью команды:

💡 Alias (псевдоним) — альтернативное имя, которое даётся функции при et импорте из файла.

```
import function_file
```

```
print(function_file.f(1)) # Целое

print(function_file.f(2.3)) # 23

print(function_file.f(28)) # None
```

Значения по умолчанию для функции

💡 В Python можно перемножать строку на число.

В данной функции есть два аргумента: ***symbol*** (символ или число) и ***count*** (число, на которое умножается первый аргумент).

Если введены оба аргумента, функция работает без ошибок. Если только символ — функция выдает ошибку.

```
def new_string(symbol, count):

     return symbol * count

 print(new_string('!', 5))    # !!!!!

 print(new_string('!')) # TypeError missing 1 required ...
```

Можно указать значение переменной count по умолчанию. Например, если значение явно не указано (нет второго аргумента), по умолчанию значение переменной count равно трем.

```
def new_string(symbol, count=3):

     return symbol * count


print(new_string('!', 5)) # !!!!!

print(new_string('!')) # !!!

print(new_string(4)) # 12
```

Возможность передачи неограниченного количества аргументов

- Можно указать любое количество значений аргумента функции.
- Перед аргументом надо поставить *.

В примере ниже функция работает со строкой, поэтому при введении чисел программа выдаёт ошибку:

```
def concatenatio(*params):
    res = ""
    for item in params:
        res += item
    return res


print(concatenatio('a', 's', 'd', 'w'))  # asdw
print(concatenatio('a', '1'))  # a1
# print(concatenatio(1, 2, 3, 4)) # TypeError: ...
```

Рекурсия

💡 Рекурсия — это функция, вызывающая сама себя.

С рекурсией Вы знакомы с C#, в Python она ничем не отличается, давай рассмотрим следующую **задачу:** Пользователь вводит число n. Необходимо вывести n - первых членов последовательности Фибоначчи.
Напоминание: Последовательно Фибоначчи, это такая последовательность, в которой каждое последующее число равно сумму 2-ух предыдущих

💡 При описании рекурсии важно указать, когда функции надо остановиться и перестать вызывать саму себя. По-другому говоря, необходимо указать базис рекурсии

Решение :

```
def fib(n):
    if n in [1, 2]:
        return 1
    return fib(n - 1) + fib(n - 2)


list_1 = []
for i in range(1, 10):
    list_1.append(fib(i))
print(list_1) # [1, 1, 2, 3, 5, 8, 13, 21, 34]
```

Внутри функции fib(n), мы сначала задаем базис, если число n равно 1 или 2, это означает, что первое число и второе число последовательности равны 1. Мы так и делаем возвращаем 1. Как мы ранее проговорили: “Последовательно Фибоначчи, это такая последовательность, в которой каждое последующее число равно сумму 2-ух предыдущих”. Так и делаем, складываем на 2 предыдущих числа друг с другом и получаем 3.

Алгоритмы

Алгоритмом называется набор инструкций для выполнения некоторой задачи. В принципе, любой фрагмент программного кода можно назвать алгоритмом, но мы с Вами рассмотрим 2 самых интересных алгоритмы сортировок:

- Быстрая сортировка
- Сортировка слиянием

Быстрая сортировка

“Программирование это разбиение чего-то большого и невозможного на что-то маленькое и вполне реальное”

Быстрая сортировка принадлежит такой стратегии, как “разделяй и властвуй”. Сначала рассмотрим пример, затем напишем программный код

Два друга решили поиграть в игру: один загадывает число от 1 до 100, другой должен отгадать. Согласитесь, что мы можем перебирать эти значения в случайном порядке, например: 32, 27, 60, 73... Да, мы можем угадать в какой-то момент, но что если мы обратиться к стратегии “разделяй и властвуй” Обозначим друзей, друг_1 это Иван, который загадал число, друг_2 это Петр, который отгадывает. Итак начнем:

Иван загадал число **77.**

Петр: Число больше 50? Иван: Да.

Петр: Число больше 75? Иван: Да.

Петр: Число больше 87? Иван: Нет.

Петр: Число больше 81? Иван: Нет.

Петр: Число больше 78? Иван: Нет.

Петр: Число больше 76? Иван: Да

Число оказалось в диапазоне $76 < x < 78$, значит это число 77. Задача решена. На самом деле мы сейчас познакомились с алгоритмом бинарного поиска, который также принадлежит стратегии “разделяй и властвуй”. Давайте перейдем к обсуждению программного кода быстрой сортировки.

```
def quicksort(array):
    if len(array) < 2:
        return array
    else:
        pivot = array[0]
        less = [i for i in array[1:] if i <= pivot]
        greater = [i for i in array[1:] if i > pivot]
        return quicksort(less) + [pivot] + quicksort(greater)

print(quicksort([10, 5, 2, 3]))
```

- **1-е** повторение рекурсии:
  - array = [10, 5, 2, 3]
  - pivot = 10
  - less = [5, 2, 3]
  - greater = []
  - return quicksort([5, 2, 3]) + [10] + quicksort([])
- **2-е** повторение рекурсии:
  - array = [5, 2, 3]
  - pivot = 5
  - less = [2, 3]
  - greater = []
  - return quicksort([2, 3]) + [5] + quicksort([]) # Важно! Не забывайте, что здесь помимо вызова рекурсии добавляется список [10]
- **3-е** повторение рекурсии:
  - array = [2, 3]
  - return [2, 3] # Сработал базовый случай рекурсии

На этом работа рекурсии завершилась и итоговый список будет выглядеть таким образом: [2, 3] + [5] + [10] = [2, 3, 5, 10]

Сортировка слиянием

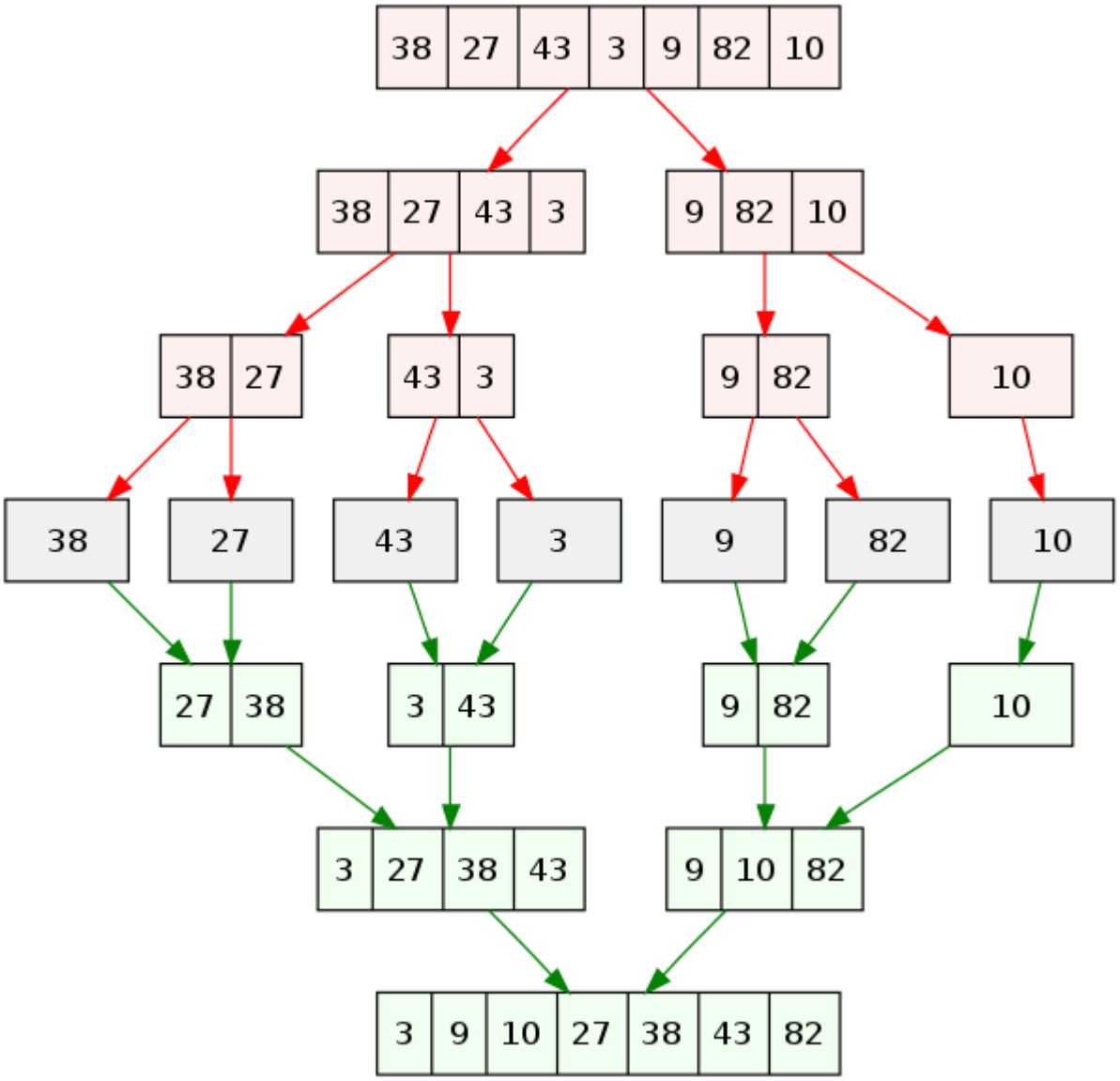
```
def merge_sort(nums):
    if len(nums) > 1:
        mid = len(nums) // 2
        left = nums[:mid]
        right = nums[mid:]
        merge_sort(left)
        merge_sort(right)
        i = j = k = 0
        while i < len(left) and j < len(right):
            if left[i] < right[j]:
                nums[k] = left[i]
                i += 1
            else:
                nums[k] = right[j]
                j += 1
            k += 1
        while i < len(left):
            nums[k] = left[i]
            i += 1
            k += 1

        while j < len(right):
            nums[k] = right[j]
            j += 1
            k += 1
```

```
nums = [38, 27, 43, 3, 9, 82, 10]

merge_sort(nums)

print(nums)
```



Итоги:

1. *Создание и использование функций*
2. *Рекурсию*
3. *Алгоритмы сортировок*