**Технологии прикладного анализа данных**

**Лабораторная работа №5**

**Функции и массивы**

# Функции и их переменные

Для работы потребуется файл data\_no\_header.txt

**Функции** позволяют выделить часть кода в отдельный объект. При этом на вход функции можно подавать набор параметров, а в ответ получать результат определенных преобразований. Таким образом можно тестировать часть кода, над которым вы работаете в данный момент. Например, если вы пишете некий алгоритм в цикле, то сначала было бы здорово написать и протестировать этот код вне цикла (без необходимости каждый раз прогонять цикл целиком). А затем просто вставить вызов этой части кода (т. е. написанной функции) в цикл. Это существенно упростит решение задачи.

Также при решении задач обработки данных мы неизбежно сталкиваемся с новыми требованиями к алгоритму расчета. С помощью функций мы можем вносить необходимые изменения в тело функции, не трогая основной цикл.

Рассмотрим следующий пример: мы хотим посчитать определенную метрику в файле. Схематично код будет выглядеть так:

with open( 'data\_no\_header.txt', 'r' ) as f:

    for line in f:

        line = line.strip().split('\t')

        # код с формированием словаря статистики

Теперь нам нужно отфильтровать строки по условиям: брать только источник 'google' и 'yandex'. Тогда код изменится:

with open( 'data\_no\_header.txt', 'r' ) as f:

    for line in f:

        line = line.strip().split('\t')

        source = line[5]

        medium = line[4]

        if source == 'google' or source == 'yandex':

            sourceType = 'search engines'

        else:

            sourceType = 'other'

        # код с формированием словаря статистики

Код растянулся на 6 строк. Для его тестирования и проверки надо прогонять весь цикл (если файл будет большой, то для проверки каждый раз придется ждать пока он отработает).

Допустим, поступило еще одно требование: надо разделять тип search engines на два типа в зависимости от значения medium. Если medium равен seo или sem, то ставить тип 'search engines seo', для значения brand - 'search engines brand':

with open( 'data\_no\_header.txt', 'r' ) as f:

    for line in f:

        line = line.strip().split('\t')

        source = line[5]

        medium = line[4]

        if source == 'google' or source == 'yandex':

            if medium == 'seo' or medium == 'sem':

                sourceType = 'search engines seo'

            elif medium == 'brand':

                sourceType = 'search engines brand'

            else:

                # если вдруг встретится другой вариант, то ставим "undefined"

                sourceType = 'search engines undefined'

        else:

            sourceType = 'other'

        # код с формированием словаря статистики

Какое значение переменной sourceType получится для последней строчки в файле?

# Упрощение при помощи функции

В итоге получилась громоздкая конструкция, проверить корректность работы, которой сложно. Если появится еще одно требование, то вносить изменения в эти условия и проверять их будет очень сложно.

Давайте вынесем наши проверки в функцию. Т.е. часть кода, на вход которой можно подавать значения source и medium, а на выходе получать значение sourceType. При этом говорят, что функция "возвращает" значение sourceType.

При написании функции пишется 'def', название функции и в скобках переменные, которые вы подаете ей на вход.

def source\_type\_classification( source, medium ):

При составлении функции неплохо добавлять описание того, что за переменные надо ей передавать. И что она возвращает. Описание функции дается в тройных кавычках:

def source\_type\_classification( source, medium ):

    """

    Функция классифицирует источник трафика в зависимости от значения source и medium.

    Возвращает название типа источника по правилам:

        - если source равен 'google' или 'yandex', то проверяем medium:

            - для medium 'seo' или 'sem' возвращаем 'search engines seo'

            - для medium 'brand' - возвращаем 'search engines brand'

            - для остальных случаев возвращаем 'search engines undefined'

        - если условие не выполнено, то возвращаем 'other'

    """

В теле функции будут прописаны наши правила. Если условие выполнено, то функция останавливает выполнение и отдает значение с помощью оператора return. Скопируем наш код из цикла в тело функции и заменим 'sourceType = ...' на 'return ...'

def source\_type\_classification( source, medium ):

    """

    Функция классифицирует источник трафика в зависимости от значения source и medium.

    Возвращает название типа источника по правилам:

        - если source равен 'google' или 'yandex', то проверяем medium:

            - для medium 'seo' или 'sem' возвращаем 'search engines seo'

            - для medium 'brand' - возвращаем 'search engines brand'

            - для остальных случаев возвращаем 'search engines undefined'

        - если условие не выполнено, то возвращаем 'other'

    """

    if source == 'google' or source == 'yandex':

        if medium == 'seo' or medium == 'sem':

            return 'search engines seo'

        elif medium == 'brand':

            return 'search engines brand'

        else:

            # если вдруг встретится другой вариант, то ставим "undefined"

            return 'search engines undefined'

    else:

        return 'other'

Что дает такой перенос кода в функцию? Теперь мы можем потестировать верность нашей классификации без необходимости прогонять весь цикл. Для source = 'google' и medium = 'seo' должны получить 'search engines seo'. Проверим так ли это:

source\_type\_classification( 'google', 'seo' )

'search engines seo'

Пока все верно. Потестируем на других значениях

print( source\_type\_classification( 'yandex', 'seo' ) ) # должны получить 'search engines seo'

search engines seo

print( source\_type\_classification( 'google', 'brand' ) ) # должны получить 'search engines brand'

search engines brand

Какое значение должна выдать функция для source = 'google' и medium = '123'?

# Использование функций в цикле

Давайте используем нашу функцию в цикле. Мы можем заменить громоздкие проверки простым вызовом функции, корректность которой мы уже проверили:

with open( 'data\_no\_header.txt', 'r' ) as f:

    for line in f:

        line = line.strip().split('\t')

        source = line[5]

        medium = line[4]

        sourceType = source\_type\_classification( source, medium )

        # код с формированием словаря статистики

Выглядит значительно проще. Теперь при появлении новых условий и требований мы будем вносить их в код функции. И тестировать ее несколькими командами без необходимости прогонять весь цикл. Код цикла при этом вообще не надо будет трогать.

Давайте посчитаем количество заказов по типам источника трафика. Т. е. в разбивке по search engines seo, search engines brand, search engines undefined (если попадется такая комбинация) и other.

source\_type\_stats = {}

with open( 'data\_no\_header.txt', 'r' ) as f:

    for line in f:

        line = line.strip().split('\t')

        source = line[5]

        medium = line[4]

        sourceType = source\_type\_classification( source, medium )

        if sourceType in source\_type\_stats:

            source\_type\_stats[ sourceType ] += 1

        else:

            source\_type\_stats[ sourceType ] = 1

Выведите на экран получившийся словарь source\_type\_stats.

Какое значение у ключа 'search engines seo'?

# Обработка строк с помощью функции

Сейчас мы передаем в функцию значения source и medium. Но может оказаться так, что новые требования потребуют учесть другие столбцы. Например, в зависимости от стоимости заказа надо будет различать тип заказа. Представим, что нам надо проанализировать откуда приходят дорогие заказы, а откуда низкий чек.

Давайте начнем с того, что сделаем функцию более универсальной и передадим ей в качестве аргумента все данные по заказу (т. е. всю строку line):

def source\_type\_classification( line ):

    """

    Пример строки на входе функции:

    ['40443', '05.10.2016 23:18', '1010', '0,000925926', 'seo', 'google', '0', '6243', '20,20']

    Функция классифицирует источник трафика в зависимости от значения source и medium.

    Возвращает название типа источника по правилам:

        - если source равен 'google' или 'yandex', то проверяем medium:

            - для medium 'seo' или 'sem' возвращаем 'search engines seo'

            - для medium 'brand' - возвращаем 'search engines brand'

            - для остальных случаев возвращаем 'search engines undefined'

        - если условие не выполнено, то возвращаем 'other'

    """

    source = line[5]

    medium = line[4]

    if source == 'google' or source == 'yandex':

        if medium == 'seo' or medium == 'sem':

            return 'search engines seo'

        elif medium == 'brand':

            return 'search engines brand'

        else:

            # если вдруг встретится другой вариант, то ставим "undefined"

            return 'search engines undefined'

    else:

        return 'other'

Проверьте, что функция работает корректно как и раньше. Вызовите функцию со следующим значением аргумента

['40443', '05.10.2016 23:18', '1010', '0,000925926', 'seo', 'google', '0', '6243', '20,20']

Какой результат она выдаст?

# Добавление цены в классификацию

Теперь в функцию можно добавить условие на стоимость заказа:

    - если стоимость больше 20 рублей, то это заказ типа 'expensive'

    - если меньше - 'cheap'

Нужно возвращать два значения - sourceType и orderType. Добавим это условие в функцию:

if amount\_paid >= 20:

   orderType = 'expensive'

else:

   orderType = 'cheap'

И возвращать теперь будем пару значений:

return sourceType, orderType

Итоговый код функции:

def source\_type\_classification( line ):

    source = line[5]

    medium = line[4]

    amount\_paid = float( line[-1].replace(',', '.') )

    if amount\_paid >= 20:

        orderType = 'expensive'

    else:

        orderType = 'cheap'

    if source == 'google' or source == 'yandex':

        if medium == 'seo' or medium == 'sem':

            sourceType = 'search engines seo'

        elif medium == 'brand':

            sourceType = 'search engines brand'

        else:

            # если вдруг встретится другой вариант, то ставим "undefined"

            sourceType = 'search engines undefined'

    else:

        sourceType = 'other'

    return sourceType, orderType

В итоге для подсчета количества заказов в разбивке по типам трафика и стоимости заказа нам потребуется следующий цикл:

source\_type\_stats = {}

with open( 'data\_no\_header.txt', 'r' ) as f:

    for line in f:

        line = line.strip().split('\t')

        sourceType, orderType = source\_type\_classification( line )

        source\_order\_type = sourceType + ' ' + orderType

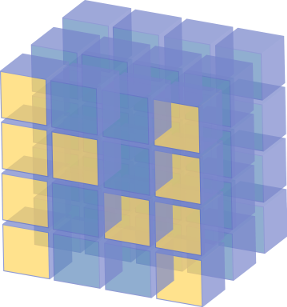
        if source\_order\_type in source\_type\_stats:

            source\_type\_stats[ source\_order\_type ] += 1

        else:

            source\_type\_stats[ source\_order\_type ] = 1

Какое значение получим в словаре source\_type\_stats для ключа 'search engines seo cheap'?



# Numerical Python

NumPy – это библиотека языка Python, добавляющая поддержку больших многомерных массивов и матриц, вместе с большой библиотекой высокоуровневых (и очень быстрых) математических функций для операций с этими массивами.

Основным объектом NumPy является однородный многомерный массив (в numpy называется numpy.ndarray). Это многомерный массив элементов (обычно чисел), одного типа.

Наиболее важные атрибуты объектов ndarray:

**ndarray.ndim** - число измерений (чаще их называют "оси") массива.

**ndarray.shape** - размеры массива, его форма. Это кортеж натуральных чисел, показывающий длину массива по каждой оси. Для матрицы из n строк и m столбов, shape будет (n,m). Число элементов кортежа shape равно ndim.

**ndarray.size** - количество элементов массива. Очевидно, равно произведению всех элементов атрибута shape.

**ndarray.dtype** - объект, описывающий тип элементов массива. Можно определить dtype, используя стандартные типы данных Python. NumPy здесь предоставляет целый букет возможностей, как встроенных, например: bool\_, character, int8, int16, int32, int64, float8, float16, float32, float64, complex64, object\_, так и возможность определить собственные типы данных, в том числе и составные.

**ndarray.itemsize** - размер каждого элемента массива в байтах.

# Создание массивов

В NumPy существует много способов создать массив. Один из наиболее простых - создать массив из обычных списков или кортежей Python, используя функцию numpy.array() (array - функция, создающая объект типа ndarray):

import numpy as np

a = np.array([1, 2, 3])

a

# строка вывода

type(a)

# строка вывода

Функция array() трансформирует вложенные последовательности в многомерные массивы. Тип элементов массива зависит от типа элементов исходной последовательности (но можно и переопределить его в момент создания).

b = np.array([[1.5, 2, 3], [4, 5, 6]])

b

array([[1.5, 2. , 3. ],

[4. , 5. , 6. ]])

Можно также переопределить тип элементов в момент создания:

b = np.array([[1.5, 2, 3], [4, 5, 6]], dtype=np.complex)

b

array([[1.5+0.j, 2. +0.j, 3. +0.j],

[4. +0.j, 5. +0.j, 6. +0.j]])

Функция array() не единственная функция для создания массивов. Обычно элементы массива вначале неизвестны, а массив, в котором они будут храниться, уже нужен. Поэтому имеется несколько функций для того, чтобы создавать массивы с каким-то исходным содержимым (по умолчанию тип создаваемого массива – float64).

Функция zeros() создает массив из нулей, а функция ones() – массив из единиц. Обе функции принимают кортеж с размерами, и аргумент dtype:

np.zeros((3, 5))

np.ones((2, 2, 2))

Функция eye() создаёт единичную матрицу (двумерный массив):

Функция empty() создает массив без его заполнения. Исходное содержимое случайно и зависит от состояния памяти на момент создания массива (то есть от того мусора, что в ней хранится):

np.empty((3, 3)) – запустить в двух разных строчках и сравнить результаты.

Для создания последовательностей чисел, в NumPy имеется функция arange(), аналогичная встроенной в Python range(), только вместо списков она возвращает массивы, и принимает не только целые значения:

np.arange(10, 30, 5)

array([10, 15, 20, 25])

np.arange(0, 1, 0.1)

Вообще, при использовании arange() с аргументами типа float, сложно быть уверенным в том, сколько элементов будет получено (из-за ограничения точности чисел с плавающей запятой). Поэтому, в таких случаях обычно лучше использовать функцию linspace(), которая вместо шага в качестве одного из аргументов принимает число, равное количеству нужных элементов:

np.linspace(0, 2, 9) # 9 чисел от 0 до 2 включительно

fromfunction(): применяет функцию ко всем комбинациям индексов

def f1(i, j):

return 3 \* i + j

np.fromfunction(f1, (3, 4))

array([[0., 1., 2., 3.],

[3., 4., 5., 6.],

[6., 7., 8., 9.]])

Что будет результатом выполнения np.fromfunction(f1, (3, 3))?

# Печать массивов

Если массив слишком большой, чтобы его печатать, NumPy автоматически скрывает центральную часть массива и выводит только крайние значения.

print(np.arange(0, 3000, 1))

Если вам действительно нужно увидеть весь массив, используйте функцию numpy.set\_printoptions:

np.set\_printoptions(threshold=np.nan)

У данной функции много аргументов, которые позволяют настроить печать массива самыми различными способами. Более подробно можно почитать в документации. <https://docs.scipy.org/doc/numpy/reference/>

# Базовые операции

Математические операции над массивами выполняются поэлементно. Создается новый массив, который заполняется результатами действия оператора.

a = np.array([20, 30, 40, 50])

b = np.arange(4)

a + b

array([20, 31, 42, 53])

Выполните все основные арифметические операции для данных массивов (\*, /, -, //, \*\*,%).

Естественным ограничением подобных операций является то, что массивы должны быть одинаковых размеров.

c = np.array([[1, 2, 3], [4, 5, 6]])

d = np.array([[1, 2], [3, 4], [5, 6]])

c + d

# должны получить ошибку

Также можно производить математические операции между массивом и числом. В этом случае к каждому элементу прибавляется (или другая, используемая арифметическая операция) это число.

a + 1

array([21, 31, 41, 51])

Выполните:

a \*\* 3

a < 35

NumPy также предоставляет множество математических операций для обработки массивов (что очень упрощает жизнь):

np.cos(a)

np.arctan(a)

np.sinh(a)

Полный список таких функций представлен в <https://docs.scipy.org/doc/numpy/reference/routines.math.html>

Многие унарные операции, такие как, например, вычисление суммы всех элементов массива, представлены также и в виде методов класса ndarray.

a = np.array([[1, 2, 3], [4, 5, 6]])

np.sum(a)

a.sum()

a.min()

a.max()

По умолчанию, эти операции применяются к массиву, как если бы он был списком чисел, независимо от его формы. Однако, указав параметр axis, можно применить операцию для указанной оси массива:

a.min(axis=0) # Наименьшее число в каждом столбце

a.min(axis=1) # Наименьшее число в каждой строке

# Индексы, срезы, итерации

Одномерные массивы осуществляют операции индексирования, срезов и итераций очень схожим образом с обычными списками (list) и другими последовательностями Python (разве что удалять с помощью срезов нельзя).

a = np.arange(10) \*\* 3

Для данного массива выведите первый элемент массива, элементы массива со второго по шестой, включительно. Затем замените значения элементов со второго по шестой на 8 и выведите весь массив. После этого выведите массив в обратном порядке.

У многомерных массивов на каждую ось приходится один индекс. Индексы передаются в виде последовательности чисел, разделенных запятыми (то есть, кортежами):

b = np.array([[ 0, 1, 2, 3],

[10, 11, 12, 13],

[20, 21, 22, 23],

[30, 31, 32, 33],

[40, 41, 42, 43]])

b[2,3] # Вторая строка, третий столбец

b[(2,3)]

b[2][3]

b[:,2] # Третий столбец

b[:2] # Первые две строки

b[1:3, : : ] # Вторая и третья строки

Когда индексов меньше, чем осей, отсутствующие индексы предполагаются дополненными с помощью срезов:

b[-1] # Последняя строка. Эквивалентно b[-1,:]

b[i] можно читать как b[i, <столько символов ':', сколько нужно>]. В NumPy это также может быть записано с помощью точек, как b[i, ...].

Например, если x имеет ранг 5 (то есть у него 5 осей), тогда

* x[1, 2, ...] эквивалентно x[1, 2, :, :, :],
* x[... , 3] то же самое, что x[:, :, :, :, 3] и
* x[4, ... , 5, :] это x[4, :, :, 5, :].

# Манипуляции с формой

Как уже говорилось, у массива есть форма (shape), определяемая числом элементов вдоль каждой оси:

a = np.array(([[0, 1, 2], [10, 12, 13]], [[100, 101, 102], [110, 112, 113]]))

a.shape

(2, 2, 3)

Форма массива может быть изменена с помощью различных команд:

a.ravel() # Делает массив плоским

a.transpose() # Транспонирование

a.reshape((3, 4)) # Изменение формы

Порядок элементов в массиве в результате функции ravel() соответствует обычному "C-стилю", то есть, чем правее индекс, тем он "быстрее изменяется": за элементом a[0,0] следует a[0,1]. Если одна форма массива была изменена на другую, массив переформировывается также в "C-стиле". Функции ravel() и reshape() также могут работать (при использовании дополнительного аргумента) в FORTRAN-стиле, в котором быстрее изменяется более левый индекс.

a.reshape((3, 4), order='F')

Метод reshape() возвращает ее аргумент с измененной формой, в то время как метод resize() изменяет сам массив:

a.resize((2, 6))

a.reshape((3, -1))