Тема работы: «Разработка, отладка и испытание программ, работающих со списками больших объемов»

1 Цель работы

Научить создавать и обрабатывать списки больших объемов данных с помощью библиотеки NumPy.

2 Задание

Выполните следующие действия над массивами с помощью модуля NumPy:

- ✓ преобразовать список в массив;
- ✓ преобразовать массив в список;
- ✓ найти количество различных элементов массива;

3 Оснащение работы

Задание по варианту, ЭВМ, среда разработки **Python 3.7, IDLE**.

4 Основные теоретические сведения

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.

Установка NumPy

На linux – пакет python3-numpy (или аналогичный для вашей системы), или через pip. Ну или же собирать из исходников https://sourceforge.net/projects/numpy/files/NumPy/.

На Windows на том же сайте есть exe установщики. Или, если возникают проблемы, рекомендую ещё хороший сборник библиотек http://www.lfd.uci.edu/~gohlke/pythonlibs/#numpy.

Начинаем работу

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.

ndarray.data – буфер, содержащий фактические элементы массива. Обычно не нужно использовать этот атрибут, так как обращаться к элементам массива проще всего с помощью индексов.

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() (запомните: array – функция, создающая объект типа ndarray):

```
>>>
>>> import numpy as np
>>> a = np.array([1, 2, 3])
>>> a
array([1, 2, 3])
>>> type(a)
<class 'numpy.ndarray'>
```

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 в момент создания:

```
>>>
>>> b = np.array([[1.5, 2, 3], [4, 5, 6]], dtype=np.complex)
>>> b
array([[ 1.5+0.j,  2.0+0.j,  3.0+0.j],
       [ 4.0+0.j,  5.0+0.j,  6.0+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))
array([[ 0.,  0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.,  0.]])
>>> np.ones((2, 2, 2))
array([[[ 1.,  1.],
        [ 1.,  1.]],

       [[ 1.,  1.],
        [ 1.,  1.]]])
```

Функция eye() создаёт единичную матрицу (двумерный массив)

```
>>>
>>> np.eye(5)
array([[ 1.,  0.,  0.,  0.,  0.],
       [ 0.,  1.,  0.,  0.,  0.],
       [ 0.,  0.,  1.,  0.,  0.],
       [ 0.,  0.,  0.,  1.,  0.],
       [ 0.,  0.,  0.,  0.,  1.]])
```

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))
array([[  6.93920488e-310,   6.93920488e-310,   6.93920149e-310],
       [  6.93920058e-310,   6.93920058e-310,   6.93920058e-310],
       [  6.93920359e-310,   0.00000000e+000,   6.93920501e-310]])
>>> np.empty((3, 3))
array([[  6.93920488e-310,   6.93920488e-310,   6.93920147e-310],
       [  6.93920149e-310,   6.93920146e-310,   6.93920359e-310],
       [  6.93920359e-310,   0.00000000e+000,   3.95252517e-322]])
```

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)
array([ 0. ,  0.1,  0.2,  0.3,  0.4,  0.5,  0.6,  0.7,  0.8,  0.9])
```

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
array([  0. ,  0.25,  0.5 ,  0.75, 1. , 1.25, 1.5 , 1.75, 2. ])
```

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.]])
>>> np.fromfunction(f1, (3, 3))
array([[ 0.,  1.,  2.],
       [ 3.,  4.,  5.],
       [ 6.,  7.,  8.]])
```

Печать массивов

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

```
>>>
>>> print(np.arange(0, 3000, 1))
[   0    1    2 ..., 2997 2998 2999]
```

Если вам действительно нужно увидеть весь массив, используйте функцию numpy.set_printoptions:

```
np.set_printoptions(threshold=np.nan)
```

И вообще, с помощью этой функции можно настроить печать массивов «под себя». Функция numpy.set_printoptions принимает несколько аргументов:

precision: количество отображаемых цифр после запятой (по умолчанию 8).

threshold: количество элементов в массиве, вызывающее обрезание элементов (по умолчанию 1000).

edgeitems: количество элементов в начале и в конце каждой размерности массива (по умолчанию 3).

linewidth: количество символов в строке, после которых осуществляется перенос (по умолчанию 75).

suppress: если True, не печатает маленькие значения в scientific notation (по умолчанию False).

nanstr: строковое представление NaN (по умолчанию 'nan').

infstr: строковое представление inf (по умолчанию 'inf').

formatter: позволяет более тонко управлять печатью массивов.

Базовые операции

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

```
>>>
>>> import numpy as np
>>> a = np.array([20, 30, 40, 50])
>>> b = np.arange(4)
>>> a + b
array([20, 31, 42, 53])
>>> a - b
array([20, 29, 38, 47])
>>> a * b
array([  0,  30,  80, 150])
>>> a / b  # При делении на 0 возвращается inf (бесконечность)
array([       inf,  30.       ,  20.       ,  16.66666667])
<string>:1: RuntimeWarning: divide by zero encountered in true_divide
>>> a ** b
array([     1,     30,   1600, 125000])
>>> a % b  # При взятии остатка от деления на 0 возвращается 0
<string>:1: RuntimeWarning: divide by zero encountered in remainder
array([0, 0, 0, 2])
```

Для этого, естественно, 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
Traceback (most recent call last):
  File "<input>", line 1, in <module>
ValueError: operands could not be broadcast together with shapes (2,3) (3,2)
```

Также можно производить математические операции между массивом и числом. В этом случае к каждому элементу прибавляется (или что вы там делаете) это число.

```
>>>
>>> a + 1
array([21, 31, 41, 51])
>>> a ** 3
array([  8000,  27000,  64000, 125000])
>>> a < 35  # И фильтрацию можно проводить
array([ True,  True, False, False], dtype=bool)
```

NumPy также предоставляет множество математических операций для обработки массивов:

```
>>>
>>> np.cos(a)
array([ 0.40808206,  0.15425145, -0.66693806,  0.96496603])
>>> np.arctan(a)
array([ 1.52083793,  1.53747533,  1.54580153,  1.55079899])
>>> np.sinh(a)
```

```
array([  2.42582598e+08,   5.34323729e+12,   1.17692633e+17,
         2.59235276e+21])
```

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)
21
>>> a.sum()
21
>>> a.min()
1
>>> a.max()
6
```

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
array([1, 2, 3])
>>> a.min(axis=1)  # Наименьшее число в каждой строке
array([1, 4])
```

5 Порядок выполнения работы

1. Выделить ключевые моменты задачи.
2. Построить алгоритм решения задачи.
3. Запрограммировать полученный алгоритм.
4. Провести тестирование полученной программы.

6 Форма отчета о работе

Лабораторная работа № ___
Номер учебной группы _____
Фамилия, инициалы учащегося: _____
Дата выполнения работы: _____
Тема работы: _____
Цель работы: _____
Оснащение работы: _____
Результат выполнения работы: _____
_____

7 Контрольные вопросы и задания

1. Назовите назначение модуля NymPy.

2. Опишите функции для работы с массивами.
3. Опишите математические операции над массивами.
4. Перечислите способы создания массивов?

8 Рекомендуемая литература

Плас, Дж. В. Python для сложных задач. Наука о данных и машинное обучение / Дж.В. Плас. – СПб: Питер, 2018.

Прохоренок, Н.А. Python 3. Самое необходимое / Н.А Прохоренок, В.А. Дронов – СПб.: БВХ-Петербург, 2016.