Теоретическая часть.

Модуль NumPy - одна из основных причин популярности Python среди ученых (http://www.numpy.org/). Динамические языки вроде Python зачастую медленнее компилирующих языков вроде C или даже других интерпретируемых языков вроде Java. NumPy был написан для предоставления доступа к быстрым многомерным массивам по аналогии с научными языками вроде FORTRAN. Вы получаете скорость C и дружественность к разработчикам Python.

Для того чтобы начать работу с NumPy, вы должны понять устройство основной структуры данных, многомерного массива ndarray (от N-dimensional array — «N-мерный массив») или просто array. В отличие от списков и кортежей в Python, все элементы должны иметь одинаковый тип.

NumPy называет количество измерений массива его рангом. Одномерный массив похож на ряд значений, двухмерный — на таблицу с рядами и колонками, а трехмерный — на кубик Рубика. Длина измерений может не быть одинаковой

ПРИМЕЧАНИЕ:

Array в NumPy и array в Python — это не одно и то же. В дальнейшем в этойлабораторной работемы будем работать только с массивами NumPy.

Но зачем нам нужны массивы?

- Научные данные зачастую представляют собой большие последовательности.
- Научные подсчеты для таких данных часто выполняются с использованием матричной математики, регрессии, симуляции и других приемов, которые обрабатывают множество фрагментов данных одновременно.
- NumPy обрабатывает массивы гораздо быстрее, чем стандартные списки или кортежи Python.

Существует множество способов создать массив NumPy.

```
>>> import numpy as np
>>> b = np.array([2, 4, 6, 8])
>>> b
array([2, 4, 6, 8])
```

Атрибут ndim возвращает ранг массива:

```
>>> b.ndim
1
```

Общее число значений можно получить с помощью атрибута size:

```
>>> b.size
4
```

Количество значений каждого ранга возвращает атрибут shape:

```
>>> b.shape
(4,)
```

Оборудование и материалы.

Персональный компьютер, среда разработки Python.

Указания по технике безопасности:

Соответствуют технике безопасности по работе с компьютерной техникой.

Задания

Создание массива с помощью функции arange()

Метод arange() похож на стандартный метод range(). Если вы вызовете метод arange(), передав ему один целочисленный аргумент num, он вернет ndarray от 0 до num-1:

```
>>> import numpy as np
>>> a = np.arange(10)
>>> a
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
>>> a.ndim
1
>>> a.shape
(10,)
>>> a.size
10
```

С помощью двух значений он создаст массив от первого элемента до последнего минус один:

```
>>> a = np.arange(7, 11)
>>> a
array([ 7,  8,  9, 10])
```

Вы также можете передать как третий параметр размер шага, который будет использован вместо единицы:

```
>>> a = np.arange(7, 11, 2)
>>> a
array([7, 9])
```

До сих пор мы показывали примеры лишь с целыми числами, но метод arrange() работает и с числами с плавающей точкой:

```
>>> f = np.arange(2.0, 9.8, 0.3)
>>> f
array([ 2. ,  2.3,  2.6,  2.9,  3.2,  3.5,  3.8,  4.1,  4.4,  4.7,  5. ,
        5.3,  5.6,  5.9,  6.2,  6.5,  6.8,  7.1,  7.4,  7.7,  8. ,  8.3,
        8.6,  8.9,  9.2,  9.5,  9.8])
```

И последний прием: аргумент dtype указывает функции arrange(), какого типа значения следует создать:

```
>>> g = np.arange(10, 4, -1.5, dtype=np.float)
>>> g
array([ 10. ,   8.5,   7. ,   5.5])
```

Создание массива с помощью функций zeros(), ones() и random()

Метод zeros() возвращает массив, все значения которого равны 0. В эту функцию вам нужно передать аргумент, в котором будет указана желаемая форма массива.

Так создается одномерный массив:

```
>>> a = np.zeros((3,))
>>> a
array([ 0.,  0.,  0.])
>>> a.ndim
1
>>> a.shape
(3,)
>>> a.size
3
```

Этот массив имеет ранг 2:

```
>>> b = np.zeros((2, 4))
>>> b
array([[ 0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.]])
>>> b.ndim
2
>>> b.shape
(2, 4)
>>> b.size
8
```

Другой особой функцией, заполняющей массив одинаковыми значениями, является ones():

```
>>> import numpy as np
>>> k = np.ones((3, 5))
>>> k
array([[ 1.,  1.,  1.,  1.,  1.],
       [ 1.,  1.,  1.,  1.,  1.],
       [ 1.,  1.,  1.,  1.,  1.]])
```

Последняя функция создает массив и заполняет его случайными значениями из промежутка от 0,0 до 1,0:

```
>>> m = np.random.random((3, 5))
>>> m
```

```
array([[  1.92415699e-01,   4.43131404e-01,   7.99226773e-01,
          1.14301942e-01,   2.85383430e-04],
       [  6.53705749e-01,   7.48034559e-01,   4.49463241e-01,
          4.87906915e-01,   9.34341118e-01],
       [  9.47575562e-01,   2.21152583e-01,   2.49031209e-01,
          3.46190961e-01,   8.94842676e-01]])
```

Изменяем форму массива с помощью метода reshape()

До этого момента массив не особо отличался от списка или кортежа. Одним из различий между ними является возможность изменять его форму с помощью функции reshape():

```
>>> a = np.arange(10)
>>> a
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
>>> a = a.reshape(2, 5)
>>> a
array([[0, 1, 2, 3, 4],
       [5, 6, 7, 8, 9]])
>>> a.ndim
2
>>> a.shape
(2, 5)
>>> a.size
10
```

Вы можете изменять форму массива разными способами:

```
>>> a = a.reshape(5, 2)
>>> a
array([[0, 1],
       [2, 3],
       [4, 5],
       [6, 7],
       [8, 9]])
>>> a.ndim
2
>>> a.shape
(5, 2)
>>> a.size
10
```

Присваиваем кортеж, указывающий параметры формы, атрибуту shape:

```
>>> a.shape = (2, 5)
>>> a
array([[0, 1, 2, 3, 4],
       [5, 6, 7, 8, 9]])
```

Единственное ограничение — произведение рангов должно быть равным количеству значений (в нашем случае 10):

```
>>> a = a.reshape(3, 4)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: total size of new array must be unchanged
```

Получаем элемент с помощью конструкции []

Одномерный массив работает как список:

```
>>> a = np.arange(10)
>>> a[7]
7
>>> a[-1]
9
```

Но если массив имеет другую форму, используйте индексы, разделенные запятыми:

```
>>> a.shape = (2, 5)
>>> a
array([[0, 1, 2, 3, 4],
       [5, 6, 7, 8, 9]])
>>> a[1,2]
7
```

Это отличается от двухмерного списка:

```
>>> l = [ [0, 1, 2, 3, 4], [5, 6, 7, 8, 9] ]
>>> l
[[0, 1, 2, 3, 4], [5, 6, 7, 8, 9]]
>>> l[1,2]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: list indices must be integers, not tuple
>>> l[1][2]
7
```

Еще один момент: разбиение работает, но опять же только внутри множества, заключенного в один набор квадратных скобок. Снова создадим привычный проверочный

массив:

```
>>> a = np.arange(10)
>>> a = a.reshape(2, 5)
>>> a
array([[0, 1, 2, 3, 4],
       [5, 6, 7, 8, 9]])
```

Используйте разбиение, чтобы получить первый ряд — элементы начиная со смещения 2, до конца:

```
>>> a[0, 2:]
array([2, 3, 4])
```

Теперь получим последний ряд — все элементы вплоть до третьего с конца:

```
>>> a[-1, :3]
array([5, 6, 7])
```

Вы также можете присвоить значение более чем одному элементу с помощью разбиения. Следующее выражение присваивает значение 1000 колонкам (смещениям) 2 и 3 каждого ряда:

```
>>> a[:, 2:4] = 1000
>>> a
array([[   0,    1, 1000, 1000,    4],
       [   5,    6, 1000, 1000,    9]])
```

Математика массивов

Создание и изменение формы массивов так нас увлекли, что мы почти забыли сделать с ними что-то более полезное. Для начала используем переопределенный в NumPy оператор умножения (*), чтобы умножить все значения массива за раз:

```
>>> from numpy import *
>>> a = arange(4)
>>> a
array([0, 1, 2, 3])
>>> a *= 3
>>> a
array([0, 3, 6, 9])
```

Если вы пытались умножить каждый элемент обычного списка Python на число, вам бы понадобились цикл или включение:

```
>>> plain_list = list(range(4))
>>> plain_list
[0, 1, 2, 3]
>>> plain_list = [num * 3 for num in plain_list]
>>> plain_list
[0, 3, 6, 9]
```

Такое поведение применимо также к сложению, вычитанию, делению и другим функциям библиотеки NumPy. Например, вы можете инициализировать все элементы массива любым значением с помощью функции zeros() и оператора сложения:

```
>>> from numpy import *
>>> a = zeros((2, 5)) + 17.0
>>> a
array([[ 17.,  17.,  17.,  17.,  17.],
       [ 17.,  17.,  17.,  17.,  17.]])
```

Линейная алгебра

NumPy содержит множество функций линейной алгебры. Например, определим такую систему линейных уравнений:

$$4x + 5y = 20$$
$$x + 2y = 13$$

Как мы можем найти x и y? Создадим два массива:

- коэффициенты (множители для x и y);
- зависимые переменные (правая часть уравнения):

```
>>> import numpy as np
>>> coefficients = np.array([ [4, 5], [1, 2] ])
>>> dependents = np.array([20, 13])
```

Теперь используем функцию solve() модуля linalg:

```
>>> answers = np.linalg.solve(coefficients, dependents)
>>> answers
array([ -8.33333333,  10.66666667])
```

В результате получим, что x примерно равен –8.3, а y примерно равен 10.6. Являются ли эти числа решениями уравнения?

```
>>> 4 * answers[0] + 5 * answers[1]
20.0
>>> 1 * answers[0] + 2 * answers[1]
13.0
```

Так и есть. Для того чтобы напечатать меньше текста, вы также можете указать NumPy найти скалярное произведение массивов:

```
>>> product = np.dot(coefficients, answers)
>>> product
array([ 20.,  13.])
```

Если решение верно, значения массива product должны быть близки к значениям массива dependents. Вы можете использовать функцию allclose(), чтобы проверить, являются ли массивы хотя бы приблизительно равными (они могут быть не полностью равными из-за округления чисел с плавающей точкой):

```
>>> np.allclose(product, dependents)
True
```

NumPy также имеет модули для работы с многочленами, преобразованиями Фурье, статистикой и распределением вероятностей.

Контрольные вопросы

1. Какими командами создаются массивы? Какие атрибуты у этих команд?
2. Как изменить форму массива?
3. Как получить элемент массива?
4. Какие операции линейной алгебры реализует NumPy?

Список литературы, рекомендуемый к использованию по данной теме:

1. Орлов, С. А. Программная инженерия. Технологии разработки программного обеспечения : учебник / С.А. Орлов. - 5-е изд., обновл. и доп. - СПб. : Питер, 2017. - 640 с.
2. Гагарина, Л. Г. Современные проблемы информатики и вычислительной техники : [учеб. пособие] / Л.Г. Гагарина, А.А. Петров. - М. : Форум, 2016. - 368 с.
3. Шкляр, М. Ф. Основы научных исследований : учеб. пособие / М.Ф. Шкляр. - 6-е изд. - М. : Дашков и Ко, 2016. - 208 с.
4. Михеева, Е. В. Информационные технологии в профессиональной деятельности : учеб. пособие / Е.В. Михеева. - 14-е изд., стер. - М. : Академия, 2016. - 384 с.
5. Любанович Билл. Простой Python. Современный стиль программирования. — СПб.: Питер, 2016. — 480 с.