ЛАБОРАТОРНАЯ РАБОТА №1
РАБОТА С МАССИВАМИ В NumPy И ПОСТРОЕНИЕ ГРАФИКОВ В Matplotlib

ОСНОВЫ

NumPy – это специальная библиотека для работы с многомерными массивами. Как правило это матрицы элементов (в основном числа) одного типа, где для индексации используются целочисленные кортежи. Например, координаты точки в трехмерном пространстве [1, 2, 1] имеют только одну ось. Эта ось имеет три элемента, следовательно, его длина равняется 3. В примере ниже массив имеет две оси. Первая ось имеет размер равный 2, а вторая трем.

```
In [1]: [[ 1., 0., 0.],
   ...:  [ 0., 1., 2.]]
```

Массивы в *NumPy* – это класс *ndarray*. Их так же называют псевдомассивами. Необходимо помнить, что массивы в *NumPy* не относятся к классу *array.array*, который поддерживает только одномерные массивы и обладает меньшей функциональностью. Основные атрибуты *ndarray*:

ndarray.ndim

Размерность массива (количество осей)

ndarray.shape

Размер массива по всем осям, возвращается как кортеж. Для матрицы с *n* строками и *m* столбцами, атрибут *shape* будет *(n, m)*. Длина атрибута *shape* всегда будет количество осей (размерность), т.е. атрибут *ndim*.

ndarray.size

Общее количество элементов в массиве. Данный атрибут будет равносилен произведению всех элементов в атрибуте *shape*

ndarray.dtype

Возвращает тип элементов в массиве. Можно создать как свой собственный тип, а так же использовать встроенные типы Python. Так же можно использовать специальные типы библиотеки, например, *numpy.int32*, *numpy.int16* и *numpy.float64*

ndarray.itemsize

Размер в байтах для каждого элемента в массиве. Например, для массива содержащего элементы типа *float64* атрибут *itemsize* будет 8 (=64/8).

ndarray.data

Буфер, содержащий текущие значения массива. Обычно, этот атрибут не используется, т.к. можно извлечь необходимый элемент, используя индексацию массива.

Пример создания матрицы размером 3 на 5, содержащую числа от 1 до 15.

```
import numpy as np #импорт библиотеки
a = np.arange(15).reshape(3, 5)
b = np.array([(1.5,2,3), (4,5,6)])
```

Задание 1. Используя приведенный пример, создайте матрицу *a*, найдите ее размерность, тип данных, размер элемента в матрице, размер матрицы, количество элементов. Создайте массив *b*, содержащий одну строку [6, 7, 8].

СОЗДАНИЕ МАССИВОВ

Есть несколько способов для создания массивов.

Например, можно создать массив, используя стандартные типы в Python такие как лист или кортеж, используя функцию array.

```
In [6]: import numpy as np

In [7]: a = np.array([2,3,4])

In [8]: a
Out[8]: array([2, 3, 4])

In [9]: a.dtype
Out[9]: dtype('int32')

In [10]: b = np.array([1.2, 3.5, 5.1])

In [11]: b.dtype
Out[11]: dtype('float64')
```

Функция *array* преобразовывает последовательность из последовательностей в двухмерный массив и т.д.

```
In [14]: b = np.array([(1.5,2,3), (4,5,6)])

In [15]: b
Out[15]:
array([[1.5, 2. , 3. ],
       [4. , 5. , 6. ]])
```

Тип данных в массиве можно задать при его создании.

```
In [18]: c = np.array( [ [1,2], [3,4] ], dtype=complex )

In [19]: c
Out[19]:
array([[1.+0.j, 2.+0.j],
       [3.+0.j, 4.+0.j]])
```

Чаще всего элементы массива неизвестны, однако чаще всего известен их тип. Поэтому в *NumPy* есть ряд функций для создания массивов с заранее определенными значениями по умолчанию. Для создания матрицы содержащей нулевые значения, используется функция *zeros*.

```
a = np.zeros((3,4))
b = np.zeros((3,4), dtype = np.int16 )
```

Аналогично, используя функцию *ones*, можно создать матрицу, содержащую единицы.

Для создания последовательности из чисел в *NumPy* есть функция *range*, которая возвращает массив, а не список.

```
c = np.arange( 10, 30, 5 )
print(c)
d = np.arange( 0, 2, 0.3 )
print(d)
```

На практике при работе с числами с плавающей точкой чаще используется функция *linspace*, которая в качестве аргументов получает количество элементов, которое необходимо, вместо шага.

```
from numpy import pi
e = np.linspace( 0, 2, 9 )
x = np.linspace( 0, 2*pi, 100 )
f = np.sin(x)
print(e, x, f)
```

При выводе массива на печать, *NumPy* использует следующий шаблон:

1) Значения выводятся слева направо и сверху вниз
2) Для многомерных массивов каждый срез выводится отдельно

Для задания способа вывода используется функция *set_printoptions* библиотеки *NumPy*.

Задание 2. Наберите код, приведенный в разделе «Создание массивов» в отдельном файле “myArray.py”. Выведите все массивы на экран, сравните вывод двухмерных и трехмерных массивов. Самостоятельно изучите работу функций *array*, *zeros*, *zeros_like*, *ones*, *ones_like*, *empty*, *empty_like*, *arange*, *linspace*, *numpy.random.rand*, *numpy.random.randn*, *fromfunction*, *fromfile*.

БАЗОВЫЕ ОПЕРАЦИИ

Арифметические операции над массивами выполняются поэлементно. При этом создается новый массив, который заполняется вычисленными значениями.

```
a = np.array( [20,30,40,50] )
b = np.arange(4)
c = a-b
d = b**2 #возведение в степень
f = 10*np.sin(a)
k = a<35
```

По умолчанию оператор * выполняет поэлементное произведение элементов матрицы. Для выполнения произведения матриц согласно правилам линейной алгебры, используется функция *dot*.

```
A = np.array( [[1,1], [0,1]] )
B = np.array( [[2,0], [3,4]] )
D = A * B
print('D = ',D)
P = np.dot(A, B)
print('P = ', P)
```

Такие операции как += и *= не создают новый массив, а изменяют исходный. При использовании этих операторов данные в массивах должны быть одного типа.

Если вычисления происходят с массивами с разными типами данных, тип данных в результирующем массиве будет более точный из двух.

Некоторые унарные операции такие как вычисление суммы (*sum*), кумулятивно суммы (*cumsum*), минимума (*min*) или максимума (*max*) элементов в массиве, реализованы как методы объектов. По умолчанию эти функции работают со всем массивом, однако их можно использовать для каждой отдельной оси.

```
b = np.arange(12).reshape(3,4)
print(b)
s1 = b.sum(axis=0) #сумма по столбцам
print(s1)
s2 = b.sum(axis=1) #сумма по строкам
print(s2)
```

В *NumPy* реализованы математические функции такие как *sin*, *cos* и *exp*. В *NumPy* они называются универсальные функции. Эти функции поэлементно выполняют операции над элементами массива, возвращая другой массив.

Задание 3. Реализуйте код приведенный в примерах. Для матрицы b найдите максимальный элемент по всей матрице, по строкам, по столбцам.

РАБОТА С ИНДЕКСАМИ

Пример работы с массивами, используя индексы и срезы.

```
a = np.arange(10)**3
print(a[2]) #третий элемент
print(a[2:5]) #'элементы с третьего по пятый
a[:6:2] = -1000 # равнсильно a[0:6:2] = -1000
#начинаем с шестого элемента, изменяя каждый второй объект
print(a)
print(a[ : :-1]) #распечатываем массив в обратном порядке
for i in a: #цикл по массиву a
    print(i**(1/3.)) #возводим каждый элемент в 1/3 степень
```

Многомерные массивы могут иметь только один индекс для каждой оси. Эти индексы задаются как кортеж, перечисляемые через запятую.

```
b = np.arange(20, dtype = float).reshape(5, 4)
print(b[2,3])
print(b[0:5, 1])
print(b[ : ,1])
print(b[1:3, : ])
```

Если количество, используемых индексов, меньше чем размерность массива, то пропущенные индексы дополняются срезами.

```
print(b[-1]) # равносильно b[-1,:]
```

Выражение внутри скобок *b[i]* воспринимается как будто после *i*, перечислено столько срезов сколько необходимо для описания оставшихся осей. Вместо *:*, в *NumPy* можно так же использовать

троеточие (...) *b[i, ...]*. Троеточие обозначает, что необходимо добавить столько индексов сколько необходимо добавить для описания всех осей массива. Например, если у массива x пять элементов, то:

1) *x[1,2,...]* эквивалентно *x[1,2,:,:,:]*
2) *x[...,3]* равносильно *x[:,:,:,:,3]*
3) *x[4,...,5,:]* соответствует *x[4,:,:,5,:]*

```
#создаем трехмерный массив
c = np.array( [[[  0,  1,  2],
          [ 10, 12, 13]],
         [[100,101,102],
          [110,112,113]]])
print(c)
print(c[1,...]) #равносильно c[1,:,:] или c[1]
print(c[...,2]) #так же как c[:,:,2]
```

Итерации начинаются с первого индекса.

```
for row in b:
    print(row)
```

Однако, если необходимо операции над каждым элементом, необходимо использовать атрибут *flat*.

```
for element in b.flat:
    print(element)
```

Задание 4. Наберите код, приведенный в примерах.

ПРЕОБРАЗОВАНИЕ ФОРМЫ МАТРИЦ

Изменение формы массива. Массив имеет форму, задаваемую количеством элементов по каждой оси.

```
a = np.floor(10*np.random.random((3,4)))
print(a)
print(a.shape)
```

Для изменения формы существуют различные команды. Приведенные ниже три команды возвращают измененный массив, но не изменяют исходный массив.

```
print(a.ravel())
print(a.reshape(6, 2))
print(a.T)
print(a.T.shape)
print(a.shape)
```

Функция *reshape* возвращает другой массив с измененной формой, метод *ndarray.resize* модифицирует сам массив. Если в качестве индекса указано значение -1, то размерность вычисляется автоматически.

```
a = np.floor(10*np.random.random((3,4)))
print(a)
a.resize((2,6))
print(a)
a.reshape(3,-1)
print(a)
b = a.reshape(3,-1)
print(b)
```

Задание 5. Реализуйте код, приведенный в примере. Изучите работу функции *ndarray.shape, reshape, resize, ravel*.

ОБЪЕДИНЕНИЕ И РАЗБИЕНИЕ МАССИВОВ

Несколько массивов могут быть объединены вместе вдоль одной из размерностей.

```
a = np.floor(10*np.random.random((2,2)))
b = np.floor(10*np.random.random((2,2)))
v = np.vstack((a,b))
h = np.hstack((a,b))
print(v)
print(h)
```

Функция *column_stack* объединяет два одномерных массива в один двухмерный массив. Эта функция эквивалентна *hstack*, но только для двухмерных массивов. Функция *newaxis* добавляет еще одну размерность к массиву.

```
from numpy import newaxis
d = np.column_stack((a,b))
print(d)
a = np.array([4.,2.])
b = np.array([3.,8.])
d = np.column_stack((a,b))
print(d)
d = np.hstack((a,b))
print(a[:,newaxis])
print(np.column_stack((a[:,newaxis], b[:,newaxis])))
print(np.hstack((a[:,newaxis], b[:,newaxis])))
```

В целом, для массивов с размерностью больше, чем два, функция *hstack*, объединяет массивы вдоль второй размерности. Функция *vstack* вдоль первой размерности. Функция *concatenate* позволяет указывать в качестве аргументов ось, по которой будет происходить объединение.

Используя функцию *hsplit*, можно разделить массив вдоль горизонтальной оси, при этом необходимо указать то число, на которое должен быть разбит массив, либо указать номера колонок, после которых необходимо выполнить разбиение.

```
a = np.floor(10*np.random.random((2,12)))
d1 = np.hsplit(a,3) #разбивает на три массива
print(d)
d2 = np.hsplit(a,(3,4)) #разбивает массив
#после 3-ей и 4-ой строчки
```

Функция *vsplit* разделяет массив по вертикали, для указания по какой оси необходимо разбивать массив используется функция *array_split*.

Задание 6. Реализуйте код, приведенный в примере. Изучите работу функций: *hstack, vstack, column_stack, concatenate, c_, r_*.

КОПИРОВАНИЕ МАССИВОВ

При копировании массивов данные в них иногда копируются в новый массив, а иногда нет. В *NumPy* существует три способа для копирования массивов. Простое присваивание не копирует массив и данные в нем. При передаче массива в функцию не создается его копия.

```
a = np.arange(12)
b = a
print(b is a) #проверяем, что b -это a
b.shape = 3,4
print(b.shape)
print(a.shape)
def f(x):
    print(id(x))
print(id(a))
f(a)
```

Различные объекты массива могут использовать одни и те же данные. Метод *view* создает новый массив, который использует те же данные, что исходный объект. Срезы работают так же как метод *view*.

```
c = a.view()
print(c is a)
print(c.base is a) #c содержит данные наследованные из a
print(c.flags.owndata)
c.shape = 2,6 #изменяем форму c
print(a.shape) #a не изменилось
c[0,4] = 1234 #но, изменяя данные в a
print(a) #изменяются данные в a
```

Для того чтобы скопировать данные необходимо использовать функцию *copy*.

```
d = a.copy()
print(d is a)
print(d.base is a)
d[0,0] = 9999
print(a)
```

Задание 7. Реализуйте код, написанный в примерах, закомментируйте строки без комментариев.

ИНДЕКСАЦИЯ ЧЕРЕЗ МАССИВЫ

Для индексации массивов могут использоваться массивы целых чисел и булевых переменных. Если индексированный массив используется для многомерных массивов, то индексы применяются только по первой размерности. Массивы индексов можно создавать для нескольких

```
a = np.arange(12)**2 #числа в квадрате
i = np.array([1,1,3,8,5]) #массив индексов
print(a[i])
j = np.array([[3,4],[9,7]]) #двухмерный массив индексов
print(a[j])
```

Если для индексации используются булевые значения, в этом случае мы явно задаем, какие массивы используем, а какие нет. Наиболее общим образом использованием булевых индексов является создание массива такого же размера как исходный массив.

```
a = np.arange(12).reshape(3,4)
b = a > 4
print(b)
print(a[b])
```

Задание 8. Реализуйте код, написанный в примерах.

ПЕРВЫЙ ГРАФИК

Matplotlib.pyplot – это набор функций, которые заставляют работать *matplotlib* так же как MATLAB. Каждая команда *pyplot* вносит какие-то изменения на окно: создает новое окно, создает область для построения графиков, строит графики, добавляет подписи и т.д.

Выполнить построение графика с *pyplot* очень просто: здесь строится график со значениями от 1 до 4 по оси y и от 0 до 3 по оси *x*.

```
import matplotlib.pyplot as plt
plt.plot([1, 2, 3, 4])
plt.ylabel('some numbers')
plt.show()
```

Если в *plot* передан массив или список, *matplotlib* воспринимает как набор значений по оси *y* и автоматически создает набор значений оп оси *x*. Поскольку в *Python* нумерация начинается с 0, то по определению *x* будет такой же размерности как *y* и содержать значения [0, 1, 2, 3].

pyplot – разносторонняя команда, у которой много аргументов. Для построения зависимости между массивами в явном виде, можно использовать такой способ:

```
plt.plot([1, 2, 3, 4], [1, 4, 9, 16])
```

Задание 10. Реализуйте код, приведенный в примерах.

ЗАДАНИЕ СТИЛЯ ГРАФИКА

Для каждой пары *(x, y)* есть так же третий аргумент в виде строки, который задает цвет и тип линии графика, которая задается так же как в *MATLAB*. По определению это значение ‘*b*-’ – голубая сплошная линия. В примере показано задание красных круглых маркеров для построения графика.

```
plt.plot([1, 2, 3, 4], [1, 4, 9, 16], 'ro')
plt.axis([0, 6, 0, 20])
plt.show()
```

Команда *axis()* принимает список *[xmin, xmax, ymin, ymax]* и задает вид окна просмотра.

По умолчанию все последовательности, которые передаются в функции *matplotlib*, по умолчанию преобразовываются в массивы *NumPy*.

```
import numpy as np
t = np.arange(0., 5., 0.2)
plt.plot(t, t, 'r--', t, t**2, 'bs', t, t**3, 'g^')
plt.show()
```

Задание 11. Реализуйте код, приведенный в примерах. Напишите комментарии с тем, что реализовано в каждой строчке.

ПОСТРОЕНИЕ РАЗЛИЧНЫХ ТИПОВ ГРАФИКОВ

В Python есть некоторые объекты, где доступ к элементам осуществляется через строки. Например, в *numpy.recarray* и *pandas.DataFrame*. *Matplotlib* позволяет сопровождать объекты используя ключи. Если необходимо то, затем можно подписать эти данные.

```
data = {'a': np.arange(50),
        'c': np.random.randint(0, 50, 50),
        'd': np.random.randn(50)}
data['b'] = data['a'] + 10 * np.random.randn(5
data['d'] = np.abs(data['d']) * 100

plt.scatter('a', 'b', c='c', s='d', data=data)
plt.xlabel('entry a')
plt.ylabel('entry b')
plt.show()
```

Так же можно создавать графики с категориями объектов.

```
names = ['group_a', 'group_b', 'group_c']
values = [1, 10, 100]
plt.figure(1, figsize=(9, 3))
plt.subplot(131)
plt.bar(names, values)
plt.subplot(132)
plt.scatter(names, values)
plt.subplot(133)
plt.plot(names, values)
plt.suptitle('Categorical Plotting')
plt.show()
```

Задание 12. Реализуйте код, приведенный в примерах.

УПРАВЛЕНИЕ СВОЙСТВАМИ ЛИНИЙ

У линий есть множество атрибутов, которые можно задавать: ширина линии, тип штриха, гладкость (*matplotlib.lines.Line2D*). Есть несколько способов для задания этих свойств.

1) Используя именованные аргументы
```
x = np.arange(0., 5., 0.2)
y = x**2
plt.plot(x, y, linewidth=2.0)
```
2) Используя сеттер экземпляра объекта *Line2D*. plot возвращает список объектов типа *Line2D*, т.е. *line1, line2 = plot(x1, y1, x2, y2)*. Для того, чтобы взять первый элемент из объекта *line* его необходимо рассматривать как кортеж.
```
line, = plt.plot(x, y, '-')
# отключить сглаживание
line.set_antialiased(False)
```
3) Использовать команду *setp()*, которая работает со списком или с одним объектом. В этой функции можно использовать как ключевые слова *Python* или *MATLAB*-стиль.
```
x1 = np.arange(0., 5., 0.2)
y1 = x1**(0.5)
x2 = np.arange(1., 5., 0.1)
y2 = x2**(2)
lines = plt.plot(x1, y1, x2, y2)
# используя именованные аргументы
plt.setp(lines, color='r', linewidth=2.0)
# Или в стиле MATLAB
plt.setp(lines, 'color', 'r', 'linewidth', 2.0)
```

Для того, чтобы получить список свойств линии, необходимо вызвать функцию *setp()* от *line* или *lines*.

Задание 13. Реализуйте код, приведенный в примерах.

РАБОТА С НЕСКОЛЬКИМИ ГРАФИКАМИ НА ОДНОМ ОКНЕ

Как *MATLAB*, так и *pyplot* поддерживают концепцию работы с текущим окном и текущими осями. Все команды, используемые для построения, применяются к текущим осям. Функция *gca()* возвращает текущий объект типа *matplotlib.axes.Axes*, функция *gcf()* возвращает текущий объект типа *matplotlib.figure.Figure*.

```
def f(t):
    return np.exp(-t) * np.cos(2*np.pi*t)

t1 = np.arange(0.0, 5.0, 0.1)
t2 = np.arange(0.0, 5.0, 0.02)

plt.figure(1)
plt.subplot(211)
plt.plot(t1, f(t1), 'bo', t2, f(t2), 'k')

plt.subplot(212)
plt.plot(t2, np.cos(2*np.pi*t2), 'r--')
plt.show()
```

По умолчанию команда *figure()* создает только одно окно (*figure(1)*), аналогично как *subplot(111)* будет вызываться оп умолчанию. Функция *subplot()* задает количество строк, колонок и графиков в окне. Вызов команды *subplot(211)* равносилен вызову *subplot(2, 1, 1)*.

Можно создавать произвольное количество осей и графиков на одном окне. Если необходимо вручную задать расположение очей и графиков, необходимо использовать функцию *axes()* следующим образом *axes([left, bottom, width, height])*, где все значения задаются на интервале от *0* до *1*. Для того, чтобы очистить текущее окно используется функция *clf()*, для очистки осей функция *cla()*.

```
import matplotlib.pyplot as plt
plt.figure(1)                # первое окно
plt.subplot(211)             # первый график в окне
plt.plot([1, 2, 3])
plt.subplot(212)             # второй график в окне
plt.plot([4, 5, 6])


plt.figure(2)                # второе окнопо умолчанию subplot(111)
plt.plot([4, 5, 6])

plt.figure(1)                # текущие окно 1 (figure(1)) и график subplot(212);
plt.subplot(211)             # устанавливаем график 211 как текущий в окне 1
plt.title('Easy as 1, 2, 3') # Подписываем график (211)
```

Задание 14. Реализуйте код, приведенный в примере.

РАБОТА С ТЕКСТОМ

Команда *text()* используется для добавления текста в любом месте. Такие функции как *xlabel()*, *ylabel()* и *title()* используются для обозначения определенных областей. Все функции текста возвращают объекты типа *matplotlib.text.Text*.

```
mu, sigma = 100, 15
x = mu + sigma * np.random.randn(10000)
# the histogram of the data
n, bins, patches = plt.hist(x, 50, density=1, facecolor='g', alpha=0.75)
plt.xlabel('Smarts')
plt.ylabel('Probability')
plt.title('Histogram of IQ')
plt.text(60, .025, r'$\mu=100,\ \sigma=15$')
plt.axis([40, 160, 0, 0.03])
plt.grid(True)
plt.show()
```

Аналогично как при работе с линиями задать свойства текста можно использовать именованные аргументы и функцию *setp()*.

```
t = plt.xlabel('my data', fontsize=14, color='red')
```

Функция *annotate()* позволяет упростить работу с текстами за счет аннотации. При использовании аннотации необходимо учитывать два фактора: кортеж *xy* задает расположение объекта для аннотирования; кортеж *xytext* задает координаты текста при аннотировании.

```
ax = plt.subplot(111)
t = np.arange(0.0, 5.0, 0.01)
s = np.cos(2*np.pi*t)
line, = plt.plot(t, s, lw=2)
plt.annotate('local max', xy=(2, 1), xytext=(3, 1.5),
             arrowprops=dict(facecolor='black', shrink=0.05))
plt.ylim(-2, 2)
plt.show()
```

Задание 15. Реализуйте код, приведенный в примере.

ЛОГАРИФМИЧЕСКАЯ И ДРУГИЕ ОСИ

matplotlib.pyplot поддерживает не только линейные оси, а так же логарифмические и степенные оси. Чаще всего эти оси используются, если динамический диапазон данных имеет большой разброс. Чтобы изменить шкалу необходимо использовать конструкцию: *plt.xscale('log')*.

```
y = np.random.normal(loc=0.5, scale=0.4, size=1000)
y = y[(y > 0) & (y < 1)]
y.sort()
x = np.arange(len(y))
plt.figure(1)

# линейный
plt.subplot(121)
plt.plot(x, y)
plt.yscale('linear')
plt.title('linear')
plt.grid(True)
# логорафмический
plt.subplot(122)
plt.plot(x, y)
plt.yscale('log')
plt.title('log')
plt.grid(True)
```

Задание 16. Реализуйте код, приведенный в примере.