ТЕМА 1. NumPy

Мета. Навчитися виконувати математичні та числові операції над масивами і матрицями, користуючись модулем NumPy (Numeric Python).

Вступ. NumPy – це модуль з відкритим кодом для Python, що надає загальні математичні та числові операції в виді скомпільованих, швидких функцій. Вони об'єднуються у високорівневі пакети, які забезпечують функціональність, що порівнюється з можливостями MatLab. NumPy (Numeric Python) надає базові методи для маніпулювання великими масивами і матрицями.

План.

1. Встановлення NumPy
2. Імпорт модуля NumPy
3. Масиви
4. Математичні операції над масивами
5. Перебирання елементів масиву
6. Базові операції над масивами
7. Оператори порівняння і тестування значень
8. Вибірка елементів масиву і маніпуляція з ними
9. Векторна і матрична математика
10. Математика многочленів
11. Статистика
12. Випадкові числа

1. Встановлення NumPy

Для подальшого вивчення пакету NumPy потрібно встановити Python3 і систему керування пакунками `pip` (Python Package Index) , яка використовується для встановлення та управління програмними пакетами, які написані на Python.

Більшість дистрибутивів Python вже містять `pip`. Якщо `pip` відсутній, то його можна інсталювати за допомогою системи керування пакунками.

```
sudo apt-get python-pip
sudo apt-get update && sudo apt-get upgrade
```

Або користуючись утилітою `curl` для завантаження через інтернет.

```
curl https://bootstrap.pypa.io/get-pip.py | python
sudo apt-get update && sudo apt-get upgrade
```

Встановлення NumPy.

```
pip install numpy
sudo apt-get update
```

2.Імпорт модуля NumPy

Існує декілька шляхів імпорту .
Стандартний метод-це використання простого виразу

```
import numpy
```

Але при виклику багатьох функцій набагато доцільніше імпортувати пакет наступним шляхом:

```
import numpy as np
```

Такий вираз дозволить звертатись до numpy об`єктів користуючись `np.X` замість `numpy.X`.

Також можна імпортувати numpy прямо в заданий простір імен, що дозволить взагалі не викликати функцію через оператор “крапка”, а викликати напряму:

```
from numpy import *
```

Однак, використання цього методу не є доцільним у випадку використання деяких корисних структур даних, що надаються пакетом.

3. Масиви

Головною особливістю NumPy є об’єкт *array* – масив, який подібний до списків Python. Однак з масивами можна виконувати числові операції з великими обсягами інформації в рази швидше і, головне, набагато ефективніше ніж із списками.

Приклад створення масиву із списку:

```
import numpy as np

a = np.array([1, 2, 3, 4], float)
>>> a
array([ 1.,  2.,  3.,  4.])
>>> type(a)
<class 'numpy.ndarray'>
```

Як можна зауважити, функція *array* приймає два аргументи – список і тип масиву. Однак, при ініціалізації об’єкту кожен з елементів у списку повинен мати тип, вказаний другим аргументом при виклику функції. Чому так? Як було раніше зазначено, NumPy – це по суті колекція скомпільованих функцій. Більшість з цих функцій написані на мові C, яка має низькорівневі можливості, що забезпечує високу продуктивність.

Розглянемо невелику програму реалізації цього ж масиву, але на мові C:

```
#include <stdlib.h>
#include <stdio.h>

#define FLOAT sizeof(float)
int main(int argc,char** argv)
{
  long size=FLOAT*4; //Кількість памяті на 4 елементи
  float * array=malloc(size); //Виділення памяті
  *array=1.0f;
  *(array+1)=2.0f;
  *(array+2)=3.0f;
  *(array+3)=4.0f;
  int n=size/FLOAT;//Кількість елементів у масиві
  for(int i=0;i<n;i++)
  {
    printf("%f ",*(array+i));
  }
 return 0;
}
```

В цій програмі, з використанням стандартної бібліотеки stdlib.h , динамічно виділяється пам’ять *(sizeof(float)*4)* на 4 елементи і повертається адреса першого елементу типу *float* адресній змінній *array*. Таким чином можна ініціалізувати таку структуру даних, як масив, за допомогою оператора *розіменування*. До адресних змінних також можна застосовувати арифметичні операції. Наприклад: *array+1* буде вказувати на адресу *array+sizeof(float)*1*. Тому пакет NumPy використовує список з однаковим типом.

До всіх елементів такого об’єкту можна отримати доступ і маніпулювати ними, як з звичайним списком.

```
>>> a[:2]
array([ 1.,  4.])
>>> a[3]
8.0
>>> a[0] = 5.
>>> a
array([ 5.,  4.,  5.,  8.])
```

Масиви також можуть бути багатовимірними (матриця). Приклад матриці 2x3:

```
>>> a = np.array([[1, 2, 3], [4, 5, 6]], float)
>>> a
array([[ 1.,  2.,  3.],
[ 4.,  5.,  6.]])
>>> a[0,0]
1.0
>>> a[0,1]
2.0
```

Зрізи (*Array slicing*) працюють з багатовимірними масивами аналогічно, як і з одновимірними, застосовуючи кожний зріз, як фільтр.

```
>>> a = np.array([[1, 2, 3], [4, 5, 6]], float)
>>> a[1,:]
array([ 4.,  5.,  6.])
>>> a[:,2]
array([ 3.,  6.])
>>> a[-1:, -2:]
array([[ 5.,  6.]])
```

Метод *shape* повертає кортеж з кількістю рядків і стовбців у матриці:

```
>>> a.shape
(2, 3)
```

Метод *dtype* повертає тип змінних, що зберігаються в масиві:

```
>>> a.dtype
dtype('float64')
```

Метод *len* повертає кількість рядків у масиві:

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

Метод *in* використовується для перевірки на наявність елемента у масиві:

```
>>> a = np.array([[1, 2, 3], [4, 5, 6]], float)
>>> 2 in a
True
>>> 0 in a
False
```

Масиви можна переформатувати за допомогою метода *reshape*, що задає новий багатовимірний масив. Приклад переформатування одновимірного масиву 0..9, що неявно створений за допомогою вбудованої функції *range(n)* в матрицю 5×2:

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

Примітка: метод *reshape* не модифікує оригінальний масив, а повертає новий. Списки також можна створювати з масивів:

```
>>> a = np.array([1, 2, 3], float)
>>> a.tolist()
[1.0, 2.0, 3.0]
>>> list(a)
[1.0, 2.0, 3.0]
```

Також можна переконвертувати масив у бінарну стрічку за допомогою метода *tostring*. Що є корисним для збереження великої кількості інформації у файлах для зчитування у майбутньому, що реалізується зворотнього методу *fromstring*.

```
>>> a = array([1, 2, 3], float)
>>> s = a.tostring()
>>> s
'\x00\x00\x00\x00\x00\x00\xf0?\x00\x00\x00\x00\x00\x00\x00@\x00\x00\x00\x00\x00\x00\x08@'
>>> np.fromstring(s)
array([ 1.,  2.,  3.])
```

Іноді потрібно ініціалізувати масив одним значенням для подальшої роботи з ним.

```
>>> a = array([1, 2, 3], float)
>>> a
array([ 1.,  2.,  3.])
>>> a.fill(0)
>>> a
array([ 0.,  0.,  0.])
```

Можна також транспонувати масив, при цьому створюється новий масив.

```
>>> a = np.array(range(6), float).reshape((2, 3))
>>> a
array([[ 0.,  1.,  2.],
[ 3.,  4.,  5.]])
>>> a.transpose()
array([[ 0.,  3.],
[ 1.,  4.],
[ 2.,  5.]])
```

Багатовимірний масив можна переконвертувати в одновимірний масив за допомогою методу *flatten:*

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

Можна зчепити два і більше масивів за допомогою методу *concatenate*:

```
>>> a = np.array([1,2], float)
>>> b = np.array([3,4,5,6], float)
>>> c = np.array([7,8,9], float)
>>> np.concatenate((a, b, c))
array([1., 2., 3., 4., 5., 6., 7., 8., 9.])
```

Якщо ж масиви не одномірні то є можливість задати вісь по якій буде здійснюватись конкатенація. За замовчуванням конкатенація здійснюється за першим виміром.

```
>>> a = np.array([[1, 2], [3, 4]], float)
>>> b = np.array([[5, 6], [7,8]], float)
>>> np.concatenate((a,b))
array([[ 1.,  2.],
[ 3.,  4.],
[ 5.,  6.],
[ 7.,  8.]])
>>> np.concatenate((a,b), axis=0)
array([[ 1.,  2.],
[ 3.,  4.],
[ 5.,  6.],
[ 7.,  8.]])
>>>
np.concatenate((a,b), axis=1)
array([[ 1.,  2.,  5.,  6.],
[ 3.,  4.,  7.,  8.]])
```

4. Математичні операції над масивами

При виконанні стандартних математичних операцій над масивами – кількість рядків і стовпців першого масиву має бути рівною кількості рядків і стовпців другого масиву. Це означає, що масиви повинні бути однакового розміру при здійсненні бінарних математичних операцій над ними (додавання, віднімання, множення і тому подібне).

```
>>> a = np.array([1,2,3], float)
>>> b = np.array([5,2,6], float)
>>> a + b
array([6., 4., 9.])
>>> a - b
array([-4., 0., -3.])
>>> a * b
array([5., 4., 18.])
>>> b / a
array([5., 1., 2.])
>>> a % b
array([1., 0., 3.])
>>> b**a
array([5., 4., 216.])
```

Коли розмір одного масиву не співпадає з іншим – виникає виняток *Exception*:

```
>>> a = np.array([1,2,3], float)
>>> b = np.array([4,5], float)
>>> a + b
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
ValueError: operands could not be broadcast together with shapes (3,) (2,)
```

В *numpy* також включена бібліотека стандартних математичних функцій, які можуть бути застосовані до кожного елементу до масиву: *abs, sign, sqrt, log, log10, exp, sin, cos, tan, arcsin, arccos, arctan, sinh, cosh, tanh, arcsinh, arccosh* і *arctanh.*

```
>>> a = np.array([1, 4, 9], float)
>>> np.sqrt(a)
array([ 1.,  2.,  3.])
```

Функції *floor*, *ceil* і *rint* повертають нижнє, верхнє і округлене значення.

```
>>> a = np.array([1.1, 1.5, 1.9], float)
>>> np.floor(a)
array([ 1.,  1.,  1.])
>>> np.ceil(a)
array([ 2.,  2.,  2.])
>>> np.rint(a)
array([ 1.,  2.,  2.])
```

Також в *numpy* включені дві важливі математичні константи (число і число e):

```
>>> np.pi
3.1415926535897931
>>> np.e
2.7182818284590451
```

5. Перебирання елементів масиву

Масиви ітеруються так як списки:

```
>>> a = np.array([1, 4, 5], int)
>>> for x in a:
...   print x
1
4
5
```

Для багатовимірних масивів ітерація проводиться по першій вісі і кожний прохід циклу повертає масив.

```
>>> a = np.array([[1, 2], [3, 4], [5, 6]], float)
>>> for x in a:
...   print x
[ 1.  2.]
[ 3.  4.]
[ 5.  6.]
```

Для перебору двовимірного масиву можна застосувати вкладений цикл:

```
>>> a = np.array([[1, 2], [3, 4], [5, 6]], int)
>>> for x in a:
...   for y in x:
          print y

1
2
3
4
5
6
```

Перебір елементів двовимірного масиву можна виконати в одній ітерації зразу за двома змінними:

```
>>> a = np.array([[1, 2], [3, 4], [5, 6]], float)
>>> for (x, y) in a:
...   print x * y
2.0
12.0
30.0
```

6. Базові операції над масивами

Для отримання деякий властивостей масивів існує багато функцій. Елементи масиву можуть бути просумовані або перемножені за допомогою відповідних методів *sum* і *prod*:

```
>>> a = np.array([2, 4, 3], float)
>>> a.sum()
9.0
>>> a.prod()
24.0
```

Деякі функції надають можливість оперувати статистичними даними. Це такі функції, як середнє арифметичне (*mean*), дисперсія (*var*) і стандартне відхилення (*std).*

```
>>> a = np.array([2, 1, 9], float)
>>> a.mean()
4.0
>>>
a.var()
12.666666666666666
>>> a.std()
3.5590260840104371
```

Можна знайти мінімальне (*min)* і максимальне (*max*) числове значення у масиві:

```
>>> a = np.array([2, 1, 9], float)
>>> a.min()
1.0
>>> a.max()
9.0
```

Функції *argmin* і *argmax* повертають індекс мінімального або максимального елемента у масиві:

```
>>> a = np.array([2, 1, 9], float)
>>> a.argmin()
1
>>> a.argmax()
2
```

Як списки, так і масиви можна відсортувати. Для цього є функція *sort*. За замовчуванням сортування відбувається за алгоритмом QuickSort. Однак при сортування великих обсягів з використанням цього алгоритму має показник *обчислювальної складності* $O(n^2)$.

NumPy підтримує три алгоритми сортування – QuickSort , MergeSort, HeapSort. Алгоритм сортування вказується третім параметром у функції *sort*, після задання вісі, вздовж якої буде здійснене сортування.

Вибір алгоритму сортування залежить від кількості елементів у масиві, наявності часткової відсортованості і обсягу пам'яті.

Сортування одновимірного масиву з використанням алгоритму QuickSort:

```
>>> a = np.array([6, 2, 5, -1, 0], float)
>>> a.sort()
>>> a
array([-1.,  0.,  2.,  5.,  6.])
```

Унікальні елементи масиву можна отримати за допомогою функції *unique.*

```
>>> a = np.array([1, 1, 4, 5, 5, 5, 7], float)
>>> np.unique(a)
array([ 1.,  4.,  5.,  7.])
```

Для двовимірного масиву діагональ можна отримати за допомогою функції *diagonal*.

```
>>> a = np.array([[1, 2], [3, 4]], float)
>>> a.diagonal()
array([ 1.,  4.])
```

7.Оператори порівняння і тестування значень

Булеве поелементне порівняння може бути застосоване до масивів однакового розміру

Кожен з такий бінарних операторів порівняння повертає масив булевих значень True/False:

```
>>> a = np.array([1, 3, 0], float)
>>> b = np.array([0, 3, 2], float)
>>> a > b
array([ True, False, False], dtype=bool)
>>> a == b
array([False,  True, False], dtype=bool)
>>> a <= b
array([False,  True,  True], dtype=bool)
```

Відповідно результат порівняння може бути збереженим у масив:

```
>>> c = a > b
>>> c
array([ True, False, False], dtype=bool)
```

Також масиви можуть бути порівняні з одиночним значенням:

```
>>> a = np.array([1, 3, 0], float)
>>> a > 2
array([False,  True, False], dtype=bool)
```

Оператори *any* і *all* можуть бути використані для визначення чи істинний хоча б один елемент чи всі елементи масиву:

```
>>> c = np.array([ True, False, False], bool)
>>> any(c)
True
>>> all(c)
False
```

Комбіновані булеві вирази можуть бути застосовані до масивів за принципом елемент-елемент використовуючи спеціальні функції *logical_and, logical_or і logical_not*:

```
>>> a = np.array([1, 3, 0], float)
>>> np.logical_and(a > 0, a < 3)
array([ True, False, False], dtype=bool)
>>> b = np.array([True, False, True], bool)
>>> np.logical_not(b)
array([False,  True, False], dtype=bool)
>>> c = np.array([False, True, False], bool)
>>> np.logical_or(b, c)
array([ True,  True,  False], dtype=bool)
```

8. Вибірка елементів масиву і маніпуляція з ними

Масиви, як і списки, підтримують звернення до елементів за індексом. Однак, на відмінну від списків, масиви також дозволяють вибирати елементи використовуючи інші масиви.

Це означає, що можна використовувати масив для фільтрації специфічних підмножин елементів інших масивів.

Булеві масиви також можуть використовуватися як масиви для фільтрації:

```
>>> a = np.array([[6, 4], [5, 9]], float)
>>> a >= 6
array([[ True, False],
       [False,  True]], dtype=bool)
>>> a[a >= 6]
array([ 6.,  9.])
```

Варто зауважити, що коли передається булевий масив a>=6, як індекс для операції доступу за індексом до масиву *a*, повернений масив буде містити тільки True значення.

Також можна записати масив для фільтрації у змінну:

```
>>> a = np.array([[6, 4], [5, 9]], float)
>>> sel = (a >= 6)
>>> a[sel]
array([ 6.,  9.])
```

Для більш гнучкої фільтрації використовуються булеві вирази:

```
>>> a[np.logical_and(a > 5, a < 9)]
>>> array([ 6.])
```

Вибирати елементи з масиву можна також з використанням цілочисельних масивів або списків. В цьому випадку, цілочисельний масив зберігає індекси вибраних елементів:

```
>>> a = np.array([2, 4, 6, 8], float)
>>> b = np.array([0, 0, 1, 3, 2, 1], int)
>>> a[b]
array([ 2.,  2.,  4.,  8.,  6.,  4.])
```

Для багатовимірних масивів, необхідно передати декілька одновимірних цілочисельних масивів в оператор доступу за індексом (в даному випадку індекси це масиви) для кожної вісі. При індексації перший елемент першого масиву є індексом ряду, а перший елемент другого масиву є індексом стовпця.

```
>>> a = np.array([[1, 4], [9, 16]], float)
>>> b = np.array([0, 0, 1, 1, 0], int)
>>> c = np.array([0, 1, 1, 1, 1], int)
>>> a[b,c]
array([  1.,   4.,  16.,  16.,   4.])
```

Спеціальна функція *take* дозволяє індексувати масиви з використанням масиву цілочисельних індексів:

```
>>> a = np.array([2, 4, 6, 8], float)
>>> b = np.array([0, 0, 1, 3, 2, 1], int)
>>> a.take(b)
array([ 2.,  2.,  4.,  8.,  6.,  4.])
```

Зворотньою до функції *take* є функція *put*, яка бере значення з вхідного масиву і записує їх у інший масив за заданими індексами:

```
>>> a = np.array([0, 1, 2, 3, 4, 5], float)
>>> b = np.array([9, 8, 7], float)
>>> a.put([0, 3], b)
>>> a
array([ 9.,  1.,  2.,  8.,  4.,  5.])
```

Значення 7 з вхідного масиву *b* не використано, так як задано тільки два індекси 0 і 3 для запису у вихідний масив. Замість вхідного масиву може бути задане значення:

```
>>> a = np.array([0, 1, 2, 3, 4, 5], float)
>>> a.put([0, 3], 5)
>>> a
array([ 5.,  1.,  2.,  5.,  4.,  5.])
```

9. Векторна і матрична математика

Пакет NumPy широко використовується вирішення задач лінійної алгебри, так як він надає багато функцій для роботи з векторами і матрицями.

Функція *dot* повертає скалярний добуток векторів:

```
>>> a = np.array([1, 2, 3], float)
>>> b = np.array([0, 1, 1], float)
>>> np.dot(a, b)
5.0
```

Також можна отримати скалярний, тензорний і зовнішній добуток векторів і матриць. Потрібно зауважити, що для векторів внутрішній і скалярний добуток співпадає.

```
>>> a = np.array([1, 4, 0], float)
>>> b = np.array([2, 2, 1], float)
>>> np.outer(a, b)
array([[ 2.,  2.,  1.],
       [ 8.,  8.,  4.],
       [ 0.,  0.,  0.]])
>>> np.inner(a, b)
10.0
>>> np.cross(a, b)
array([ 4., -1., -6.])
```

Також функція *dot* служить для перемноження матриць:

```
>>> a = np.array([[0, 1], [2, 3]], float)
>>> b = np.array([2, 3], float)
>>> c = np.array([[1, 1], [4, 0]], float)
>>> a
array([[ 0.,  1.],
       [ 2.,  3.]])
>>> np.dot(b, a)
array([  6.,  11.])
>>> np.dot(a, b)
array([  3.,  13.])
```

```
>>> np.dot(a, c)
array([[  4.,   0.],
       [ 14.,   2.]])
>>> np.dot(c, a)
array([[ 2.,  4.],
       [ 0.,  4.]])
```

NumPy має набір вбудованих функцій для роботи з лінійною алгеброю (модуль linalg). Визначення детермінанту матриці:

```
>>> a = np.array([[4, 2, 0], [9, 3, 7], [1, 2, 1]], float)
>>> a
array([[ 4.,  2.,  0.],
       [ 9.,  3.,  7.],
       [ 1.,  2.,  1.]])
>>> np.linalg.det(a)
-53.999999999999993
```

Знаходження власного вектора і власного значення матриці:

```
>>> vals, vecs = np.linalg.eig(a)
>>> vals
array([ 9.        ,  2.44948974, -2.44948974])
>>> vecs
array([[-0.3538921 , -0.56786837,  0.27843404],
       [-0.88473024,  0.44024287, -0.89787873],
       [-0.30333608,  0.69549388,  0.34101066]])
```

Знаходження невиродженої матриці:

```
>>> b = np.linalg.inv(a)
>>> b
array([[ 0.14814815,  0.07407407, -0.25925926],
       [ 0.2037037 , -0.14814815,  0.51851852],
       [-0.27777778,  0.11111111,  0.11111111]])
>>> np.dot(a, b)
array([[  1.00000000e+00,   5.55111512e-17,   2.22044605e-16],
       [  0.00000000e+00,   1.00000000e+00,   5.55111512e-16],
       [  1.11022302e-16,   0.00000000e+00,   1.00000000e+00]])
```

10. Математика многочленів

NumPy має методи для роботи з поліномами. Передаючи список коренів у функцію *poly*, можна отримати коефіцієнти нерівності:

```
>>> np.poly([-1, 1, 1, 10])
array([ 1, -11,   9,  11, -10])
```

В цьому випадку масив повертає коефіцієнти відповідного рівняння:

$$x^4 - 11x^3 + 9x^2 + 11x - 10 = 0$$

Можна виконати і зворотну операція – передати список коефіцієнтів полінома у функцію *roots* і отримати корені рівняння:

```
>>> np.roots([1, 4, -2, 3])
array([-4.57974010+0.j        ,  0.28987005+0.75566815j,
        0.28987005-0.75566815j])
```

Можна інтегрувати рівняння, наприклад $1/4x^4 + 1/3x^3 + 1/2x^2 + C$, де константа C звичайно дорівнює нулю:

```
>>> np.polyint([1, 1, 1, 1])
array([ 0.25      ,  0.33333333,  0.5       ,  1.        ,  0.        ])
```

Аналогічно, можуть бути взяті і похідні:

```
>>> np.polyder([1./4., 1./3., 1./2., 1., 0.])
array([ 1.,  1.,  1.,  1.])
```

Функції *polyadd, polysub, polymul* і *polydiv* виконують додавання, віднімання, множення і ділення коефіцієнтів многочлена відповідно.

Функція *polyval* підставляє в многочлен задане значення, наприклад многочлен $x^3 - 2x^2 + 2$, для x=4:

```
>>> np.polyval([1, -2, 0, 2], 4)
34
```

Функція *polyfit* виконує підбір(інтерполяцію) многочлена заданого порядку для набору значень:

```
>>> x = [1, 2, 3, 4, 5, 6, 7, 8]
>>> y = [0, 2, 1, 3, 7, 10, 11, 19]
>>> np.polyfit(x, y, 2)
array([ 0.375     , -0.88690476,  1.05357143])
```

11. Статистика

Окрім раніше зазначених функцій *mean, var, std* NumPy має додаткові методи для роботи з статистичними даними в масивах.

Знаходження медіани:

```
>>> a = np.array([1, 4, 3, 8, 9, 2, 3], float)
>>> np.median(a)
3.0
```

Коефіцієнти кореляції для деяких змінних із масивів [[*x*1,*x*2, ...],[*y*1,*y*2, ...],[*z*1,*z*2, ...], ...], де *x*, *y*, *z* різні вибірки спостережень:

```
>>> a = np.array([[1, 2, 1, 3], [5, 3, 1, 8]], float)
>>> c = np.corrcoef(a)
>>> c
array([[ 1.        ,  0.72870505],
[ 0.72870505,  1.        ]])
```

Результат в масиві c[i, j], який містить кореляційний коефіцієнт для *i*-тих і *j*-тих спостережень.

Аналогічно, знаходиться коваріаційний момент:

```
>>> np.cov(a)
array([[ 0.91666667,  2.08333333],
[ 2.08333333,  8.91666667]])
```

12.Випадкові числа

При моделюванні і аналізі часто використовуються випадкові числа. Наприклад, перевірка правильності алгоритму сортування незалежно від вхідних даних.

Випадкові числа генеруються за певним математичним алгоритмом і тому і тому є псевдовипадковими. Одним з таких методів генерування псевдовипадкових чисел є лінійний конгруентний метод:

```
x(n) = (a * x(n - 1) + c) % m
```

Для генерації псевдовипадкових чисел у NumPy використовується особливий алгоритм, що має назву Mersenne Twister. Сам генератор псевдовипадкових чисел вбудований у NumPy у модулі *random*.

Масив випадкових чисел з інтервалу [0.0, 1.0] можна згенерувати за допомогою функції *rand*:

```
>>> np.random.rand(5)
array([ 0.40783762,  0.7550402 ,  0.00919317,  0.01713451,  0.95299583])
```

Функція *rand* також може генерувати двовимірні масиви або одновимірні і перетворювати їх у двовимірні за допомогою функції *reshape*:

```
>>> np.random.rand(2,3)
array([[ 0.50431753,  0.48272463,  0.45811345],
       [ 0.18209476,  0.48631022,  0.49590404]])
>>> np.random.rand(6).reshape((2,3))
array([[ 0.72915152,  0.59423848,  0.25644881],
       [ 0.75965311,  0.52151819,  0.60084796]])
```

Одне випадкове числа в інтервалі [0.0,1.0] генерує функція *random*:

```
>>> np.random.random()
0.70110427435769551
```

Для генерації випадкового цілочисельного числа (int) в діапазоні [min, max) використовується функцію *randint*(*min*, *max*):

```
>>> np.random.randint(5, 10)
9
```

Висновки.

Модуль NumPy є надзвичайно швидким і ефективним інструментом для вирішення задач статистики, матричної і векторної геометрії і виконання загальних математичних розрахунків.

Література.

1. Дж. Вандер Плас. Python для сложных задач. Наука о данных и машинное обучение. – Питер, 2017. – 576 с.
2. Лутц М. Изучаем Python, 4-t издание. Пер. с англ.. СПб.: Символ-Плюс, 2011. – 1280 с.
3. www.numpy.org
2. https://docs.scipy-lectures.org/intro/numpy/numpy.html - tutorial on numpy
3. https://www.tutorialspoint.com/numpy/index.htm - NumPy Tutorial

Запитання.

1. Які недоліки і переваги імпортування модуля numpy з використання інструкцій `import numpy i from numpy import`

2.Чому при ініціалізації об’єкту *array* кожен з елементів у списку повинен мати тип, вказаний другим аргументом при виклику функції?

3. Як працюють зрізи у багатовимірних масивах?

4. Як визначити кількість рядків і стовпців у матриці?

5. Базові операції над масивами.

6. Які алгоритми сортування функція підтримує функція sort?

7. Який тип даних повертають оператори порівняння об’єктів типу array? Як це використовується для фільтрування елементів масиву?

8. Функції для роботи з векторами.

9. Функції для роботи з матрицями.

10. Методи для роботи з поліномами.

11. За допомогою яких функцій можна здійснювати диференціювання і інтегрування?

12. Функції і методи статистичних обчислень.

13. Функції для генерування випадкових чисел.