Инструменты обработки и визуализации данных

Тема №1 – Введение в обработку данных

Среда для выполнения лабораторных работ

Для установки среды для выполнения лабораторных работ скачайте с сайта https://www.anaconda.com и установите на компьютер дистрибутив Anaconda (Individual Edition). В качестве редактора/среды исполнения при выполнении лабораторных работ будет использоваться приложение Jupiter Notebook.

Если на локальном компьютере ведется разработка программ в среде Python, то для лабораторных работ рекомендуется создание виртуального окружения.

Например, для виртуального окружения `dm` выполняем команды (в теминале):

conda create -n dm

pip install --user ipykernel

python -m ipykernel install --user --name=dm

Тогда в Jupiter Notebook появляется виртуальное окружение `dm`.

Структуры данных языка Python

Список (List)

Список в Python – это последовательность элементов, разделенных между собой запятой и заключенных в квадратные скобки.

Список является изменяемым упорядоченным типом данных.

Примеры списков:

In [ ]:
```
[10, 20, 30, 40]
```

In [ ]:
```
["один", "два", "три"]
```

In [ ]:
```
[100, 3.14159, 'пи']
```

Список может быть создан с помощью литерала (выражения, создающего объект):

In [ ]:
```
list0 = [100, 200, 300, 400]
list0
```

Также список может быть создан с помощью функции `list()`:

In [ ]:
```
list1 = list("ЗПИбд-01-24")
list1
```

Так как список - это упорядоченный тип данных, то в списках можно обращаться к элементу списка по номеру и делать срезы:

In [ ]:
```
list1[0]
```

In [ ]:
```
list1[-1],list1[-2]
```

In [ ]:
```
list1[1:3]
```

In [ ]:
```
list1[::2]
```

Перевернуть список наоборот можно с помощью среза `[::-1]` или метода `reverse()`:

In [ ]:
```
list1[::-1]
```

In [ ]:
```
list1.reverse()
list1
```

Так как список является изменяемым типом данных, элементы списка можно менять:

In [ ]:
```
list1[9] = "Ф"
list1.reverse()
list1
```

При присваивании имени списка другой переменной создается новая переменная, которое ссылается на тот же список, и при изменении элементов "нового" списка изменяется и исходный список:

In [ ]:
```
list2 = list1
list2[1] = 'П'
list1
```

Для создания копии списка можно использовать срез `[:]` или метод `copy()`:

In [ ]:
```
list2 = list1[:]
list2[1] = 'Ф'
print(list1)
print(list2)
```

Можно создать список, состоящий из списков и, как и в обычном списке, можно обращаться к элементам во вложенных списках:

In [ ]:
```
gruppa = [[1, "Иванов", 1999],[2, "Петров", 2000],[3, "Сидоров", 1998]]
print(gruppa)
gruppa[1][2]
```

Функция `len()` возвращает количество элементов в списке:

In [ ]:
```
len(list1)
```

Функция `sorted()` сортирует элементы списка по возрастанию и возвращает новый список с отсортированными элементами:

In [ ]:
```
sorted(list1)
```

Методы для работы со списками

Список - это изменяемый тип данных, поэтому большинство методов для работы со списками изменяют исходный список, ничего не возвращая.

Метод `join()` собирает список строк в одну строку с разделителем, который указан перед `join`:

In [ ]:
```
print(''.join(list1))
print('_'.join(list1))
```

Метод `append()` добавляет в конец списка указанный элемент:

In [ ]:
```
gruppa.append([4, "Васильев", 1995])
print(gruppa)
```

Если нужно объединить два списка, то можно использовать два способа: метод `extend()` и операцию сложения. Отличие способов: `extend()` меняет список, к которому применен метод, а суммирование возвращает новый список, который состоит из двух списков.

In [ ]:
```
list2 = [2, 4, 6, 8, 10]
list3 = [3, 5, 7, 9]
print(list2 + list3)
list3.extend(list2)
list3
```

Метод `pop()` удаляет элемент, индекс которого соответствует указанному в качестве аргумента номеру, при этом метод возвращает этот элемент:

In [ ]:
```
list3.pop(0)
print(list3.pop())
list3
```

Если номер не указан, то удаляется последний элемент списка.

Метод `remove()` удаляет указанный элемент (не возвращая его):

In [ ]:
```
list1.remove("3")
list1
```

В методе `remove()` указывается сам элемент, который надо удалить, а не его номер в списке.

Метод `index()` позволяет найти номер элемента в списке:

In [ ]:
```
list3.index(2)
```

Метод `insert()` позволяет вставить элемент на определенное место в списке:

In [ ]:
```
list3.insert(0,3) # значение 3 на место с индексом 0
list3
```

Метод `sort()` сортирует сам список:

In [ ]:
```
list3.sort()
list3
```

Кортеж (Tuple)

Кортеж в Python – это последовательность элементов, которые разделены между собой запятой и заключены в круглые скобки.

Кортеж является неизменяемым упорядоченным типом данных, т.е. кортеж - это список, который нельзя изменить.

Создадим пустой кортеж:

In [ ]:
```
tuple0 = tuple()
tuple0
```

Создать кортеж из одного элемента можно так (обратите внимание на запятую):

In [ ]:
```
tuple1 = (2025,)
tuple1
```

Кортеж можно получить из списка:

In [ ]:
```
tuple2 = tuple([2025,2026,2027])
tuple2
```

К объектам в кортеже можно обращаться, как и к объектам списка, по порядковому номеру:

In [ ]:
```
tuple2[1:]
```

Так как кортеж неизменяем, присвоить новое значение элементу кортежа нельзя:

In [ ]:
```
tuple2[2] = 2028
```

С конструкцией кортежа связана операция распаковки переменных, когда в присваивании слева от знака равенства указываются несколько переменных:

In [ ]:
```
(u, v) = (2., 3.)
u,v
```

In [ ]:
```
u, v, w = [7, 8, 9]
u,v,w
```

In [ ]:
```
u, v, _, w, _ = [5, 6, 7, 8, 9]
u,v,w
```

Такие конструкции часто применяются в функциях, когда необходимо вернуть несколько значений.

Фирменная конструкция языка Python, чтобы поменять местами две переменные:

In [ ]:
```
u,v = v,u
u,v
```

Применение кортежей:

- защита данных (кортеж защищен от изменений, как намеренных, так и случайных)
- возможность использовать кортежи в качестве ключей словаря (см. далее)
- кортежи занимают меньше места, чем списки

Словарь (Dictionary)

Словари в Python - это неупорядоченные коллекции произвольных объектов с доступом по ключу.

Словари характеризуются следующими свойствами:
• это изменяемый неупорядоченный тип данных
• данные в словаре - это пары **ключ: значение**
• доступ к значениям осуществляется по ключу, а не по номеру, как в списках
• так как словари изменяемы, то элементы словаря можно менять, добавлять, удалять
• ключ должен быть объектом неизменяемого типа: число, строка, кортеж
• значение может быть данными любого типа

Пример словарей:

In [ ]:
```
d1 = {} # пустой словарь
d1
```

In [ ]:
```
d2 = { "John": 100, "Kate": 200 }
d2
```

In [ ]:
```
d3 = dict( Nick = "Boston", Sara = "Seattle" )
d3
```

In [ ]:
```
d99 = {(1,2):12}
d99
```

Проверка наличия ключа в словаре производится с помощью оператора `in`:

In [ ]:
```
name = "Kate"
if name in d2:
    print("Имя "+name+" в словаре")
```

Доступ к элементу словаря, осуществляется как же как доступ к элементу списка, только в качестве индекса указывается ключ, а именно, конструкция `<имя словаря>[<ключ>]`:

In [ ]:
```
d2[ "John" ], d3[ "Nick" ]
```

Для добавления нового значения в словарь d2 достаточно написать:

In [ ]:
```
d2[ "Willy" ] = 500
d2
```

В словаре в качестве значения можно использовать другой словарь (или, скажем, список).

Методы для работы со словарями

При присваивании имени словаря другой переменной создается еще одно имя, которое ссылается на тот же словарь.

Для создания полной копии словаря можно использовать метод `copy()`:

In [ ]:
```
d4 = d2.copy()
d2['Willy'] = 300
d4
```

Для очистки словаря можно использовать метод `clear()`:

In [ ]:
```
d2.clear()
d2
```

Если при обращении к словарю указывается ключ, которого нет в словаре, возникает ошибка. Метод `get()` запрашивает ключ, и если его нет, вместо ошибки возвращает значение `None`:

In [ ]:
```
d4["Peter"]
```

In [ ]:
```
print(d4.get("Peter")) # нужна функция print, так как иначе не будет вывода значения
```

Проверка в операторе `if` выглядит так:

In [ ]:
```
if d4.get("Peter") is None:
    print("Peter не входит в словарь")
```

Также можно указывать другое значение вместо `None`:

In [ ]:
```
d4.get("James","Ooops")
```

Метод `setdefault()` ищет ключ, и если его нет, вместо ошибки создает ключ со значением `None`:

In [ ]:
```
d4.setdefault("James")
d4
```

Второй аргумент позволяет указать, какое значение должно соответствовать ключу:

In [ ]:
```
d4.setdefault("Piter2",400)
d4
```

Методы `keys()`, `values()`, `items()` работают так:

In [ ]:
```
d4.keys()
```

In [ ]:
```
d4.values()
```

In [ ]:
```
d4.items()
```

In [ ]:
```
list(d4.items())
```

Все три метода возвращают специальные объекты, которые отображают ключи, значения и пары ключ-значение словаря соответственно, причем эти объекты меняются с изменением словаря.

Если нужно зафиксировать, скажем, список ключей, то достаточно конвертировать ключи в список: `list(d4.keys())`.

Для удаления ключа и значения можно использовать команду `del`:

In [ ]:
```
del d4['James']
d4
```

Метод `update()` позволяет добавлять в словарь содержимое другого словаря:

In [ ]:
```
d4.update({"Carl":600,"Lolita":650})
d4
```

или обновить значения:

In [ ]:
```
d4.update({'John': 150, 'Kate': 250})
d4
```

Если нужно создать словарь с известными ключами, но пока что пустыми значениями (или одинаковыми значениями), то удобна функция `fromkeys()`:

In [ ]:
```
aero_keys = ["SVO","DME","VNU"]
d5 = dict.fromkeys(aero_keys, 0)
d5
```

Перебор элементов словаря в цикле

Для перебора элементов словаря применяются цикл `for` и некоторые методы словаря:

In [ ]:
```
print("Перебор ключей:")
for key in d4.keys():
    print(key)
```

In [ ]:
```
print("Перебор значений:")
for val in d4.values():
    print(val, end=' ')
```

In [ ]:
```
print("Перебор пар:")
for key,val in d4.items():
    print("{} --> {}".format(key,val))
```

Множество (Set)

Множество в Python - это изменяемый неупорядоченный тип данных. В множестве всегда содержатся только уникальные элементы. Множество в Python - это последовательность элементов, которые разделены между собой запятой и заключены в фигурные скобки.

С помощью множества можно легко убрать повторяющиеся элементы:

In [ ]:
```
a_list = [10, 20, 30, 40, 20, 30, 40, 50]
a_set = set(a_list)
a_set
```

Метод `add()` добавляет элемент во множество:

In [ ]:
```
a_set.add(100)
a_set
```

Метод `discard()` позволяет удалять элементы, не выдавая ошибку, если элемента в множестве нет:

In [ ]:
```
a_set.discard(90)
a_set.discard(100)
a_set
```

Метод `clear()` очищает множество:

In [ ]:
```
a_set.clear()
a_set
```

Операции с множествами

Множества полезны тем, что с ними можно делать различные операции и находить объединение множеств, пересечение и так далее. Объединение множеств можно получить с помощью метода `union()` или оператора `|`:

In [ ]:
```
set1 = {1,2,3,4,5,6,7,8,9,10}
set2 = {2,4,6,8,10,12,14,16,18,20}
print(set1.union(set2))
print(set1 | set2)
```

Пересечение множеств можно получить с помощью метода `intersection()` или оператора `&`:

In [ ]:
```
print(set1.intersection(set2))
print(set1 & set2)
```

Создание множества

Нельзя создать пустое множество с помощью литерала (так как в таком случае это будет не множество, а словарь):

In [ ]:
```
set0 = {}
print(set0)
type(set0)
```

Пустое множество можно создать при помощи вызова `set()`:

In [ ]:
```
set0 = set()
print(set0)
type(set0)
```

Множество из строки будет содержать символы, использованные в строке:

In [ ]:
```
set("Привет, мир")
```

Фиксированное множество `frozenset`

Единственное отличие объекта `set` от объекта `frozenset` заключается в том, что `set` - изменяемый тип данных, а `frozenset` - нет.

In [ ]:
```
set1 = set('Привет')
set2 = frozenset('Привет')
set1 == set2
```

In [ ]:
```
set2.add('!')
```

Полезные функции последовательностей

Функция enumerate

Встроенная функция `enumerate` возвращает последовательность кортежей вида `(index, value)`:

In [ ]:
```
lst = ["Вася", "Петя", "Леша"]
enumerate(lst)
```

In [ ]:
```
list(enumerate(lst))
```

In [ ]:
```
for index, value in enumerate(lst):
    print(str(index) + ". " + value)
```

Функция zip

Функция `zip` «сшивает» элементы нескольких списков, кортежей или других последовательностей, создавая последовательность кортежей:

In [ ]:
```
lst2 = ["Иванов", "Петров", "Сидоров"]
zip(lst, lst2)
```

In [ ]:
```
list(zip(lst, lst2))
```

In [ ]:
```
list(zip(lst, lst2, [1,2,3,4,5]))
```

In [ ]:
```
for idx, (name, fname) in enumerate(zip(lst, lst2)):
    print(f"{idx}. {fname} {name}")
```

Списковое и другие включения (comprehensions)

Списковым включением называется способ конструирования списков при помощи цикла `for` внутри квадратных скобок. Основная синтаксическая форма спискового включения такова:

```
[expr for val in collection if condition]
```

Это эквивалентно следующему циклу `for`:

```
result = []
for val in collection:
    if condition:
        result.append(expr)
```

Условие фильтрации можно опустить.

Например, список целых чисел от 0 до 9 может быть получен так:

In [ ]:
```
[i for i in range(1,10)]
```

Составим список квадратов целых чисел от 0 до 9:

In [ ]:
```
[i**2 for i in range(10)]
```

В списковом включении можно использовать условный оператор `if` для выбора только тех элементов, которые удовлетворяют заданному условию.

Например,

In [ ]:
```
[i**2 for i in range(100) if i % 7==2]
```

При помощи списковых включений можно создавать списки из строковых значений:

In [ ]:
```
['ЗПИбд-{}-24'.format(num) for num in range(1,11)]
```

Возможна и более сложная конструкция генератора списков с несколькими `for`:

In [ ]:
```
name_list = [["Алексей", "Валерий", "Михаил", "Сергей", ],
             ["Анна", "Елизавета", "Мария", "Полина", ]]

[name for names in name_list for name in names if name.count("е") >= 2]
```

Словарные включения порождают словари, например:

In [ ]:
```
{str(num):2**num for num in range(20)}
```

А **множественные включения** порождают множества:

In [ ]:
```
text = "Пусть всегда будет солнце"
{sym for sym in text}
```

Если конструкция `for ... in ...` применяется в круглых скобках, то это **генераторное выражение**, которое создает итератор с отложенными вычислениями.

In [ ]:
```
gen = (sym for sym in text)
gen
```

In [ ]:
```
next(gen)
```

Тест на включения

Пусть **N** – последняя цифра суммы двух последних цифр Вашего студенческого билета. Решите задание **N**, используя включения Python, в течение __ минут и направьте программный код и результат его выполнения через Мессенджер Яндекса на адрес `shorokhov_sg@pfur.ru`

```
fruits = ['mango', 'kiwi', 'strawberry', 'guava', 'pineapple', 'mandarin orange']

numbers = [2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 13, 17, 19, 23, 256, -8, -4, -2, 5, -9]
```

0. create a list that contains each fruit with first letter capitalized (output like ['Mango', 'Kiwi', 'Strawberry', etc...])
1. create a dictionary that contains only the fruits (as keys) and their number of vowels (as values) for fruits with more than two vowels
2. create a set that contains only the fruits with exactly two vowels
3. make a list that contains each fruit with more than 5 characters
4. create a dictionary that contains only the fruits (as keys) and their number of characters (as values) for fruits with exactly 5 characters
5. make a list that contains fruits that have less than 5 characters
6. make a set that contains only the fruits that contain the letter "i"
7. make a list that holds only the even and positive numbers
8. make a set that holds only the odd and negative numbers
9. make a dictionary that contains only the numbers (as keys) and their squares (as values) that are both odd and negative

Библиотека NumPy

NumPy (Numerical Python) — это библиотека Python с поддержкой многомерных массивов (включая матрицы) и математических функций, предназначенных для работы с многомерными массивами. NumPy является основой для многих других библиотек Python, в т.ч. библиотек машинного обучения.

Для использования NumPy импортируем библиотеку командой `import`.

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

Создание объекта ndarray

Объект `ndarray` (многомерный массив) может быть создан из списка или кортежа.

Объект `ndarray` имеет такие свойства как:

• `ndim` – ранг (число осей или измерений) массива

• `shape` – кортеж, показывающий длину массива по каждой из осей (размеры массива). Для матрицы из `n` строк и `m` столбов, свойство shape равно `(n,m)`. Число элементов в кортеже `shape` равно рангу массива, то есть свойству `ndim`.

• `size` – число элементов в массиве, равное произведению всех элементов кортежа `shape`.

• `dtype` – тип элементов массива. Тип `dtype` можно определить явно, используя стандартные типы данных Python или типы, предусмотренные библиотекой NumPy, например, `dtype=np.int8`. Тип `dtype` также может быть составным, например, целое число и число с плавающей точкой.

In [ ]:
```
oneDim = np.array([1,2,3,4,5],dtype=np.float16) # одномерный массива (вектор)
print(oneDim)
print("Ранг =", oneDim.ndim)
print("Размеры =", oneDim.shape)
print("Число элементов =", oneDim.size)
print("Тип массива =", oneDim.dtype)
```

In [ ]:
```
twoDim = np.array([[1,2],[3,4],[5,6],[7,8]]) # двумерный массив (матрица)
print(twoDim)
print("Ранг =", twoDim.ndim)
print("Размеры =", twoDim.shape)
print("Число элементов =", twoDim.size)
print("Тип массива =", twoDim.dtype)
```

In [ ]:
```
arrFromTuple = np.array([(1,'a',3.0),(2,'b',3.5)]) # создание ndarray из кортежа
print(arrFromTuple)
print("Ранг =", arrFromTuple.ndim)
print("Размеры =", arrFromTuple.shape)
print("Число элементов =", arrFromTuple.size)
print("Тип массива =", arrFromTuple.dtype)
```

В NumPy имеется ряд встроенных функций для создания объектов `ndarray`.

In [ ]:
```
np.random.rand(5)          # случайные числа с равномерным распределением на [0,1]
```

In [ ]:
```
np.random.randn(5)         # случайные числа со стандартным нормальным распределением
```

In [ ]:
```
np.arange(-10,10)        # аналогично range, возвращается объект ndarray вместо списка
```

In [ ]:
```
np.linspace(0,1,11)        # разбиение интервала [0,1] на 11 равноудаленных значений
```

In [ ]:
```
np.zeros((2,3))            # матрица из нулей
```

In [ ]:
```
np.ones((3,2),dtype=int)   # матрица из единиц
```

In [ ]:
```
np.eye(4,dtype=np.int16)   # единичная матрица размером 4 x 4 (со значениями типа np.int1
```

Поэлементные операции

Можно применять стандартные операции, такие как сложение и умножение, к каждому элементу объекта `ndarray`, при этом создается новый массив, который заполняется

результатами действия оператора.

In [ ]:
```
x = np.array([1,2,3,4,5]) # ndarray из списка Python

print('x + 1 =', x + 1)      # сложение
print('x - 1 =', x - 1)      # вычитание
print('x * 2 =', x * 2)      # умножение
print('x // 2=', x // 2)     # целая часть от деления
print('x ** 2=', x ** 2)     # возведение в квадрат
print('x % 2 =', x % 2)      # остаток от деления
print('1 / x =', 1 / x)      # деление
```

In [ ]:
```
x = np.array([2,4,6,8,10])
y = np.array([1,2,3,4,5])

print('x + y =', x + y)
print('x - y =', x - y)
print('x * y =', x * y)
print('x / y =', x / y)
print('x // y=', x // y)
print('x ** y=', x ** y)
```

Если требуется создать пользовательскую функцию, которая может быть применена к элементам массива, то можно воспользоваться функцией `vectorize()` из NumPy:

In [ ]:
```
add5 = lambda z: z + 5
vect_add5 = np.vectorize(add5)
vect_add5(x)
```

Индексы и срезы

Существуют различные способы выбрать определенные элементы в массиве `ndarray`. Нумерация элементов в массиве (как и в других структурах данных) начинается с нуля.

In [ ]:
```
x = np.arange(-5,5)
print(x)
```

In [ ]:
```
y = x[3:5]     # y - это срез или указатель на подмассив в x
print("y =",y)
y[:] = 1000    # изменение y приведет к изменению x
print("y =",y)
print("x =",x)
```

In [ ]:
```
z = x[3:5].copy()   # делаем копию подмассива
print("z =",z)
z[:] = 500          # изменение z не влияет на x
print("z =",z)
print("x =",x)
```

Срез массива представляет собой представление тех же самых данных (ссылку на данные), а для создания копии данных массива можно воспользоваться методом `copy()`.

Изменение формы массива (метод `reshape`)

Приведем три разных метода изменения формы массива:

In [ ]:
```
X = np.floor(10 * np.random.randn(3,4)) # случайная матрица 3 x 4
X
```

In [ ]:
```
X.ravel() # выпрямление массива в одномерный из 12 элементов
```

In [ ]:
```
X.flatten() # то же самое
```

In [ ]:
```
X.reshape(2, 6) # преобразование в матрицу 2 x 6
```

In [ ]:
```
X.T # транспонирование в матрицу 4 x 3
```

Вместо метода `reshape()`, который возвращает модифицированный массив, можно использовать метод `resize()`, который модифицирует сам массив:

In [ ]:
```
X.resize((6, 2))
X
```

Если в операции изменения формы указано значение -1, то фактическое значение вычисляется автоматически:

In [ ]:
```
X.reshape(4, -1)
```

На практике достаточно часто метод `reshape()` применяется для перехода от матрицы с одним столбцом/строкой к вектору и обратно:

In [ ]:
```
X.reshape(-1) # одномерный вектор
```

In [ ]:
```
X.reshape(-1, X.size) # матрица размерами 1 x 12
```

In [ ]:
```
X.reshape(X.size, -1) # матрица размерами 12 x 1
```

Транслирование массивов (broadcasting)

Транслирование (broadcasting) — набор правил, которые позволяют применять функции из NumPy для выполнения бинарных операций (таких как сложение, вычитание, умножение и т. д.) с массивами разных форм и размеров. Например, добавление скалярного значения к массиву можно рассматривать как транслирование, когда скалярное значение дублируется в массив:

In [ ]:
```
a = np.array([0, 1, 2])
a + 5
```

Посмотрите на результат сложения одномерного и двумерного массивов:

In [ ]:
```
M = np.ones((3, 3))
M
```

In [ ]:
```
M + a
```

Здесь одномерный массив `a` растягивается (транслируется) на второе измерение, чтобы соответствовать форме массива `M`.

Более сложные случаи могут включать транслирование обоих массивов:

In [ ]:
```
a = np.arange(3)
a
```

In [ ]:
```
b = np.arange(3).reshape((-1,1))
# b = np.arange(3)[:, np.newaxis]
b
```

In [ ]:
```
a + b
```

Транслирование в библиотеке NumPy следует набору правил, определяющему взаимодействие двух массивов:

- **Правило 1**: если размерности двух массивов отличаются, то форма массива с меньшей размерностью дополняется единицами с ведущей (левой) стороны.
- **Правило 2**: если формы двух массивов не совпадают в каком-то измерении, то массив с формой, равной 1 в данном измерении, растягивается вплоть до соответствия форме другого массива.
- **Правило 3**: если в каком-либо измерении размеры массивов различаются и ни один не равен 1, то генерируется ошибка.

Объединение массивов

Массивы могут быть объединены по вертикали (при помощи функции `vstack`) и по горизонтали (при помощи функции `hstack`):

In [ ]:
```
X1 = np.arange(1, 10).reshape(3, 3)
X1
```

In [ ]:
```
X2 = np.arange(11, 20).reshape(3, 3)
X2
```

In [ ]:
```
np.vstack((X1, X2))
```

In [ ]:
```
np.hstack((X1, X2))
```

Индексирование массивов

Объект `ndarray` допускает **индексирование массивами** или списками целых чисел (прихотливое индексирование, fancy indexing):

In [ ]:
```
X = np.arange(10)**2
X
```

In [ ]:
```
idx = np.array([2, 3, 2, 8, 6]) # одномерный массив индексов (начиная с нуля)
X[idx]
```

In [ ]:
```
idx2 = np.array([[2, 3], [8, 6]]) # двумерный массив индексов (начиная с нуля)
X[idx2]
```

Объект `ndarray` также поддерживает **булево индексирование**:

In [ ]:
```
X = np.arange(1,13,1).reshape(3,4)
X
```

In [ ]:
```
X_cond = X > 5
X_cond
```

In [ ]:
```
X[X_cond]
```

In [ ]:
```
X[X_cond] = 0
X
```

Арифметические и статистические функции Numpy

В NumPy встроены многие математические функции для манипулирования элементами `ndarray`.

In [ ]:
```
y = np.array([-1.4, 0.4, -3.2, 2.5, 3.4])    # массив чисел
y
```

In [ ]:
```
np.abs(y)          # абсолютные значения
```

In [ ]:
```
np.sqrt(abs(y))    # квадратный корень из абсолютных значений
```

In [ ]:
```
np.sign(y)         # знак каждого элемента
```

In [ ]:
```
np.exp(y)          # экспонента
```

In [ ]:
```
np.sort(y)         # сортировка
```

In [ ]:
```
y = np.array([-3.2, -1.4, 0.4, 2.5, 3.4])    # массив чисел
print(y)

print("Минимум  =", np.min(y))
print("Максимум =", np.max(y))
print("Среднее  =", np.mean(y))
print("Стандартное отклонение =", np.std(y))
print("Сумма =", np.sum(y))
```

Когда функции `max()`, `min()` и др. применяются к двумерному массиву, то они могут быть применены ко всем элементам, к строкам или столбцам:

In [ ]:
```
X = np.array([[1,2,3],[4,5,6],[7,8,9]])
X
```

In [ ]:
```
np.max(X) # максимум по всем элементам
```

In [ ]:
```
np.max(X, axis=0) # максимум по столбцам
```

In [ ]:
```
np.max(X, axis=1) # максимум по строкам
```

Линейная алгебра в Numpy

Numpy поддерживает многие операции линейной алгебры, например, транспонирование матрицы и матричное умножение.

In [ ]:
```
X = np.random.randn(2,4) # случайная матрица 2 x 4
print(X)
print(X.T)               # транспонирование матрицы X^T
```

In [ ]:
```
y = np.random.randn(4) # случайный вектор
y
```

In [ ]:
```
X.dot(y)        # умножение матрицы на вектор   X * y
```

In [ ]:
```
X @ y           # альтернативный вариант умножения X * y
```

In [ ]:
```
X.dot(X.T)      # умножение матрицы на матрицу  X * X^T
```

In [ ]:
```
X.T.dot(X)      # умножение матрицы на матрицу  X^T * X
```

Также поддерживается вычисление определителя, обратной матрицы, собственных значений и собственных векторов матрицы.

In [ ]:
```
X = np.random.randn(5,3)     # матрица 5 x 3
X
```

In [ ]:
```
C = X.T.dot(X)               # C = X^T * X - это квадратная матрица
C
```

In [ ]:
```
C.diagonal() # диагональные элементы матрицы
```

In [ ]:
```
C.trace() # след - сумма диагональных элементов
```

In [ ]:
```
C.diagonal(offset=1) # элементы со смещением вверх от главной диагонали
```

In [ ]:
```
C.diagonal(offset=-1) # элементы со смещением вниз от главной диагонали
```

In [ ]:
```
invC = np.linalg.inv(C)      # обратная матрица
invC
```

In [ ]:
```
detC = np.linalg.det(C)      # определитель
detC
```

In [ ]:
```
S, U = np.linalg.eig(C)      # собственные значения S и собственные вектора U
print('S =',S)
print('U =',U)
```

In [ ]:
```
np.linalg.matrix_rank(X) # ранг матрицы
```

Сравним скорость выполнения операции сложения векторов в Python (для списков) и в NumPy (для одномерных массивов):

In [ ]:
```python
import time
import numpy as np

size_of_vec = 1000000 # один миллион элементов

# версия на чистом Python
start_time_python = time.time()
x_python = list(range(size_of_vec))
y_python = list(range(size_of_vec))
z_python = [x_python[i] + y_python[i] for i in range(len(x_python))]
end_time_python = time.time()
time_python = end_time_python - start_time_python

# версия NumPy
start_time_numpy = time.time()
x_numpy = np.arange(size_of_vec)
y_numpy = np.arange(size_of_vec)
z_numpy = x_numpy + y_numpy
end_time_numpy = time.time()
time_numpy = end_time_numpy - start_time_numpy

print(f"Время для версии на Python: {time_python:.6f} сек.")
print(f"Время для версии на NumPy:  {time_numpy:.6f} сек.")
print(f"NumPy примерно в {time_python / time_numpy:.2f} раз быстрее.")
```

Тест на использование NumPy

Пусть **N** – последняя цифра суммы двух последних цифр Вашего студенческого билета. Решите задание **N**, используя средства библиотеки NumPy, в течение __ минут и направьте программный код и результат его выполнения через Мессенджер Яндекса на адрес `shorokhov_sg@pfur.ru`

0. Write a NumPy program to create a 3x3 identity matrix and stack it vertically and horizontally.
1. Write a NumPy program to create a 4x4 array with random values and find the sum of each row.
2. Write a NumPy program to create a 5x5 array with random values and subtract the mean of each row from each element.
3. Write a NumPy program to create a 3x3 array with random values and subtract the mean of each column from each element.
4. Write a NumPy program to create a 4x4 array with random values and sort each row.
5. Write a NumPy program to create a 5x5 array with random values and sort each column.
6. Write a NumPy program to create a 4x4 array with random values and find the second-largest value in each row.
7. Write a NumPy program to create a 5x5 array with random values and find the second-largest value in each column.
8. Write a NumPy program to create a 3x3 array with random values and replace the maximum value with 0.
9. Write a NumPy program to create a 4x4 array with random values and replace the minimum value with 100.

Библиотека Pandas

Библиотека Pandas содержит две удобных структуры данных для хранения и манипуляций с данными – объекты `Series` и `DataFrame`. Объект `Series` подобен одномерному массиву, в то время, как объект `DataFrame` более напоминает матрицу или таблицу.

In [ ]:
```
import pandas as pd
```

Объект Series

Объект `Series` состоит из одномерного массива значений (свойство `values`), к элементам которого можно обращаться при помощи массива индексов (свойство `index`). Все элементы в объекте `Series` **имеют один и тот же тип**. Объект `Series` может быть создан из списка, массива NumPy или словаря Python.

In [ ]:
```
s = pd.Series([2000, 2004, 2008, 2012, 2016, 2020]) # объект Series из списка
s
```

In [ ]:
```
s.values, s.index
```

Массив индексов объекта `Series` может быть динамически изменен на другой:

In [ ]:
```
s.index = range(27,33)
s
```

In [ ]:
```
s = pd.Series([3.1, 2.4, -1.7, 0.2, -2.9, 4.5])   # объект Series из списка
print(s)
print('Values =', s.values)     # значения объекта Series
print('Index =', s.index)       # индексы объекта Series
```

In [ ]:
```
s2 = pd.Series(data=np.random.randn(6))  # объект Series из массива numpy
s2
```

In [ ]:
```
lst = [[10,20,30]]
lst*3
```

In [ ]:
```
s3 = pd.Series(data = lst*6,
               index = ['Jan 1','Jan 2','Jan 3','Jan 4','Jan 5','Jan 6',])
s3
```

In [ ]:
```
capitals = {'MI': 'Lansing', 'CA': 'Sacramento', 'TX': 'Austin', 'MN': 'St Paul'}

s4 = pd.Series(capitals)   # объект Series из словаря
s4
```

Доступ к элементам объекта `Series` возможен по номеру объекта или по ключу:

In [ ]:
```
print('\ns3[2]=', s3[2])             # доступ по номеру
print('s3[\'Jan 3\']=', s3['Jan 3']) # доступ по индексу

print('\ns3[1:3]=')                  # срез объекта Series
print(s3[1:3])
```

Большинство функций NumPy могут применяться к объекту `Series`.

In [ ]:
```
print('shape =', s2.shape)  # размеры Series
print('size =', s2.size)    # число элементов в Series
```

In [ ]:
```
print(s2[s2 < 0])   # фильтр для выбора элементов Series
```

In [ ]:
```
print(s2 + 4)       # операции числового объекта Series со скалярными величинами
print(s2 / 4)
```

In [ ]:
```
print(np.log(s2 + 4))    # функция numpy от числового объекта Series
```