📓 Введение в NumPy: что это и зачем нужно

🔍 Что такое NumPy?

NumPy (от *Numerical Python*) — это фундаментальная библиотека Python для научных вычислений, инженерных задач и анализа данных. Она предоставляет высокооптимизированную структуру данных для хранения и обработки многомерных числовых массивов, а также обширный набор математических функций, реализованных на уровне C и Fortran.

Библиотека была создана в 2005 году как объединение проектов `Numeric` и `Numarray` и с тех пор стала **де-факто стандартом** для любых числовых вычислений в Python.

📦 Главный объект: `ndarray`

Основа NumPy — объект `numpy.ndarray` (n-dimensional array). Это:

- **Однородный массив**: все элементы имеют один тип данных (`int32`, `float64`, `bool` и т.д.)
- **Фиксированного размера**: изменение размера создаёт новый массив
- **Непрерывный в памяти**: данные хранятся в одном блоке RAM, а не как набор разрозненных Python-объектов
- **Поддержка N измерений**: вектор (1D), матрица (2D), тензор (3D+)

```
import numpy as np

a = np.array([1, 2, 3, 4])  # 1D-массив (вектор)
b = np.array([[1, 2], [3, 4]])  # 2D-массив (матрица)
```

⚡ Ключевые преимущества NumPy

Преимущество	Суть	Почему это важно
🚀 **Векторизация операций**	Применение функций ко всему массиву сразу без циклов `for`	Код работает в 10–100× быстрее, выглядит как математическая формула
💾 **Эффективная работа с памятью**	Непрерывное хранение + фиксированные типы данных	Меньше накладных расходов, предсказуемое потребление RAM
🔄 **Broadcasting (трансляция)**	Автоматическое согласование форм массивов при операциях	Позволяет складывать/умножать массивы разных размерностей без явного копирования

Преимущество	Суть	Почему это важно
📐 **Богатый математический арсенал**	Линейная алгебра, статистика, тригонометрия, Фурье, случайные числа	Заменяет ручную реализацию алгоритмов, снижает риск ошибок
🌍 **Экосистемная совместимость**	Основа для `pandas`, `scipy`, `matplotlib`, `scikit-learn`, `torch`, `tensorflow`	Без NumPy невозможен современный стек Data Science и ML

Самое главное - это быстро!

```
import numpy as np
import time

N = 10**6

# Чистый Python
t0 = time.perf_counter()
py_list = list(range(N))
py_result = 0
for x in py_list:
  py_result += x*x
py_time = time.perf_counter() - t0

# NumPy
t0 = time.perf_counter()
np_arr = np.arange(N)
np_result = np.sum(np_arr**2)
np_time = time.perf_counter() - t0

print(f"⏱️ Python : {py_time:.4f} сек")
print(f"⏱️ NumPy  : {np_time:.4f} сек")
print(f"🔥 Ускорение: ~{py_time/np_time:.0f}x")
print(f"✅ Результаты совпадают: {abs(py_result - np_result) < 1e-9}")
```

> ⏱️ Python : 0.2478 сек ⏱️ NumPy : 0.0086 сек 🔥 Ускорение: ~29x ✅ Результаты совпадают: True

📓 Блок 0. Начало работы

1. Установить NumPy

```
pip install numpy
```

2. Использовать (импортировать)

```
import numpy as np
```

📓 Блок 1. Создание и работа с векторами (1D-массивы)

🔍 Что такое вектор в NumPy?

В контексте NumPy **вектор** — это одномерный массив (`ndarray` с `ndim == 1`). Это не математический объект в строгом смысле, а **линейная структура данных фиксированной длины**, хранящая элементы одного типа в непрерывном блоке памяти.

🛠️ Ключевые функции создания векторов

Функция	Назначение	Пример
`np.array(<массив>)`	Из списка/кортежа	`np.array([1, 2, 3])`
`np.arange(<от>, <до (не включительно)>, <шаг>)`	Диапазон с шагом	`np.arange(0, 10, 2)` → `[0, 2, 4, 6, 8]`
`np.linspace(<от>, <до (включительно)>, <кол-во точек>)`	Равномерная сетка (вкл. границы)	`np.linspace(0, 1, 5)` → `[0., 0.25, 0.5, 0.75, 1.]`
`np.zeros(<размер>)` / `np.ones()`	Заполнение нулями/единицами	`np.zeros(4)`
`np.full(<размер>, <чем заполнять>)`	Заполнение определенным значением	`np.full(3, 7.5)`
`np.random.randint(<от>, <до (не включительно)>, size=<размер>)`	Случайные целые числа	`np.random.randint(1, 100, size=5)`
`np.random.random(<размер>)`	Случайные числа из равномерного распределения на `[0.0, 1.0)`	`np.random.randint(5)`

📐 Атрибуты вектора

```
v = np.array([10, 20, 30, 40], dtype=np.float32)
print(v.ndim)   # 1 (измерение)
print(v.shape)  # (4,) (обратите внимание: кортеж из 1 элемента)
print(v.size)   # 4 (общее количество элементов)
print(v.dtype)  # float32
print(v.nbytes) # 16 (4 элемента × 4 байта)
```

💮 Задания 1

1.1. Создайте вектор `v1` из списка `[5, 10, 15, 20.0, 25]`. Выведите его `.dtype`.

1.2. Создайте `v2` - вектор чисел от `0` до `30` с шагом `3`.

1.3. Создайте вектор `v3` из 15 случайных целых чисел в диапазоне `[10, 50)`. Выведите для него `.ndim`, `.size`, `.shape`.

1.4. Создайте вектор `v4` длиной 8, полностью заполненный числом `-1`.

1.5. Сгенерируйте вектор `v5` из 10 чисел, равномерно распределённых на отрезке `[-5, 5)`.

📓 Блок 2. Индексация и срезы

🔍 Теория: Как устроен доступ к элементам

1️⃣ Базовая индексация

```
v = np.array([10, 20, 30, 40, 50])
print(v[0])    # 10  (первый элемент)
print(v[-1])   # 50  (последний)
print(v[2])    # 30
v[3] = 99      # Замена элемента на месте array([10, 20, 30, 99, 50])
```

2️⃣ Срезы (Slicing): `start:stop:step`

Возвращает **представление (view)**, а не копию. Изменение среза изменяет исходный массив.

```
v = np.arange(10)           # [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
print(v[2:7])               # [2, 3, 4, 5, 6]
print(v[::2])               # [0, 2, 4, 6, 8]  (каждый второй)
print(v[::-1])              # [9, 8, ..., 0]   (разворот)
a = v[1:4]
a[:] = -1
print(v)                    # [0, -1, -1, -1, 4, 5, 6, 7, 8, 9]
```

3 Продвинутая (Fancy) индексация

Передача массива индексов.

```
idx = np.array([0, 3, 4])
print(v[idx])               # Элементы на позициях 0, 3, 4 > [ 0 -1  4]
print(v[[0,3,4]])
v[idx] = [100, 200, 300]    # Замена по списку позиций
print(v)                    # [100, -1, -1, 200, 300, 5, 6, 7, 8, 9]
```

4 Булева индексация (Маски)

Самый мощный инструмент для **векторизованной фильтрации**.

```
v = np.array([5, 12, 3, 8, 15, 2])
mask = v > 7                # [False,  True, False,  True,  True, False]
print(v[mask])              # [12,  8, 15]
v[v < 5] = 0                # Замена по условию: [5, 12, 0, 8, 15, 0]
```

5 `np.where` для условного выбора

- `np.where(condition)` → возвращает кортеж индексов, где условие истинно
- `np.where(condition, x <если да>, y <иначе>)` → векторизованный `if-else`

```
v = np.array([1, 5, 3, 8, 2])
indices = np.where(v > 3)[0]
print(indices)                  # [1, 3]
result = np.where(v > 3, v * 10, v)
print(result)                   # [1, 50, 3, 80, 2]
```

Задания 2

2.1. Извлеките срез элементов `v2` с индексами `2` по `6` (не включая `6`).

2.2. Получите каждый третий элемент массива `v5`.

2.3. Разверните массив `v2` с помощью среза.

2.4. Замените элементы массива `v4` с индексами `1`, `4`, `7` на значение `99`.

2.5. Создайте массив из 12 нулей. С помощью среза запишите `5` в позиции `3:9`.

2.6. Получите первые 4 и последние 4 элемента массива `np.arange(20)`.

2.7. Для массива `v3` используйте `np.where`, чтобы заменить значения `< 25` на `-1`, а `≥ 25` на `1`.

2.8. Замените все элементы массива `v5` в диапазоне индексов `[5:10]` на их среднее арифметическое.

📓 Блок 3. Векторизация операций

(Все задания и примеры строго запрещают использование `for`/`while`. Цель: научиться мыслить массивами.)

🔍 Теория: Что такое векторизация?

Векторизация — Векторизация — это стиль программирования, при котором операции применяются ко всему массиву одновременно, а не к каждому элементу в цикле `for`.

🛠️ Основные механизмы векторизации

1️⃣ Поэлементные арифметические операции

Операторы `+`, `-`, `*`, `/`, `**`, `%`, `//` применяются к каждой паре элементов с одинаковыми индексами.

```
x = np.array([1, 2, 3])
y = np.array([10, 20, 30])
print(x + y)   # [11, 22, 33]
print(x * y)   # [10, 40, 90]
print(x ** 2)  # [1, 4, 9]
```

2️⃣ Универсальные функции (ufuncs)

Встроенные функции, оптимизированные для массивов:

Функция	**Назначение**
`np.sqrt()`, `np.square()`	Корень / Квадрат
`np.exp()`, `np.log()`	Экспонента / Натуральный логарифм
`np.sin()`, `np.cos()`, `np.tan()`	Тригонометрия
`np.abs()`	Модуль
`np.round()`	Округление
`np.min()`, `np.argmin()`, `np.max()`, `np.argmax()`,	Максимум / минимум
`np.sum()`, `np.mean()`, `np.std()`	Сумма, среднее, среднеквадратичное отклонение

3️⃣ Сравнение массивов и логические операции

Возвращают **булевы массивы**. Комбинируются побитовыми операторами `&` (AND), `|` (OR), `~` (NOT), `^` (XOR). ⚠️ **Важно:** Условия обязательно брать в скобки из-за приоритета операторов.

```
a = np.array([1, 5, 3, 8])
mask = (a > 2) & (a < 7)  # [False,  True,  True, False]
print(a[mask])            # [5, 3]
print(np.all(a > 0))      # True  (все элементы > 0)
print(np.any(a == 8))     # True  (хотя бы один == 8)
```

🧩 Задания 3

Явные циклы `for`/`while` запрещены.

3.1. Создайте два одномерных массива `a = [1, 2, 3]` и `b = [4, 5, 6]`. Выполните три независимые поэлементные операции:

- Сложение соответствующих элементов
- Умножение соответствующих элементов
- Возведение каждого элемента `a` в степень соответствующего элемента `b` Выведите три полученных массива.

3.2. Дан массив `[1, 4, 9]`. Вычислите для каждого его элемента квадратный корень и экспоненту. Используйте специализированные математические функции библиотеки.

3.3. Сгенерируйте 5 чисел, равномерно распределённых на отрезке `[0, pi]`. Вычислите синус для каждой из этих точек. Выведите исходные точки и их синусы.

3.4. Даны массивы `x = [1, 2, 3]` и `y = [3, 2, 1]`. Проверьте поэлементное условие: `x[i] > y[i]`. Выведите полученный массив логических значений (`True`/`False`).

3.5. Сгенерируйте вектор из 8 случайных целых чисел в диапазоне `[-10, 10]`. Замените все отрицательные значения на их абсолютные величины, не трогая неотрицательные элементы. Реализуйте замену одной векторизованной операцией.

3.6. Округлите значения массива `[1.234, 2.567, 3.999]` до двух знаков после запятой, используя встроенную функцию округления для массивов.

3.7. Сгенерируйте два случайных вектора длиной 10. Вычислите их поэлементное среднее арифметическое: `(a_i + b_i)/2` для всех `i`. Выведите исходные массивы и результат.

3.8. Дан произвольный вектор целых чисел. Отфильтруйте из него только те элементы, которые одновременно строго больше `2` и строго меньше `8`. Используйте логическую операцию «И» для формирования условия.

📓 Блок 4. Матрицы (2D-массивы): форма, оси, транспонирование и операции

🔍 Теория: Матрицы в NumPy

1️⃣ Создание и базовые атрибуты

Матрица — это `ndarray` с `ndim == 2`. Форма задаётся кортежем `(строки, столбцы)`.

```
M = np.array([[1, 2, 3],
              [4, 5, 6]])
print(M.shape)  # (2, 3)
print(M.size)   # 6
print(M.ndim)   # 2
```

Быстрое создание:

Функция	Пример
`np.zeros((m,n))`	Нулевая матрица `m×n`
`np.eye(n)`	Единичная матрица `n×n`
`np.diag([1,2,3])`	Диагональная матрица
`np.random.rand(m,n)`	Равномерные числа `[0, 1)`
`np.arange(12).reshape(3,4)`	Последовательность → матрица

2️⃣ Изменение формы (`reshape`, `flatten`, `ravel`)

- `reshape(new_shape)` меняет форму **без копирования данных** (возвращает `view`), если возможно.
- Магическое `-1`: NumPy сам вычислит недостающую размерность.
- `flatten()` → упрощает массив в вектор, всегда возвращает **копию** (1D).
- `ravel()` → упрощает массив в вектор, возвращает `view`, если возможно (быстрее).

```
A = np.arange(12).reshape(3, 4)      # (3, 4)
print(A)                             # [[ 0  1  2  3]
                                     #  [ 4  5  6  7]
                                     #  [ 8  9 10 11]]

B = A.reshape(-1, 2)                 # (6, 2) (-1 → 12/2 = 6)
B[0, 0] = 999                        # A[0, 0] тоже изменится! (view)
print(A)                             # [[999   1   2   3]
                                     #  [  4   5   6   7]
                                     #  [  8   9  10  11]]

print(A.ravel())                     # [999   1   2   3   4   5   6   7   8   9  10  11]
print(A.flatten())
```

3️⃣ Транспонирование и оси

- `.T` или `np.transpose()` меняет `(m, n)` → `(n, m)`. **Всегда view.**
- **Оси в 2D:**
  - `axis=0` → движение **вниз** (по столбцам). Агрегация "схлопывает" строки.
  - `axis=1` → движение **вправо** (по строкам). Агрегация "схлопывает" столбцы.

```
M = np.array([[10, 20],
              [30, 40],
              [50, 60]])  # shape (3, 2)
print(M.T.shape)          # shape (2, 3)
print(M.T)                # [[10 30 50]
                          #  [20 40 60]]
M.sum(axis=0)  # [90, 120]  (сумма по столбцам)
M.sum(axis=1)  # [30, 70, 110]  (сумма по строкам)
```

4 Поэлементные vs Матричные операции

Оператор	Поведение	Пример
`*`, `+`, `-`, `/`	Поэлементно (Hadamard)	`A * B` требует одинаковой формы
`@`, `np.matmul()`	Матричное умножение	`A @ B`: `(m×k) @ (k×n) → (m×n)`
`np.dot()`	Для 2D эквивалентно `@`, для 1D → скалярное произведение	`np.dot(A, B)`

⚠ **Главная ловушка:** `A * B` ≠ `A @ B`. Первое умножает соответствующие элементы, второе выполняет линейную алгебру.

5 Объединение и разделение

Функция	Поведение
`np.vstack([A, B])`	Вертикальное склеивание (`axis=0`)
`np.hstack([A, B])`	Горизонтальное склеивание (`axis=1`)
`np.concatenate([A, B], axis=1)`	Аналог `hstack`, но явная ось
`np.split(M, indices_or_sections, axis=0)`	Разделение по оси

```
A = np.array([[1, 2], [3, 4]])        # (2, 2)
B = np.array([[5, 6], [7, 8]])        # (2, 2)

V = np.vstack([A, B])              # (4, 2) — требует совпадения столбцов
H = np.hstack([A, B])              # (2, 4) — требует совпадения строк
C = np.concatenate([A, B], axis=1)  # (2, 4) — полный аналог hstack
S = np.split(V, 2, axis=0)
# [(2,2), (2,2)] — режет V на 2 равные части

print("vstack:\n", V)
# [[1 2]
#  [3 4]
#  [5 6]
#  [7 8]]

print("hstack:\n", H)
# [[1 2 5 6]
#  [3 4 7 8]]

print("split → список view-массивов:")
for i, part in enumerate(S):
    print(f"  Часть {i}:\n{part}")
#  Часть 0: [[1 2] [3 4]]
#  Часть 1: [[5 6] [7 8]]
```

🧩 Задания 4

4.1. Создайте матрицу `m1` размера `3×4` из последовательных чисел `0..11`. Выведите `.shape` и `.ndim`.

4.2. Транспонируйте матрицу `m1`.

4.3. Создайте матрицу `2×6` случайных чисел и преобразуйте её в `4×3`.

4.4. Вычислите сумму элементов матрицы `m1` по столбцам и по строкам. Сравните длины результатов.

4.5. Создайте единичную матрицу `me1 3×3` и добавьте к ней матрицу из случайных чисел `3×3`.

4.6. Извлеките главную диагональ матрицы `me1`.

4.7. Замените всю вторую строку матрицы `me1` на нули.

4.8. Любым способом создайте две матрицы `2×3` и объедините их вертикально и горизонтально. Найдите максимальный элемент в каждой строке полученных матриц.

4.9. Создайте матрицу `4×4` и разделите её на четыре блока `2×2` с помощью срезов `[0:2, 0:2]` и т.д.

4.10. Вычислите среднее значение каждой строки, затем вычтите эти средние из исходных строк (центрирование 4.строк).

📓 Блок доп. Для любопытных. (не обязательный)

Broadcasting в NumPy (🔧 Фундамент векторизации)

🔍 Простое объяснение: что такое Broadcasting?

Broadcasting (трансляция) — это механизм NumPy, который автоматически "растягивает" массивы разных форм до совместимых размеров, чтобы выполнить поэлементную операцию.

📐 3 золотых правила Broadcasting

NumPy проверяет совместимость **справа налево** по каждой размерности:

Правило	Условие	Пример
1. Выравнивание	Дополняем формы слева единицами до одинаковой длины осей	`(3,)` → `(1, 3)`, `()` → `(1, 1)`
2. Совместимость	Размерности равны **или** одна из них равна `1`	`(4, 3)` и `(1, 3)` ✅, `(4, 3)` и `(4, 2)` ❌
3. Результат	Берётся максимум по каждой оси	`(4, 1) + (1, 3)` → `(4, 3)`

🔍 Примеры проверки совместимости

Форма A	Форма B	Совместимы?	Результат

Форма A	Форма B	Совместимы?	Результат
`(5,)`	`()`	✅ (скаляр)	`(5,)`
`(3,4)`	`(4,)`	✅ `(4,)` → `(1,4)`	`(3,4)`
`(3,1)`	`(1,4)`	✅	`(3,4)`
`(2,3)`	`(3,2)`	❌	`ValueError`
`(100, 1)`	`(100,)`	✅ `(100,)` → `(1,100)` ❌ Нет! `(100,)` дополняется до `(1,100)`, но `100≠1` по оси 0 → ❌	`ValueError`

💡 **Лайфхак:** Всегда выводите `print(a.shape, b.shape)` перед операцией. Если видите `ValueError: operands could not be broadcast together` → проверяйте правила справа налево.

🛠️ Как управлять формой на практике

Чтобы заставить broadcasting работать, нужно явно добавить измерения:

```
v = np.array([1, 2, 3])          # shape (3,)

# Превращаем в столбец (3, 1)
col = v[:, np.newaxis]           # или v.reshape(-1, 1)

# Превращаем в строку (1, 3)
row = v[np.newaxis, :]           # или v.reshape(1, -1)
```

🗝️ 3 частых паттерна

1. **Нормализация столбцов:** `X - X.mean(axis=0)` → `(N, D) - (D,)` работает идеально
2. **Внешняя операция:** `col + row` → `(N, 1) + (1, M) = (N, M)` (матрица сумм/произведений)
3. **Поэлементное сравнение с порогом:** `arr > threshold` → `(N,) > ()` → булев массив

⚠️ Типичные ошибки

Ошибка	Почему возникает	Как исправить
`ValueError: shapes (3,) and (3,1) not aligned`	`(3,)` дополняется до `(1,3)`, а не `(3,1)`	Явно добавлять ось: `v[:, None]` или `v.reshape(-1,1)`
Ожидание `(n,) + (n,) → (n,n)`	Broadcasting не создаёт новые оси сам	Использовать `a[:, None] + b[None, :]`
Случайное растягивание в цикле	Неправильный `axis` при агрегации	Проверять `result.shape` после `mean/sum`

Ошибка	Почему возникает	Как исправить
`np.tile()` вместо broadcasting	Незнание механизма трансляции	Заменить `np.tile(v, (5,1))` на `v[None, :]` + операция

Задания по уровням сложности

Базовый уровень

1. **Сумма строки и столбца**
   Создайте `row = np.array([1, 2, 3])` и `col = np.array([[10], [20], [30]])`. Сложите их одной строкой. Выведите результат и `result.shape`. В комментарии укажите, как NumPy автоматически расширил формы `(1,3)` и `(3,1)` до `(3,3)`.

2. **Внешнее произведение через `np.newaxis`**
   Сгенерируйте `a = np.arange(4)` и `b = np.arange(5)`. Используя только `np.newaxis` (или `None`) и оператор `*`, вычислите матрицу `M`, где `M[i, j] = a[i] * b[j]`. Выведите `M` и убедитесь, что `M.shape == (4, 5)`.

3. **Масштабирование столбцов матрицы**
   Создайте случайную матрицу `mat = np.random.rand(3, 4)` и вектор множителей `scale = np.array([2.0, 0.5, 3.0, 1.5])`. Умножьте каждый столбец `mat` на соответствующий элемент `scale` без циклов. Проверьте: `print(mat[:, 0] / scale[0])` должно совпадать с `mat[:, 0] / 2.0`.

4. **Центрирование по столбцам**
   Создайте матрицу `A = np.arange(1, 26).reshape(5, 5).astype(float)`. Вычислите среднее каждого столбца и вычтите его из соответствующего столбца исходной матрицы. Выведите средние значения результата: `print(result.mean(axis=0))` (должно быть близко к `0.0`).

5. **Комбинирование трёх измерений**
   Создайте матрицу нулей `M = np.zeros((4, 3))`. Используя broadcasting, одновременно добавьте к ней вектор-строку `r = np.array([1, 2, 3])` и вектор-столбец `c = np.array([[10], [20], [30], [40]])`. Выведите итоговую матрицу. Проверьте элемент `[2, 1]`: он должен равняться `0 + 2 + 30 = 32`.

Средний уровень

6. **Z-score нормализация признаков**
   Дана матрица `X = np.random.randn(50, 4)` (признаки в столбцах). Приведите каждый столбец к распределению `N(0,1)` по формуле `(X - μ) / σ`. Обязательно используйте `keepdims=True` при вычислении среднего и std. Проверьте: `X_norm.mean(axis=0).round(3)` и `X_norm.std(axis=0).round(3)` должны быть `[0., 0., 0., 0.]` и `[1., 1., 1., 1.]`.

7. **Бинаризация по порогам строк**
   Для матрицы `A = np.random.rand(6, 4)` замените все элементы, превышающие среднее значение *своей строки*, на `1.0`, остальные на `0.0`. Используйте `keepdims=True` для сохранения 2D-формы порога и булеву индексацию. Выведите сумму единиц в каждой строке.

8. **Матрица попарных разностей**
   Дан вектор координат `pts = np.array([0, 5, 10, 15])`. Постройте матрицу `4×4`, где `D[i, j] = np.abs(pts[i] - pts[j])`, используя только broadcasting. Проверьте симметричность (`np.allclose(D, D.T)`) и нулевую диагональ (`np.allclose(np.diag(D), 0)`).

9. **Взвешивание строк с сохранением формы**
   Создайте матрицу `data = np.ones((10, 5)) * np.arange(1, 6)` и вектор весов `w = np.arange(1, 11)`. Умножьте каждую `i`-ю строку `data` на `w[i]`, сохранив исходную форму `(10, 5)`. Проверьте сумму первой строки до и после умножения.

10. **Безопасное деление с маскированием**
    Создайте матрицу `M = np.random.randint(1, 10, (5, 3)).astype(float)` и делитель `div = np.array([2.0, 0.0, 5.0])`. Выполните поэлементное деление `M / div`, заменив результаты деления на `0` на `np.nan` **без генерации предупреждений**. Используйте `np.divide(..., out=..., where=...)` или `np.where`. Проверьте наличие `NaN` только во втором столбце.