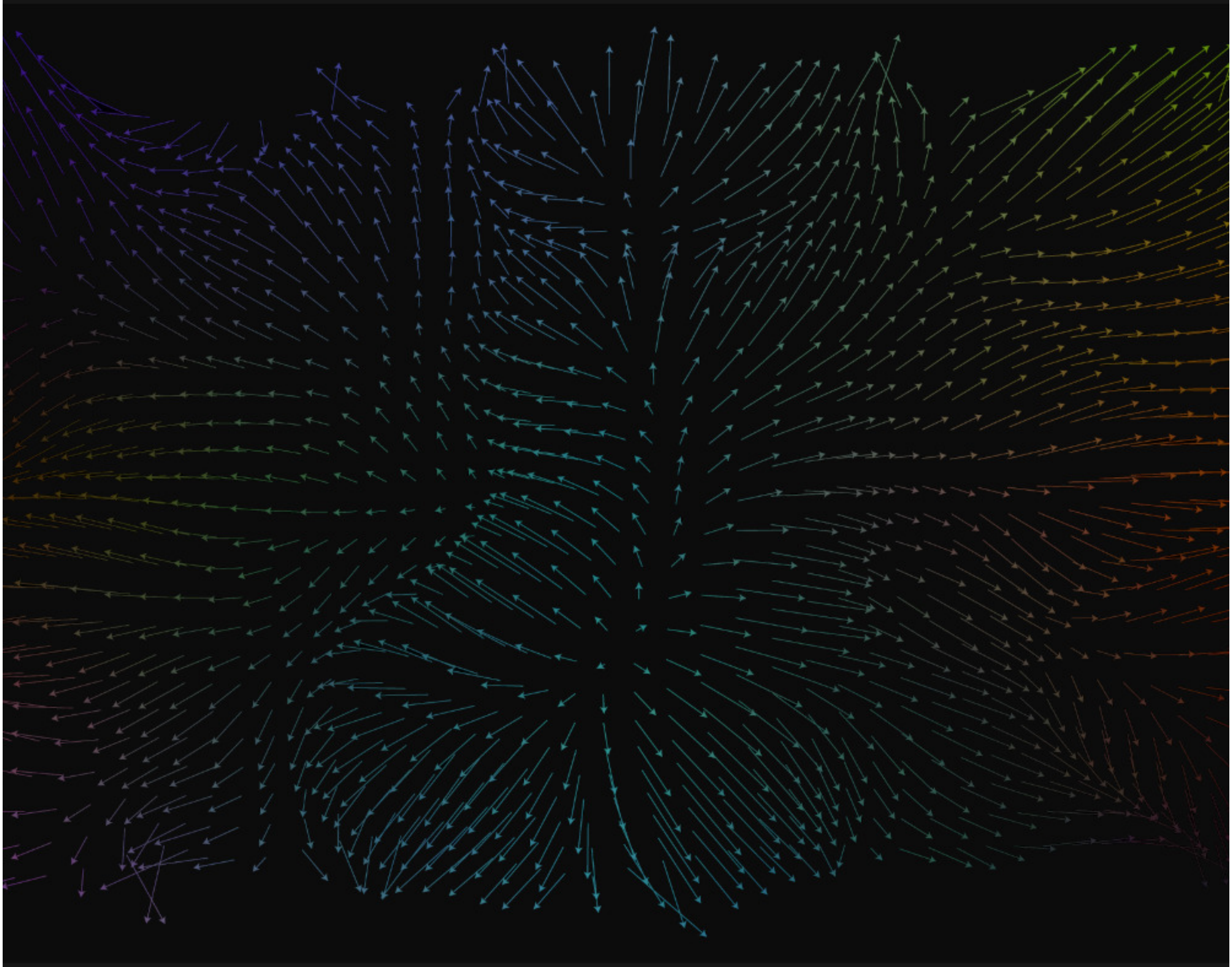
Python Series

# NumPy
# அறிமுகம்



முனைவர்

## 0.NumPy – அறிமுகம் மற்றும் வரலாறு

ஒரு சாதாரண காரில் பயணிப்பதற்கும், ஒரு ஃபார்முலா 1 பந்தயக் காரில் சீறிப் பாய்வதற்கும் உள்ள வித்தியாசத்தை உங்களால் உணர முடிகிறதா? Python நிரலாக்க மொழியில், அதன் வழக்கமான `list`-களைப் பயன்படுத்துவது சாதாரண காரில் பயணம் செய்வது போல. ஆனால், **NumPy** என்ற நூலகத்தைப் பயன்படுத்துவது, ஒரு பந்தயக் காரின் வேகத்தை உங்களுக்குக் கொடுப்பது போல!

**NumPy** (Numerical Python என்பதன் சுருக்கம்) என்பது Python மொழியில் அறிவியல் மற்றும் கணிதக் கணக்கீடுகளைச் செய்வதற்கான ஒரு திறந்த மூல (open-source) நூலகம். இது வெறும் ஒரு நூலகம் மட்டுமல்ல, Python-ஐ தரவு அறிவியல் (Data Science), இயந்திர கற்றல் (Machine Learning) போன்ற துறைகளின் முடிசூடா மன்னனாக மாற்றிய ஒரு புரட்சிகரமான கருவி. இதன் மையப் புள்ளி, அதன் சக்திவாய்ந்த **பன்முக அணி (multi-dimensional array)** அமைப்புதான்.

### 0.க. ஒரு புதிய கருவியின் தேவை

கதை 1990-களில் தொடங்குகிறது. அப்போது, அறிவியலாளர்களும் பொறியாளர்களும் Python-இன் எளிமையால் ஈர்க்கப்பட்டு, அதைத் தங்கள் ஆய்வுக் கணக்கீடுகளுக்குப் பயன்படுத்த விரும்பினர். ஆனால் ஒரு சிக்கல் இருந்தது. Python, இயல்பாகவே பெரிய அளவிலான எண்களைக் கொண்ட அணிகளைக் (arrays) கையாள்வதில் மிகவும் மெதுவாக இருந்தது. ஆயிரக்கணக்கான எண்களைக் கொண்ட ஒரு பட்டியலை (list) வைத்துக்கொண்டு கணிதச் செயல்பாடுகளைச் செய்வது, ஒரு ஆமை வேகப் பந்தயத்தில் ஓடுவது போல இருந்தது.

இந்தத் தடையைத் தகர்க்க, இரண்டு முக்கிய முயற்சிகள் முளைத்தன:

1. **Numeric:** இது NumPy-யின் முன்னோடி. முதன்முறையாக Python-ல் அணிகளைக் கொண்டு அறிவியல் கணக்கீடுகளைச் செய்ய வழிவகுத்தது. இது ஒரு மிகச்சிறந்த தொடக்கமாக இருந்தாலும், அதன் செயல்திறனிலும், நெகிழ்வுத்தன்மையிலும் சில குறைகள் இருந்தன.
2. **Numarray:** Numeric-இன் குறைகளைச் சரிசெய்யும் நோக்கில் இது உருவாக்கப்பட்டது. இது பெரிய அளவிலான தரவுகளைக் கையாள்வதில் சிறந்து விளங்கியது. ஆனாலும், அதன் செயல்பாடுகளிலும் சில சிக்கல்கள் நீடித்தன.

இந்த இரண்டு நூலகங்களும் பயனுள்ளதாக இருந்தபோதிலும், அவை இரண்டும் தனித்தனியாகச் செயல்பட்டதால், Python சமூகத்தில் ஒருவித குழப்பம் நிலவியது. ஒரு திட்டத்திற்கு எதைப் பயன்படுத்துவது என்ற தெளிவு இல்லாமல் இருந்தது.

### 0.உ. NumPy-யின் உதயம்: ஒரு ஒருங்கிணைந்த சக்தி

இந்தச் சூழலில்தான், 2005-ஆம் ஆண்டு **டிராவிஸ் ஒலிஃபன்ட் (Travis Oliphant)** என்ற தொலைநோக்கு சிந்தனையாளர், இந்த இரண்டு உலகங்களின் சிறந்த அம்சங்களை ஒன்றிணைத்து, அவற்றின் குறைகளைக் களைந்து, **NumPy** என்ற ஒற்றை, சக்திவாய்ந்த நூலகத்தை உருவாக்கினார்.

NumPy-யின் இதயத்துடிப்பு என்று அழைக்கப்படுவது அதன் **ndarray (N-dimensional array)** தான். இது வெறும் ஒரு தரவு அமைப்பு அல்ல; அது ஒரு செயல்திறன் அதிசயம்.

### 0.ங. ndarray: ஏன் இவ்வளவு சக்தி வாய்ந்தது?

Python-இன் சாதாரண `list`-ஐ ஒரு மளிகைக் கடைப் பை போல கற்பனை செய்துகொள்ளுங்கள். அதில் நீங்கள் பழம், காய்கறி, பிஸ்கட் எனப் பலதரப்பட்ட பொருட்களை வைக்கலாம். ஆனால் NumPy-யின் `ndarray` என்பது முட்டைகளை வைக்கும் அட்டைப் பெட்டி போன்றது. அதில் ஒரே மாதிரியான, ஒரே அளவிலான பொருட்களை (எண்களை) மட்டுமே வைக்க முடியும்.

இந்தக் கட்டுப்பாடுதான் அதன் மிகப்பெரிய பலம். ஒரே வகையான தரவுகள் வரிசையாக நினைவகத்தில் (memory) சேமிக்கப்படுவதால், கணினியால் மிக மிக வேகமாகச் செயல்பட முடிகிறது. ஒரு `list`-ல் உள்ள ஒவ்வொரு எண்ணுக்கும் தனித்தனியாகக் கட்டளையிடுவதற்குப் பதிலாக, NumPy-யின் `ndarray`-ஐப் பயன்படுத்தி, இலட்சக்கணக்கான எண்களைக் கொண்ட ஒரு அணிக்கு **ஒரே கட்டளையில்** ஒரு கணிதச் செயல்பாட்டைச் செய்ய முடியும். இதுதான் NumPy-யின் அபார வேகத்திற்குக் காரணம்.

இந்த **ndarray** என்ற அஸ்திவாரத்தின் மீதுதான் இன்று நாம் காணும் தரவு அறிவியல் உலகமே கட்டமைக்கப்பட்டுள்ளது. வாருங்கள், இந்த எண்களின் வேகமான உலகத்திற்குள் பயணிப்போம்!

நிச்சயமாக! நீங்கள் வழங்கிய விளக்கத்தை, வாசகர்கள் எளிதில் புரிந்து கொள்ளும் வண்ணம், மேலும் தெளிவான எடுத்துக்காட்டுகளுடன் மெருகேற்றலாம். இதோ ஒரு மேம்படுத்தப்பட்ட வடிவம்:

## 0.ச. ஏன் நமக்கு N-பரிமாண அணிகள் தேவை?

நம்மைச் சுற்றியுள்ள உலகம் தட்டையானது அல்ல. அது நீளம், அகலம், உயரம், ஆழம், நேரம் எனப் பல பரிமாணங்களைக் கொண்டது. இந்த சிக்கலான, பல அடுக்குகள் கொண்ட உலகத்தின் தகவல்களைக் கணினிக்குப் புரியவைக்க, ஒரு எளிய பட்டியல் (list) அல்லது ஒரு சாதாரண அட்டவணை (table) போதுமானதல்ல. இங்குதான் **N-பரிமாண அணிகள் (N-dimensional arrays)** ஒரு சக்திவாய்ந்த கருவியாக நமக்கு உதவுகின்றன.

எளிமையாகச் சொன்னால், நிஜ உலகின் பல அடுக்குத் தகவல்களை, கணினி புரிந்துகொள்ளும் மொழியில் கட்டமைக்கவே நமக்கு N-பரிமாண அணிகள் தேவை. இதை இரண்டு அற்புதமான எடுத்துக்காட்டுகள் மூலம் புரிந்துகொள்வோம்.

**எடுத்துக்காட்டு 1: ஒரு புகைப்படத்திற்கு உயிர் கொடுப்பது எப்படி?**

நாம் பார்க்கும் ஒவ்வொரு டிஜிட்டல் படமும் (digital image) எண்களால் ஆன ஒரு அணிதான்.

- **கருப்பு-வெள்ளைப் படம் (Grayscale Image):** ஒரு கருப்பு-வெள்ளைப் படத்தை, சதுரங்கப் பலகை (chessboard) போலக் கற்பனை செய்துகொள்ளுங்கள். அதில் நீளம் (rows), அகலம் (columns) என இரண்டு பரிமாணங்கள் மட்டுமே இருக்கும். ஒவ்வொரு கட்டத்திலும் (pixel), கருப்பின் அடர்த்தியைக் குறிக்க ஒரு எண் இருக்கும் (உதாரணமாக, 0 என்பது முழுமையான கருப்பு, 255 என்பது முழுமையான வெள்ளை). இது ஒரு **2D அணி (2D array)**.
    - `அணியின் வடிவம்: (உயரம், அகலம்)`
- **வண்ணப் படம் (Color Image):** ஆனால், நாம் பார்க்கும் பெரும்பான்மை படங்கள் வண்ணமயமானவை. கணினியைப் பொறுத்தவரை, ஒவ்வொரு வண்ணமும் **சிவப்பு (Red), பச்சை (Green), நீலம் (Blue)** ஆகிய மூன்று அடிப்படை வண்ணங்களின் கலவையே.

    சரி, இந்தக் கூடுதல் வண்ணத் தகவலை எங்கே சேமிப்பது? இங்குதான் மூன்றாவது பரிமாணம் வருகிறது. ஒரு வண்ணப் படம் என்பது, ஒரே அளவுள்ள மூன்று 2D அணிகளை ஒன்றன் மேல் ஒன்றாக அடுக்கி வைப்பதைப் போன்றது.

    1. முதல் அடுக்கு: படத்தில் உள்ள சிவப்பு நிறத்தின் அளவைக் குறிக்கும்.
    2. இரண்டாவது அடுக்கு: பச்சை நிறத்தின் அளவைக் குறிக்கும்.
    3. மூன்றாவது அடுக்கு: நீல நிறத்தின் அளவைக் குறிக்கும்.

    இந்த மூன்று அடுக்குகளும் சேரும்போதுதான் நமக்கு ஒரு முழுமையான வண்ணப் படம் கிடைக்கிறது. எனவே,

ஒரு வண்ணப் படம் என்பது ஒரு **3D அணி (3D array)**.

  - `அணியின் வடிவம்: (உயரம், அகலம், வண்ண அடுக்குகள்)`

இப்படிப் படங்களை அணிகளாக மாற்றுவதால்தான், நம்மால் ஒரு படத்தின் பிரகாசத்தை மாற்றுவது, வண்ணங்களைச் சரிசெய்வது போன்ற எண்ணற்ற Image Processing வேலைகளை எளிதாகச் செய்ய முடிகிறது.

**எடுத்துக்காட்டு 2: வானிலையைக் கணிப்பது**

ஒரு நாட்டின் வானிலை என்பது மிகவும் சிக்கலான ஒரு விஷயம். வெப்பநிலை, ஈரப்பதம், காற்றின் வேகம் எனப் பல தகவல்கள் ஒவ்வொரு இடத்திலும், ஒவ்வொரு நேரத்திலும் மாறிக்கொண்டே இருக்கும். இந்தத் தகவலை எப்படி ஒரே இடத்தில் சேமிப்பது?

யோசித்துப் பாருங்கள்:

1. **பரிமாணம் 1 & 2 (இடம்):** முதலில், ஒரு நாட்டின் வரைபடத்தை **அட்சரேகை (latitude)** மற்றும் **தீர்க்கரேகை (longitude)** கொண்டு ஒரு 2D கட்டமாக உருவாக்குகிறோம்.
2. **பரிமாணம் 3 (நேரம்):** அடுத்து, ஒரு நாளின் 24 மணிநேரத்திற்கும் தகவல்களைச் சேமிக்க வேண்டும். இது நமது மூன்றாவது பரிமாணமாக, அதாவது **நேரம்**, அமைகிறது.
3. **பரிமாணம் 4 (தரவு வகை):** இப்போது, ஒவ்வொரு இடத்திலும், ஒவ்வொரு மணி நேரத்திற்கும், நாம் வெப்பநிலை, ஈரப்பதம் போன்ற வெவ்வேறு அளவீடுகளைச் சேமிக்க வேண்டும். இது நமது நான்காவது பரிமாணம்.

ஆக, ஒரு குறிப்பிட்ட பிராந்தியத்தின் ஒரு நாள் வானிலை அறிக்கையை முழுமையாகச் சேமிக்க, நமக்கு ஒரு **4D அணி (4D array)** தேவைப்படுகிறது.

- `அணியின் வடிவம்: (அட்சரேகை இடங்கள், தீர்க்கரேகை இடங்கள், நேரம், அளவீடுகள்)`

இந்தக் கட்டமைப்பைப் பயன்படுத்துவதன் மூலம், “நேற்று மதியம் 3 மணிக்கு, நாட்டின் வடக்குப் பகுதியில் சராசரி வெப்பநிலை என்ன?” என்பது போன்ற சிக்கலான கேள்விகளுக்கு நம்மால் உடனடியாக விடை காண முடியும்.

சுருக்கமாக, **N-பரிமாண அணிகள்** என்பவை சிக்கலான நிஜ உலகத் தரவுகளைக் கையாளவும், பகுப்பாய்வு செய்யவும், அறிவியல் கணக்கீடுகளை வேகமாகவும் திறமையாகவும் செய்யவும் நமக்குக் கிடைத்த ஒரு மிகச்சிறந்த கருவியாகும்.

**குறிப்பு:**

Array programming with NumPy

**GitHub:** https://github.com/tamil-phy/NumPy_Book_Tamil

## அத்தியாயம் க: NumPy அணியின் குணாதிசயங்கள் (Array Attributes)

கற்பனை செய்துகொள்ளுங்கள், உங்களிடம் ஒரு சக்திவாய்ந்த, மர்மமான தரவுப் பெட்டி (data container) கிடைக்கிறது. அந்தப் பெட்டிக்குள் இருக்கும் பொருட்களைப் பயன்படுத்துவதற்கு முன்பு, பெட்டியைப் பற்றிய சில அடிப்படை விஷயங்களைத் தெரிந்துகொள்வது அவசியம் அல்லவா?

- அந்தப் பெட்டியின் வடிவம் என்ன? (What is its shape?)
- அது தட்டையானதா அல்லது ஆழமானதா? (Is it flat or deep?)
- உள்ளே இருக்கும் ஒவ்வொரு பொருளும் எவ்வளவு எடை கொண்டது? (How much does each item weigh?)
- அதைப் பாதுகாப்பாகக் கையாள்வதற்கான விதிகள் என்ன? (What are the handling instructions?)

NumPy-யில் நாம் உருவாக்கும் ஒவ்வொரு அணியும் (`array`) அப்படிப்பட்ட ஒரு தரவுப் பெட்டிதான். அந்த அணியின் குணாதிசயங்களை (`attributes`) விசாரிப்பது, அந்தப் பெட்டியின் மீது ஒட்டப்பட்டிருக்கும் “விவரச் சீட்டை” (`specification label`) படிப்பது போன்றது. வாருங்கள், அந்த விவரச் சீட்டில் என்னென்ன இருக்கிறது என்று பார்ப்போம்.

### க.க. `ndarray.shape` – அணியின் கட்டமைப்பு

ஒரு பெட்டியின் நீளம், அகலம், உயரம் எப்படி அதன் அளவைச் சொல்கிறதோ, அதுபோல **`shape`** (வடிவம்) என்ற குணம், ஒரு NumPy அணியின் கட்டமைப்பை நமக்குத் தெளிவாகக் காட்டுகிறது. எளிமையாகச் சொன்னால், ஓர் அணியில் **எத்தனை வரிசைகள் (`rows`)** மற்றும் **எத்தனை நிரல்கள் (`columns`)** உள்ளன என்பதைச் சொல்லும் முகவரிதான் `shape`.

**`shape`**-ஐ சோதிப்பது, நாம் எந்த மாதிரியான தரவுக் களத்தில் வேலை செய்யப் போகிறோம் என்பதைத் தெரிந்துகொள்ளும் முதல் படி.

**எடுத்துக்காட்டு:**

Python

```
import numpy as np

# இரண்டு வரிசைகள் மற்றும் மூன்று நிரல்கள் கொண்ட ஒரு அணியை உருவாக்குவோம்
arr = np.array([[1, 2, 3], [4, 5, 6]])

print("அணியின் வடிவம் (Shape of array):", arr.shape)
```

**வெளியீடு:**

```
அணியின் வடிவம் (Shape of array): (2, 3)
```

இந்த `(2, 3)` என்பது, நமது தரவுப் பெட்டியில் **2 வரிசைகளும்**, ஒவ்வொரு வரிசையிலும் **3 உறுப்புகளும்** உள்ளன என்பதைத் தெளிவாகக் காட்டுகிறது.

### க.உ. `ndarray.ndim` – பரிமாணங்களின் எண்ணிக்கை

`shape` நமக்கு அணியின் அளவுகளைச் சொன்னால், **`ndim`** (N-Dimensions என்பதன் சுருக்கம்) அந்த அணிக்கு எத்தனை பரிமாணங்கள் (`dimensions`) உள்ளன என்பதைச் சொல்கிறது.

- `ndim = 1` என்றால், அது ஒரு தட்டையான பட்டியல் போன்ற **1D அணி**.
- `ndim = 2` என்றால், அது ஒரு அட்டவணை போன்ற **2D அணி**.
- `ndim = 3` என்றால், அது ஒன்றன் மேல் ஒன்றாக அடுக்கி வைக்கப்பட்ட பல அட்டவணைகள் போன்ற **3D அணி**.

நமது தரவுப் பெட்டி ஒரு கடிதம் போல தட்டையானதா (1D), ஒரு புகைப்படம் போல இரு பரிமாணம் கொண்டதா (2D), அல்லது ஒரு அட்டைப்பெட்டி போல முப்பரிமாணம் கொண்டதா (3D) என்பதை `ndim` ஒரு நொடியில் சொல்லிவிடும்.

**எடுத்துக்காட்டு:**

Python

```python
import numpy as np

arr = np.array([[1, 2, 3], [4, 5, 6]])

print("பரிமாணங்களின் எண்ணிக்கை (Number of dimensions):", arr.ndim)
```

**வெளியீடு:**

```
பரிமாணங்களின் எண்ணிக்கை (Number of dimensions): 2
```

இந்த வெளியீடு, நாம் ஒரு 2D அணியுடன் (அட்டவணை போன்ற அமைப்பு) வேலை செய்கிறோம் என்பதை உறுதிப்படுத்துகிறது.

### க.ரு. `itemsize` – ஒவ்வொரு உறுப்பின் நினைவக அளவு

நமது தரவுப் பெட்டிக்குள் இருக்கும் ஒவ்வொரு பொருளும் ஒரே மாதிரியானவை என்று பார்த்தோம். **`itemsize`** (உறுப்பின் அளவு) என்ற குணம், அந்த அணியில் உள்ள **ஒவ்வொரு தனி உறுப்பும் (`element`)** கணினியின் நினைவகத்தில் (`memory`) எவ்வளவு இடத்தை (பைட்களில் `bytes`) எடுத்துக்கொள்கிறது என்பதைக் கூறுகிறது.

மிகப்பெரிய தரவுகளைக் கையாளும்போது, நினைவகப் பயன்பாட்டைக் கணக்கிட இந்தக் குணம் மிகவும் அவசியம்.

**எடுத்துக்காட்டு:**

Python

```python
import numpy as np

# இயல்பாக, முழு எண்கள் 64-bit integer ஆக சேமிக்கப்படும்
arr = np.array([1, 2, 3, 4, 5])

print("ஒவ்வொரு உறுப்பின் அளவு (Item size of array):", arr.itemsize, "bytes")
```

**வெளியீடு:**

```
ஒவ்வொரு உறுப்பின் அளவு (Item size of array): 8 bytes
```

இதன் பொருள், இந்த அணியில் உள்ள ஒவ்வொரு எண்ணும் கணினி நினைவகத்தில் 8 பைட்களை எடுத்துக்கொள்கிறது. இலட்சக்கணக்கான எண்களைக் கொண்ட ஒரு அணியை உருவாக்கும்போது, இந்த எண் எவ்வளவு முக்கியம் என்பது புரியும்!

### க.ச. `flags` – அணியின் கையாளும் விதிகள்

ஒவ்வொரு பெட்டியின் மீதும் “Fragile” (உடையக்கூடியது), “This side up” (இந்தப் பக்கம் மேலே) போன்ற சில கையாளும் விதிமுறைகள் எழுதப்பட்டிருக்கும் அல்லவா? அதுபோல, **`flags`** (கொடிக்குறிப்புகள்) என்ற குணம், ஒரு NumPy அணியின் நினைவக அமைப்பு (`memory layout`) மற்றும் அதன் உள்ளார்ந்த பண்புகளைப் பற்றிய தொழில்நுட்ப விவரங்களைத் தருகிறது.

இது ஒரு ஆழமான தொழில்நுட்பப் பகுதி என்றாலும், இதன் மூலம் ஓர் அணி நினைவகத்தில் எவ்வாறு சேமிக்கப்பட்டுள்ளது, அதில் உள்ள தரவை நம்மால் மாற்ற முடியுமா (`writeable`) போன்ற முக்கியமான தகவல்களை அறியலாம்.

**எடுத்துக்காட்டு:**

Python

```
import numpy as np

arr = np.array([1, 2, 3, 4, 5])

print("அணியின் கொடிக்குறிப்புகள் (Flags of the array):\n", arr.flags)
```

**வெளியீடு:**

```
அணியின் கொடிக்குறிப்புகள் (Flags of the array):
  C_CONTIGUOUS : True
  F_CONTIGUOUS : True
  OWNDATA : True
  WRITEABLE : True
  ALIGNED : True
  WRITEBACKIFCOPY : False
```

இந்த வெளியீட்டில் உள்ள சில முக்கியக் குறிப்புகள்:

- **`C_CONTIGUOUS`**: தரவுகள் வரிசை வரிசையாக (C மொழி போல) நினைவகத்தில் தொடர்ச்சியாகச் சேமிக்கப்பட்டுள்ளன.
- **`WRITEABLE`**: `True` என இருப்பதால், இந்த அணியில் உள்ள மதிப்புகளை நம்மால் மாற்ற முடியும்.
- **`OWNDATA`**: இந்த அணி தனக்கென சொந்தமாக நினைவகத்தை ஒதுக்கியுள்ளது.

இந்தக் குணாதிசயங்கள், NumPy-யின் செயல்திறனுக்குப் பின்னால் இருக்கும் ரகசியங்களை நமக்குக் காட்டுகின்றன. ஒரு அணியின் இந்த “விவரச் சீட்டை” புரிந்துகொள்வது, அதை மேலும் திறமையாகக் கையாள நமக்கு உதவும்.

## அத்தியாயம் ௨: NumPy அணிகளை உருவாக்கும் வழிகள் (Array Creation Routines)

ஒரு சிறந்த ஓவியத்தை வரைவதற்கு முன், ஓவியர் தனக்கு ஏற்ற கேன்வாஸைத் (canvas) தேர்ந்தெடுப்பார். சில சமயங்களில் வேக வேகமாக ஒரு ஓவியத்தை வரையத் தொடங்க, ஏற்கெனவே இருக்கும் ஒரு கேன்வாஸைப் பயன்படுத்துவார். சில சமயங்களில், முற்றிலும் புதிதாக, கறுப்பு அல்லது வெள்ளை நிறத்தில் ஒரு கேன்வாஸைத் தயார் செய்வார்.

NumPy-யிலும் அப்படித்தான். நம்முடைய தரவுப் பகுப்பாய்வு அல்லது கணக்கீடுகள் என்ற ஓவியத்தை வரைவதற்கு முன், நமக்குத் தேவையான அணியை (`array`) என்ற கேன்வாஸை உருவாக்க வேண்டும். நமது தேவைக்கேற்ப, அந்த கேன்வாஸை உருவாக்க NumPy பல எளிய வழிகளைத் தருகிறது. வாருங்கள், அவற்றில் முக்கியமான மூன்றைப் பற்றிப் பார்ப்போம்.

### ௨.க. `numpy.empty` – மின்னல் வேகத்தில் ஒரு தளம்

ஒரு ஓவியர் மிக அவசரமாக ஒரு ஓவியத்தை வரையத் தொடங்க வேண்டும் என்று வைத்துக்கொள்வோம். அவர் புதிதாக ஒரு கேன்வாஸை உருவாக்கக் காத்திருக்காமல், ஸ்டுடியோவில் ஏற்கெனவே இருக்கும் ஒரு கேன்வாஸை எடுத்து அப்படியே வரையத் தொடங்கிவிடுவார். அந்தப் பழைய கேன்வாஸில் ஏற்கெனவே சில கிறுக்கல்கள் இருக்கலாம், ஆனால் அவர் அதைப்பற்றிக் கவலைப்பட மாட்டார், ஏனெனில் அவர் அதன் மேல் முழுமையாகப் புதிய வண்ணங்களைப் பூசப் போகிறார்.

**`numpy.empty`** என்ற செயற்கூறு (`function`) அப்படித்தான் செயல்படுகிறது. இது ஒரு புதிய அணியை மிக மிக வேகமாக உருவாக்கும். ஆனால், அது அந்த அணிக்கான இடத்தை நினைவகத்தில் (`memory`) ஒதுக்குமே தவிர, அதில் உள்ள பழைய மதிப்புகளை நீக்காது. அதனால், அந்த இடத்தில் ஏற்கெனவே இருந்த குப்பை மதிப்புகளே (`junk values`) அந்த அணியில் இருக்கும்.

**எப்போது பயன்படுத்த வேண்டும்?** ஓர் அணியை உருவாக்கிய அடுத்த நொடியே, அதில் உள்ள எல்லா இடங்களையும் உங்கள் சொந்த மதிப்புகளைக் கொண்டு நிரப்பப் போகிறீர்கள் என்றால், தொடக்க மதிப்புகள் பற்றி உங்களுக்குக் கவலை இல்லை. அந்தச் சமயத்தில், வேகத்திற்காக `numpy.empty`-ஐப் பயன்படுத்தலாம்.

**எடுத்துக்காட்டு:**

Python

```python
import numpy as np

# 2 வரிசை, 3 நிரல்களுடன் ஒரு காலி அணியை உருவாக்குதல்
empty_array = np.empty((2, 3))

print("காலி அணி (Empty array):\n", empty_array)
```

**வெளியீடு:**

```
காலி அணி (Empty array):
 [[2.12199579e-314 6.36598737e-315 8.48798316e-315]
 [1.06099790e-314 1.27319747e-314 1.48539705e-314]]
```

**கவனம்:** ⚠️ இங்கு வரும் மதிப்புகள் ஒவ்வொரு முறையும் மாறக்கூடியவை. இவை கணினியின் நினைவகத்தில் அந்த நேரத்தில் இருந்த தேவையற்ற மதிப்புகள். இந்த அணியின் மதிப்புகளை நம்பி எந்தக் கணக்கீட்டையும் செய்யக்கூடாது.

### உ.உ. `numpy.zeros` – ஒரு சுழியத் தொடக்கம்

சில சமயங்களில், ஓவியர் தன் ஓவியத்தை ஒரு முழுமையான கறுப்பு நிறப் பின்னணியில் இருந்து தொடங்க விரும்புவார். அது ஓவியத்திற்கு ஒரு புதிய பரிமாணத்தைக் கொடுக்கும்.

**`numpy.zeros`** என்ற செயற்கூறு, அப்படிப்பட்ட ஒரு கறுப்பு கேன்வாஸை நமக்குத் தருகிறது. இது நாம் கேட்கும் வடிவில் (`shape`) ஓர் அணியை உருவாக்கி, அதன் அனைத்து உறுப்புகளையும் (`elements`) **சுழியம் (`0`)** என்ற மதிப்பைக் கொண்டு நிரப்பும். இது மிகவும் பாதுகாப்பான மற்றும் கணிக்கக்கூடிய ஒரு தொடக்கத்தைத் தருகிறது.

**எப்போது பயன்படுத்த வேண்டும்?** நீங்கள் ஒரு கணக்கீட்டின் தொடக்க மதிப்பை சுழியத்திலிருந்து தொடங்க விரும்பும்போது, (உதாரணமாக, மதிப்பெண்களைக் கூட்டத் தொடங்கும் முன் மொத்த மதிப்பை 0 என வைப்பது போல) `numpy.zeros` மிகவும் பயனுள்ளதாக இருக்கும்.

**எடுத்துக்காட்டு:**

Python

```
import numpy as np

# 2x2 வடிவில் சுழியங்கள் நிறைந்த அணியை உருவாக்குதல்
zeros_array = np.zeros((2, 2))

print("சுழிய அணி (Zeros array):\n", zeros_array)
```

**வெளியீடு:**

```
சுழிய அணி (Zeros array):
 [[0. 0.]
 [0. 0.]]
```

பார்த்தீர்களா? எந்தக் குழப்பமும் இல்லாத, சுத்தமான, கணிக்கக்கூடிய ஒரு தொடக்கம்.

### உ.ரு. `numpy.ones` – ஒன்றிலிருந்து தொடங்குவோம்

சில கணக்கீடுகளுக்கு, நமக்கு சுழியம் ஒரு நல்ல தொடக்கமாக இருக்காது. பெருக்கல் போன்ற சில கணிதச் செயல்பாடுகளுக்கு, தொடக்க மதிப்பை `1` என வைப்பது பயனுள்ளதாக இருக்கும். இதை, ஓவியர் தன் கேன்வாஸை வெள்ளை நிறப் பின்னணியில் இருந்து தொடங்குவதுடன் ஒப்பிடலாம்.

**`numpy.ones`** என்ற செயற்கூறு, நாம் கேட்கும் வடிவில் ஓர் அணியை உருவாக்கி, அதன் அனைத்து உறுப்புகளையும் **ஒன்று (`1`)** என்ற மதிப்பைக் கொண்டு நிரப்புகிறது.

**எடுத்துக்காட்டு:**

Python

```
import numpy as np

# 3x3 வடிவில் ஒன்றுகள் நிறைந்த அணியை உருவாக்குதல்
ones_array = np.ones((3, 3))

print("ஒன்று அணி (Ones array):\n", ones_array)
```

**வெளியீடு:**

```
ஒன்று அணி (Ones array):
 [[1. 1. 1.]
 [1. 1. 1.]
 [1. 1. 1.]]
```

சுருக்கமாகச் சொன்னால், உங்கள் தேவைக்கேற்ப சரியான கேன்வாஸைத் தேர்ந்தெடுங்கள்:

- **அதிகபட்ச வேகம் தேவையெனில்:** `np.empty()` (பயன்படுத்திய கேன்வாஸ்)
- **பாதுகாப்பான, சுழியத் தொடக்கம் தேவையெனில்:** `np.zeros()` (கறுப்பு கேன்வாஸ்)
- **குறிப்பிட்ட, ஒன்றென்ற தொடக்கம் தேவையெனில்:** `np.ones()` (வெள்ளை கேன்வாஸ்)

## அத்தியாயம் ௬: NUMPY – ARRAY FROM EXISTING DATA (ஏற்கெனவே உள்ள தரவிலிருந்து ஓர் அணியை உருவாக்குதல்)

ஒரு திறமையான சிற்பி, தன்னிடம் வரும் மூலப்பொருட்கள் எந்த வடிவில் இருந்தாலும்—அது ஒரு பாறையாக இருந்தாலும், மரத்துண்டாக இருந்தாலும், அல்லது உடைந்த சிலைகளின் பாகங்களாக இருந்தாலும்—அதை நேர்த்தியாகச் செதுக்கி ஒரு அற்புதமான சிற்பமாக மாற்றுவார்.

NumPy-யும் அப்படித்தான். நம்முடைய தரவுகள் Python-ல் பல வடிவங்களில் இருக்கலாம். ஒரு சாதாரணமான பட்டியல் (`list`), கணினியின் நினைவகத்தில் இருக்கும் ஒரு மூலத் தரவுப் பகுதி (`buffer`), அல்லது தேவைக்கேற்ப மதிப்புகளை உருவாக்கும் ஒரு நீரோட்டை (`iterable`) என எப்படி இருந்தாலும், NumPy அவற்றை சக்திவாய்ந்த அணிகளாக (`arrays`) மாற்றும் கருவிகளைக் கொண்டுள்ளது.

இந்த அத்தியாயத்தில், வெவ்வேறு மூலப்பொருட்களை NumPy என்ற சிற்பி எப்படித் தனக்கு ஏற்றவாறு செதுக்கிக் கொள்கிறார் என்று பார்ப்போம்.

### ௬.க. `numpy.asarray` – மளிகைப் பையிலிருந்து மாற்றுதல்

நம்மிடம் இருக்கும் தரவுகளில் மிகவும் பொதுவானது Python `list`. இதை ஒரு மளிகைப் பை (`grocery bag`) என்று கற்பனை செய்துகொள்ளுங்கள். **`asarray`** என்ற செயற்கூறு (`function`), அந்தப் பையில் இருக்கும் பொருட்களை எடுத்து, நேர்த்தியாக NumPy என்ற பாத்திரத்தில் வைக்கும் ஒரு புத்திசாலியான வேலையாள்.

இதன் மிகப்பெரிய சிறப்பு என்னவென்றால், நீங்கள் கொடுக்கும் பொருள் ஏற்கெனவே ஒரு NumPy பாத்திரத்தில் (அணியில்) இருந்தால், இந்த வேலையாள் அதை மீண்டும் ஒரு புதிய பாத்திரத்திற்கு மாற்ற மாட்டார். இதனால், தேவையற்ற நகல் எடுப்பது தவிர்க்கப்பட்டு, நினைவகம் (`memory`) சேமிக்கப்படுகிறது.

**எடுத்துக்காட்டு:**

Python

```python
import numpy as np

# நமது மளிகைப் பை (Python list)
list_data = [10, 20, 30, 40, 50]

# அதை NumPy பாத்திரத்திற்கு (array) மாற்றுவோம்
array_data = np.asarray(list_data)

print("பட்டியலிலிருந்து உருவான அணி (Array from list):", array_data)
print("அணியின் வகை (Type of array):", type(array_data))
```

**வெளியீடு:**

```
பட்டியலிலிருந்து உருவான அணி (Array from list): [10 20 30 40 50]
அணியின் வகை (Type of array): <class 'numpy.ndarray'>
```

சாதாரண `list`-ஆக இருந்தது, இப்போது NumPy-யின் சக்திவாய்ந்த செயல்பாடுகளுக்குத் தயாராகிவிட்டது.

### ௬.உ. `numpy.frombuffer` – மூலப் பொட்டலத்தைப் பிரித்தல்

சில சமயங்களில், தரவுகள் நமக்கு நேரடியாகப் புரியும் வடிவில் இருக்காது. அவை கணினியின் நினைவகத்தில் பைனரி (`binary`) வடிவில், அதாவது ஒருவித மூலக் குறியீடு போல, ஒரு பொட்டலத்தில் (`buffer`) அடைக்கப்பட்டிருக்கும்.

**`frombuffer`** என்ற கருவி, அந்தப் பொட்டலத்தைப் பிரித்து, உள்ளே இருக்கும் மூலத் தரவை நாம் குறிப்பிடும் வடிவில் (எடுத்துக்காட்டாக, ஒவ்வொரு எழுத்தாக) தனித்தனி உறுப்புகளாக மாற்றி ஓர் அணியை உருவாக்கும் ஒரு சிறப்பு நிபுணர். இது பெரும்பாலும் கோப்புகளைப் படிக்கும்போதும், பிணையத் தரவுகளைக் கையாளும்போதும் பயன்படும்.

**எடுத்துக்காட்டு:**

Python

```python
import numpy as np

# b'...' என்பது ஒரு பைனரி தரவுப் பொட்டலம்
buffer_data = b'Gemini AI'

# அந்தப் பொட்டலத்தை ஒவ்வொரு எழுத்தாக ('S1') பிரிக்கச் சொல்கிறோம்
array_buffer = np.frombuffer(buffer_data, dtype='S1')

print("பொட்டலத்திலிருந்து உருவான அணி (Array from buffer):", array_buffer)
```

**வெளியீடு:**

```
பொட்டலத்திலிருந்து உருவான அணி (Array from buffer):
 [b'G' b'e' b'm' b'i' b'n' b'i' b' ' b'A' b'I']
```

பார்த்தீர்களா, `frombuffer` அந்த மூலப் பொட்டலத்தை நேர்த்தியாகப் பிரித்து, ஒவ்வொரு உறுப்பையும் தனித்தனியாகக் கொடுத்துவிட்டது.

### ௩.௩. `numpy.fromiter` – நீரோடையிலிருந்து சேகரித்தல்

சில நேரங்களில், நமக்குத் தேவையான எல்லாத் தரவுகளும் ஒரே நேரத்தில் கிடைக்காது. அவை ஒரு நீரோடை (`stream`) அல்லது நகரும் பட்டை (`conveyor belt`) போல, தேவைக்கேற்ப ஒவ்வொன்றாக வந்து கொண்டிருக்கும். இப்படிப்பட்ட `iterable` அல்லது `generator` வகை தரவுகளைக் கையாளவே **`fromiter`** என்ற செயற்கூறு உதவுகிறது.

இது மிகவும் நினைவகத் திறன்கொண்டது (`memory-efficient`). ஏனெனில், இது எல்லாத் தரவுகளும் வரும் வரை காத்திருந்து ஒரு பெரிய பட்டியலை உருவாக்காது. மாறாக, நீரோடையில் இருந்து வரும் ஒவ்வொரு மதிப்பையும் உடனடியாக எடுத்து, நேராக NumPy அணிக்குள் சேகரித்துவிடும்.

**எடுத்துக்காட்டு:**

range(5)-ல் உள்ள ஒவ்வொரு எண்ணையும் இரண்டால் பெருக்கி வரும் மதிப்புகளைக் கொண்ட ஒரு நீரோடையை (iterable) உருவாக்குவோம்.

Python

```
import numpy as np

# இது ஒரு நீரோடை. மதிப்புகள் தேவைப்படும்போது மட்டுமே உருவாகும்.
iterable_data = (x * 2 for x in range(5))

# fromiter, அந்த ஓடையில் இருந்து மதிப்புகளைச் சேகரிக்கிறது
array_iter = np.fromiter(iterable_data, dtype='int32')

print("நீரோடையிலிருந்து உருவான அணி (Array from iterable):", array_iter)
```

**வெளியீடு:**

```
நீரோடையிலிருந்து உருவான அணி (Array from iterable): [0 2 4 6 8]
```

`fromiter` அந்த நீரோடையில் இருந்து ஒவ்வொரு மதிப்பாக (0, 2, 4, 6, 8) எடுத்து, ஒரு முழுமையான NumPy அணியை உருவாக்கிவிட்டது.

**சுருக்கம்: சரியான கருவியைத் தேர்ந்தெடுங்கள்**

| செயற்கூறு (Function) | உவமை (Analogy) | எப்போது பயன்படுத்த வேண்டும்? |
|---|---|---|
| `np.asarray()` | மளிகைப் பை (Grocery Bag) | Python `list` அல்லது `tuple`-களை மாற்ற. (நினைவகச் சிக்கனம்) |
| `np.frombuffer()` | மூலப் பொட்டலம் (Sealed Packet) | பைனரி (`binary`) தரவுகளைக் கையாள. (மேம்பட்ட பயன்பாடு) |
| `np.fromiter()` | நகரும் பட்டை (Conveyor Belt) | `iterable`-களை நினைவகத் திறனுடன் மாற்ற. (பெரிய தரவுகளுக்கு) |

## அத்தியாயம் ௪: NUMPY – ARRAY FROM NUMERICAL RANGES (எண் வரம்புகளிலிருந்து ஓர் அணியை உருவாக்குதல்)

தரவு அறிவியலில் பல நேரங்களில், நம்மிடம் தரவுகள் இருக்காது; மாறாக, நாமே ஒரு எண் தொடரை (`sequence of numbers`) உருவாக்க வேண்டியிருக்கும். இதை ஒரு நீண்ட பாதையில், குறிப்பிட்ட இடங்களில் கற்களை வைப்பதற்கு ஒப்பிடலாம். பாதையின் தொடக்கம் எங்கே, முடிவு எங்கே, ஒவ்வொரு கல்லுக்கும் எவ்வளவு இடைவெளி இருக்க வேண்டும் என்பதை நாம்தான் தீர்மானிக்க வேண்டும்.

NumPy, இப்படிப்பட்ட எண் பாதைகளை உருவாக்க மூன்று விதமான கருவிகளை நமக்குத் தருகிறது. ஒவ்வொன்றும் ஒரு குறிப்பிட்ட முறையில் கற்களை வைப்பதற்கு உதவுகிறது.

### ௪.௧. `numpy.arange` – குறிப்பிட்ட இடைவெளியில் கல் வைத்தல்

இது பாதையில் கல் வைப்பதற்கான மிகவும் நேரடியான முறை. **`arange`** (Arrange என்பதன் சுருக்கம்) என்ற கருவியைப் பயன்படுத்தும்போது, நீங்கள் மூன்று விஷயங்களைக் குறிப்பிடுவீர்கள்:

1. **தொடக்கம் (`start`):** முதல் கல்லை எங்கே வைக்க வேண்டும்?
2. **முடிவு (`stop`):** எந்த இடத்திற்கு முன்புவரை கற்களை வைக்க வேண்டும்?

3. **இடைவெளி (`step`):** ஒவ்வொரு கல்லுக்கும் இடையில் எவ்வளவு தூரம் இருக்க வேண்டும்?

இது Python-ல் உள்ள `range()` செயற்கூறு போன்றதே. ஆனால் இது மிதக்கும் புள்ளி எண்களையும் (`floating point numbers`) கையாளும் மற்றும் NumPy அணியை உருவாக்கும்.

எடுத்துக்காட்டு:

1-லிருந்து தொடங்கி, 10-க்குள், ஒவ்வொரு கல்லுக்கும் 2 அடி இடைவெளி விட்டு வைப்போம்.

Python

```
import numpy as np

# arange(தொடக்கம், முடிவு, இடைவெளி)
array_arange = np.arange(1, 10, 2)

print("arange மூலம் உருவான அணி:", array_arange)
```

**வெளியீடு:**

```
arange மூலம் உருவான அணி: [1 3 5 7 9]
```

பார்த்தீர்களா? 1-ல் தொடங்கி, ஒவ்வொரு முறையும் 2 அடி இடைவெளி விட்டு, 10-ஐத் தாண்டுவதற்கு முன்புவரை கற்கள் அழகாக அடுக்கப்பட்டுவிட்டன.

**ச.உ. `numpy.linspace` – சமமான எண்ணிக்கையில் கல் வைத்தல்**

இப்போது, முறையைச் சற்று மாற்றுவோம். எவ்வளவு இடைவெளி என்பதைக் குறிப்பிடாமல், ஒரு பாதையின் தொடக்கத்தையும் முடிவையும் சொல்லி, **"சரியாக இத்தனை கற்களை மட்டும் சமமான இடைவெளியில் வைக்க வேண்டும்"** என்று கூறினால் எப்படி? அதுதான் **`linspace`** (Linearly Spaced என்பதன் சுருக்கம்) செய்யும் வேலை.

நீங்கள் குறிப்பிடும் எண்ணிக்கையில் கற்களை சமமான இடைவெளியில் வைக்க, ஒவ்வொரு கல்லுக்கும் எவ்வளவு இடைவெளி தேவை என்பதை `linspace` தானாகவே கணக்கிட்டுக் கொள்ளும்.

**முக்கிய குறிப்பு:** `arange` போலல்லாமல், `linspace` **முடிவு மதிப்பையும் (`stop` value) அணியில் சேர்த்துக்கொள்ளும்.**

எடுத்துக்காட்டு:

0-விலிருந்து 10 மீட்டர் நீளமுள்ள ஒரு பாதையில், சரியாக 5 கற்களை சமமான இடைவெளியில் வைப்போம்.

Python

```
import numpy as np

# linspace(தொடக்கம், முடிவு, கற்களின் எண்ணிக்கை)
array_linspace = np.linspace(0, 10, 5)

print("linspace மூலம் உருவான அணி:", array_linspace)
```

**வெளியீடு:**

```
linspace மூலம் உருவான அணி: [ 0.   2.5  5.   7.5 10. ]
```

0-விற்கும் 10-க்கும் இடையில் 5 கற்களை சமமாக அடுக்க, ஒவ்வொரு கல்லுக்கும் 2.5 மீட்டர் இடைவெளி தேவை என்பதை `linspace` தானாகவே கணக்கிட்டு, கற்களை வைத்துவிட்டது.

### ௪.௩. `numpy.logspace` – அதிவேகமாக கல் வைத்தல்

சில நேரங்களில், பாதையின் தொடக்கத்தில் கற்களை நெருக்கமாகவும், போகப்போக கற்களின் இடைவெளியை அதிவேகமாக (exponentially) அதிகரித்துக்கொண்டே செல்ல வேண்டியிருக்கும். இதை ஒரு ராக்கெட் கிளம்புவது போல கற்பனை செய்துகொள்ளுங்கள்; முதலில் மெதுவாகவும், பின்னர் மிக வேகமாகவும் பயணிக்கும்.

**`logspace`** (Logarithmically Spaced என்பதன் சுருக்கம்) இந்த வேலையைச் செய்கிறது. இது மடக்கை அளவில் (`logarithmic scale`) சமமான இடைவெளியில் எண்களை உருவாக்கும். இது அதிர்வெண் பகுப்பாய்வு (`frequency analysis`) போன்ற அறிவியல் கணக்கீடுகளில் மிகவும் பயனுள்ளதாக இருக்கும்.

சுருக்கமாக, `logspace(1, 3, 5)` என்றால், 101-க்கும் 103-க்கும் இடையில், மடக்கை அளவில் 5 சம இடைவெளிகளை உருவாக்கு என்று பொருள்.

எடுத்துக்காட்டு:

101 (10) க்கும் 103 (1000) க்கும் இடையில் 5 கற்களை மடக்கை அளவில் வைப்போம்.

Python

```python
import numpy as np

# logspace(தொடக்கம்^, முடிவு^, எண்ணிக்கை) -> 10^1, 10^3
array_logspace = np.logspace(1, 3, 5)

print("logspace மூலம் உருவான அணி:", array_logspace)
```

**வெளியீடு:**

```
logspace மூலம் உருவான அணி: [  10.          31.6227766   100.         316.22776602
 1000.        ]
```

இங்கே, தொடக்கத்தில் இடைவெளி குறைவாகவும் (10-க்கும் 31-க்கும் இடையில்), போகப்போக இடைவெளி அதிவேகமாக அதிகரிப்பதையும் (316-க்கும் 1000-க்கும் இடையில்) காணலாம்.

**சுருக்கம்: எந்தக் கருவி எப்போது?**

| **செயற்கூறு (Function)** | **மையக்கரு (Core Idea)** | **எதைக் கட்டுப்படுத்துகிறோம்?** |
|---|---|---|
| `np.arange()` | குறிப்பிட்ட படி அளவு (Fixed Step Size) | ஒவ்வொரு படிக்கும் உள்ள **இடைவெளியை.** |
| `np.linspace()` | குறிப்பிட்ட படிகளின் எண்ணிக்கை (Fixed Number of Steps) | பாதையில் வைக்கப்படும் **மொத்த படிகளின் எண்ணிக்கையை.** |
| `np.logspace()` | அதிவேகப் படிகள் (Exponential Steps) | பரந்த எண் வரம்பை (பல மடங்கு மதிப்புகள்) ஆராய. |

## அத்தியாயம் ௫: NumPy – உங்களுக்குத் தேவையானதைத் தேர்ந்தெடுத்தல் (Indexing & Slicing)

ஒரு பெரிய நூலக அலமாரியில் (`bookshelf`) ஆயிரக்கணக்கான புத்தகங்கள் வரிசையாக அடுக்கப்பட்டிருப்பதாகக் கற்பனை செய்துகொள்ளுங்கள். NumPy அணி என்பதும் அப்படியான ஒரு நேர்த்தியான அலமாரிதான். இப்போது, அந்த அலமாரியிலிருந்து உங்களுக்குத் தேவையான புத்தகங்களை எப்படி எடுப்பீர்கள்?

- **ஒரு குறிப்பிட்ட புத்தகம் வேண்டுமானால்:** அதன் சரியான இடத்தைச் சொல்லி எடுப்பீர்கள். (உதாரணமாக, “மேலிருந்து இரண்டாவது வரிசையில், இடமிருந்து ஐந்தாவது புத்தகம்”).
- **ஒரு பகுதி புத்தகங்கள் வேண்டுமானால்:** “இந்த வரிசையில் உள்ள 3-வது புத்தகத்திலிருந்து 7-வது புத்தகம் வரை அனைத்தையும் கொடு” என்று கேட்பீர்கள்.

இந்த இரண்டு வழிகள்தான் NumPy-யில் **Indexing** மற்றும் **Slicing**. இது தரவுகளைப் பிரித்தெடுப்பதற்கும், கையாளுவதற்கும் மிகவும் அடிப்படையான மற்றும் சக்திவாய்ந்த முறையாகும்.

### ௫.௧. ஒரு வரிசை அலமாரி (1D Array) – அடிப்படைகள்

முதலில், ஒரே ஒரு வரிசை கொண்ட எளிய அலமாரியைப் பார்ப்போம்.

Python

```python
import numpy as np

# நமது ஒற்றை வரிசை அலமாரி
arr = np.array([10, 20, 30, 40, 50, 60])
```

### ௫.௨. Indexing: ஒரு குறிப்பிட்ட புத்தகத்தை எடுத்தல்

அலமாரியில் உள்ள ஒரு குறிப்பிட்ட புத்தகத்தை அதன் வரிசை எண்ணைச் சொல்லி எடுப்பதே **Indexing**.

**மிக முக்கியமான விதி:** NumPy (மற்றும் பெரும்பாலான நிரலாக்க மொழிகளில்) எண்ணுவது `0`-விலிருந்து தொடங்கும். எனவே, முதல் புத்தகம் `0`-வது இடத்திலும், இரண்டாவது புத்தகம் `1`-வது இடத்திலும் இருக்கும்.

Python

```
# 3-வது இடத்தில் (index 2) உள்ள புத்தகத்தை எடுப்போம்
print("3-வது இடத்தில் உள்ள உறுப்பு:", arr[2])

# கடைசிப் புத்தகத்தை எடுக்க ஒரு தந்திரம்: எதிர்மறை குறியீடு (negative index)
print("கடைசி உறுப்பு:", arr[-1])
print("கடைசியிலிருந்து இரண்டாவது உறுப்பு:", arr[-2])
```

**வெளியீடு:**

```
3-வது இடத்தில் உள்ள உறுப்பு: 30
கடைசி உறுப்பு: 60
கடைசியிலிருந்து இரண்டாவது உறுப்பு: 50
```

### ௫.௩. Slicing: ஒரு பகுதி புத்தகங்களை எடுத்தல்

அலமாரியின் ஒரு குறிப்பிட்ட பகுதியை அப்படியே வெட்டி எடுப்பதே **Slicing**. இதற்கு `[தொடக்கம் : முடிவு]` என்ற வடிவமைப்பைப் பயன்படுத்துவோம்.

**மிக முக்கியமான விதி:** `முடிவு` (`stop`) என்று குறிப்பிடும் இடத்திற்கு **முந்தைய இடம் வரை** மட்டுமே உறுப்புகள் எடுக்கப்படும். `arr[1:4]` என்றால், 1-வது இடத்திலிருந்து தொடங்கி, 4-வது இடத்திற்கு முன்புவரை (அதாவது 1, 2, 3) உள்ள உறுப்புகளை எடு என்று பொருள்.

Python

```
# 2-வது இடத்திலிருந்து (index 1) 5-வது இடத்திற்கு (index 4) முன்புவரை எடுப்போம்
print("ஒரு பகுதி (Slice):", arr[1:4])

# தொடக்கத்திலிருந்து 3-வது இடம் வரை
print("தொடக்கத்திலிருந்து ஒரு பகுதி:", arr[:3])

# 3-வது இடத்திலிருந்து கடைசி வரை
print("ஒரு இடத்திலிருந்து கடைசி வரை:", arr[2:])
```

**வெளியீடு:**

```
ஒரு பகுதி (Slice): [20 30 40]
தொடக்கத்திலிருந்து ஒரு பகுதி: [10 20 30]
ஒரு இடத்திலிருந்து கடைசி வரை: [30 40 50 60]
```

### ௫.௪. பல அடுக்கு அலமாரி (2D Array) – உண்மையான சக்தி

NumPy-யின் உண்மையான பலம், பல அடுக்குகள் கொண்ட அலமாரிகளை (2D அணிகளை) கையாளும்போதுதான் வெளிப்படுகிறது. இங்கே ஒரு புத்தகத்தை எடுக்க, அது **எந்த வரிசையில் (row)**, **எந்த இடத்தில் (column)** இருக்கிறது என்று சொல்ல வேண்டும்.

Python

```
# 3 வரிசைகள், 4 நிரல்கள் கொண்ட பல அடுக்கு அலமாரி
bookcase = np.array([[10, 11, 12, 13],
                     [20, 21, 22, 23],
                     [30, 31, 32, 33]])
```

### ஒரு குறிப்பிட்ட உறுப்பை எடுத்தல்

`[வரிசை, நிரல்]` என்ற குறியீட்டைப் பயன்படுத்துவோம்.

Python

```python
# 2-வது வரிசையில் (index 1), 3-வது இடத்தில் (index 2) உள்ள புத்தகம்
element = bookcase[1, 2]
print("குறிப்பிட்ட உறுப்பு:", element)  # வெளியீடு: 22
```

### ஒரு முழு வரிசையை (Row) எடுத்தல்

ஒரு வரிசையில் உள்ள அனைத்துப் புத்தகங்களையும் எடுக்க, நிரல் (`column`) இடத்தில் `:` குறியீட்டைப் பயன்படுத்தவும்.

Python

```python
# 2-வது வரிசையை (index 1) முழுமையாக எடுப்போம்
row_slice = bookcase[1, :]
print("ஒரு முழு வரிசை:", row_slice)  # வெளியீடு: [20 21 22 23]
```

### ஒரு முழு நிரலை (Column) எடுத்தல்

அலமாரியில் உள்ள ஒவ்வொரு வரிசையிலிருந்தும், ஒரு குறிப்பிட்ட இடத்தில் உள்ள புத்தகத்தை மட்டும் எடுக்க, வரிசை (`row`) இடத்தில் `:` குறியீட்டைப் பயன்படுத்தவும்.

Python

```python
# 3-வது நிரலை (index 2) முழுமையாக எடுப்போம்
col_slice = bookcase[:, 2]
print("ஒரு முழு நிரல்:", col_slice)  # வெளியீடு: [12 22 32]
```

### ஒரு பகுதி அலமாரியை எடுத்தல் (Sub-matrix)

அலமாரியின் ஒரு குறிப்பிட்ட செவ்வகப் பகுதியை வெட்டி எடுக்கலாம்.

Python

```python
# முதல் இரண்டு வரிசைகளில், 2-வது மற்றும் 3-வது நிரல்களை எடுப்போம்
sub_matrix = bookcase[0:2, 1:3]
print("ஒரு பகுதி அலமாரி:\n", sub_matrix)
```

**வெளியீடு:**

```
ஒரு பகுதி அலமாரி:
 [[11 12]
 [21 22]]
```

Indexing மற்றும் Slicing-ல் தேர்ச்சி பெறுவது, NumPy-யில் தரவுகளைக் கையாள்வதற்கான திறவுகோல். இதன் மூலம், எவ்வளவு பெரிய தரவுக் கட்டமைப்பாக இருந்தாலும், நமக்குத் தேவையான பகுதியைத் துல்லியமாகப் பிரித்தெடுக்க முடியும்.

## அத்தியாயம் ௬: NumPy – மேம்பட்ட தேர்வு முறைகள் (Advanced Indexing)

கடந்த அத்தியாயத்தில், நமது நூலக அலமாரியிலிருந்து ஒரு குறிப்பிட்ட புத்தகத்தையோ (`Indexing`) அல்லது ஒரு நேர்த்தியான பகுதிப் புத்தகங்களையோ (`Slicing`) எப்படி எடுப்பது என்று பார்த்தோம்.

ஆனால், உங்கள் தேவை இன்னும் சிக்கலானதாக இருந்தால் என்ன செய்வது?

- “முதல் வரிசையில் 2-வது புத்தகம், மூன்றாவது வரிசையில் 4-வது புத்தகம், இரண்டாவது வரிசையில் 1-வது புத்தகம் வேண்டும்” என்று **ஒரே நேரத்தில் பல இடங்களில் இருந்து** புத்தகங்களைக் கேட்க வேண்டுமானால்?
- “இந்த அலமாரியில், 500 பக்கங்களுக்கு மேல் உள்ள எல்லாப் புத்தகங்களையும் கொடுங்கள்” என்று **ஒரு நிபந்தனையின் அடிப்படையில்** புத்தகங்களைத் தேர்ந்தெடுக்க வேண்டுமானால்?

இப்படிப்பட்ட சிக்கலான தேர்வுகளைச் செய்வதற்கான மந்திரக் கருவிகள்தான் **மேம்பட்ட தேர்வு முறைகள் (Advanced Indexing)**.

### ௬.௧. Integer Array Indexing – உங்கள் விருப்பத் தேர்வுப் பட்டியல்

இது நூலகரிடம் ஒரு **தனிப்பயன் தேர்வுப் பட்டியலைக் (custom shopping list)** கொடுப்பது போன்றது. அந்தப் பட்டியலில், உங்களுக்குத் தேவையான புத்தகங்களின் வரிசை மற்றும் நிரல் எண்களை (`row` and `column` coordinates) துல்லியமாகக் குறிப்பிடுவீர்கள். நூலகர் (NumPy) அந்தப் பட்டியலைப் பார்த்து, அலமாரியின் வெவ்வேறு மூலைகளில் இருக்கும் அந்தப் புத்தகங்களை மட்டும் உங்களுக்காக எடுத்து வருவார்.

எடுத்துக்காட்டு:

நம்மிடம் உள்ள பல அடுக்கு அலமாரியிலிருந்து, குறிப்பிட்ட மூன்று புத்தகங்களை ஒரே நேரத்தில் எடுப்போம்.

Python

```python
import numpy as np

# நமது பல அடுக்கு அலமாரி
bookcase = np.array([[10, 11, 12, 13],
                     [20, 21, 22, 23],
                     [30, 31, 32, 33]])

# நமது தேர்வுப் பட்டியல்:
# நமக்கு வேண்டிய புத்தகங்களின் வரிசை எண்கள்: [0, 2, 1]
# நமக்கு வேண்டிய புத்தகங்களின் நிரல் எண்கள்:  [1, 3, 0]
rows = [0, 2, 1]
cols = [1, 3, 0]

# முதல் தேர்வு: வரிசை 0, நிரல் 1 -> 11
```

```
# இரண்டாம் தேர்வு: வரிசை 2, நிரல் 3 -> 33
# மூன்றாம் தேர்வு: வரிசை 1, நிரல் 0 -> 20

# பட்டியலை NumPy-யிடம் கொடுப்போம்
selected_books = bookcase[rows, cols]

print("தேர்வுப் பட்டியலிலிருந்து கிடைத்தவை:", selected_books)
```

**வெளியீடு:**

```
தேர்வுப் பட்டியலிலிருந்து கிடைத்தவை: [11 33 20]
```

ஒரே கட்டளையில், அலமாரியின் வெவ்வேறு இடங்களில் இருந்த புத்தகங்களை நம்மால் துல்லியமாகத் தேர்ந்தெடுக்க முடிந்தது. இதுதான் Integer Array Indexing-இன் சக்தி.

### ௯.௨. Boolean Array Indexing – மந்திர வடிப்பான்

இது ஒரு மிகவும் சக்திவாய்ந்த முறை. இதை அலமாரியின் மீது வைக்கும் ஒரு **மந்திர வடிப்பானுக்கு (magical filter)** ஒப்பிடலாம். நீங்கள் ஒரு நிபந்தனையை உருவாக்குவீர்கள். அந்த நிபந்தனை, அலமாரியில் உள்ள ஒவ்வொரு புத்தகத்தின் மீதும் சோதிக்கப்படும்.

1. **நிபந்தனை (Condition):** முதலில், “25-ஐ விட பெரிய எண்கள் எவை?” என்பது போன்ற ஒரு நிபந்தனையை உருவாக்குவோம்.
2. **வடிப்பான் உருவாதல் (Mask Creation):** NumPy, இந்த நிபந்தனையை ஒவ்வொரு உறுப்பின் மீதும் செலுத்தி, `True` (நிபந்தனை சரி) அல்லது `False` (நிபந்தனை தவறு) என்ற மதிப்புகள் கொண்ட ஒரு புதிய அணியை (வடிப்பானை) உருவாக்கும்.
3. **வடித்தெடுத்தல் (Filtering):** இந்த வடிப்பானை அசல் அணியின் மீது செலுத்தும்போது, `True` என்று குறிக்கப்பட்ட இடங்களில் உள்ள உறுப்புகள் மட்டுமே நமக்குக் கிடைக்கும்.

எடுத்துக்காட்டு:

நம் அலமாரியில், 25-ஐ விட பெரிய மதிப்புகள் கொண்ட புத்தகங்களை மட்டும் எடுப்போம்.

Python

```
# நமது அலமாரி
arr = np.array([[10, 20, 30],
                [40, 25, 15],
                [50, 60, 22]])

# படி 1: நிபந்தனையை உருவாக்குதல்
condition = arr > 25
print("மந்திர வடிப்பான் (Boolean Mask):\n", condition)

# படி 2: வடிப்பானைப் பயன்படுத்துதல்
filtered_array = arr[condition]
print("\nவடித்தெடுத்த பின் கிடைத்தவை:", filtered_array)
```

**வெளியீடு:**

```
மந்திர வடிப்பான் (Boolean Mask):
 [[False False  True]
 [ True False False]
 [ True  True False]]

வடித்தெடுத்த பின் கிடைத்தவை: [30 40 50 60]
```

`True` என்று எங்கெல்லாம் இருந்ததோ, அந்த இடங்களில் உள்ள மதிப்புகள் (`30, 40, 50, 60`) மட்டும் நமக்குக் கிடைத்துவிட்டன. இந்த முறை, தரவுகளை ஆராய்ந்து, குறிப்பிட்ட நிபந்தனைகளின் அடிப்படையில் அவற்றை வடிகட்டுவதற்கு மிக மிக அவசியம்.

இந்த மேம்பட்ட தேர்வு முறைகள், NumPy-ஐ ஒரு சாதாரணமான எண் கணிப்பானிலிருந்து, ஒரு சக்திவாய்ந்த தரவுப் பகுப்பாய்வுக் கருவியாக மாற்றுகின்றன.

## அத்தியாயம் எ: NumPy – மந்திர விரிவாக்கம் (Broadcasting)

சாதாரண கணிதத்தில், `10 + 5` என்று இரண்டு எண்களைக் கூட்ட முடியும். ஆனால், ஒரு பெட்டி நிறைய ஆப்பிள்கள் (`[🍎,🍎,🍎]`) இருக்கும்போது, அதனுடன் `+ 5` என்று கூட்ட முடியுமா? முடியாது. ஒவ்வொரு ஆப்பிளுடனும் 5-ஐத் தனித்தனியாகக் கூட்ட, நாம் ஒரு `for loop`-ஐப் பயன்படுத்த வேண்டும்.

ஆனால், NumPy உலகில் ஒரு மந்திரம் இருக்கிறது. அதன் பெயர் **Broadcasting**. இந்த மந்திரத்தின் மூலம், வெவ்வேறு வடிவங்கள் மற்றும் அளவுகள் கொண்ட அணிகளை நம்மால் எளிதாகக் கூட்டவோ, கழிக்கவோ முடியும். இதற்கு NumPy எந்த `loop`-களையும் பயன்படுத்துவதில்லை.

இதை ஒரு **மந்திர பெயிண்ட் ரோலருக்கு (magical paint roller)** ஒப்பிடலாம். நீங்கள் ஒரு துளி பெயிண்ட்டை (ஒரு எண்) எடுத்து, ஒரு பெரிய சுவரில் (ஓர் அணி) தேய்க்கும்போது, அந்த ஒரு துளி பெயிண்ட்டே சுவரின் ஒவ்வொரு அங்குலத்திற்கும் பரவுவது போல, Broadcasting செயல்படுகிறது.

### எ.க. Broadcasting எப்படி வேலை செய்கிறது?

Broadcasting என்பது, கணிதச் செயல்பாட்டின்போது, சிறிய அணியை, பெரிய அணியின் வடிவத்திற்குப் பொருந்தும் வகையில், **நினைவகத்தில் நகல் எடுக்காமலேயே, கிட்டத்தட்ட விரிவடையச் செய்யும்** ஒரு நுட்பமாகும். NumPy இந்த வேலையைத் தானாகவே சில விதிகளைப் பின்பற்றிச் செய்கிறது.

**எடுத்துக்காட்டு:** ஒரு வரிசை அணியையும் (`row vector`), ஒரு நிரல் அணியையும் (`column vector`) கூட்டுவோம்.

Python

```
import numpy as np

array1 = np.array([[10, 20, 30]])  # வடிவம்: (1, 3) - ஒரு நீண்ட படுக்கை துண்டு
array2 = np.array([[1], [2], [3]])   # வடிவம்: (3, 1) - ஒரு நீண்ட செங்குத்து துண்டு

# இந்த இரண்டின் வடிவங்களும் வெவ்வேறாக உள்ளன. ஆனாலும் NumPy இவற்றைக் கூட்டும்!
result = array1 + array2

print("Array1 வடிவம்:", array1.shape)
print("Array2 வடிவம்:", array2.shape)
print("\nBroadcasting-க்குப் பின் கிடைத்த அணி:\n", result)
print("\nமுடிவின் வடிவம்:", result.shape)
```

**வெளியீடு:**

```
Array1 வடிவம்: (1, 3)
Array2 வடிவம்: (3, 1)

Broadcasting-க்குப் பின் கிடைத்த அணி:
 [[11 21 31]
 [12 22 32]
 [13 23 33]]

முடிவின் வடிவம்: (3, 3)
```

**திரைக்குப் பின்னால் நடந்த மந்திரம் என்ன?**

1. NumPy இரண்டு அணிகளின் வடிவங்களையும் ஒப்பிட்டது: `(1, 3)` மற்றும் `(3, 1)`.
2. `array1` என்ற படுக்கை பட்டியை, `array2`-வின் உயரத்திற்குப் பொருந்தும் வகையில், **கீழ்நோக்கி கிட்டத்தட்ட விரிவாக்கியது.**
3. `array2` என்ற செங்குத்து பட்டியை, `array1`-வின் அகலத்திற்குப் பொருந்தும் வகையில், **பக்கவாட்டில் கிட்டத்தட்ட விரிவாக்கியது.**

- `array1` இப்படி மாறியது (நினைவில் அல்ல, கற்பனையில்):

  ```
  [[10, 20, 30],
   [10, 20, 30],
   [10, 20, 30]]
  ```

- `array2` இப்படி மாறியது (நினைவில் அல்ல, கற்பனையில்):

  ```
  [[1, 1, 1],
   [2, 2, 2],
   [3, 3, 3]]
  ```

இப்போது, இரண்டு அணிகளும் ஒரே வடிவத்தில் (`3, 3`) இருப்பதால், NumPy அவற்றை உறுப்பு வாரியாக எளிதாகக் கூட்டி, இறுதி விடையைத் தந்தது.

### எ.உ. Broadcasting ஏன் இவ்வளவு முக்கியம்?

Broadcasting என்பது வெறும் ஒரு வசதி மட்டுமல்ல, அது NumPy-யின் செயல்திறனுக்கு முதுகெலும்பாக இருக்கிறது.

- நினைவகச் சிக்கனம் (Memory Efficiency):

  மேலே பார்த்த “விரிவாக்குதல்” என்பது உண்மையிலேயே நடக்காது; NumPy அப்படி நடப்பதாகக் கற்பனை செய்துகொண்டு கணக்கீடுகளைச் செய்யும். இதனால், பெரிய அணிகளை நகல் எடுக்கத் தேவையில்லாததால், மிக மிகக் குறைந்த நினைவகமே பயன்படுத்தப்படுகிறது.

- எளிமையான குறியீடு (Code Simplification):

  for loop-கள் எழுதி நூறு வரிகளில் செய்ய வேண்டிய ஒரு வேலையை, array1 + array2 என்று ஒற்றை வரியில் அழகாக எழுதிவிட முடிகிறது. இது குறியீட்டைத் தெளிவாகவும், படிக்க எளிதாகவும் மாற்றுகிறது.

- அபார வேகம் (High Performance):

  இந்தக் கணிதச் செயல்பாடுகள் அனைத்தும், பின்னணியில் மிக வேகமாக இயங்கும் C மொழியில் எழுதப்பட்ட vectorized operations-ஐப் பயன்படுத்துவதால், சாதாரண Python loop-களை விடப் பல நூறு மடங்கு வேகத்தில் இயங்கும்.

சுருக்கமாக, Broadcasting என்பது வெவ்வேறு வடிவ அணிகளைக் கையாளும் ஒரு திறமையான, வேகமான மற்றும் நினைவகச் சிக்கனமான வழியாகும். இதுவே NumPy-ஐ Data Scienceக்கு தேவையான கருவியாக மாற்றுகிறது.

## அத்தியாயம் அ: NumPy அணியில் வலம் வருதல் (Iterating Over Array)

ஒரு NumPy அணியை, ஒரு தொழிற்சாலையின் **உற்பத்திப் பட்டை (Production Line)** 🏭 என்று கற்பனை செய்துகொள்ளுங்கள். அந்தப் பட்டையில், வரிசையாகப் பொருட்கள் (`products`) நகர்ந்து வருகின்றன. ஒவ்வொரு பொருளுக்கும் சில குணாதிசயங்கள் (`attributes`) உள்ளன; உதாரணமாக, அதன் விலை, விற்கப்பட்ட எண்ணிக்கை போன்றவை.

ஒரு மேலாளராக, அந்தப் பட்டையில் உள்ள ஒவ்வொரு பொருளையும் நாம் ஆய்வு செய்ய வேண்டும். இந்த ஆய்வுப் பயணத்தை (`iteration`) நாம் பல வழிகளில் மேற்கொள்ளலாம். ஒரு சாதாரண ஆய்வாளரைப் போல மெதுவாகச் செல்வதா, அல்லது ஒரு அதிவேக ரோபோவைப் (`robot`) போலத் திறமையாகவும் வேகமாகவும் ஆய்வு செய்வதா என்பதை நாம்தான் முடிவு செய்ய வேண்டும்.

### க. சாதாரண ஆய்வு முறை (Standard `for` Loop)

இதுதான் மிகவும் நேரடியான முறை. ஆய்வாளர், உற்பத்திப் பட்டையில் வரும் ஒவ்வொரு பொருளையும் (`row`) முழுமையாக எடுத்து, அதன் அனைத்து குணாதிசயங்களையும் ஆய்வு செய்வார்.

**எடுத்துக்காட்டு:** நம்மிடம் மூன்று பொருட்களின் விற்பனைத் தரவுகள் உள்ளன. ஒவ்வொரு பொருளுக்கும் `[அடையாள எண், விற்கப்பட்ட எண்ணிக்கை, ஒரு பொருளின் விலை]` என்ற விவரங்கள் உள்ளன.

Python

```python
import numpy as np

# நமது உற்பத்திப் பட்டை
sales_data = np.array([[101, 50, 2.50],  # பொருள் 101, 50 விற்றது, விலை $2.50
                       [102, 35, 15.00], # பொருள் 102, 35 விற்றது, விலை $15.00
                       [103, 80, 1.75]]) # பொருள் 103, 80 விற்றது, விலை $1.75
```

```
print("ஒவ்வொரு பொருளின் மொத்த வருவாய்:")
# ஒவ்வொரு பொருளாக ஆய்வு செய்கிறோம்
for product in sales_data:
    quantity = product[1]
    price = product[2]
    revenue = quantity * price
    print(f"பொருள் #{int(product[0])}: ${revenue:.2f}")
```

**வெளியீடு:**

```
ஒவ்வொரு பொருளின் மொத்த வருவாய்:
பொருள் #101: $125.00
பொருள் #102: $525.00
பொருள் #103: $140.00
```

இந்த முறை சிறிய தரவுகளுக்கு ஏற்றது. ஆனால், இலட்சக்கணக்கான பொருட்கள் பட்டையில் வரும்போது, இந்த ஆய்வாளர் (Python `for` loop) மிகவும் மெதுவாகச் செயல்படுவார்.

### உ. அதிவேக ரோபோ ஆய்வு (`np.nditer`)

இலட்சக்கணக்கான பொருட்களை ஆய்வு செய்ய, நமக்கு ஒரு அதிவேக, புத்திசாலியான ரோபோ தேவை. அந்த ரோபோதான் `np.nditer`. இது பின்னணியில் C மொழியின் வேகத்தில் இயங்குவதால், நம்பமுடியாத அளவிற்குத் திறமையாகச் செயல்படும்.

**ஆய்வின்போதே பொருட்களை மாற்றுதல்**

இந்த ரோபோவால், பொருட்களை ஆய்வு செய்வது மட்டுமல்லாமல், அவற்றை மாற்றியமைக்கவும் முடியும். உதாரணமாக, ஒரு நிபந்தனையின் அடிப்படையில் பொருட்களின் எண்ணிக்கையை மாற்றுவோம்: “ஒரு பொருளின் விலை $5-க்குக் குறைவாக இருந்தால், அதன் விற்பனை எண்ணிக்கையில் 10-ஐக் கூட்டு; இல்லையென்றால் 5-ஐக் கூட்டு.”

இதற்கு, `op_flags=['readwrite']` என்ற சிறப்பு அனுமதியை ரோபோவிற்கு வழங்க வேண்டும்.

Python

```
# 'readwrite' அனுமதியுடன் ரோபோவை அனுப்புவோம்
for product in np.nditer(sales_data, flags=['refs_ok'], op_flags=['readwrite']):
    # product என்பது முழு வரிசையையும் குறிக்கும் ஒரு object
    price = product[...][2]  # அந்த வரிசையில் உள்ள விலையை எடுக்கிறோம்

    if price < 5.0:
        product[...][1] += 10 # எண்ணிக்கையை 10 கூட்டுகிறோம்
    else:
        product[...][1] += 5  # எண்ணிக்கையை 5 கூட்டுகிறோம்

print("\nநிபந்தனையின்படி மாற்றப்பட்ட விற்பனை எண்ணிக்கை:\n", sales_data)
```

**வெளியீடு:**

```
நிபந்தனையின்படி மாற்றப்பட்ட விற்பனை எண்ணிக்கை:
 [[101.   60.    2.5]
 [102.   40.   15.  ]
 [103.   90.    1.75]]
```

இப்படிப்பட்ட சிக்கலான, நிபந்தனைகளுக்குட்பட்ட மாற்றங்களைச் செய்ய `nditer` மிகவும் சக்திவாய்ந்தது.

### ரோபோவின் ஆய்வுப் பாதையை மாற்றுதல்

ரோபோ வழக்கமாக பொருள் வாரியாக (`row-wise`) ஆய்வு செய்யும். ஆனால், **`order='F'`** கட்டளை மூலம், அதன் ஆய்வுப் பாதையை மாற்றி, முதலில் எல்லாப் பொருட்களின் அடையாள எண்களையும், அடுத்து விற்கப்பட்ட எண்ணிக்கையையும், கடைசியாக விலைகளையும் ஆய்வு செய்யும்படி (`column-wise`) நிரல்படுத்தலாம். **`flags=['external_loop']`** என்பதைச் சேர்க்கும்போது, ரோபோ ஒவ்வொரு நிரலையும் ஒரே மூச்சில், அதிவேகமாக ஆய்வு செய்யும்.

Python

```python
# அசல் தரவை மீண்டும் பயன்படுத்துவோம்
sales_data = np.array([[101, 50, 2.50], [102, 35, 15.00], [103, 80, 1.75]])

print("\nரோபோவின் நிரல் வாரியான அதிவேக ஆய்வு:")
for column_block in np.nditer(sales_data, flags=['external_loop'], order='F'):
    print("ஒரு முழு நிரல்:", column_block)
```

**வெளியீடு:**

```
ரோபோவின் நிரல் வாரியான அதிவேக ஆய்வு:
ஒரு முழு நிரல்: [101. 102. 103.]
ஒரு முழு நிரல்: [50. 35. 80.]
ஒரு முழு நிரல்: [ 2.5  15.    1.75]
```

தரவின் ஒரு குறிப்பிட்ட குணாதிசயத்தை மட்டும் ஆய்வு செய்ய இந்த முறை மிகவும் வேகமானது.

### ௩. இரண்டு உற்பத்திப் பட்டைகளை ஒப்பிடுதல் (Broadcasting Iteration)

நம்மிடம் இரண்டு வெவ்வேறு உற்பத்திப் பட்டைகள் உள்ளன. ஒன்றில் பொருட்களின் விலைகள் வருகின்றன. மற்றொன்றில், தள்ளுபடி சதவிகிதங்கள் (`discount percentages`) வருகின்றன. ஒவ்வொரு தள்ளுபடிக்கும், ஒவ்வொரு பொருளின் இறுதி விலை என்ன என்பதைக் கணக்கிட வேண்டும்.

Python

```python
product_prices = np.array([100.0, 200.0, 50.0])
discount_multipliers = np.array([[0.9], [0.8]]) # 10% மற்றும் 20% தள்ளுபடி

print("\nஒவ்வொரு தள்ளுபடிக்கும் இறுதி விலைகள்:")
# இரண்டு பட்டைகளையும் ஒரே நேரத்தில் ஆய்வு செய்ய ரோபோவை அனுப்புவோம்
for price, discount in np.nditer([product_prices, discount_multipliers]):
    final_price = price * discount
    print(f"₹{price:.2f} கொண்ட பொருள் {int((1-discount)*100)}% தள்ளுபடியில்: ₹
{final_price:.2f}")
```

**வெளியீடு:**

```
ஒவ்வொரு தள்ளுபடிக்கும் இறுதி விலைகள்:
₹100.00 கொண்ட பொருள் 10% தள்ளுபடியில்: ₹90.00
₹200.00 கொண்ட பொருள் 10% தள்ளுபடியில்: ₹180.00
₹50.00 கொண்ட பொருள் 10% தள்ளுபடியில்: ₹45.00
₹100.00 கொண்ட பொருள் 20% தள்ளுபடியில்: ₹80.00
₹200.00 கொண்ட பொருள் 20% தள்ளுபடியில்: ₹160.00
₹50.00 கொண்ட பொருள் 20% தள்ளுபடியில்: ₹40.00
```

Broadcasting விதிகளைப் பயன்படுத்தி, `nditer` ரோபோ இரண்டு வெவ்வேறு வடிவங்கள் கொண்ட தரவுப் பட்டைகளையும் எப்படி ஒருங்கிணைத்து ஆய்வு செய்ய வேண்டும் என்பதைத் தானாகவே புரிந்துகொண்டு செயல்படுகிறது.

| **முறை (Method)** | **உவமை (Analogy)** | **எப்போது பயன்படுத்த வேண்டும்?** |
|---|---|---|
| Basic `for` loop | சாதாரண ஆய்வாளர் | எளிய, பொருள் வாரியான ஆய்வுகளுக்கு (சிறிய தரவு). |
| `np.nditer` | அதிவேக ரோபோ | வேகம், சிக்கலான நிபந்தனைகள், மற்றும் மாற்றங்கள் தேவைப்படும்போது. |

## அத்தியாயம் ௯: NumPy அணியைக் கையாளுதல் (Array Manipulation)

ஒரு NumPy அணியை, ஒரு சிற்பியின் கையில் இருக்கும் **களிமண் தொகுதி (block of clay)** என்று கற்பனை செய்துகொள்ளுங்கள். ஒரு திறமையான சிற்பி, அந்தக் களிமண்ணை வெட்டி, நீட்டி, சுழற்றி, இணைத்து, பிரித்து ஒரு அழகான சிற்பத்தை உருவாக்குவார். அதுபோல, NumPy நமக்குத் தரும் கருவிகளைக் கொண்டு, நம்முடைய தரவு என்ற களிமண்ணை நமக்குத் தேவையான வடிவத்திற்கு மாற்றியமைக்க முடியும்.

இந்த அத்தியாயத்தில், ஒரு சிற்பியைப் போல, நம் தரவுத் தொகுதியை வெவ்வேறு கருவிகளைக் கொண்டு எப்படிச் செதுக்குவது என்று படிப்படியாகக் கற்றுக்கொள்வோம்.

### பகுதி ௧: வடிவத்தைச் செதுக்குதல் (Reshaping and Transposing)

முதலில், நம் களிமண் தொகுதியின் அடிப்படை வடிவத்தை மாற்றுவதற்கான கருவிகளைப் பார்ப்போம்.

`numpy.reshape` – களிமண்ணுக்குப் புதிய வடிவம் கொடுத்தல்

இது ஒரு சிற்பியின் மிக அடிப்படையான கருவி. ஒரு நீண்ட, தட்டையான களிமண் பட்டையை, ஒரு கனசதுர வடிவத்திற்கு மாற்றுவது போல, `reshape` ஒரு பரிமாண அணியை (`1D array`) பல பரிமாண அணியாக (`multi-dimensional array`) மாற்ற உதவுகிறது.

**முக்கிய விதி:** களிமண்ணின் மொத்த அளவு மாறக்கூடாது. அதாவது, அணியில் உள்ள மொத்த உறுப்புகளின் எண்ணிக்கை, புதிய வடிவத்திற்கும் பொருந்த வேண்டும்.

Python

```
import numpy as np

# 12 உறுப்புகள் கொண்ட ஒரு நீண்ட களிமண் பட்டை
clay_strip = np.arange(12)  # [0, 1, 2, ..., 11]
print("பழைய வடிவம்:", clay_strip.shape)

# அதை 3 வரிசைகள், 4 நிரல்கள் கொண்ட செவ்வகமாக மாற்றுவோம்
clay_block = clay_strip.reshape(3, 4)
print("\nபுதிய வடிவம் (3, 4):\n", clay_block)
```

**வெளியீடு:**

```
பழைய வடிவம்: (12,)

புதிய வடிவம் (3, 4):
 [[ 0  1  2  3]
 [ 4  5  6  7]
 [ 8  9 10 11]]
```

`numpy.transpose` மற்றும் `.T` – சிற்பத்தைத் திருப்புதல்

இப்போது, நாம் உருவாக்கிய களிமண் சிற்பத்தின் வரிசைகளை நிரல்களாகவும், நிரல்களை வரிசைகளாகவும், அதாவது கிடைமட்டத்தை செங்குத்தாகவும், செங்குத்தைக் கிடைமட்டமாகவும் திருப்ப வேண்டும். இதற்கு `transpose` உதவுகிறது. இதன் சுருக்கமான வழிதான் `.T`.

Python

```
# (3, 4) வடிவ சிற்பத்தை திருப்புவோம்
transposed_block = clay_block.transpose()
# அல்லது சுருக்கமாக: transposed_block = clay_block.T

print("\nதிருப்பப்பட்ட வடிவம் (4, 3):\n", transposed_block)
```

**வெளியீடு:**

```
திருப்பப்பட்ட வடிவம் (4, 3):
 [[ 0  4  8]
 [ 1  5  9]
 [ 2  6 10]
 [ 3  7 11]]
```

**பகுதி உ: தட்டையாக்குதல் மற்றும் விரித்தல் (Flattening and Expanding)**

சில நேரங்களில், நம்முடைய பல அடுக்கு சிற்பத்தை ஒரே நேர்கோடாகத் தட்டையாக்க வேண்டியிருக்கும், அல்லது ஒரு நேர்கோட்டிற்குப் புதிய பரிமாணத்தைச் சேர்க்க வேண்டியிருக்கும்.

`flatten` மற்றும் `ravel` – சிற்பத்தை ஒரே நேர்கோடாக்குதல்

இந்த இரண்டு கருவிகளுமே, பல பரிமாண அணியை ஒற்றைப் பரிமாண அணியாக (`1D array`) மாற்றும். ஆனால், அவற்றுக்கு இடையே ஒரு முக்கியமான வேறுபாடு உள்ளது.

- `flatten`: இது சிற்பத்தின் ஒரு **புதிய நகலை (new copy)** உருவாக்கி அதைத் தட்டையாக்கும். அசல் சிற்பத்தில் மாற்றம் செய்தால், இந்த நகல் மாறாது.
- `ravel`: இது பெரும்பாலும் சிற்பத்தின் ஒரு **பார்வையை (view)** மட்டுமே தரும். அசல் சிற்பத்தில் மாற்றம் செய்தால், இந்தத் தட்டையான வடிவமும் மாறும். இது நினைவகத்தைச் சிக்கனப்படுத்தும்.

Python

```python
# (3, 4) வடிவ சிற்பத்தைத் தட்டையாக்குவோம்
flattened_copy = clay_block.flatten()
raveled_view = clay_block.ravel()

print("Flatten மூலம்:", flattened_copy)
print("Ravel மூலம்:", raveled_view)

# இப்போது அசல் சிற்பத்தில் ஒரு மாற்றத்தைச் செய்வோம்
clay_block[0, 0] = 99

print("\nமாற்றத்திற்குப் பின் Ravel:", raveled_view)
print("மாற்றத்திற்குப் பின் Flatten:", flattened_copy)
```

**வெளியீடு:**

```
...
மாற்றத்திற்குப் பின் Ravel: [99  1  2  3  4  5  6  7  8  9 10 11]
மாற்றத்திற்குப் பின் Flatten: [ 0  1  2  3  4  5  6  7  8  9 10 11]
```

**முக்கியக் குறிப்பு:** நினைவகச் சிக்கனம் தேவைப்படும்போது `ravel`-ஐப் பயன்படுத்துவது சிறந்தது.

`expand_dims` மற்றும் `squeeze` – புதிய பரிமாணத்தைச் சேர்த்தல் மற்றும் நீக்குதல்

- `expand_dims`: ஒரு தட்டையான களிமண் பட்டைக்கு, உயரத்தைக் கொடுத்து அதை ஒரு 3D பொருளாக மாற்றுவது போல, இது ஒரு அணிக்கு புதிய பரிமாணத்தை (`axis`) சேர்க்கிறது.
- `squeeze`: தேவையற்ற, ஒற்றை உறுப்பு மட்டுமே கொண்ட பரிமாணங்களை (உதாரணமாக, ஒரு பெட்டிக்குள் இன்னொரு பெட்டி, அதற்குள் ஒரே ஒரு பொருள் இருப்பது போல) நீக்கி, அணியைச் சுருக்குகிறது.

Python

```python
strip = np.array([1, 2, 3]) # வடிவம்: (3,)

# ஒரு புதிய பரிமாணத்தைச் சேர்ப்போம்
expanded = np.expand_dims(strip, axis=0)
print("விரிவாக்கப்பட்ட அணி:", expanded)
print("புதிய வடிவம்:", expanded.shape) # வடிவம்: (1, 3)

# தேவையற்ற பரிமாணத்தை நீக்குவோம்
squeezed = np.squeeze(expanded)
print("\nசுருக்கப்பட்ட அணி:", squeezed)
print("மீண்டும் பழைய வடிவம்:", squeezed.shape) # வடிவம்: (3,)
```

### பகுதி ௩: இணைத்தல் மற்றும் பிரித்தல் (Joining and Splitting)

ஒரு சிற்பி, பல களிமண் துண்டுகளை இணைத்து ஒரு பெரிய சிற்பத்தையோ, அல்லது ஒரு பெரிய துண்டைப் பிரித்து பல சிறிய சிற்பங்களையோ உருவாக்குவார்.

`concatenate`, `vstack`, `hstack` – களிமண் துண்டுகளை இணைத்தல்

- **`concatenate`**: இரண்டு அல்லது அதற்கு மேற்பட்ட அணிகளை, குறிப்பிட்ட அச்சு (`axis`) வழியாக ஒன்றன் பின் ஒன்றாக இணைக்கிறது.
- **`vstack`**: (Vertical Stack) அணிகளை ஒன்றன் கீழ் ஒன்றாக, செங்குத்தாக அடுக்கி இணைக்கிறது.
- **`hstack`**: (Horizontal Stack) அணிகளை ஒன்றன் பின் ஒன்றாக, கிடைமட்டமாக இணைக்கிறது.

Python

```python
clay1 = np.array([[1, 2], [3, 4]])
clay2 = np.array([[5, 6], [7, 8]])

# செங்குத்தாக இணைத்தல்
v_stacked = np.vstack((clay1, clay2))
print("செங்குத்து இணைப்பு (vstack):\n", v_stacked)

# கிடைமட்டமாக இணைத்தல்
h_stacked = np.hstack((clay1, clay2))
print("\nகிடைமட்ட இணைப்பு (hstack):\n", h_stacked)
```

`split`, `vsplit`, `hsplit` – களிமண் தொகுதியைப் பிரித்தல்

இது இணைத்தலுக்கு நேர்மாறான செயல். `split`, ஒரு அணியை பல சமமான சிறிய அணிகளாகப் பிரிக்கிறது. `vsplit` செங்குத்தாகவும், `hsplit` கிடைமட்டமாகவும் பிரிக்கிறது.

Python

```python
large_block = np.arange(16).reshape(4, 4)

# இரண்டாகச் செங்குத்தாகப் பிரித்தல்
v_splitted = np.vsplit(large_block, 2)
print("செங்குத்துப் பிரிவுகள்:\n", v_splitted)
```

### பகுதி ௪: உறுப்புகளைச் சேர்த்தல், நீக்குதல் மற்றும் சீராக்குதல்

இறுதியாக, சிற்பத்தைச் செதுக்கி, தேவையற்ற பகுதிகளை நீக்கி, புதிய பகுதிகளைச் சேர்த்து, அதை மெருகேற்றுவோம்.

`append`, `insert`, `delete` – சேர்த்தல் மற்றும் நீக்குதல்

- **`append`**: அணியின் இறுதியில் புதிய உறுப்புகளைச் சேர்க்கிறது.
- **`insert`**: அணியின் குறிப்பிட்ட இடத்தில் (`index`) புதிய உறுப்புகளைச் செருகுகிறது.
- **`delete`**: அணியின் குறிப்பிட்ட இடத்தில் உள்ள உறுப்புகளை நீக்குகிறது.

**முக்கியக் குறிப்பு:** இந்த செயல்பாடுகள், அசல் அணியை மாற்றாது; ஒரு **புதிய அணியை** உருவாக்கும்.

Python

```
arr = np.array([1, 2, 5, 6])

# இறுதியில் 7, 8-ஐச் சேர்த்தல்
appended = np.append(arr, [7, 8])
print("Append:", appended)

# 2-வது இடத்தில் (index 2) 3, 4-ஐச் செருகுதல்
inserted = np.insert(arr, 2, [3, 4])
print("Insert:", inserted)

# 1-வது இடத்தில் (index 1) உள்ள உறுப்பை நீக்குதல்
deleted = np.delete(arr, 1)
print("Delete:", deleted)
```

`unique` – நகல்களை நீக்கி மெருகேற்றுதல்

ஒரு சிற்பத்தில் ஒரே மாதிரியான பகுதிகள் மீண்டும் மீண்டும் வந்தால், அதைச் சீராக்குவது போல, `unique` ஓர் அணியில் மீண்டும் மீண்டும் வரும் உறுப்புகளை (`duplicates`) நீக்கி, தனித்துவமான உறுப்புகளை மட்டுமே தருகிறது.

Python

```
messy_clay = np.array([1, 2, 2, 3, 4, 1, 4, 5])
unique_elements = np.unique(messy_clay)
print("\nதனித்துவமான உறுப்புகள்:", unique_elements)
```

**வெளியீடு:**

```
தனித்துவமான உறுப்புகள்: [1 2 3 4 5]
```

இந்தக் கருவிகளைக் கொண்டு, உங்கள் தரவு என்ற களிமண்ணை, உங்கள் கற்பனைக்கு ஏற்றவாறு எந்த வடிவத்திலும் செதுக்க முடியும்.

## அத்தியாயம் ௰: NumPy – இருமச் செயல்பாடுகள் (Binary Operators)

நாம் கணினிக்குக் கொடுக்கும் ஒவ்வொரு எண்ணும், தகவலும், அதன் இதயத்தில் `0` மற்றும் `1` என்ற இரும எண்களாகவே சேமிக்கப்படுகிறது. இதை, ஒரு **கட்டுப்பாட்டுப் பலகையில் (Control Panel)** உள்ள சுவிட்சுகள் (`switches`) என்று கற்பனை செய்துகொள்ளுங்கள். `1` என்றால் சுவிட்ச் **ON** நிலையில் உள்ளது, `0` என்றால் **OFF** நிலையில் உள்ளது.

NumPy-யின் இருமச் செயல்பாடுகள், இந்தக் கட்டுப்பாட்டுப் பலகையில் உள்ள ஒவ்வொரு சுவிட்சையும் தனித்தனியாகக் கையாளும் கருவிகளை நமக்குத் தருகின்றன. இதன் மூலம், மிகத் திறமையாகவும் வேகமாகவும் தரவைக் கையாள முடியும்.

**நமது எடுத்துக்காட்டு:** ஒரு கோப்பிற்கான அனுமதிகளை (`permissions`) 4 சுவிட்சுகள் கொண்ட ஒரு கட்டுப்பாட்டுப் பலகையில் சேமிப்போம்.

- `8 (`23`)`: படிக்கும் அனுமதி (READ)
- `4 (`22`)`: எழுதும் அனுமதி (WRITE)

- `2 (`21`)`: இயக்கும் அனுமதி (EXECUTE)
- `1 (`20`)`: பகிரும் அனுமதி (SHARE)

ஒரு பயனரின் அனுமதி `13` என்றால், அதன் இரும வடிவம் `1101`. அதாவது, அவருக்கு READ (`8`), WRITE (`4`), SHARE (`1`) அனுமதிகள் உள்ளன, ஆனால் EXECUTE (`2`) அனுமதி இல்லை (`8 + 4 + 0 + 1 = 13`).

**க. தர்க்கரீதியான சுவிட்சுகள் (Logical Switches)**

`numpy.bitwise_and` – ஒரு குறிப்பிட்ட சுவிட்ச் ON-ல் உள்ளதா எனச் சோதிக்க

ஒரு பயனருக்கு `WRITE` அனுமதி உள்ளதா என எப்படிச் சோதிப்பது?

`WRITE` சுவிட்சை மட்டும் ON செய்து, ஒரு “மாஸ்க்” (`mask`) உருவாக்குவோம் (`WRITE_MASK = 4`, அதாவது `0100`). இப்போது, பயனரின் அனுமதியையும் இந்த மாஸ்கையும் `AND` செய்தால், அந்தச் சுவிட்ச் ON-ல் இருந்தால் மட்டுமே நமக்கு ஒரு மதிப்பு கிடைக்கும்.

**விதி:** `AND` செயல்பாட்டில், இரண்டு சுவிட்சுகளும் `1` (ON) ஆக இருந்தால் மட்டுமே, வெளியீடு `1` (ON) ஆக இருக்கும்.

Python

```python
import numpy as np

user_permission = 13  # இருமம்: 1101
WRITE_MASK = 4       # இருமம்: 0100

# AND செயல்பாட்டைச் செய்வோம்
has_write_permission = np.bitwise_and(user_permission, WRITE_MASK)

print(f"பயனர் அனுமதி (1101) AND மாஸ்க் (0100) = {has_write_permission} (இருமம்:
{has_write_permission:04b})")

if has_write_permission:
    print("பயனருக்கு WRITE அனுமதி உள்ளது.")
```

**வெளியீடு:**

```
பயனர் அனுமதி (1101) AND மாஸ்க் (0100) = 4 (இருமம்: 0100)
பயனருக்கு WRITE அனுமதி உள்ளது.
```

`numpy.bitwise_or` – ஒரு குறிப்பிட்ட சுவிட்சை ON செய்தல்

ஒரு பயனருக்கு, மற்ற அனுமதிகளை மாற்றாமல், `EXECUTE` அனுமதியை மட்டும் புதிதாக வழங்குவது எப்படி?

`EXECUTE` சுவிட்சை மட்டும் ON செய்து, ஒரு மாஸ்க் உருவாக்குவோம் (`EXECUTE_MASK = 2`, அதாவது `0010`). இதை, பயனரின் அனுமதியுடன் `OR` செய்தால், அந்தச் சுவிட்ச் மட்டும் ON நிலைக்கு மாறிவிடும்.

**விதி:** `OR` செயல்பாட்டில், ஏதாவது ஒரு சுவிட்ச் `1` (ON) ஆக இருந்தாலும், வெளியீடு `1` (ON) ஆக இருக்கும்.

Python

```
user_permission = 13  # இருமம்: 1101
EXECUTE_MASK = 2    # இருமம்: 0010

# OR செயல்பாட்டைச் செய்வோம்
new_permission = np.bitwise_or(user_permission, EXECUTE_MASK)

print(f"புதிய அனுமதி: {new_permission} (இருமம்: {new_permission:04b})")
```

**வெளியீடு:**

```
புதிய அனுமதி: 15 (இருமம்: 1111)
```

`EXECUTE` சுவிட்ச் இப்போது ON ஆகிவிட்டது (`1101` -> `1111`).

`numpy.invert` – எல்லா சுவிட்சுகளையும் திருப்புதல்

`invert` (அல்லது `bitwise_not`) என்ற கருவி, கட்டுப்பாட்டுப் பலகையில் உள்ள **எல்லா சுவிட்சுகளையும்** ஒரே நேரத்தில் தலைகீழாக மாற்றும். ON-ல் இருப்பவை OFF ஆகும், OFF-ல் இருப்பவை ON ஆகும்.

**குறிப்பு:** இது, கணினி பயன்படுத்தும் மொத்த பிட்களின் (bits) அடிப்படையில் செயல்படும். நாம் `uint8` (8-bit integer) பயன்படுத்தினால், 8 சுவிட்சுகளும் தலைகீழாக மாறும்.

Python

```
# 8-bit கட்டுப்பாட்டுப் பலகையில் நமது அனுமதி
permission_8bit = np.array([13], dtype=np.uint8) # இருமம்: 00001101

inverted_permission = np.invert(permission_8bit)

print(f" தலைகீழ் அனுமதி: {inverted_permission[0]} (இருமம்: {inverted_permission[0]:08b})")
```

**வெளியீடு:**

```
 தலைகீழ் அனுமதி: 242 (இருமம்: 11110010)
```

### உ . இடமாறும் சுவிட்சுகள் (Shifting Switches)

இந்தக் கருவிகள், கட்டுப்பாட்டுப் பலகையில் உள்ள சுவிட்சுகளை இடதுபுறமாகவோ வலதுபுறமாகவோ நகர்த்தும். இது, எண்களை 2-இன் அடுக்குகளால் பெருக்குவதற்கும் வகுப்பதற்கும் ஒரு அதிவேக வழியாகும்.

`numpy.left_shift` – மதிப்பை இரட்டிப்பு செய்தல்

சுவிட்சுகளை ஒரு இடம் இடதுபுறம் நகர்த்துவது, எண்ணை `2`-ஆல் பெருக்குவதற்குச் சமம். இரண்டு இடங்கள் நகர்த்துவது, `4`-ஆல் ($2^2$) பெருக்குவதற்குச் சமம்.

Python

```
value = 5  # இருமம்: 0101
# 2 இடங்கள் இடதுபுறம் நகர்த்துவோம் (5 * 4 = 20)
shifted_left = np.left_shift(value, 2)

print(f"{value} (0101) இடதுபுறம் நகர்த்தினால்: {shifted_left} (இருமம்: 10100)")
```

**வெளியீடு:**

```
5 (0101) இடதுபுறம் நகர்த்தினால்: 20 (இருமம்: 10100)
```

`numpy.right_shift` – மதிப்பை பாதியாகக் குறைத்தல்

சுவிட்சுகளை ஒரு இடம் வலதுபுறம் நகர்த்துவது, எண்ணை `2`-ஆல் வகுப்பதற்குச் சமம்.

Python

```
value = 40  # இருமம்: 101000
# 3 இடங்கள் வலதுபுறம் நகர்த்துவோம் (40 / 8 = 5)
shifted_right = np.right_shift(value, 3)

print(f"{value} (101000) வலதுபுறம் நகர்த்தினால்: {shifted_right} (இருமம்: 000101)")
```

**வெளியீடு:**

```
40 (101000) வலதுபுறம் நகர்த்தினால்: 5 (இருமம்: 000101)
```

இந்த இருமச் செயல்பாடுகள், மிகக் குறைந்த அளவில் தரவைக் கையாள்வதால், இவை தரவு சுருக்கம்(compression), image processing மற்றும் உயர் செயல்திறன் தேவைப்படும் பல இடங்களில் அடித்தளமாக விளங்குகின்றன.

## ௰௧. NumPy – எழுத்துக்களைச் செதுக்குதல் (String Functions)

NumPy-ஐ எண்களின் உலகம் என்று பார்த்தோம். ஆனால், எழுத்துக்கள் மற்றும் சொற்கள் நிறைந்த தரவுகளைக் கையாளவும் NumPy ஒரு சக்திவாய்ந்த கருவியைக் கொண்டுள்ளது. இதை ஒரு **டிஜிட்டல் பதிப்பகத்தின் (Digital Publishing House)** ஆசிரியர் குழுவாகக் கற்பனை செய்துகொள்ளுங்கள்.

உங்கள் கையில் ஆயிரக்கணக்கான வரிகள் கொண்ட ஒரு கையெழுத்துப் பிரதி (`manuscript`) இருக்கிறது. அதில் உள்ள எல்லாத் தலைப்புகளையும் பெரிய எழுத்துக்களாக (`uppercase`) மாற்ற வேண்டும், அல்லது ஒரு குறிப்பிட்ட பெயரை மாற்றி எழுத வேண்டும். ஒரு பதிப்பாசிரியராக இருந்தால், நீங்கள் வரி வரியாகச் சென்று இந்த மாற்றங்களைச் செய்ய வேண்டும். இது மிகவும் மெதுவான செயல்.

ஆனால், NumPy-யின் `numpy.char` என்ற ஆசிரியர் குழு, ஒரு **அதிவேக, தானியங்கி இயந்திரம்** போலச் செயல்படும். நீங்கள் ஒரு கட்டளையைக் கொடுத்தால் போதும், அது கையெழுத்துப் பிரதியில் உள்ள எல்லா வரிகளுக்கும் ஒரே நேரத்தில் அந்த மாற்றத்தைச் செய்துவிடும். இந்த **வெக்டரைசேஷன் (vectorization)** திறன்தான், NumPy-ஐ எழுத்துக்களைக் கையாள்வதில் மிக வேகமானதாக மாற்றுகிறது.

### க. எழுத்து வடிவத்தை மாற்றுதல் (Changing Case)

கையெழுத்துப் பிரதியின் தோற்றத்தை அழகுபடுத்துவது போல, எழுத்துக்களின் வடிவத்தை (`case`) மாற்றுவது முதல் படி.

Python

```
import numpy as np

# நமது கையெழுத்துப் பிரதியில் உள்ள சில வரிகள்
manuscript = np.array(['anna university', 'iit madras', 'data science course'])
```

- `np.char.upper()`: அனைத்தையும் பெரிய எழுத்துக்களாக மாற்ற.

  Python

  ```
  print("Upper Case:", np.char.upper(manuscript))
  # வெளியீடு: ['ANNA UNIVERSITY' 'IIT MADRAS' 'DATA SCIENCE COURSE']
  ```

- `np.char.lower()`: அனைத்தையும் சிறிய எழுத்துக்களாக மாற்ற.

  Python

  ```
  print("Lower Case:", np.char.lower(np.array(['TAMIL', 'NUMPY'])))
  # வெளியீடு: ['tamil' 'numpy']
  ```

- `np.char.title()`: ஒவ்வொரு வார்த்தையின் முதல் எழுத்தையும் பெரிய எழுத்தாக மாற்ற.

  Python

  ```
  print("Title Case:", np.char.title(manuscript))
  # வெளியீடு: ['Anna University' 'Iit Madras' 'Data Science Course']
  ```

- `np.char.capitalize()`: ஒரு வரியின் முதல் எழுத்தை மட்டும் பெரிய எழுத்தாக மாற்ற.

  Python

  ```
  print("Capitalize:", np.char.capitalize(manuscript))
  # வெளியீடு: ['Anna university' 'Iit madras' 'Data science course']
  ```

### உ. சொற்களை இணைத்தல் மற்றும் பிரித்தல் (Joining and Splitting)

ஆசிரியர், வாக்கியங்களை இணைப்பதும் பிரிப்பதும் போல, நாமும் சொற்களைக் கையாளலாம்.

- `np.char.add()`: இரண்டு அணிகளில் உள்ள சொற்களை ஒன்றோடு ஒன்று இணைக்க.

  Python

  ```
  first_names = np.array(['ஆதித்யா', 'கவின்'])
  last_names = np.array([' கரிகாலன்', ' மலர்'])
  full_names = np.char.add(first_names, last_names)
  print("முழுப் பெயர்கள்:", full_names)
  # வெளியீடு: ['ஆதித்யா கரிகாலன்' 'கவின் மலர்']
  ```

- `np.char.split()`: ஒரு வரியை, குறிப்பிட்ட பிரிப்பான் (`separator`) மூலம் சொற்களாகப் பிரிக்க.

Python

```
full_text = np.array(['NumPy-For-Data-Science', 'Learn-Python-Easily'])
split_text = np.char.split(full_text, sep='-')
print("பிரிக்கப்பட்ட சொற்கள்:", split_text)
# வெளியீடு: [list(['NumPy', 'For', 'Data', 'Science']) list(['Learn', 'Python',
'Easily'])]
```

- `np.char.join()`: ஒரு பிரிப்பானை, ஒரு வரியில் உள்ள ஒவ்வொரு எழுத்திற்கும் இடையில் சேர்க்க.

  Python

  ```
  joined_text = np.char.join('-', np.array(['numpy', 'ai']))
  print("இணைக்கப்பட்ட எழுத்துக்கள்:", joined_text)
  # வெளியீடு: ['n-u-m-p-y' 'a-i']
  ```

### ௩. தேடுதல் மற்றும் மாற்றுதல் (Finding and Replacing)

கையெழுத்துப் பிரதியில் ஒரு குறிப்பிட்ட சொல்லைத் தேடுவதும், அதை மற்றொரு சொல்லால் மாற்றுவதும் மிக முக்கியமான பணி.

- `np.char.replace()`: ஒரு வரியில் உள்ள ஒரு சொல்லை, வேறொரு சொல் கொண்டு மாற்ற.

  Python

  ```
  text = 'NumPy is a powerful tool. NumPy is easy.'
  replaced_text = np.char.replace(text, 'NumPy', 'Pandas')
  print("மாற்றப்பட்ட வரி:", replaced_text)
  # வெளியீடு: Pandas is a powerful tool. Pandas is easy.
  ```

- `np.char.count()`: ஒரு வரியில் ஒரு சொல் எத்தனை முறை வந்துள்ளது எனக் கணக்கிட.

  Python

  ```
  text_array = np.array(['data data science', 'big data analytics'])
  print("Count 'data':", np.char.count(text_array, 'data'))
  # வெளியீடு: [2 1]
  ```

### ௪. சுத்தம் செய்தல் (Cleaning and Trimming)

வெளியிடுவதற்கு முன், கையெழுத்துப் பிரதியில் உள்ள தேவையற்ற காலி இடங்களை (`whitespaces`) நீக்கிச் சுத்தம் செய்வது அவசியம்.

- `np.char.strip()`: ஒரு சொல்லின் இருபுறமும் உள்ள காலி இடங்களை நீக்க.

  Python

  ```
  dirty_data = np.array(['  chennai  ', '\tmadurai\n', ' coimbatore '])
  clean_data = np.char.strip(dirty_data)
  print("சுத்தம் செய்யப்பட்ட தரவு:", clean_data)
  # வெளியீடு: ['chennai' 'madurai' 'coimbatore']
  ```

`lstrip()` இடதுபுறமும், `rstrip()` வலதுபுறமும் உள்ள காலி இடங்களை மட்டும் நீக்கும்.

NumPy-யின் இந்த எழுத்துச் செயல்பாடுகள், குறிப்பாக பெரிய உரைத் தரவுகளை (`text data`) கையாளும்போது, Python-இன் சாதாரண `loop`-களை விடப் பன்மடங்கு வேகமாகவும், நினைவகத்தைச் சிக்கனப்படுத்தியும் செயல்படும்.

## ௰௨. NumPy – கணிதக் கருவூலம் (Mathematical Functions)

சாதாரண கால்குலேட்டரில், ஒரு நேரத்தில் ஒரு கணக்கீடுதான் செய்ய முடியும். ஆனால், உங்களிடம் பல்லாயிரம் எண்கள் கொண்ட ஒரு தரவுத் தொகுதி இருந்தால் என்ன செய்வது? ஒவ்வொரு எண்ணுக்கும் தனித்தனியாகக் கணக்கிடுவது பல மணி நேரங்கள் ஆகும்.

இங்குதான் NumPy, ஒரு **சூப்பர்சார்ஜ் செய்யப்பட்ட அறிவியல் கால்குலேட்டராக** மாறுகிறது. நீங்கள் பல்லாயிரம் எண்கள் கொண்ட ஒரு முழு அணியையும் (`array`) ஒரே ஒரு கட்டளையில் கொடுத்தால் போதும், அது நொடிப்பொழுதில் அத்தனை எண்களுக்கும் கணிதச் செயல்பாட்டைச் செய்து முடித்துவிடும். இந்த **வெக்டரைசேஷன் (vectorization)** எனும் மந்திரம்தான் NumPy-யின் கணிதக் கருவூலத்தின் திறவுகோல்.

### ௰௨.௧. முக்கோணவியல் செயல்பாடுகள் (Trigonometric Functions)

அலைகள், வட்டங்கள், சுழற்சிகள் என இயற்கையின் வடிவங்களைப் புரிந்துகொள்ளவும், உருவாக்கவும் பயன்படும் கருவிகளே முக்கோணவியல் செயல்பாடுகள்.

**எடுத்துக்காட்டு:** வெறும் எண்களைக் கொண்டு ஒரு வட்டத்தை வரைவோம்!

Python

```python
import numpy as np
import matplotlib.pyplot as plt

# 0-விலிருந்து 360 டிகிரி (2*pi ரேடியன்) வரை 100 சமமான கோணங்களை உருவாக்குவோம்
angles = np.linspace(0, 2 * np.pi, 100)

# அந்த 100 கோணங்களுக்கும் ஒரே நேரத்தில் sin மற்றும் cos மதிப்புகளைக் கணக்கிடுவோம்
x_coords = np.cos(angles)
y_coords = np.sin(angles)

# வட்டத்தை வரைந்து பார்ப்போம்
plt.figure(figsize=(5,5))
plt.plot(x_coords, y_coords)
plt.title("NumPy மூலம் உருவான வட்டம்")
plt.grid(True)
plt.show()
```

**ஒரு சிறிய விளக்கம்:** `np.sin(np.pi)`-இன் மதிப்பு சரியாக `0` என்று வராமல், `1.224647e-16` (அதாவது $1.22 \times 10^{-16}$) என்று மிக மிகச் சிறிய எண்ணாக வருவதைக் காணலாம். இது NumPy-யின் பிழை அல்ல. கணினிகள் மிதக்கும் புள்ளி எண்களை (`floating-point numbers`) கையாளும்போது ஏற்படும் மிகச்சிறிய துல்லியப் பிழை (precision error) ஆகும். நடைமுறையில், இந்த எண்ணை நாம் பூஜ்யமாகவே கருத வேண்டும்.

### ௰௨.௨. எண்களை முழுமையாக்குதல் (Rounding Functions)

ஒரு தச்சர், மரத்துண்டின் முனைகளை மென்மையாக்க அவர் ஒரு உளியைப் பயன்படுத்துகிறார். போல, இந்தக் கருவிகள் தசம எண்களை தேவைக்கேற்ப முழுமையாக்கி, தரவைச் சுத்தம் செய்ய உதவுகின்றன.

Python

```
float_array = np.array([2.1, 3.8, 5.5, -1.2])
```

- `np.floor()` **– கீழே தள்ளு**: ஒரு எண்ணுக்குக் கீழே உள்ள மிக நெருக்கமான முழு எண்ணுக்கு மாற்றும்.

  Python

  ```
  print("Floor:", np.floor(float_array)) # வெளியீடு: [ 2.  3.  5. -2.]
  ```

- `np.ceil()` **– மேலே தள்ளு**: ஒரு எண்ணுக்கு மேலே உள்ள மிக நெருக்கமான முழு எண்ணுக்கு மாற்றும்.

  Python

  ```
  print("Ceil:", np.ceil(float_array))  # வெளியீடு: [ 3.  4.  6. -1.]
  ```

- `np.round()` **– அருகாமைக்கு முழுமையாக்கு**: மிக அருகாமையில் உள்ள முழு எண்ணுக்கு மாற்றும். `.5` என்று முடிந்தால், அருகாமையில் உள்ள இரட்டைப்படை எண்ணுக்கு மாற்றும்.

  Python

  ```
  print("Round:", np.round(float_array)) # வெளியீடு: [ 2.  4.  6. -1.]
  ```

### ௰௨.௩. அடுக்கு மற்றும் மடக்கைச் செயல்பாடுகள் (Exponential and Logarithmic Functions)**

இந்தக் கருவிகள், அதிவேக வளர்ச்சியையும், மிகப் பெரிய எண் வரம்புகளைச் சுருக்கிப் பார்ப்பதற்கும் உதவுகின்றன.

`numpy.exp()` – இயற்கையான வளர்ச்சி

`e` $(\approx 2.718)$ என்ற எண்ணின் அடிப்படையில் ஏற்படும் அதிவேக வளர்ச்சியை இது கணக்கிடுகிறது. இது கூட்டு வட்டி, மக்கள் தொகை வளர்ச்சி போன்ற கணக்கீடுகளில் பயன்படுகிறது.

Python

```
growth_factors = np.array([1, 2, 5])
exp_growth = np.exp(growth_factors)
print("அதிவேக வளர்ச்சி:", exp_growth)
# வெளியீடு: [  2.71828183   7.3890561  148.4131591 ]
```

`numpy.log()` – பிரம்மாண்டத்தைச் சுருக்கும் கருவி

மில்லியன், பில்லியன் என மிகப் பெரிய அளவில் வேறுபடும் எண்களை, ஒரே வரைபடத்தில் (`graph`) ஒப்பிடுவது கடினம். **மடக்கை (Logarithm)**, இந்த பிரம்மாண்டமான எண் வரம்புகளைச் சுருக்கி, எளிதாக ஒப்பிடக்கூடியதாக மாற்றுகிறது.

**எடுத்துக்காட்டு:**

- ஒரு எறும்பின் எடை: 10 மில்லிகிராம்
- ஒரு மனிதனின் எடை: 70,000,000 மில்லிகிராம் (70 கிலோ)
- ஒரு யானையின் எடை: 5,000,000,000 மில்லிகிராம் (5000 கிலோ)

இந்த எண்களை `log10` எடுத்தால்:

Python

```
weights = np.array([10, 70000000, 5000000000])
log_weights = np.log10(weights)
print("சுருக்கப்பட்ட எடை அளவுகள்:", log_weights)
```

**வெளியீடு:**

```
சுருக்கப்பட்ட எடை அளவுகள்: [1.         7.845098  9.69897   ]
```

`1, 7.8, 9.7` என்ற இந்த எண்களை ஒரு வரைபடத்தில் ஒப்பிடுவது மிகவும் எளிது. இதுவே தரவுப் பகுப்பாய்வில் மடக்கையின் சக்தி.

## ௰௩. NumPy – எண்கணிதச் செயல்பாடுகள் (Arithmetic Operations)

NumPy-யின் உண்மையான மந்திரம், அது கணிதக் குறியீடுகளை (`+`, `-`, `*`, `/`) கையாளும் விதத்தில்தான் உள்ளது. நீங்கள் இரண்டு NumPy அணிகளுக்கு இடையில் ஒரு கணிதக் குறியீட்டைப் பயன்படுத்தும்போது, NumPy ஒரு `for loop`-ஐப் பயன்படுத்துவதில்லை. மாறாக, அது ஒரு இணைச் செயல்பாட்டை (parallel operation) நடத்துகிறது.

இதை, இரண்டு அணிகளையும் அருகருகே வைத்து, அவற்றின் ஒத்த உறுப்புகளை இணைத்துக் கோடுகள் வரைவது போலக் கற்பனை செய்துகொள்ளுங்கள். முதல் உறுப்பு முதல் உறுப்புடனும், இரண்டாம் உறுப்பு இரண்டாம் உறுப்புடனும்—இப்படி ஒவ்வொரு ஜோடி உறுப்புகளின் மீதும் ஒரே நேரத்தில் செயல்பாடு நடக்கிறது. இந்த உறுப்பு வாரியான (element-wise) அணுகுமுறை, பல்லாயிரக்கணக்கான கணக்கீடுகளை நொடிப்பொழுதில் செய்ய உதவுகிறது.

### ௰௩.௧. அடிப்படைச் செயல்பாடுகள் (Basic Operations)

இவை நமது சூப்பர் கால்குலேட்டரின் முக்கிய பொத்தான்கள். `np.add(a, b)` போன்ற செயல்பாடுகளைப் பயன்படுத்தலாம் அல்லது `a + b` போன்ற குறியீடுகளை நேரடியாகப் பயன்படுத்தலாம். குறியீடுகளைப் பயன்படுத்துவதே எளிதானது.

**எடுத்துக்காட்டு:** ஒரு கடையில் விற்கப்பட்ட பொருட்களின் எண்ணிக்கையையும், వాటీ விலைகளையும் கொண்டு மொத்த வருவாயைக் கணக்கிடுவோம்.

Python

```
import numpy as np

# விற்கப்பட்ட பொருட்களின் எண்ணிக்கை
quantities = np.array([10, 25, 8])
# ஒரு பொருளின் விலை
prices = np.array([5.0, 2.0, 12.5])

# உறுப்பு வாரியாகக் கணக்கீடுகள்
total_revenue = quantities * prices # np.multiply(quantities, prices)
print(f"ஒவ்வொரு பொருளின் வருவாய்: {total_revenue}")

total_items = np.sum(quantities) # எல்லா உறுப்புகளையும் கூட்ட np.sum()
print(f"மொத்த விற்கப்பட்ட பொருட்கள்: {total_items}")

# கழித்தல் மற்றும் வகுத்தல்
```

```
a = np.array([20, 30, 40])
b = np.array([10, 5, 8])
print(f"கழித்தல் (a - b): {a - b}") # [10 25 32]
print(f"வகுத்தல் (a / b): {a / b}") # [2. 6. 5.]
```

**வெளியீடு:**

```
ஒவ்வொரு பொருளின் வருவாய்: [ 50.  50. 100.]
மொத்த விற்கப்பட்ட பொருட்கள்: 43
கழித்தல் (a - b): [10 25 32]
வகுத்தல் (a / b): [2. 6. 5.]
```

### ௰௩.௨. அடுக்கு மற்றும் மீதி (Power and Modulo)

இவை நமது கால்குலேட்டரின் சிறப்புப் பொத்தான்கள்.

`numpy.power()` – அடுக்கு காணுதல்

ஒரு அணியில் உள்ள ஒவ்வொரு எண்ணிற்கும், மற்றொரு அணியில் உள்ள எண்ணை அடுக்காக (`power`) வைத்து மதிப்பு காணும்.

Python

```
base = np.array([2, 3, 5])
exponent = np.array([3, 2, 2]) # 2^3, 3^2, 5^2

power_values = np.power(base, exponent)
print(f"அடுக்கு மதிப்புகள்: {power_values}") # வெளியீடு: [ 8  9 25]
```

`numpy.mod()` அல்லது `%` – மீதி காணுதல்

ஒரு எண்ணை மற்றொரு எண்ணால் வகுத்து, மீதியை மட்டும் கண்டறியும். இது எண்கள் இரட்டைப்படையா, ஒற்றைப்படையா எனக் கண்டறிவது போன்ற செயல்களுக்குப் பயன்படும்.

Python

```
numbers = np.array([10, 21, 30, 45])
remainders = np.mod(numbers, 2) # numbers % 2
print(f"2-ஆல் வகுத்த மீதி: {remainders}") # வெளியீடு: [0 1 0 1]
```

`0` என்று வரும் எண்கள் இரட்டைப்படை, `1` என்று வரும் எண்கள் ஒற்றைப்படை.

### ௰௩.௩. தலைகீழ் மதிப்பு (Reciprocal)

`numpy.reciprocal()` – $\frac{1}{x}$ கணக்கிடுதல்

இது ஒரு அணியில் உள்ள ஒவ்வொரு உறுப்பு `x`-க்கும், $\frac{1}{x}$ என்ற தலைகீழ் மதிப்பைக் கணக்கிடும்.

**ஒரு முக்கியமான எச்சரிக்கை:** இந்தச் செயல்பாட்டை முழு எண் (`integer`) அணிகளில் பயன்படுத்தும்போது கவனமாக இருக்க வேண்டும். கணினியைப் பொறுத்தவரை, ஒரு முழு எண்ணை மற்றொரு முழு எண்ணால் வகுத்தால், விடை ஒரு முழு எண்ணாகவே இருக்க வேண்டும்; தசமப் பகுதி நீக்கப்படும்.

**சரியான முறை (மிதக்கும் புள்ளி எண்களுடன்):**

Python

```
float_array = np.array([1.0, 2.0, 4.0, 5.0])
reciprocal_values = np.reciprocal(float_array)
print(f"சரியான தலைகீழ் மதிப்புகள்: {reciprocal_values}")
```

**வெளியீடு:**

```
சரியான தலைகீழ் மதிப்புகள்: [1.   0.5  0.25 0.2 ]
```

**தவறான புரிதலுக்கு வழிவகுக்கும் முறை (முழு எண்களுடன்):**

Python

```
int_array = np.array([1, 2, 4, 5])
reciprocal_values_int = np.reciprocal(int_array)
print(f"முழு எண் தலைகீழ் மதிப்புகள்: {reciprocal_values_int}")
```

**வெளியீடு:**

```
முழு எண் தலைகீழ் மதிப்புகள்: [1 0 0 0]
```

இங்கு, 1/2=0.5, ஆனால் முழு எண்ணாக மாற்றும்போது அது `0` ஆகிவிடும். எனவே, துல்லியமான தலைகீழ் மதிப்புகள் தேவையெனில், எப்போதும் மிதக்கும் புள்ளி (`float`) அணிகளையே பயன்படுத்துங்கள்.

## ௰௪. NumPy – புள்ளியியல் செயல்பாடுகள் (Statistical Functions)

தரவு என்பது வெறும் எண்களின் தொகுப்பு அல்ல; அது ஒரு **கதை சொல்லும் புதையல்**. அந்தப் புதையலைத் திறந்து, அதன் ரகசியங்களைப் புரிந்துகொள்ள உதவும் வரைபடங்களும், திறவுகோல்களும்தான் புள்ளியியல் செயல்பாடுகள்.

**நமது எடுத்துக்காட்டு:** ஐந்து மாணவர்களின் மூன்று பாடங்களின் மதிப்பெண்களைக் கொண்ட ஒரு தரவுத் தொகுதியை (`array`) வைத்துக்கொள்வோம். ஒவ்வொரு வரிசையும் ஒரு மாணவரையும், ஒவ்வொரு நிரலும் ஒரு பாடத்தையும் (தமிழ், ஆங்கிலம், கணிதம்) குறிக்கிறது.

Python

```
import numpy as np

# மதிப்பெண் பட்டியல்: மாணவர்கள் x பாடங்கள்
scores = np.array([[85, 90, 92],  # மாணவர் 1
                   [72, 65, 70],  # மாணவர் 2
                   [95, 98, 99],  # மாணவர் 3
                   [68, 75, 72],  # மாணவர் 4
                   [88, 82, 91]]) # மாணவர் 5
```

இந்த மதிப்பெண்களின் கதையை NumPy-யின் கருவிகளைக் கொண்டு அறிந்துகொள்வோம்.

### ௰௪.௧. தரவின் எல்லைகள்: பெரிது & சிறிது (Measures of Range)

இந்தக் கருவிகள், நமது தரவின் உச்சத்தையும் ஆழத்தையும் கண்டறிய உதவுகின்றன. வகுப்பில் முதல் மதிப்பெண் என்ன? மிகக் குறைந்த மதிப்பெண் என்ன?

- **`np.amax()`**: அதிகபட்ச மதிப்பைக் கண்டறிய.
- **`np.amin()`**: குறைந்தபட்ச மதிப்பைக் கண்டறிய.
- **`np.ptp()`**: (Peak-to-Peak) அதிகபட்ச மற்றும் குறைந்தபட்ச மதிப்பிற்கு உள்ள வித்தியாசத்தைக் (range) கண்டறிய.

**`axis` என்ற மந்திரம்:** இந்தக் கருவிகளின் உண்மையான சக்தி, `axis` என்ற அம்சத்தில்தான் உள்ளது.

- **`axis=0`**: நிரல் வாரியாக (column-wise) செயல்படும் (ஒவ்வொரு பாடத்தின் புள்ளிவிவரம்).
- **`axis=1`**: வரிசை வாரியாக (row-wise) செயல்படும் (ஒவ்வொரு மாணவனின் புள்ளிவிவரம்).

Python

```python
# மொத்தத் தேர்வில் அதிகபட்ச மதிப்பெண்
print(f"அனைத்துப் பாடங்களிலும் உச்ச மதிப்பெண்: {np.amax(scores)}")

# ஒவ்வொரு பாடத்திலும் உச்ச மதிப்பெண் (நிரல் வாரியாக)
print(f"பாடம் வாரியான உச்ச மதிப்பெண்: {np.amax(scores, axis=0)}")

# ஒவ்வொரு மாணவனின் குறைந்தபட்ச மதிப்பெண் (வரிசை வாரியாக)
print(f"மாணவர் வாரியான குறைந்தபட்ச மதிப்பெண்: {np.amin(scores, axis=1)}")

# கணிதப் பாடத்தின் மதிப்பெண் வீச்சு (Range)
math_scores = scores[:, 2] # கணித நிரலை மட்டும் எடுத்தல்
print(f"கணித மதிப்பெண் வீச்சு (ptp): {np.ptp(math_scores)}")
```

**வெளியீடு:**

```
அனைத்துப் பாடங்களிலும் உச்ச மதிப்பெண்: 99
பாடம் வாரியான உச்ச மதிப்பெண்: [95 98 99]
மாணவர் வாரியான குறைந்தபட்ச மதிப்பெண்: [85 65 95 68 82]
கணித மதிப்பெண் வீச்சு (ptp): 29
```

### யச.உ. மையப்புள்ளி எது? (Measures of Central Tendency)

வகுப்பின் “வழக்கமான” அல்லது “சராசரி” செயல்திறன் எப்படி இருக்கிறது என்பதை இந்தக் கருவிகள் கூறுகின்றன.

- **`np.mean()`**: சராசரி (average) மதிப்பைக் கணக்கிடுகிறது.
- **`np.median()`**: இடைநிலை (median) மதிப்பைக் கண்டறிகிறது. மதிப்பெண்களை வரிசைப்படுத்தும்போது, சரியாக நடுவில் வரும் மதிப்பு.

**`mean` vs `median`:** ஒரு மாணவர் மட்டும் தேர்வில் 0 மதிப்பெண் வாங்கினால், வகுப்பின் `mean` (சராசரி) வலுவாகக் குறையும். ஆனால் `median` (இடைநிலை) பெரிய அளவில் மாறாது. எனவே, `outliers` (விளிம்புநிலை மதிப்பெண்கள்) இருக்கும்போது `median` ஒரு சிறந்த அளவீடு ஆகும்.

Python

```
# ஒவ்வொரு பாடத்தின் சராசரி மதிப்பெண் (நிரல் வாரியாக)
print(f"பாடம் வாரியான சராசரி: {np.mean(scores, axis=0)}")

# ஒவ்வொரு மாணவனின் இடைநிலை மதிப்பெண் (வரிசை வாரியாக)
print(f"மாணவர் வாரியான இடைநிலை: {np.median(scores, axis=1)}")
```

**வெளியீடு:**

```
பாடம் வாரியான சராசரி: [81.6 82.  84.8]
மாணவர் வாரியான இடைநிலை: [90. 70. 98. 72. 88.]
```

- `np.average()` **(Weighted Average):** சில பாடங்களுக்கு அதிக முக்கியத்துவம் (`weights`) கொடுத்து சராசரியைக் கணக்கிட இது உதவுகிறது.

### ௰௪.௩. தரவு எவ்வளவு பரவியுள்ளது? (Measures of Dispersion)

எல்லா மாணவர்களும் ஏறக்குறைய ஒரே மாதிரி மதிப்பெண் வாங்கியுள்ளார்களா, அல்லது முதல் மற்றும் கடைசி மாணவருக்கு இடையே மிகப் பெரிய இடைவெளி உள்ளதா?

- `np.std()` **(Standard Deviation):** மதிப்பெண்கள் சராசரியிலிருந்து எவ்வளவு தூரம் விலகி உள்ளன என்பதைக் காட்டுகிறது.
    - **குறைந்த** `std`: மாணவர்கள் அனைவரும் சராசரியைச் சுற்றி, நெருக்கமாக மதிப்பெண் பெற்றுள்ளனர் (consistent performance).
    - **அதிக** `std`: மதிப்பெண்கள் பரவலாக உள்ளன; சிலர் மிக அதிகம், சிலர் மிகக் குறைவு (inconsistent performance).

Python

```
# ஒவ்வொரு பாடத்தின் பரவல் (standard deviation)
print(f"பாடம் வாரியான பரவல் (std): {np.std(scores, axis=0)}")
```

**வெளியீடு:**

```
பாடம் வாரியான பரவல் (std): [10.05186544  11.528660   10.6845683]
```

ஆங்கிலப் பாடத்தின் (`11.52`) பரவல் மற்றவற்றை விடச் சற்று அதிகமாக உள்ளது.

### ௰௪.௪. தரவின் படிநிலைகள் (Measures of Position)

- `np.percentile()`: வகுப்பில் முதல் 10% மாணவர்கள் எவ்வளவு மதிப்பெண் பெற்றுள்ளனர்? கடைசி 25% மாணவர்களின் மதிப்பெண் என்ன? போன்ற கேள்விகளுக்கு இது பதிலளிக்கிறது.

`50`-வது பெர்சன்டைல் (`percentile`) என்பது `median`-ஐக் குறிக்கும்.

Python

```
# கணிதத்தில் 75-வது பெர்சன்டைல் மதிப்பெண் என்ன?
# (அதாவது, 75% மாணவர்கள் இந்த மதிப்பெண்ணுக்குக் கீழ்தான் உள்ளனர்)
print(f"கணிதத்தில் 75வது பெர்சன்டைல்: {np.percentile(scores[:, 2], 75)}")
```

**வெளியீடு:**

```
கணிதத்தில் 75வது பெர்சன்டைல்: 92.0
```

இந்தப் புள்ளியியல் கருவிகளைக் கொண்டு, வெறும் எண்களாக இருந்த `scores` என்ற அணியிலிருந்து, மாணவர்களின் செயல்திறன், பாடங்களின் கடினத்தன்மை போன்ற பல கதைகளை நம்மால் கண்டறிய முடிகிறது.

## ௰௫. NumPy – வரிசைப்படுத்துதல், தேடுதல் மற்றும் கண்டறிதல்

ஒரு NumPy அணியை, கலைந்து கிடக்கும் புத்தகங்கள் நிறைந்த ஒரு **நூலக அலமாரி** என்று கற்பனை செய்துகொள்ளுங்கள். ஒரு திறமையான நூலகராக, நமது முதல் வேலை, அந்தப் புத்தகங்களை நேர்த்தியாக **வரிசைப்படுத்துவது**, பின்னர் பார்வையாளர்கள் கேட்கும் புத்தகங்களைத் துல்லியமாகத் **தேடிக் கொடுப்பது**, மற்றும் குறிப்பிட்ட புத்தகங்களைக் **கண்டறிவது**.

NumPy, இந்த நூலகப் பணிகளைச் செய்ய நமக்குச் சக்திவாய்ந்த கருவிகளைத் தருகிறது.

### ௰௫.௧. நூல்களை வரிசைப்படுத்துதல் (Sorting the Books)

`numpy.sort()` – எளிய வரிசைப்படுத்துதல்

இது நூலகரின் முதல் மற்றும் அடிப்படையான வேலை. புத்தகங்களை அதன் பக்கங்களின் எண்ணிக்கையைப் பொறுத்து, சிறியதிலிருந்து பெரியதாக வரிசைப்படுத்துவது போல.

Python

```
import numpy as np

# புத்தகப் பக்கங்களின் எண்ணிக்கை
pages = np.array([350, 120, 480, 210, 300])
sorted_pages = np.sort(pages)

print(f"வரிசைப்படுத்தப்பட்ட பக்கங்கள்: {sorted_pages}")
```

**வெளியீடு:**

```
வரிசைப்படுத்தப்பட்ட பக்கங்கள்: [120 210 300 350 480]
```

`numpy.argsort()` – நூலகரின் அட்டவணை அட்டை

`sort` புத்தகங்களை வரிசைப்படுத்திவிடும். ஆனால், ஒரு புத்தகத்தின் விலையை மட்டும் வரிசைப்படுத்தினால், அது எந்தப் புத்தகம் என்ற தொடர்பு அறுந்துவிடும் அல்லவா?

`argsort`, புத்தகங்களை நேரடியாக வரிசைப்படுத்தாது. மாறாக, அவற்றை எப்படி வரிசைப்படுத்த வேண்டும் என்ற **இடங்களின் பட்டியலை (list of indices)**, ஒரு அட்டவணை அட்டை போல நமக்குத் தரும். இந்த அட்டையைப் பயன்படுத்தி, நாம் தொடர்புடைய எல்லாத் தரவுகளையும் ஒருசேர வரிசைப்படுத்தலாம்.

**எடுத்துக்காட்டு:**

Python

```
titles = np.array(['புலிதேசம்', 'கடல் நாகம்', 'யவன ராணி'])
prices = np.array([450, 520, 380])

# விலையின் அடிப்படையில் வரிசைப்படுத்துவதற்கான அட்டவணையை உருவாக்குவோம்
price_order_card = np.argsort(prices)
print(f"விலை வாரியான வரிசை அட்டை: {price_order_card}")

# இப்போது, இந்த அட்டையைப் பயன்படுத்தி தலைப்புகளையும் விலைகளையும் வரிசைப்படுத்துவோம்
print(f"வரிசைப்படுத்தப்பட்ட தலைப்புகள்: {titles[price_order_card]}")
print(f"வரிசைப்படுத்தப்பட்ட விலைகள்: {prices[price_order_card]}")
```

**வெளியீடு:**

```
விலை வாரியான வரிசை அட்டை: [2 0 1]
வரிசைப்படுத்தப்பட்ட தலைப்புகள்: ['யவன ராணி' 'புலிதேசம்' 'கடல் நாகம்']
வரிசைப்படுத்தப்பட்ட விலைகள்: [380 450 520]
```

`numpy.lexsort( )` – பல அடுக்கு வரிசைப்படுத்துதல்

ஒரு நூலகர், புத்தகங்களை எழுத்தாளர் பெயரின்படி (A-Z) வரிசைப்படுத்திய பின், ஒரே எழுத்தாளரின் புத்தகங்களை மட்டும், அவை வெளியான ஆண்டின்படி வரிசைப்படுத்துவார் அல்லவா? இப்படி, **பல அடுக்குகளில் அல்லது பல நிபந்தனைகளின் அடிப்படையில்** ஒன்றை ஒன்று சார்ந்து வரிசைப்படுத்தவே `lexsort` உதவுகிறது.

இது, நேரடியாக மதிப்புகளை வரிசைப்படுத்தாது. மாறாக, `argsort` போலவே, சரியான வரிசைக்கான **இடங்களின் பட்டியலை (list of indices)** ஒரு அட்டவணை அட்டையாக நமக்குத் தரும்.

மிக முக்கியமான விதி

`lexsort`-ஐப் பயன்படுத்தும்போது, நீங்கள் கொடுக்கும் நிபந்தனைகளின் வரிசை மிகவும் முக்கியம். நீங்கள் கொடுக்கும் பட்டியலில் (`tuple`), **கடைசியாக இருக்கும் நிபந்தனைதான் முதல் நிலையில் (primary key) எடுத்துக்கொள்ளப்படும்.**

`np.lexsort((key_C, key_B, key_A))` என்று கொடுத்தால், NumPy முதலில் `key_A`-இன் படி வரிசைப்படுத்தும். பின்னர், `key_A`-இல் ஒரே மதிப்புள்ள உறுப்புகளை, `key_B`-இன் படி வரிசைப்படுத்தும். அதிலும் ஒரே மதிப்பு இருந்தால், `key_C`-ஐப் பயன்படுத்தும்.

மாணவர்களின் பெயர்களையும், அவர்கள் படிக்கும் வகுப்புகளையும் கொண்ட இரண்டு பட்டியல்கள் உள்ளன. நமது நோக்கம்: **முதலில் வகுப்பின்படியும் (Class A, Class B), பின்னர் ஒவ்வொரு வகுப்பிற்குள்ளும் மாணவர்களின் பெயர்களை அகர வரிசைப்படியும்** வரிசைப்படுத்த வேண்டும்.

Python

```
import numpy as np

# மாணவர்களின் பெயர்கள்
student_names = np.array(['கீதா', 'அன்பு', 'பாலா', 'திவ்யா'])
# அவர்கள் படிக்கும் வகுப்பு
```

```
student_classes = np.array(['வகுப்பு B', 'வகுப்பு A', 'வகுப்பு A', 'வகுப்பு B'])

# நமது முதல் நிபந்தனை: வகுப்பு (Class)
# நமது இரண்டாம் நிபந்தனை: பெயர் (Name)

# விதிப்படி, முதல் நிபந்தனையை (வகுப்பு) கடைசியாகக் கொடுக்க வேண்டும்.
sorted_indices = np.lexsort((student_names, student_classes))

print(f"பல அடுக்கு வரிசைக்கான அட்டவணை அட்டை: {sorted_indices}")

# இப்போது இந்த அட்டையைப் பயன்படுத்தி மாணவர்களை வரிசைப்படுத்துவோம்
print("\nவரிசைப்படுத்தப்பட்ட பட்டியல்:")
print(f"வகுப்பு: {student_classes[sorted_indices]}")
print(f"பெயர் : {student_names[sorted_indices]}")
```

**வெளியீடு:**

```
பல அடுக்கு வரிசைக்கான அட்டவணை அட்டை: [1 2 0 3]

வரிசைப்படுத்தப்பட்ட பட்டியல்:
வகுப்பு: ['வகுப்பு A' 'வகுப்பு A' 'வகுப்பு B' 'வகுப்பு B']
பெயர் : ['அன்பு' 'பாலா' 'கீதா' 'திவ்யா']
```

### ௰௫.௨. குறிப்பிட்ட நூல்களைத் தேடுதல் மற்றும் கண்டறிதல்

`numpy.argmax()` மற்றும் `numpy.argmin()` – உச்சத்தில் உள்ளதைக் கண்டறிதல்

அலமாரியில், **அதிக பக்கங்கள் கொண்ட புத்தகம் எங்கே உள்ளது? மிகக் குறைந்த பக்கங்கள் கொண்ட புத்தகம் எங்கே உள்ளது?** இந்தக் கேள்விகளுக்கு, இந்தச் செயல்பாடுகள் அந்தப் புத்தகத்தின் **இடத்தை (index)** நமக்குத் தெரிவிக்கும்.

Python

```
pages = np.array([350, 120, 480, 210, 300])

print(f"அதிக பக்கங்கள் கொண்ட புத்தகத்தின் இடம்: {np.argmax(pages)}")
print(f"குறைந்த பக்கங்கள் கொண்ட புத்தகத்தின் இடம்: {np.argmin(pages)}")
```

**வெளியீடு:**

```
அதிக பக்கங்கள் கொண்ட புத்தகத்தின் இடம்: 2
குறைந்த பக்கங்கள் கொண்ட புத்தகத்தின் இடம்: 1
```

`numpy.where()` – நிபந்தனையின் அடிப்படையில் தேடுதல்

இது நூலகரின் மேம்பட்ட தேடல் கருவி. “400 பக்கங்களுக்கு மேல் உள்ள புத்தகங்கள் எந்தெந்த இடங்களில் உள்ளன?” என்பது போன்ற நிபந்தனைகளின் அடிப்படையில் தேட இது உதவுகிறது.

Python

```
# 400 பக்கங்களுக்கு மேல் உள்ள புத்தகங்களின் இடங்களைத் தேடுவோம்
long_books_indices = np.where(pages > 400)
print(f"400 பக்கங்களுக்கு மேல் உள்ள புத்தகங்களின் இடங்கள்: {long_books_indices[0]}")
```

**வெளியீடு:**

```
400 பக்கங்களுக்கு மேல் உள்ள புத்தகங்களின் இடங்கள்: [2]
```

`numpy.extract()` – நிபந்தனையின் அடிப்படையில் பிரித்தெடுத்தல்

`where` நிபந்தனைக்குப் பொருந்தும் புத்தகங்களின் *இடங்களைச்* சொன்னது. `extract` அந்த நிபந்தனைக்குப் பொருந்தும் *புத்தகங்களையே* நமக்குத் தரும்.

**குறிப்பு:** `extract` செய்வது, `array[condition]` என்ற boolean indexing முறையைப் போன்றதே. இவற்றில், இரண்டாவது முறையே மிகவும் எளிமையானதும், பரவலாகப் பயன்படுத்தப்படுவதும் ஆகும்.

Python

```
# 400 பக்கங்களுக்கு மேல் உள்ள புத்தகங்களைப் பிரித்தெடுப்போம்
long_books = np.extract(pages > 400, pages)
print(f"400 பக்கங்களுக்கு மேல் உள்ள புத்தகங்கள்: {long_books}")

# எளிமையான முறை:
print(f"எளிமையான முறையில்: {pages[pages > 400]}")
```

**வெளியீடு:**

```
400 பக்கங்களுக்கு மேல் உள்ள புத்தகங்கள்: [480]
எளிமையான முறையில்: [480]
```

`numpy.nonzero()` – காலியாக இல்லாத இடங்களைக் கண்டறிதல்

நூலக அலமாரியில், புத்தகங்கள் இல்லாத இடங்கள் `0` என்றும், புத்தகங்கள் உள்ள இடங்கள் வேறு எண்களாலும் குறிக்கப்பட்டிருந்தால், புத்தகங்கள் உள்ள எல்லா இடங்களையும் கண்டறிய `nonzero` உதவுகிறது.

Python

```
shelf = np.array([1, 0, 3, 0, 5]) # 0 என்பது காலி இடம்
occupied_indices = np.nonzero(shelf)
print(f"புத்தகங்கள் உள்ள இடங்கள்: {occupied_indices[0]}")
```

**வெளியீடு:**

```
புத்தகங்கள் உள்ள இடங்கள்: [0 2 4]
```

இந்தக் கருவிகளைக் கொண்டு, கலைந்து கிடக்கும் தரவுகளை ஒரு நேர்த்தியான நூலகம் போல ஒழுங்குபடுத்தி, நமக்குத் தேவையான தகவல்களை நொடிப்பொழுதில் கண்டறிய முடியும்.

## ௰௬. NumPy – பைட்களை இடம் மாற்றுதல் (Byte Swapping)

கணினிகளுக்குள் தரவுகள் எப்படிச் சேமிக்கப்படுகின்றன என்பதில் ஒரு சிறிய கலாச்சார வேறுபாடு உள்ளது. இதை, நாம் **முகவரி எழுதும் முறையில்** உள்ள வேறுபாட்டை வைத்துப் புரிந்துகொள்ளலாம்.

**கற்பனைச் சூழல்:** உலகில் இரண்டு நாடுகள் உள்ளன.

- **நாடு A (Little-endian):** இவர்கள் முகவரியை, மிகச் சிறிய அலகிலிருந்து பெரிய அலகிற்கு எழுதுவார்கள். (உதாரணம்: `வீட்டு எண், தெரு, நகரம், நாடு`). இன்று நாம் பயன்படுத்தும் பெரும்பாலான கணினிகள் (Intel, AMD) இந்த முறையைப் பின்பற்றுகின்றன.
- **நாடு B (Big-endian):** இவர்கள் முகவரியை, மிகப் பெரிய அலகிலிருந்து சிறிய அலகிற்கு எழுதுவார்கள். (உதாரணம்: `நாடு, நகரம், தெரு, வீட்டு எண்`). இணையத்தில் передача данных (network protocols) பெரும்பாலும் இந்த முறையில் நடக்கும்.

இப்போது, நாடு A-விலிருந்து நாடு B-க்கு ஒரு கடிதம் அனுப்பினால் என்ன ஆகும்? நாடு B-யின் தபால்காரர், வீட்டு எண்ணை நாடு என்று தவறாகப் புரிந்துகொண்டு குழம்பிவிடுவார். இந்தக் குழப்பத்தைத் தீர்க்கும் **முகவரி மாற்றி எழுதும் கருவிதான்** NumPy-யின் **`byteswap()`**.

#### ௰௬.௧. எண்டியன் (Endian) என்றால் என்ன?

ஒரு எண், கணினியின் நினைவகத்தில் (`memory`) பல பைட்களாகச் (`bytes`) சேமிக்கப்படும்போது, அந்த பைட்களை எந்த வரிசையில் எழுதுவது என்பதே **Endianness** ஆகும்.

- **Little-endian (சிறியதிலிருந்து பெரியது):** ஒரு எண்ணின் முக்கியத்துவம் குறைந்த பைட் (least significant byte - *வீட்டு எண்*) முதலில் சேமிக்கப்படும்.
- **Big-endian (பெரியதிலிருந்து சிறியது):** ஒரு எண்ணின் முக்கியத்துவம் வாய்ந்த பைட் (most significant byte - *நாடு*) முதலில் சேமிக்கப்படும்.

#### ௰௬.௨. `byteswap()` – முகவரியை மாற்றி எழுதுதல்

`byteswap()` செயல்பாடு, ஒரு எண்களின் பைட்களை அப்படியே தலைகீழாக மாற்றும். Little-endian முகவரியை Big-endian ஆகவும், Big-endian-ஐ Little-endian ஆகவும் மாற்றும்.

**எடுத்துக்காட்டு:** `int16` என்பது, ஒவ்வொரு எண்ணையும் 2 பைட்கள் (16 பிட்கள்) கொண்ட ஒரு “முகவரியாக” சேமிக்கும்.

Python

```python
import numpy as np

# நமது எண்கள் (Intel கணினியில் Little-endian ஆக சேமிக்கப்படும்)
original_array = np.array([1, 256, 8755], dtype=np.int16)

print(f"அசல் எண்கள்: {original_array}")
print(f"அசல் எண்களின் நினைவக பைட் வரிசை (உதாரணம்): [01, 00], [00, 01], [33, 22]")

# பைட்களை இடம் மாற்றுவோம்
# கவனம்: byteswap() அசல் அணியை நேரடியாக மாற்றும் (in-place operation).
original_array.byteswap(inplace=True)

print(f"\nஇடம் மாற்றப்பட்ட எண்கள்: {original_array}")
print(f"புதிய பைட் வரிசை (உதாரணம்): [00, 01], [01, 00], [22, 33]")
```

**வெளியீடு:**

```
அசல் எண்கள்: [   1  256 8755]
அசல் எண்களின் நினைவக பைட் வரிசை (உதாரணம்): [01, 00], [00, 01], [33, 22]

இடம் மாற்றப்பட்ட எண்கள்: [ 256    1 13090]
புதிய பைட் வரிசை (உதாரணம்): [00, 01], [01, 00], [22, 33]
```

**விளக்கம்:**

- `1` என்ற எண் (`0x0001`), நினைவகத்தில் `[01, 00]` எனச் சேமிக்கப்பட்டது. இடம் மாற்றிய பின், `[00, 01]` ஆகி, அதன் மதிப்பு `256` (`0x0100`) ஆனது.
- `256` என்ற எண் (`0x0100`), நினைவகத்தில் `[00, 01]` எனச் சேமிக்கப்பட்டது. இடம் மாற்றிய பின், `[01, 00]` ஆகி, அதன் மதிப்பு `1` (`0x0001`) ஆனது.

**மிக முக்கியக் குறிப்பு:** `array.byteswap()` என்ற செயல்பாடு, **அசல் அணியை நேரடியாக மாற்றியமைக்கும் (in-place)**. அசல் அணி மாறாமல், புதிய நகல் வேண்டுமானால், `new_array = array.byteswap(inplace=False)` என்று பயன்படுத்த வேண்டும்.

#### ௰௬.௩. பைட் இடமாற்றம் ஏன், எப்போது அவசியம்?

1. பலதரப்பட்ட கணினி அமைப்புகளுக்கு இடையே தகவல் பரிமாற்றம் (Cross-platform Compatibility):

   ஒரு Little-endian கணினியிலிருந்து (உங்கள் லேப்டாப்), ஒரு Big-endian முறை பயன்படுத்தும் இணைய சேவையகத்திற்குத் தரவை அனுப்பும்போது, தரவு சரியாகப் புரிந்துகொள்ளப்பட byteswap அவசியம்.

2. இருமத் தரவுகளைக் கையாளுதல் (Binary Data Handling):

   கோப்புகளிலிருந்து (files) பைனரி தரவுகளைப் படிக்கும்போதோ அல்லது வலையமைப்பு (network) வழியாகத் தரவுகளைப் பெறும்போது, அவை எந்த Endian வடிவத்தில் அனுப்பப்பட்டன என்பதை அறிந்து, தேவைப்பட்டால் மாற்ற வேண்டும்.

3. தரவின் நம்பகத்தன்மை (Data Integrity):

   தவறான Endian வடிவத்தில் உள்ள தரவை நாம் பயன்படுத்தினால், எண்கள் முற்றிலும் தவறான மதிப்புகளைக் காட்டும் (தபால்காரர் வீட்டு எண்ணை நாடு என்று படிப்பது போல). இது தரவு சிதைவிற்கு (data corruption) வழிவகுக்கும். சரியான இடமாற்றம், தரவின் நம்பகத்தன்மையை உறுதி செய்கிறது.

## ௰௭. NumPy – நகல்கள் மற்றும் பார்வைகள் (Copies & Views)

NumPy-யில் தரவைக் கையாளும் முறைகளைப் புரிந்துகொள்ள, ஒரு **Google Docs ஆவணத்தை** கையாள்வதுடன் ஒப்பிடுவோம். நீங்கள் ஒருவரிடம் ஆவணத்தைப் பகிரும்போது, அதன் இணைப்பை அனுப்புகிறீர்களா, நேரடிப் பார்வையை அனுமதிக்கிறீர்களா, அல்லது ஒரு நகலைப் பதிவிறக்கம் செய்து அனுப்புகிறீர்களா என்பதைப் பொறுத்து விளைவுகள் மாறும். NumPy-யும் அப்படித்தான்.

#### ௰௭.௧. நகல் இல்லை (No Copy) – ஆவணத்தின் நேரடி இணைப்பு (Direct URL)

இது, உங்கள் மூல Google Docs ஆவணத்தின் **நேரடி URL இணைப்பை** உங்கள் நண்பருக்குக் கொடுப்பது போன்றது.

புதிய ஆவணம் எதுவும் உருவாகவில்லை. நீங்களும் உங்கள் நண்பரும் ஒரே ஆவணத்தைத்தான் பார்க்கிறீர்கள், திருத்துகிறீர்கள். உங்கள் நண்பர் அந்த இணைப்பிற்கு வேறு பெயர் (`b = a`) வைத்துக்கொண்டாலும், அது அதே மூல ஆவணத்தையே குறிக்கும். அவர் ஒரு வரியை மாற்றினால் (`b[0] = 99`), உங்கள் ஆவணத்திலும் அந்த வரி உடனடியாக மாறும்.

**சுருக்கமாக:** `b` என்பது `a`-இன் இன்னொரு புனைப்பெயர். இரண்டும் **ஒரே பொருள், ஒரே தரவு**.

**எடுத்துக்காட்டு:**

Python

```python
import numpy as np
a = np.array([10, 20, 30])
b = a  # நேரடி இணைப்பு மட்டும் கொடுக்கப்படுகிறது

b[0] = 99
print(f"நண்பர் மாற்றிய பின், உங்கள் அசல் ஆவணம்: {a}") # அசலும் மாறிவிட்டது!
```

**வெளியீடு:**

```
நண்பர் மாற்றிய பின், உங்கள் அசல் ஆவணம்: [99 20 30]
```

### யஎ.உ. பார்வை (View) – இணைக்கப்பட்ட நேரலைக் காட்சி (Synced Live View)

இது, உங்கள் மூல Google Docs ஆவணத்தை, ஒரு இணையதளத்தில் **நேரலையாக உட்பொதிப்பது (embed)** போன்றது.

இணையதளத்தில் தெரிவது (`view`) வேறு பொருள், மூல ஆவணம் வேறு பொருள். ஆனால், இரண்டின் **உள்ளடக்கமும் நேரடியாக இணைக்கப்பட்டுள்ளது**. நீங்கள் மூல ஆவணத்தில் ஒரு திருத்தம் செய்தால், இணையதளக் காட்சியிலும் அது உடனடியாகத் தெரியும். அதேபோல, இணையதளக் காட்சியில் உள்ளடக்கத்தை மாற்றினால், மூல ஆவணமும் மாறும்.

**சுருக்கமாக:** `c` மற்றும் `a` **வேறு வேறு பொருள்கள், ஆனால் ஒரே தரவைப் பகிர்ந்துகொள்கின்றன**. ஒரு அணியை `slice` செய்வதும் (`a[1:3]`) இந்த வகையைச் சேர்ந்ததே.

**எடுத்துக்காட்டு:**

Python

```python
a = np.array([10, 20, 30])
c = a.view() # நேரலைக் காட்சி உருவாக்கப்படுகிறது

c[1] = 88 # நேரலைக் காட்சியை மாற்றுகிறோம்
print(f"காட்சியை மாற்றிய பின், அசல் ஆவணம்: {a}") # அசலும் மாறிவிட்டது!

a[2] = 77 # அசல் ஆவணத்தை மாற்றுகிறோம்
print(f"அசலை மாற்றிய பின், காட்சி: {c}") # காட்சியும் மாறிவிட்டது!
```

**வெளியீடு:**

```
காட்சியை மாற்றிய பின், அசல் ஆவணம்: [10 88 30]
அசலை மாற்றிய பின், காட்சி: [10 88 77]
```

### ௰௭.௩. முழுமையான நகல் (Deep Copy) – பதிவிறக்கம் செய்யப்பட்ட பிரதி

இது, உங்கள் Google Docs ஆவணத்தை **“File -> Make a Copy”** செய்வது அல்லது **PDF ஆகப் பதிவிறக்கம்** செய்வது போன்றது.

உங்கள் நண்பருக்குக் கிடைப்பது மூல ஆவணத்தின் ஒரு **முழுமையான, தனித்தன்மை வாய்ந்த நகல்**. அவர் அந்த நகலில் என்ன மாற்றம் செய்தாலும், அது உங்கள் மூல ஆவணத்தை எந்த விதத்திலும் பாதிக்காது. இரண்டும் முற்றிலும் தனித்தனியானவை.

**சுருக்கமாக:** `d` மற்றும் `a` **வேறு வேறு பொருள்கள், வேறு வேறு தரவுகள்.**

**எடுத்துக்காட்டு:**

Python

```
a = np.array([10, 20, 30])
d = a.copy() # முழுமையான நகல் எடுக்கப்படுகிறது

d[2] = 66 # நகலை மட்டும் மாற்றுகிறோம்
print(f"நகலை மாற்றிய பின், அசல் ஆவணம்: {a}") # அசல் மாறவில்லை!
```

**வெளியீடு:**

```
நகலை மாற்றிய பின், அசல் ஆவணம்: [10 20 30]
```

## ௰௮. NumPy – அணி நூலகம் (Matrix Library)

நேரியல் இயற்கணிதம் (Linear Algebra) மற்றும் அறிவியல் கணக்கீடுகளில் **அணி (Matrix)** என்பது மிக முக்கியமான ஒரு கருத்து. இது ஒரு சிறப்பு வகை இருபரிமாண (`2D`) அமைப்பாகும். NumPy, இந்த அணிகளைக் கையாள்வதற்காகவே **`numpy.matlib`** என்ற ஒரு பிரத்யேக துணை நூலகத்தைக் கொண்டுள்ளது.

இதை, ஒரு **பழைய, பிரத்யேக பட்டறைக்கும் (Old, Specialized Workshop)**, நவீன, பல்திறன் பட்டறைக்கும் உள்ள வேறுபாடாகக் கற்பனை செய்துகொள்ளுங்கள்.

- **`numpy.matlib` (பழைய பட்டறை):** இது அணிகளை உருவாக்குவதற்காக மட்டுமே வடிவமைக்கப்பட்ட கருவிகளைக் கொண்டது. இதன் கருவிகள் அணிகளை மட்டுமே உருவாக்கும்.
- **`numpy` (நவீன பட்டறை):** இதன் கருவிகள் (`np.array`, `np.zeros`) எந்தப் பரிமாணத்திலும் (1D, 2D, 3D...) தரவுக் கட்டமைப்பை உருவாக்கும். ஒரு நவீன 2D `array`, பழைய `matrix` செய்யக்கூடிய அனைத்து வேலைகளையும், இன்னும் பலவற்றையும் செய்யவல்லது.

**முக்கியக் குறிப்பு:** `matlib` நூலகம், பழைய MATLAB பயனர்களுக்காகவும், முந்தைய குறியீடுகளுடன் இயைபுத்தன்மைக்காகவும் (backward compatibility) இன்னும் உள்ளது. ஆனால், **நவீன NumPy-யில், சாதாரண `np.array`-களையே அணிகளாகப் பயன்படுத்துவதுதான் சிறந்த மற்றும் பரிந்துரைக்கப்பட்ட முறையாகும்.**

இந்த அத்தியாயத்தில், `matlib`-இன் கருவிகளைப் பற்றியும், அவற்றுக்கான நவீன மாற்று வழிகளையும் அறிந்துகொள்வோம்.

### ௰௮.௧. அடிப்படை அணிகளை உருவாக்குதல்

`matlib.empty()`, `matlib.zeros()`, `matlib.ones()`

இந்தக் கருவிகள், முறையே மதிப்புகள் எதுவும் இல்லாத, பூஜ்யங்கள் நிறைந்த, மற்றும் ஒன்றுகள் நிறைந்த அணிகளை உருவாக்குகின்றன.

**எடுத்துக்காட்டு (`zeros`):**

Python

```python
import numpy.matlib as matlib
import numpy as np

# பழைய பட்டறை முறை
zeros_matrix = matlib.zeros((2, 3))
print("matlib மூலம் உருவான பூஜ்ய அணி:\n", zeros_matrix)
print("இதன் வகை:", type(zeros_matrix))

# நவீன பட்டறை முறை
zeros_array = np.zeros((2, 3))
print("\nnp.zeros மூலம் உருவான பூஜ்ய அணி:\n", zeros_array)
print("இதன் வகை:", type(zeros_array))
```

**வெளியீடு:**

```
matlib மூலம் உருவான பூஜ்ய அணி:
 [[0. 0. 0.]
 [0. 0. 0.]]
இதன் வகை: <class 'numpy.matrix'>

np.zeros மூலம் உருவான பூஜ்ய அணி:
 [[0. 0. 0.]
 [0. 0. 0.]]
இதன் வகை: <class 'numpy.ndarray'>
```

இரண்டும் ஒரே வேலையைச் செய்தாலும், நவீன முறை நமக்கு `ndarray`-ஐத் தருகிறது, இது மிகவும் நெகிழ்வுத்தன்மை கொண்டது.

### ௰௮.௨. அடையாள அணிகள் (Identity Matrices)

அணிக் கணிதத்தில், அடையாள அணி என்பது எண் 1 போலச் செயல்படும் ஒரு முக்கியமான அணியாகும். இதன் மூலைவிட்டத்தில் (`diagonal`) மட்டும் 1-களும், மற்ற இடங்களில் 0-களும் இருக்கும்.

`matlib.eye()` மற்றும் `matlib.identity()`

- **`eye()`**: இது ஒரு செவ்வக வடிவ (`rectangular`) அணியைக்கூட உருவாக்கும். `k` என்ற அம்சத்தைப் பயன்படுத்தி, மூலைவிட்டத்தின் இடத்தையும் மாற்றலாம்.
- **`identity()`**: இது எப்பொழுதும் ஒரு சதுர (`square`) அடையாள அணியை மட்டுமே உருவாக்கும்.

**நவீன வழி:** `np.eye()` மற்றும் `np.identity()` ஆகிய செயல்பாடுகள் பிரதான NumPy நூலகத்திலேயே உள்ளன, அவையே பரிந்துரைக்கப்படுகின்றன.

### ௰௮.௩. சீரற்ற எண்களின் அணி (Random Matrix)

`matlib.rand()`

0-விற்கும் 1-க்கும் இடையில் சீரற்ற மிதக்கும் புள்ளி எண்களைக் கொண்ட ஒரு அணியை இது உருவாக்கும்.

**நவீன வழி:** `np.random.rand()` என்ற கருவியைப் பயன்படுத்துவதே சிறந்தது. ஏனென்றால், `np.random` என்ற நவீன பட்டறையில், சீரான எண்கள், இயல்பான எண்கள் (normal distribution) எனப் பலவிதமான சீரற்ற எண்களை உருவாக்கும் கருவிகள் உள்ளன.

**`matlib` வேண்டுமா, வேண்டாமா?**

`numpy.matlib` என்பது NumPy-யின் ஒரு பகுதியாக இருந்தாலும், அது பெரும்பாலும் ஒரு மரபுவழிக் கருவியாகவே (`legacy tool`) கருதப்படுகிறது. நீங்கள் புதிதாக NumPy கற்கிறீர்கள் என்றால், **`matlib`-ஐத் தவிர்த்துவிட்டு, பிரதான NumPy செயல்பாடுகளையே (`np.zeros`, `np.eye`, `np.random.rand` போன்றவை) நேரடியாகப் பயன்படுத்துவதுதான்** சிறந்த, நவீன மற்றும் நெகிழ்வான அணுகுமுறையாகும்.

## ௰௯. NumPy – நேரியல் இயற்கணிதம் (Linear Algebra)

நேரியல் இயற்கணிதம் என்பது கணிதத்தின் ஒரு கிளை என்பதை விட, அது **தரவுகளுக்கு இடையேயான உறவுகளின் மொழி** என்று சொல்லலாம். ஒரு புதிரில் உள்ள தடயங்களை (`clues`) ஒழுங்குபடுத்தி, அவற்றிலிருந்து உண்மையைக் கண்டறியும் ஒரு **தரவுத் துப்பறிவாளரின் (Data Detective)** தர்க்கரீதியான அணுகுமுறைதான் இது.

NumPy, குறிப்பாக அதன் `linalg` என்ற துணை நூலகம், இந்தத் துப்பறியும் பணிக்குத் தேவையான அனைத்து உயர் தொழில்நுட்பக் கருவிகளையும் நமக்கு வழங்குகிறது.

### ௰௯.௧. தடயங்களை இணைத்தல்: டாட் ப்ராடக்ட் (Dot Product)

ஒரு துப்பறிவாளருக்குக் கிடைக்கும் இரண்டு வெவ்வேறு தடயப் பட்டியல்களை (`vectors`) இணைத்துப் புதிய கோணத்தில் சிந்திப்பதுதான் டாட் ப்ராடக்ட். இது நேரியல் இயற்கணிதத்தின் மிக அடிப்படையான செயல்பாடு.

1D திசையன்களுக்கு (`vectors`), `dot()`, `vdot()`, மற்றும் `inner()` ஆகிய மூன்று செயல்பாடுகளும் ஒரே விடையைத் தரும்.

**எடுத்துக்காட்டு:**

Python

```python
import numpy as np

# தடயப் பட்டியல் 1
clues_a = np.array([1, 2])
# தடயப் பட்டியல் 2
clues_b = np.array([3, 4])

# தடயங்களை இணைப்போம்
dot_product = np.dot(clues_a, clues_b)
print(f"டாட் ப்ராடக்ட்: {dot_product}")
```

**விளக்கம்:** இதன் கணக்கீடு `(1 * 3) + (2 * 4) = 3 + 8 = 11`.

### ௰௯.௨. அணிகளின் பெருக்கல்: `matmul` மற்றும் `@`

பல தடயப் பட்டியல்களைக் கொண்ட இரண்டு பெரிய தொகுப்புகளை (`matrices`) இணைத்து, ஒரு புதிய முடிவுக்கு வருவதே அணிப் பெருக்கல் (Matrix Multiplication) ஆகும்.

இதைச் செய்ய, `np.matmul(A, B)` என்ற செயல்பாட்டைப் பயன்படுத்தலாம். ஆனால், நவீன NumPy-யில், இதைவிட எளிமையான ஒரு வழி உள்ளது: `@` **குறியீடு**. இதுவே பரிந்துரைக்கப்பட்ட மற்றும் பரவலாகப் பயன்படுத்தப்படும் முறையாகும்.

**எடுத்துக்காட்டு:**

Python

```
# தடயத் தொகுப்பு 1
matrix_A = np.array([[1, 2], [3, 4]])
# தடயத் தொகுப்பு 2
matrix_B = np.array([[5, 6], [7, 8]])

# அணிப் பெருக்கல் (நவீன முறை)
matrix_product = matrix_A @ matrix_B

print(f"அணிப் பெருக்கல்:\n{matrix_product}")
```

**வெளியீடு:**

```
அணிப் பெருக்கல்:
[[19 22]
 [43 50]]
```

### ௰௯.௩. புதிரின் தன்மை: டிட்டர்மினன்ட் (Determinant)

ஒரு புதிரைத் தீர்க்கும் முன், அந்தப் புதிருக்கு **தனித்துவமான, ஒரே ஒரு விடை உள்ளதா** என்பதைத் தெரிந்துகொள்ள வேண்டும் அல்லவா? அந்தக் கேள்வியைச் சோதிக்கும் கருவிதான் **டிட்டர்மினன்ட் (`determinant`).**

`np.linalg.det()` என்ற கருவி, ஒரு சதுர அணியின் டிட்டர்மினன்ட் மதிப்பைக் கணக்கிடும்.

**விதி:**

- **டிட்டர்மினன்ட் = 0** என்றால், புதிரின் தடயங்கள் ஒன்றுக்கொன்று முரண்பாடானவை அல்லது தேவையற்றவை. புதிருக்கு முடிவிலி விடைகள் இருக்கலாம் அல்லது விடைகளே இல்லாமல் இருக்கலாம்.
- **டிட்டர்மினன்ட் ≠ 0** என்றால், புதிருக்கு **நிச்சயமாக ஒரே ஒரு தனித்துவமான விடை உண்டு.**

Python

```
determinant = np.linalg.det(matrix_A)
print(f"டிட்டர்மினன்ட்: {determinant:.2f}")
```

**வெளியீடு:**

```
டிட்டர்மினன்ட்: -2.00
```

இதன் மதிப்பு பூஜ்யம் இல்லை. எனவே, `matrix_A`-ஐப் பயன்படுத்தி ஒரு புதிரை உருவாக்கினால், அதற்கு நிச்சயம் ஒரு விடை உண்டு!

### ௰௯.௪ . புதிருக்கு விடை காணுதல்: `solve`

இதுதான் துப்பறிவாளரின் இறுதி மற்றும் சக்திவாய்ந்த கருவி. தடயங்களை ஒழுங்குபடுத்தி, விடை உண்டு என்பதை உறுதி செய்தபின், புதிருக்கான விடையை நேரடியாகக் கண்டறிய இது உதவுகிறது.

**புதிர்:**

- 3 ஆப்பிள் மற்றும் 1 வாழைப்பழம் வாங்கினால் விலை ₹9. (`3x + 1y = 9`)
- 1 ஆப்பிள் மற்றும் 2 வாழைப்பழம் வாங்கினால் விலை ₹8. (1x + 2y = 8)

  கேள்வி: ஒரு ஆப்பிளின் (x) விலை என்ன? ஒரு வாழைப்பழத்தின் (y) விலை என்ன?

Python

```
# தடயங்கள் (சமன்பாடுகளின் குணகங்கள்)
coefficients = np.array([[3, 1], [1, 2]])

# முடிவுகள்
constants = np.array([9, 8])

# புதிரை விடுவிப்போம்
solutions = np.linalg.solve(coefficients, constants)
print(f"விடை: {solutions}")
print(f"அதாவது, ஆப்பிள் (x) = {solutions[0]}, வாழைப்பழம் (y) = {solutions[1]}")
```

**வெளியீடு:**

```
விடை: [2. 3.]
அதாவது, ஆப்பிள் (x) = 2.0, வாழைப்பழம் (y) = 3.0
```

புதிர் விடுவிக்கப்பட்டது! நேரியல் இயற்கணிதம் என்ற இந்த சக்திவாய்ந்த தர்க்க மொழி மூலம், NumPy சிக்கலான புதிர்களுக்கு எளிதாக விடை காண்கிறது.

### உம்.க. NumPy – தரவின் கதையை வரைபடமாக்குதல் (Histogram)

நம்மிடம் ஆயிரக்கணக்கான எண்கள் கொண்ட ஒரு தரவுத் தொகுதி இருந்தால், அதைப் பார்த்தவுடன் நம்மால் எதையும் புரிந்துகொள்ள முடியாது. அந்தத் தரவின் **கதையைச் சொல்லும் ஒரு சித்திரம்தான் ஹிஸ்டோகிராம் (Histogram)**.

இதை, ஒரு பெரிய **காசு குவியலை வகைப்படுத்துவதுடன்** ஒப்பிடலாம்.

- **தரவு (Data):** உங்கள் முன் குவிந்து கிடக்கும் ஆயிரம் காசுகள்.
- **தொட்டிகள் (Bins):** `₹1-₹10`, `₹11-₹20`, `₹21-₹30` என எழுதப்பட்ட காலி ஜாடிகள்.
- **ஹிஸ்டோகிராம் செயல்பாடு:** ஒவ்வொரு காசாக எடுத்து, அதன் மதிப்புக்குரிய ஜாடியில் போடுகிறீர்கள்.
- **இறுதி வரைபடம்:** எல்லா காசுகளையும் போட்ட பிறகு, ஒவ்வொரு ஜாடியின் உயரத்தைப் பார்த்தால், எந்த மதிப்புள்ள காசுகள் அதிகமாக உள்ளன என்பது ஒரு நொடியில் தெரிந்துவிடும்.

`numpy.histogram()` மற்றும் `matplotlib.pyplot.hist()`

இங்கே இரண்டு முக்கியக் கருவிகள் உள்ளன:

1. **`np.histogram()` (கணக்காளர்):** இந்தக் கருவி, காசுகளை ஜாடியில் போடும் வேலையை மட்டும் செய்யும். அதாவது, ஒவ்வொரு ஜாடியிலும் (`bin`) எத்தனை காசுகள் (`frequency`) உள்ளன என்ற எண்ணிக்கையை மட்டும் நமக்குத் தரும். இது வரைபடத்தை உருவாக்காது.

2. **`plt.hist()` (ஓவியர்):** இந்தக் கருவி ஒரு முழுமையான கலைஞர். இது கணக்காளர் செய்யும் வேலையையும் செய்து (காசுகளை எண்ணி), அதை ஒரு அழகான வரைபடமாகவும் வரைந்து தந்துவிடும்.

**எடுத்துக்காட்டு (ஓவியர் `plt.hist()`):**

Python

```python
import numpy as np
import matplotlib.pyplot as plt

# இயல்நிலைப் பரவலில் (Normal Distribution) 1000 சீரற்ற எண்களை உருவாக்குவோம்
data = np.random.randn(1000)

# 30 ஜாடிகளை (bins) உருவாக்கி, தரவை ஹிஸ்டோகிராம் ஆக வரைவோம்
plt.hist(data, bins=30, alpha=0.7, color='blue')

plt.title("தரவுப் பரவலின் ஹிஸ்டோகிராம்")
plt.xlabel("மதிப்புகள்")
plt.ylabel("அதிர்வெண் (Frequency)")
plt.grid(True)
plt.show()
```

இந்த வரைபடம், நமது தரவில் உள்ள பெரும்பாலான எண்கள் `0`-ஐச் சுற்றி அமைந்துள்ளன என்ற கதையை அழகாகச் சொல்கிறது. தரவின் அடர்த்தியைப் புரிந்துகொள்ள ஹிஸ்டோகிராம் ஒரு மிகச் சிறந்த கருவியாகும்.

### ௨௰.௨. NumPy – தரவைச் சேமித்தலும் மீட்டலும் (Saving and Loading Data)

நாம் கடினமாக உழைத்து உருவாக்கிய NumPy அணிகளை, பின்னர் பயன்படுத்துவதற்காகக் கோப்புகளில் (`files`) சேமிக்க வேண்டும். இதை, **நமது பொருட்களைப் பயணத்திற்காகப் பொதி கட்டுவதுடன் ஒப்பிடலாம்.** நாம் தேர்ந்தெடுக்கும் பொதி கட்டும் முறை, நமது தேவையைப் பொறுத்தது.

#### ௨௰.௨.௧. NumPy-யின் பிரத்யேக வடிவம்: `.npy` கோப்புகள்

இது, நமது உணவுப் பொருட்களை **வெற்றிடப் பொட்டலம் (vacuum-sealing)** செய்வது போன்றது.

- **நன்மைகள்:** மிக வேகமாகவும், கச்சிதமாகவும், தரவின் துல்லியம் சற்றும் குறையாமலும் சேமிக்கப்படும்.
- **குறைபாடு:** இந்தப் பொட்டலத்தை NumPy என்ற பிரத்யேக இயந்திரத்தால் மட்டுமே திறக்க முடியும்.

**`np.save()` (சேமிக்க) மற்றும் `np.load()` (மீட்டெடுக்க):**

Python

```python
# சேமிக்க வேண்டிய அணி
a = np.array([[1, 2, 3], [4, 5, 6]])
np.save('my_array.npy', a)

# சேமித்த கோப்பிலிருந்து தரவை மீட்டெடுப்போம்
loaded_a = np.load('my_array.npy')
print("மீட்டெடுக்கப்பட்ட அணி:\n", loaded_a)
```

#### ௨௰.௨.௨. அனைவராலும் படிக்கக்கூடிய வடிவம்: `.txt` கோப்புகள்

இது, நமது சமையல் குறிப்பை ஒரு **சாதாரண காகிதத்தில்** 📝 எழுதி வைப்பது போன்றது.

- **நன்மைகள்:** இதை யார் வேண்டுமானாலும் (மனிதர்கள், Excel, பிற மென்பொருட்கள்) எளிதாகப் படிக்கலாம்.
- **குறைபாடு:** அதிக இடம் எடுக்கும், வேகம் குறைவு, மிதக்கும் புள்ளி எண்களின் துல்லியம் குறைய வாய்ப்புள்ளது.

**`np.savetxt()` (சேமிக்க) மற்றும் `np.loadtxt()` (மீட்டெடுக்க):**

Python

```python
# அணியை ஒரு டெக்ஸ்ட் கோப்பாகச் சேமிப்போம்
np.savetxt('my_array.txt', a, delimiter=',')

# அந்த டெக்ஸ்ட் கோப்பிலிருந்து தரவை மீட்டெடுப்போம்
loaded_txt = np.loadtxt('my_array.txt', delimiter=',')
print("\nடெக்ஸ்ட் கோப்பிலிருந்து மீட்டெடுக்கப்பட்ட அணி:\n", loaded_txt)
```

### ௨௰.௨.௫. பல அணிகளை ஒரே கோப்பில் சேமித்தல்: `.npz` கோப்புகள்

இது, பல வெற்றிடப் பொட்டலங்களை, ஒரே **பெரிய கொள்கலனில் (container)** வைத்துப் பூட்டுவது போன்றது. பல அணிகளை ஒரே கோப்பில் சேமிக்க இது உதவுகிறது.

**`np.savez()` (சேமிக்க) மற்றும் `np.load()` (மீட்டெடுக்க):**

Python

```python
b = np.array([7, 8, 9])

# a மற்றும் b ஆகிய இரண்டு அணிகளையும் ஒரே .npz கோப்பில் சேமிப்போம்
np.savez('my_archive.npz', array_x=a, array_y=b)

# அந்தக் கொள்கலனைத் திறந்து, உள்ளே இருக்கும் பொட்டலங்களை எடுப்போம்
archive_data = np.load('my_archive.npz')

print("\nகொள்கலனிலிருந்து மீட்டெடுக்கப்பட்ட அணி 'x':\n", archive_data['array_x'])
print("கொள்கலனிலிருந்து மீட்டெடுக்கப்பட்ட அணி 'y':\n", archive_data['array_y'])
```

`np.savez_compressed()` என்ற கருவி, இதே வேலையைச் செய்து, கொள்கலனைச் சுருக்கி (compress), மிகக் குறைந்த இடத்தில் சேமிக்க உதவும்.

| செயல்பாடு | கோப்பு வடிவம் | முக்கியப் பண்பு | எப்போது பயன்படுத்த வேண்டும்? |
|---|---|---|---|
| `np.save / np.load` | `.npy` | வேகம், துல்லியம், சிறிய அளவு | NumPy-க்கு உள்ளேயே தரவைப் பரிமாற. |
| `np.savetxt / np.loadtxt` | `.txt`, `.csv` | மனிதரால் படிக்கக்கூடியது | பிற மென்பொருட்களுடன் (Excel) பகிர. |
| `np.savez / np.load` | `.npz` | பல அணிகளின் தொகுப்பு | பல அணிகளை ஒரே கோப்பில் சேமிக்க. |