# Vectorized Operations

הטכניון
היחידה ללימודי המשך
ולימודי חוץ ע״ש עזריאלי

**Vectorized operations allow you to perform operations on entire arrays or Series (collections of data) without using explicit loops.**

- **Why It's Important**:
- **Efficiency**: Faster execution than loops.
- **Cleaner Code**: More readable and concise.
- **Parallelism**: Many vectorized operations are optimized to run on multiple CPU cores, making them faster for large datasets.

**Examples of vectorized operations in Pandas:**

- where()
- select()

# Why Vectorized Operations Matter

- **Performance**:
  - Vectorized operations are implemented in C (underlying NumPy and pandas), making them much faster than Python loops.
  - Example: Adding two large lists using loops vs. vectorized operations.
- **Memory Efficiency**:

Operations on entire arrays are performed in a single pass, avoiding the overhead of Python loops.

# Vectorized vs. loop

loop

```
a = np.array([1, 2, 3])
b = np.array([4, 5, 6])
result = []
for i in range(len(a)):
    result.append(a[i] + b[i])
result
```

```
[5, 7, 9]
```

Vectorized

```
a = np.array([1, 2, 3])
b = np.array([4, 5, 6])
result = a + b
print(result)
```

```
[5 7 9]
```

- **Loops**: Slower because Python executes the loop one iteration at a time.
- **Vectorized**: Much faster, executed in C (internally optimized).

# The apply Method -

**Purpose:** Apply a function to each element, row, or column of a DataFrame or Series.

```
Series.apply(func)
DataFrame.apply(func, axis=0)
```

Lambda or predefined function

```
# axis=0 for columns, axis=1 for rows
```

**Code example**

```
df.apply(lambda row: row['A'] + row['B'], axis=1)
```

# The apply Method -

apply is **not truly vectorized**.

- While it is more concise and easier to use than explicit loops, it still operates element-by-element or row-by-row.
- Internally, apply often calls Python functions (like lambda), which makes it slower than fully vectorized operations.
- For large datasets, the overhead of Python function calls in apply adds up and can make it significantly slower than NumPy-based or vectorized operations.

# The where Method

Purpose: Conditionally replace values in a DataFrame or Series

```
DataFrame.where(condition, other)
```

## .where()

The where() method is used to filter data, retaining values that satisfy a condition and replacing others with NaN.

```
#recreate evertihng
mask_k = cereal["mfr"]=='K'
cereal[mask_k]
cereal.where(mask_k)
```

| name | mfr | type | calories | protein | fat | sodium | fiber | carbo | sugars | potass | vitamins | shelf | weight | cups | rating |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 100% Bran | NaN | NaN | NaN | NaN | NaN | NaN | NaN | NaN | NaN | NaN | NaN | NaN | NaN | NaN | NaN |
| 100% Natural Bran | NaN | NaN | NaN | NaN | NaN | NaN | NaN | NaN | NaN | NaN | NaN | NaN | NaN | NaN | NaN |
| All-Bran | K | C | 70.0 | 4.0 | 1.0 | 260.0 | 9.0 | 7.0 | 5.0 | 320.0 | 25.0 | 3.0 | 1.0 | 0.33 | 59.425505 |
| All-Bran with Extra Fiber | K | C | 50.0 | 4.0 | 0.0 | 140.0 | 14.0 | 8.0 | 0.0 | 330.0 | 25.0 | 3.0 | 1.0 | 0.50 | 93.704912 |
| Almond Delight | NaN | NaN | NaN | NaN | NaN | NaN | NaN | NaN | NaN | NaN | NaN | NaN | NaN | NaN | NaN |
| ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... |

Condition NOT satisfied

Condition satisfied



TECHNION Azrieli Continuing Education and External Studies Division

# The select Function

Purpose: Conditionally replace values in a DataFrame or Series.

```
np.select(conditions, choices, default=0)
```

list of bool ndarrays

list of bool ndarrays

scalar, optional

```
conditions = [df['Fare']> 50, df['Fare']<20]
outcomes = ['Expensive', 'Cheep']
np.select(conditions,outcomes, 'ok')
```

# The select Function

```
conditions = [df['Fare']> 50, df['Fare']<20]
outcomes = ['Expensive', 'Cheep']
np.select(conditions,outcomes, 'ok')
```

select se example

```
pd.concat ([df,
            pd.Series(np.select(conditions,outcomes, 'ok'))]
           ,axis =1)
```

Concat the "select" column to the data frame

| | Name | Sex | Age | Fare | 0 |
|---|---|---|---|---|---|
| 0 | Mr. Owen Harris Braund | male | 22.0 | 7.2500 | Sir |
| 1 | Mrs. John Bradley (Florence Briggs Thayer) Cum... | female | 38.0 | 71.2833 | adult |
| 2 | Miss. Laina Heikkinen | female | 26.0 | 7.9250 | She/Her |
| 3 | Mrs. Jacques Heath (Lily May Peel) Futrelle | female | 35.0 | 53.1000 | adult |
| 4 | Mr. William Henry Allen | male | 35.0 | 8.0500 | Sir |
| ... | ... | ... | ... | ... | ... |

Final result

# Best Practices

- Use vectorized operations whenever possible.
- Avoid explicit loops over DataFrame rows or columns.
- Combine apply, where, and select for advanced use cases.
- Leverage NumPy functions when required for additional functionality.

| Feature | apply | where | np.select |
|---|---|---|---|
| Vectorized? | No | Yes | Yes |
| Flexibility | Very high (supports custom functions) | Medium (conditionally replaces values) | High (handles multiple conditions) |
| Performance | Slower (element-wise operations) | Faster (column/array-based) | Faster (multiple vectorized conditions) |
| Use Case | Complex logic with custom functions | Simple conditional replacements | Multiple complex conditions |

# Summary

- Vectorized operations streamline data analysis in Pandas.
- Key functions discussed:
  - **apply** for applying functions element-wise
  - **where** for conditional updates
  - **select** for multi-condition choices

# Additional Resources

- **Pandas Documentation**: https://pandas.pydata.org/docs/
- **NumPy Documentation**: https://numpy.org/doc/stable/