NumPy, scipy, pandas

Maciej Marchwiany, PhD

https://www.linkedin.com/pulse/ai-powered-physics-how-ai-revolutionizing-scientific-discovery-cain-ddbwc

Plan

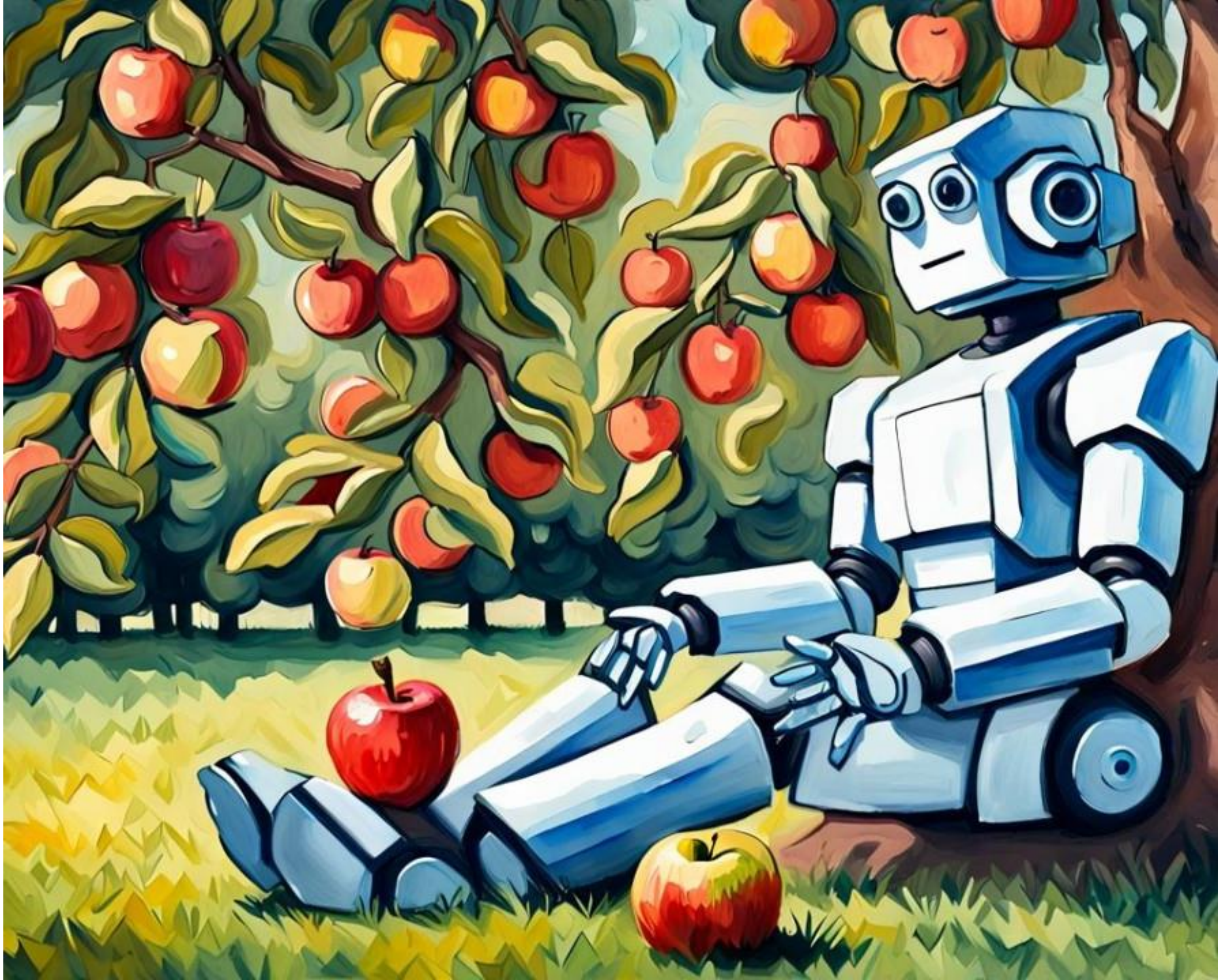
NumPy

scipy

pandas

optimization

NumPy



NumPy

NumPy (Numerical Python) is a fundamental library for numerical computing in Python. It provides support for large, multi-dimensional arrays and matrices, along with a collection of mathematical functions to operate on these data structures efficiently.

NumPy is widely used in data science, machine learning, and scientific computing due to its speed and efficiency compared to native Python lists.

NumPy is a Python library used for working with arrays. It also has functions for working in domain of linear algebra, fourier transform, and matrices.

Installation:

```
pip install numpy
```

Why Use NumPy

- **Faster** – Uses C-based implementation, much quicker than Python lists.
- **Memory Efficient** – Uses less memory than Python lists.
- **Vectorized Operations** – Perform element-wise operations without loops.
- **Built-in Math Functions** – Supports mean, std, sin, matrix multiplication, etc.
- **Multi-Dimensional Support** – Works with 2D & 3D arrays, broadcasting.
- **Used in Data Science & ML** – Essential for Pandas, Scikit-Learn, TensorFlow.

Create an array

- NumPy is used to work with arrays. The array object in NumPy is called ndarray.

```
import numpy as np
arr1 = np.array([1, 2, 3, 4, 5])

Arr2 = np.arange(5)
```

```
array([ 1,  2,  3,  4, 5])
```

```
np.array([(1.5, 2, 3), (4, 5, 6)])
```

```
array([[1.5, 2. , 3. ],
       [4. , 5. , 6. ]])
```

```
np.array([[1, 2], [3, 4]], dtype=complex)
```

```
array([[1.+0.j, 2.+0.j],
       [3.+0.j, 4.+0.j]])
```

Create an array

```
np.zeros((3, 4))
```

```
array([[0., 0., 0., 0.],
       [0., 0., 0., 0.],
       [0., 0., 0., 0.]])
```

```
np.ones((2, 3, 4), dtype=np.int16)
```

```
array([[[1, 1, 1, 1],
        [1, 1, 1, 1],
        [1, 1, 1, 1]],

       [[1, 1, 1, 1],
        [1, 1, 1, 1],
        [1, 1, 1, 1]]],
dtype=int16)
```

```
np.arange(0, 2, 0.3)
```

```
array([0. , 0.3, 0.6, 0.9, 1.2, 1.5, 1.8])
```

Shape and reshape

- ndarray.ndim - the dimension of the array.
- ndarray.shape – the dimensions of the array. This is a tuple of integers indicating the size of the array in each dimension
- ndarray.size - the total number of elements of the array.
- ndarray.reshape(<new_shape>) – reshape the array

```
import numpy as np
a = np.arange(15).reshape(3, 5)
```

```
array([[ 0,  1,  2,  3,  4],
       [ 5,  6,  7,  8,  9],
       [10, 11, 12, 13, 14]])
```

```
print(”Dimensions: ”, a.shape)
print(”Dimension: ”,a.ndim)
print(”Size: ”, a.size)
```

Basic operations

```
a = np.array([20, 30, 40, 50])
b = np.arange(4)
```

```
array([0, 1, 2, 3])
```

```
c = a - b
```

```
array([20, 29, 38, 47])
```

```
b**2
```

```
array([0, 1, 4, 9])
```

```
10 * np.sin(a)
```

```
array([ 9.12945, -9.88031,  7.45113 , -2.62374])
```

```
a < 35
```

```
array([ True,  True, False, False])
```

Basic operations

```
A = np.array([[1, 1],
              [0, 1]])
B = np.array([[2, 0],
              [3, 4]]
A * B      # elementwise product
```

```
array([[2, 0],
       [0, 4]])
```

```
A @ B      # matrix product
```

```
array([[5, 4],
       [3, 4]])
```

```
A.dot(B)   # another matrix product
```

```
array([[5, 4],
       [3, 4]])
```

Basic operations

```
a = np.random((2, 3))
```

```
array([[0.82770, 0.40919, 0.54959,
        [0.02755, 0.75351, 0.53814]])
```

```
a.sum()
```

```
3.105710952999
```

```
a.min()
```

```
0.027559113243
```

```
a.sum(axis=0)
```

```
array([[0.85525, 1.16270, 1.08763])
```

```
a.max(axis=1)
```

```
array([[0.82770, 0.75351])
```

```python
import numpy as np
a = np.array([1, 2, 3])
b = np.array([4, 5, 6])
```

```python
 #Addition
(a + b)   # [5 7 9]
# Subtraction
(a - b)   # [-3 -3 -3]
# Multiplication)
(a * b)   # [4 10 18]
# Division
(a / b)   # [0.25 0.4 0.5 ]
# Floor Division
(a // b)   # [0 0 0]
# Modulus (remainder)
(a % b)   # [1 2 3]
# Power
(a ** 2)   # [1 4 9
```

```python
(a==b)
(a>b)
np.logical_and(a>1, a<3)
np.logical_or(a>1, a<3)
```

```python
# Sum of all elements
(np.sum(a))   # 6
# Mean (average)
(np.mean(a))   # 2.0
# Median
(np.median(a))   # 2.0
# Standard deviation
(np.std(a))   # 0.816496
# Variance
(np.var(a))   # 0.666666
# Minimum and maximum
(np.min(a))   # 1
(np.max(a))   # 3
# Cumulative sum
(np.cumsum(a))   # [1 3 6]
# Cumulative product
(np.cumprod(a))   # [1 2 6]
```

```python
# Sine
(np.sin(angles))
# Cosine
(np.cos(angles))
# Tangent
(np.tan(angles))
# Inverse trigonometric
```

```python
# Floor (round down)
(np.floor(a))
# Ceil (round up)
(np.ceil(a))
# Round to nearest integer
(np.round(a))
```

```python
# Exponential (e^x)
(np.exp(x))
# Natural Log (ln)
(np.log(x))
# Log base 10
(np.log10(x))
# Log base 2
(np.log2(x))
```

```python
# Matrix multiplication
(np.dot(A, B)) = (A @ B)
# Transpose
(A.T)
# Determinant
(np.linalg.det(A
# Inverse of a matrix
(np.linalg.inv(A))
# Eigenproblem
```

```
b = np.array([4, 5, 6])
```

```
 #Addition
(a + b)  # [5 7 9]
# Subtraction
(a - b)  # [-3 -3 -3]
# Multiplication)
(a * b)  # [4 10 18]
# Division
(a / b)  # [0.25 0.4 0.5 ]
# Floor Division
(a // b)  # [0 0 0]
# Modulus (remainder)
(a % b)  # [1 2 3]
# Power
(a ** 2)  # [1 4 9
```

```
(a==b)
(a>b)
np.logical_and(a>1, a<3)
np.logical_or(a>1, a<3)
```

```
# Mean (average)
(np.mean(a))  # 2.0
# Median
(np.median(a))  # 2.0
# Standard deviation
(np.std(a))  # 0.816496
# Variance
(np.var(a))  # 0.666666
# Minimum and maximum
(np.min(a))  # 1
(np.max(a))  # 3
# Cumulative sum
(np.cumsum(a))  # [1 3 6]
# Cumulative product
(np.cumprod(a))  # [1 2 6]
```

```
# Sine
(np.sin(angles))
# Cosine
(np.cos(angles))
# Tangent
(np.tan(angles))
# Inverse trigonometric
(np.arcsin([0, 1, 0]))
```

```
# Ceil (round up)
(np.ceil(a))
# Round to nearest integer
(np.round(a))
```

```
# Exponential (e^x)
(np.exp(x))
# Natural Log (ln)
(np.log(x))
# Log base 10
(np.log10(x))
# Log base 2
(np.log2(x))
```

```
# Matrix multiplication
(np.dot(A, B)) = (A @ B)
# Transpose
(A.T)
# Determinant
(np.linalg.det(A
# Inverse of a matrix
(np.linalg.inv(A))
# Eigenproblem
eval,evec=np.linalg.eig(A)
```

Indexing, slicing and iterating

```
a = np.arange(10)**3
```

```
array([ 0,  1,  8,  27,  64, 125, 216, 343, 512, 729])
```

```
a[2]
```

```
8
```

```
A[-1]
```

```
729
```

```
a[2:5]
```

```
array([ 8, 27, 64])
```

```
a[:6:2] = -1
```

```
array([-1,  1,-1, 27, -1, 125, 216, 343, 512, 729])
```

```
a[::-1]  # reversed a
```

```
array([729, 512, 343, 216, 125, -1, 27,-1, 1,-1])
```

Stacking arrays

```
a = np.floor(10 * rg.random((2, 2)))
```

```
array([[9., 7.],
       [5., 2.]])
```

```
b = np.floor(10 * rg.random((2, 2)))
```

```
array([[1., 9.],
       [5., 1.]])
```

```
np.vstack((a, b))
```

```
array([[9., 7.],
       [5., 2.],
       [1., 9.],
       [5., 1.]])
```

```
np.hstack((a, b)
```

```
array([[9., 7., 1., 9.],
       [5., 2., 5., 1.]])
```

Copies

```
a = np.array([1, 2, 3])
```

- No copy at all

```
b = a
```

- Copy

```
c = a[:]
```

- Deep copy

```
d = a.copy()
```

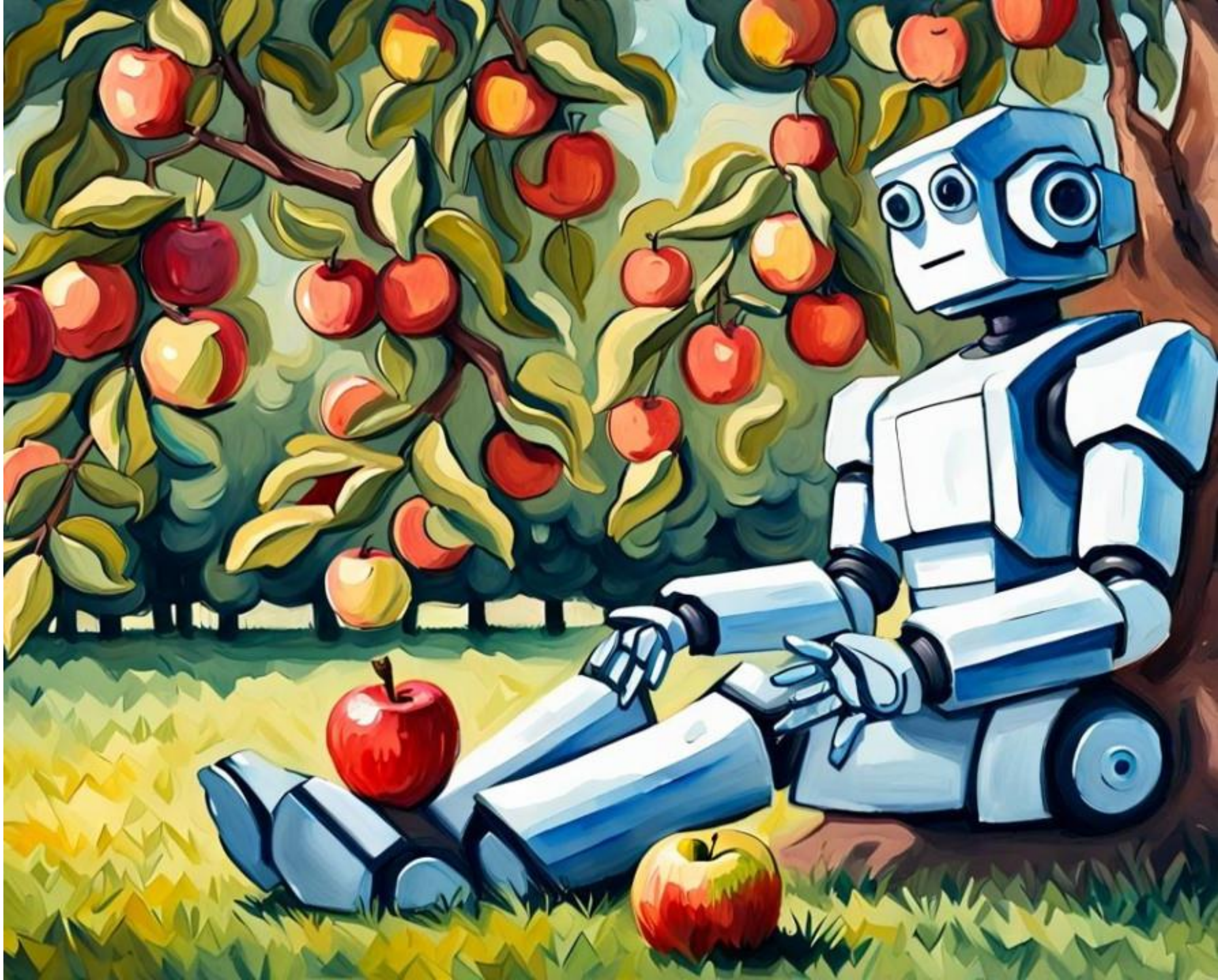
```
b[1] = 5
```

```
a = array([1, 5, 3])
b = array([1, 5, 3])
c = array([1, 2, 3])
```

```
c[1] = 8
```

```
a = array([1, 5, 3])
b = array([1, 5, 3])
c = array([1, 8, 3])
```

scipy



SciPy

- SciPy (Scientific Python) is an open-source library in Python used for scientific computing and technical computing. It is built on top of **NumPy** and provides additional functionality for optimization, integration, interpolation, eigenvalue problems, signal processing, and other advanced mathematical and scientific computations.

Key Features of SciPy

- **Optimization** (`scipy.optimize`) - Functions for minimizing or maximizing objective functions.
- **Integration** (`scipy.integrate`) - Tools for numerical integration (e.g., solving differential equations).
- **Linear Algebra** (`scipy.linalg`) - Advanced linear algebra operations, similar to those in MATLAB.
- **Interpolation** (`scipy.interpolate`) - Functions for interpolating data points.
- **Fourier Transforms** (`scipy.fft`) - Efficient Fast Fourier Transform (FFT) implementations.
- **Signal Processing** (`scipy.signal`) - Tools for filtering and analyzing signals.
- **Statistics** (`scipy.stats`) - A large number of probability distributions and statistical functions.
- **Sparse Matrices** (`scipy.sparse`) - Support for working with large, sparse matrices.

Optimization

```
import numpy as np.
from scipy.optimize import minimize


# Define the function
def func(x):
    return x**2 + 3*x + 2


# Initial guess
x0 = 0


# Minimize the function
result = minimize(func, x0)


# Print the result
print("Optimal x:", result.x)
print("Minimum value of function:", result.fun)
```

Linear Algebra

```
from scipy.linalg import solve

# Coefficient matrix (A)
A = np.array([[2, 3], [5, 7]])

# Right-hand side (b)
b = np.array([8, 19])

# Solve for x and y
solution = solve(A, b)

print("Solution:", solution)
```

Interpolation

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.interpolate import interp1d

# Given data points
x = np.array([0, 1, 2, 3, 4])
y = np.array([1, 3, 7, 13, 21])

# Create interpolation function
f = interp1d(x, y, kind='cubic')

# New x values for interpolation
x_new = np.linspace(0, 4, 50)
y_new = f(x_new)

# Plot
```

FFT

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.fft import fft, fftfreq

# Generate a sine wave
t = np.linspace(0, 1, 1000, endpoint=False)
signal = np.sin(2 * np.pi * 50 * t)  # 50 Hz sine wave

# Compute FFT
fft_values = fft(signal)
freqs = fftfreq(len(signal), 1/1000)  # Frequency bins

# Plot
plt.plot(freqs[:500], np.abs(fft_values[:500]))  # Only positive frequencies
```

Sparse Matrices

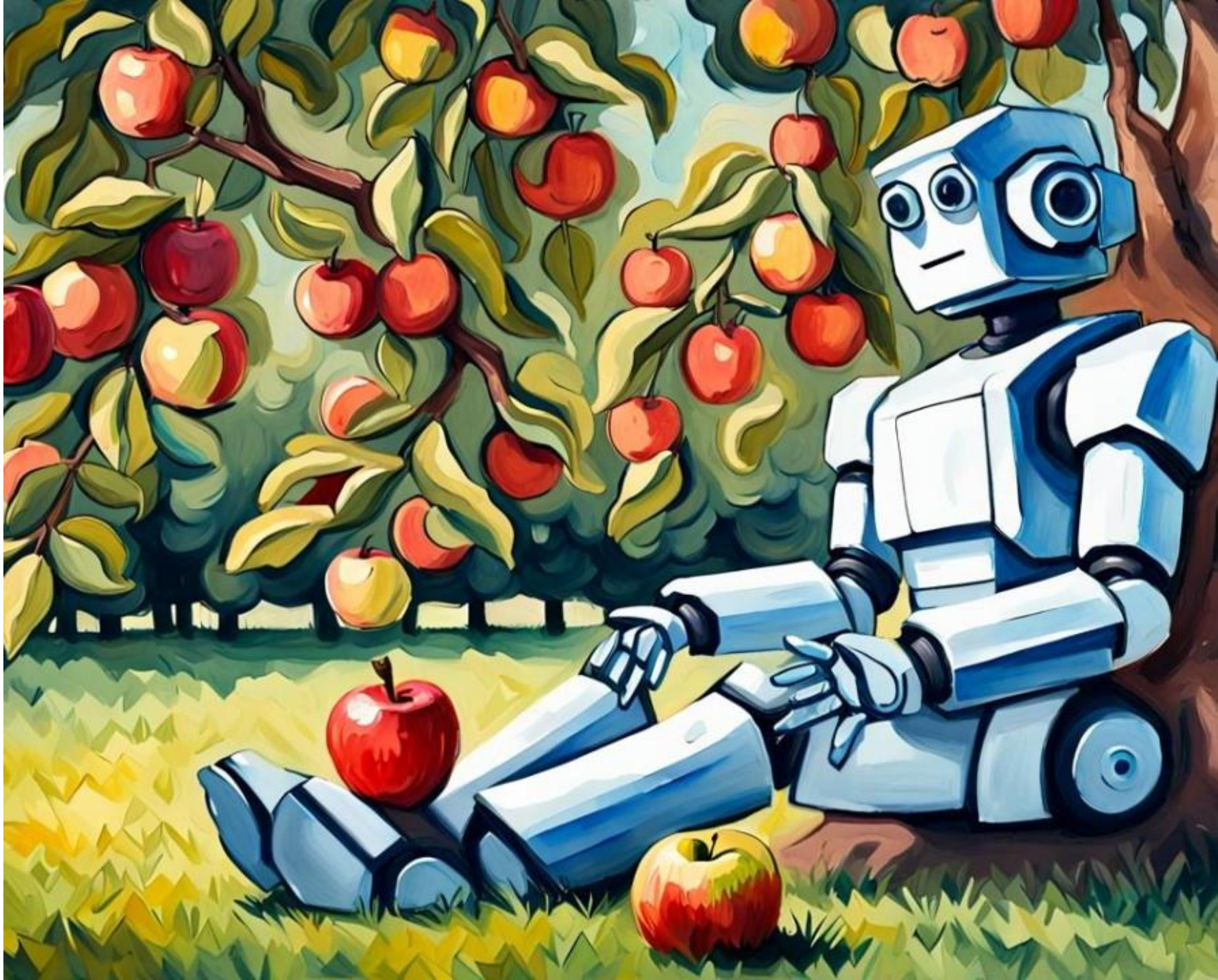
```
from scipy.sparse import csr_matrix
from scipy.sparse.linalg import spsolve

# Define sparse matrix A
A = csr_matrix([[3, 0, 2], [2, 3, 0], [0, 1, 4]])

# Define vector b
b = np.array([2, 4, 6])

# Solve Ax = b
x = spsolve(A, b)
print("Solution x:", x)
```

pandas



Pandas

Pandas is an open-source Python library used for **data manipulation, analysis, and cleaning**. It provides **fast, flexible, and powerful** data structures like **DataFrames** and **Series** that make working with structured data easy. Pandas is widely used in **data science, machine learning, and finance** for handling large datasets efficiently.
Features:

- **DataFrame & Series** – Flexible data structures for tabular and time-series data.
- **Data Cleaning** – Handling missing values, duplicates, and inconsistent data.
- **Data Transformation** – Filtering, grouping, merging, and reshaping data.
- **Data Analysis** – Built-in statistical functions (mean, median, correlation, etc.).
- **Integration** – Works well with NumPy, SciPy, Matplotlib, and SQL databases.

DataFrame

```
data = {
    "Name": ["Alice", "Bob", "Charlie"],
    "Age": [25, 30, 35],
    "Salary": [50000, 60000, 70000]
}
df = pd.DataFrame(data)
print(df)
```

```
       Name      Age    Salary
0      Alice     25     50000
1       Bob      30     60000
2      Charlie   35     70000
```

Reading Data

- CSV File

```
df = pd.read_csv("data.csv")
```

- Excel File

```
df = pd.read_excel("data.xlsx")
```

- SQL

```
import sqlite3

conn = sqlite3.connect("database.db")
df = pd.read_sql("SELECT * FROM employees", conn)
```

Data Exploration

- First 5 rows
  `df.head()`
- Last 5 rows
  `df.tail()`
- Data types and missing values
  `df.info()`
- Summary statistics
  `df.describe()`
- Print #rows, #columns
  `df.shape`
- List of column names
  `df.columns`

Data Exploratio

- First 5 rows

```
import pandas as pd

# Create a sample DataFrame
data = {
 "Age": [25, 30, 35, 40, 45],
 "Salary": [50000, 60000, 70000, 80000, 90000],
 "Experience (Years)": [2, 5, 7, 10, 12]
 }

df = pd.DataFrame(data)
# Get summary statistics
print(df.describe())
```

```
             Age        Salary  Experience (Years)
count   5.000000      5.000000            5.000000
mean   35.000000  70000.000000            7.200000
std     7.905694  15811.388301            3.962323
min    25.000000  50000.000000            2.000000
25%    30.000000  60000.000000            5.000000
50%    35.000000  70000.000000            7.000000
75%    40.000000  80000.000000           10.000000
max    45.000000  90000.000000           12.000000
```

Data Filtering

- Select a Column

```
df["Age"]
```

- Select Multiple Columns

```
df[["Name", "Salary"]]
```

- Filter Rows

```
df[df["Age"] > 30]
```

- Select Rows by Index or Position

```
import pandas as pd

data = {
    "Name": ["Alice", "Bob", "Charlie", "David", "Emma"],
    "Age": [25, 30, 35, 40, 45],
    "Salary": [50000, 60000, 70000, 80000, 90000]
}
df = pd.DataFrame(data, index=["A", "B", "C", "D", "E

print(df.loc["C"])
Print(df.iloc[2])
```

Missing Data

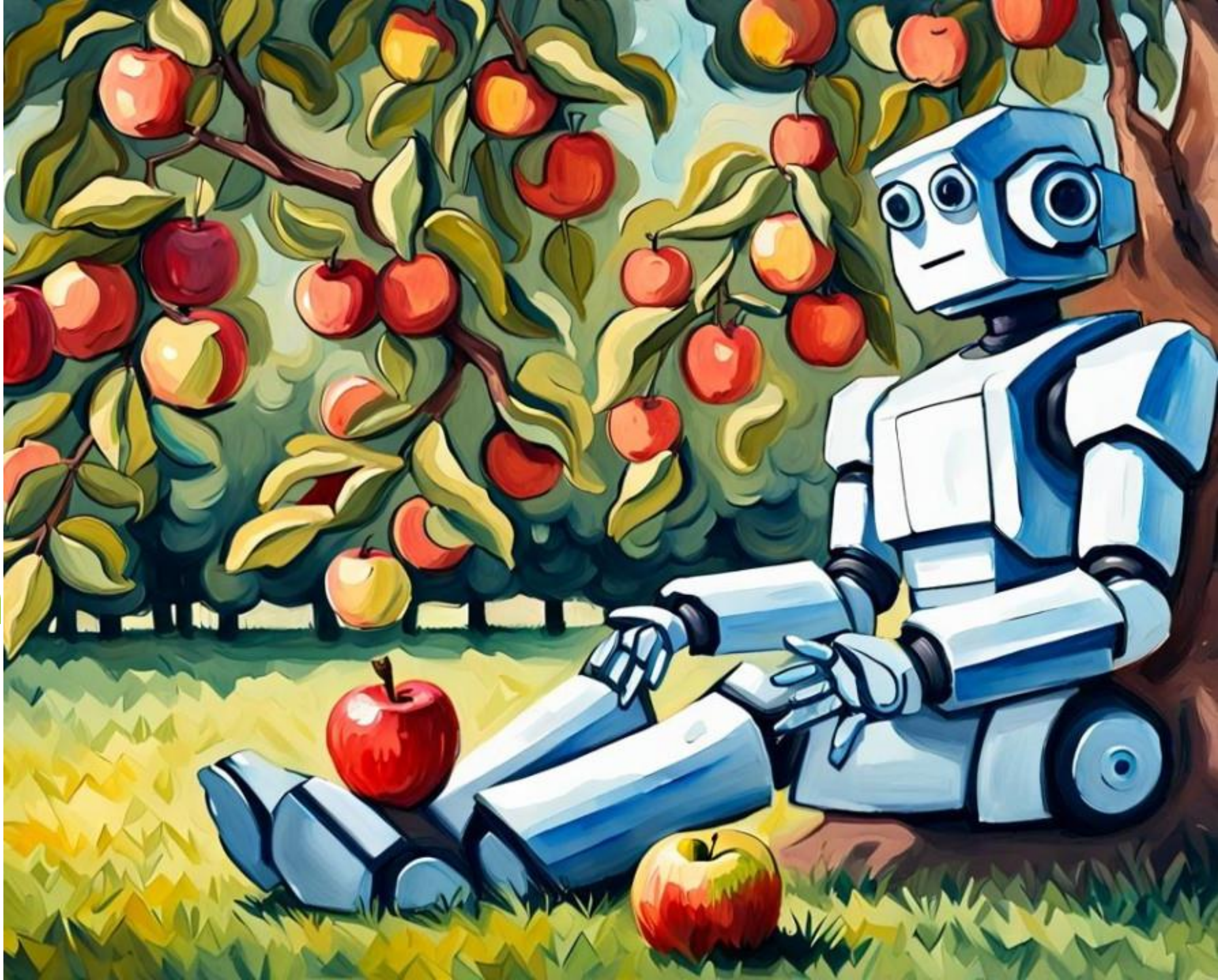
- Remove missing values

```
df.dropna()
```

- Replace missing values with 0

```
df.fillna(0)
```

Optimization



optimization

- Use right data type

```
# Defaults to float64 (8 bytes per element)
arr = np.ones(1000000)
```

```
# loat32 (4 bytes per element)
arr = np.ones(1000000, dtype=np.float32)
```

Speed Boost:
Up to 2x faster for large arrays (less cache memory usage).

optimization

- Use multi-threading

```
# Use 4 CPU cores
import os
os.environ["OMP_NUM_THREADS"] = "4"
```

Faster matrix operations, like np.dot(), np.linalg.inv(), and np.matmul().

optimization

- Use np.copy

```
arr = np.array([1, 2, 3])
arr_view = arr[:]
```

```
arr = np.array([1, 2, 3])
arr_copy = arr.copy()
```

optimization

- Use jit (Just-in-Time Compilation)

```python
def compute(arr):
    result = 0
    for x in arr:
        result += x ** 2
    return result
```

```python
from numba import jit

# Compile to fast machine code
@jit(nopython=True)
def compute(arr):
    result = 0
    for x in arr:
        result += x ** 2
    return result
```

optimization

- Optimize Memory Access Patterns

```
arr = np.random.rand(1000, 1000)

# Column-wise access (slow!)
for j in range(1000):
    for i in range(1000):
        arr[i, j] += 1
```

```
arr = np.random.rand(1000, 1000)

# Row-wise access (cache-friendly!)
for i in range(1000):
    for j in range(1000):
        arr[i, j] += 1
```

optimization

- Do not use loop, use vector operations

```
li_a =  np.random.rand(1000)
li_b =  np.random.rand(1000)

for i in range(len(li_a)):
       li_a[i] * li_b[i]
```

```
li_a =  np.random.rand(1000)
li_b =  np.random.rand(1000)

li_a[i] * li_b[i]
```

optimization

- Do not use loop, use vector function

```
li_a =  np.random.rand(1000)

prod = 0
for x in li_a:
      prod += x * 5
```

```
li_a =  np.random.rand(1000)

a_helper = li_a * 5
prod = a_helper.sum()
```

optimization

- Do not use vector operation, use broadcasting

```
arr = np.arange(12).reshape(3,4)
col_vector = np.array([5,6,7])

num_cols = arr.shape[1]

for col in range(num_cols):
        arr[:, col] += col_vector
```

```
arr = np.arange(12).reshape(3,4)
col_vector = np.array([5,6,7])

num_cols = arr.shape[1]

add_matrix = np.array([col_vector,] * num_cols).T

arr += add_matrix
```

```
[array([5, 6, 7]), ]
```

```
[array([5, 6, 7]),
 array([5, 6, 7]),
 array([5, 6, 7]),
 array([5, 6, 7])]
```

optimization

- use build-in functions

```
def relu(x):
    return x if x > 0 else 0
```

```
def relu(x):
    return max(0, x)
```

```
def abs_value(x):
    if x >= 0:
        return x
    else:
        return -x
```

```
def abs_value(x):
    return abs(x)
```

optimization

- Use Functional Programming (map and lambda)

```python
arr = [1, 2, 3, 4, 5]
result = []

for x in arr:
    if x % 2 == 0:
        result.append(x * 2)
    else:
        result.append(x * 3)
```

```python
arr = [1, 2, 3, 4, 5]
result = list(map(
    lambda x: x * 2 if x % 2 == 0 else x * 3, arr))
```

optimization

- Use NumPy Vectorization Instead of if-else Loops

```python
import numpy as np

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

for x in arr:
    if x % 2 == 0:
        result.append(x * 2)
    else:
        result.append(x * 3)

result = np.array(result)
```

```python
import numpy as np

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

# Vectorized, no loops!
result = np.where(arr % 2 == 0, arr * 2, arr * 3)
```

optimization

- Use Lookup Tables Instead of if-else Chains

```
def fruit_color(fruit):
    if fruit == "apple":
        return "red"
    elif fruit == "banana":
        return "yellow"
    elif fruit == "grape":
        return "purple"
    else:
        return "unknown"
```

```
fruit_colors = {
    "apple": "red",
    "banana": "yellow",
    "grape": "purple"
}

# O(1) lookup!
def fruit_color(fruit):
    return fruit_colors.get(fruit, "unknown")
```

```
def get_data(size = 10_000):
df = pd.DataFrame()
df['age'] = np.random.randint(0, 100, size)
df['time_at_work'] = np.random.randint(0,8,size)
df['percentage_productive'] = np.random.rand(size)
df['favorite_treat'] = np.random.choice(['ice_cream', 'boba', 'cookie'], size)
df['bad_karma'] = np.random.choice(['stub_toe', 'wifi_malfunction', 'extra_traffic'])
return df
```

```
def reward_calc(row):
  if row['age'] >= 65:
    return row ['favorite_treat']
  if (row['time_at_work'] >= 2) & (row['percentage_productive'] >= 0.5):
    return row ['favorite_treat']
  return row['bad_karma']
```


https://www.exxactcorp.com/blog/Deep-Learning/how-to-speed-up-python-pandas-by-over-300x

optimization

- Do not use looping, use appla, use vectorization

```
df = get_data()
for index, row in df.iterrows():
  df.loc[index, 'reward'] = reward_calc(row)
```

```
df = get_data()
df['reward'] = df.apply(reward_calc, axis=1)
```

```
df = get_data()
df['reward'] = df['bad_karma']
df.loc[((df['percentage_productive'] >= 0.5) &
        (df['time_at_work'] >= 2)) |
        (df['age'] >= 65), 'reward'] =
df['favorite_treat']
```

360x faster than Looping?

https://www.exxactcorp.com/blog/Deep-Learning/how-to-speed-up-python-pandas-by-over-300x

optimization

- Do not use appla, use vectorization

```
def update_record(row):
    if row['product_names'] == 'Smartphone':
        row['quantity'] *= 5
    else:
        row['quantity'] *= 2
    row['total'] = row['quantity'] * row['price']
    return row

df = df.apply(update_record, axis=1)
```

```
df['quantity'] =
np.where(df['product_names'].values == 'Smartphone’,
      df['quantity'].values * 5,
      df['quantity'].values * 2)
```

```
df.loc[df['product_names']=='Smartphone', 'quantity'] *= 5
df.loc[df['product_names']!='Smartphone', 'quantity'] *= 2

df['total'] = df['quantity'] * df['price']
```


https://towardsdatascience.com/speed-up-pandas-code-with-numpy-fdee76210794/

optimization

- Use .values or to_numpy()

```
df['new_col'] = df['col1'] + df['col2']
```

```
import numpy as np
df['new_col'] =
   np.add(df['col1'].values,df['col2'].values)
```

optimization

- Do not use Pandas, use paralel-Pandas

Moldin

```
import modin.pandas as pd

#use pandas functions
```

Pandaral·lel

```
import pandas as pd
from pandarallel import pandarallel

# Initialize Pandarallel
pandarallel.initialize()

# Define a function to apply to each row
def my_function(row):
    return row['A'] + row['B’]
#Pandas
df.apply(my_function, axis=1)
#Pandarallel
df.parallel_apply(my_function, axis=1)
```

optimization

- Do not use weak typing, use dtype and usecols

```
df = pd.read_csv("data.csv")
```

```
df = pd.read_csv("data.csv",
  usecols=['name', 'age', 'salary’],
  dtype={'age': 'int32', 'salary': 'float32'})
```

optimization

- Use C (or Cython)