Python for Developers: Databases, Web APIs and Data Analysis

Chandrashekar Babu <training@chandrashekar.info>

https://www.chandrashekar.info/

https://www.slashprog.com/

A comprehensive guide to working with data in Python: from storage to analysis.

Welcome to Day 3

Database Connectivity

Working with relational and NoSQL databases using Python

Web APIs

Building REST APIs with Flask and FastAPI

NumPy

Working with numerical data through arrays and vectorization

Pandas

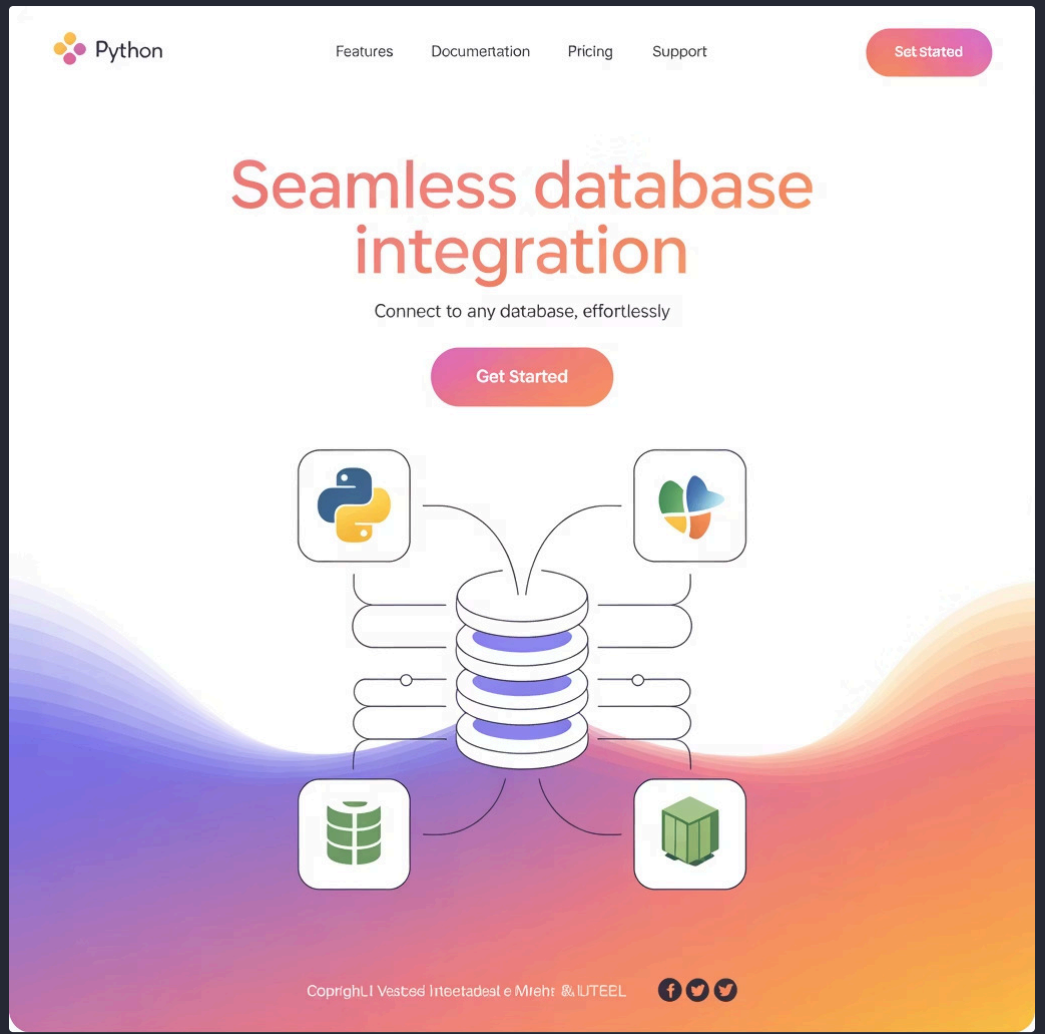
Manipulating tabular data with DataFrames

Database Connectivity in Python

Databases are the backbone of most applications, storing structured data for persistent access. Python offers multiple libraries for interacting with databases:

- sqlite3 (built-in for SQLite)
- psycopg2 (PostgreSQL)
- mysql-connector-python (MySQL)
- SQLAlchemy (ORM abstraction)
- pymongo (MongoDB)

Most Python database interfaces follow the DB-API 2.0 specification (PEP 249), providing a consistent interface regardless of the database engine.



Basic Database Connection Flow

Connect to database

Establish connection using appropriate driver

```
conn = sqlite3.connect('example.db')
```

Create cursor

Interface for executing SQL commands

```
cursor = conn.cursor()
```

Execute SQL

Run queries or statements

```
cursor.execute('SELECT * FROM users')
```

Fetch results

Retrieve data (for queries)

```
results = cursor.fetchall()
```

Commit changes

Save modifications to database

```
conn.commit()
```

Close connection

Release database resources

```
cursor.close()
conn.close()
```

CRUD Operations with Python

CRUD represents the four fundamental operations for persistent data storage:

1

Create

Inserting new records into database tables

```
cursor.execute("INSERT INTO users (name, email) VALUES (?, ?)",
               ("John Smith", "john@example.com"))
```

2

Read

Retrieving data from database tables

```
cursor.execute("SELECT * FROM users WHERE id = ?", (user_id,))
user = cursor.fetchone()
```

3

Update

Modifying existing records

```
cursor.execute("UPDATE users SET email = ? WHERE id = ?",
               ("new_email@example.com", user_id))
```

4

Delete

Removing records from database tables

```
cursor.execute("DELETE FROM users WHERE id = ?", (user_id,))
```

Preventing SQL Injection

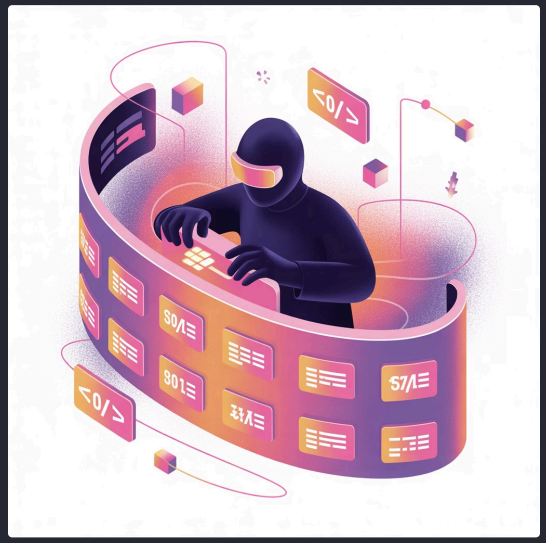
SQL injection is one of the most common security vulnerabilities in database applications. Always use parameterised queries instead of string concatenation.

Vulnerable:

```
user_id = "1; DROP TABLE users;"
query = f"SELECT * FROM users WHERE id = {user_id}"
cursor.execute(query)  # DANGEROUS!
```

Secure:

```
user_id = "1; DROP TABLE users;"
cursor.execute("SELECT * FROM users WHERE id = ?", (user_id,))  # SAFE!
```



Different database drivers use different parameterisation styles:

- SQLite/MySQL: ? placeholders
- PostgreSQL (psycopg2): %s placeholders
- Named parameters: :name or %(name)s

Database Transactions

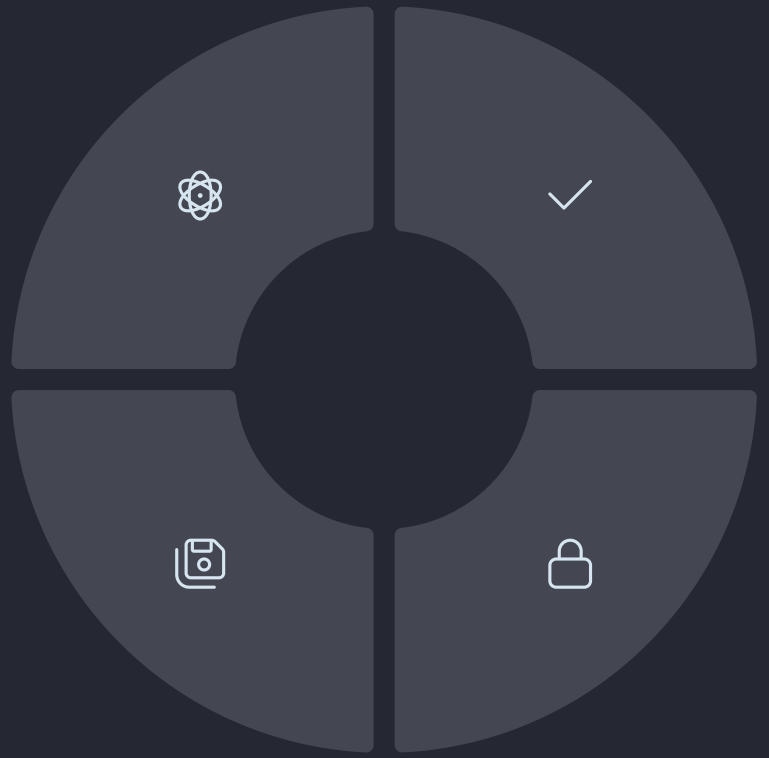
A transaction is a sequence of operations performed as a single logical unit of work. They ensure database consistency even during errors or concurrent access.

Atomicity

All operations complete successfully or none do

Consistency

Database remains in valid state before and after transaction



Durability

Once committed, changes persist even after system failures

Isolation

Concurrent transactions don't interfere with each other

Implementing Transactions in Python

```
import sqlite3

conn = sqlite3.connect('banking.db')
try:
    # Start transaction (implicit in most DB APIs)
    cursor = conn.cursor()

    # Multiple operations that should happen together
    cursor.execute("UPDATE accounts SET balance = balance - 100 WHERE id = ?", (account1_id,))
    cursor.execute("UPDATE accounts SET balance = balance + 100 WHERE id = ?", (account2_id,))

    # Commit transaction if all operations succeed
    conn.commit()
    print("Transfer successful")
except Exception as e:
    # Rollback all changes if any operation fails
    conn.rollback()
    print(f"Transfer failed: {e}")
finally:
    # Always close connections
    conn.close()
```

This pattern ensures that money transfers are atomic—either both accounts are updated or neither is, preventing inconsistent states.

Context Managers for Database Connections

Python's **with** statement provides cleaner syntax for managing database connections. It automatically handles closing connections even when exceptions occur.

Recommended approach:

```
import sqlite3

with sqlite3.connect('example.db') as conn:
    cursor = conn.cursor()
    cursor.execute("SELECT * FROM users")
    for row in cursor:
        print(row)
    # conn automatically closed
    # when block exits
```

Some database libraries provide their own context managers for both connections and cursors:

```
import psycopg2

with psycopg2.connect(dsn) as conn:
    with conn.cursor() as cursor:
        cursor.execute(
            "SELECT * FROM users")
        for row in cursor:
            print(row)
    # Both cursor and
    # connection properly closed
```

This approach eliminates resource leaks and simplifies error handling considerably.

SQLAlchemy: Python's SQL Toolkit and ORM

SQLAlchemy provides a full SQL abstraction layer that automates the tedious aspects of database interaction whilst maintaining flexibility.

Core (SQL Expression Language)

A pythonic way to generate SQL expressions programmatically without writing raw SQL strings.

```
from sqlalchemy import select, create_engine,
MetaData, Table, Column, Integer, String

engine = create_engine('sqlite:///example.db')
metadata = MetaData()
users = Table('users', metadata,
    Column('id', Integer, primary_key=True),
    Column('name', String),
    Column('email', String)
)

query = select(users).where(users.c.id > 5)
with engine.connect() as conn:
    result = conn.execute(query)
    for row in result:
        print(row)
```

ORM (Object Relational Mapper)

Maps database tables to Python classes, allowing you to work with objects rather than SQL.

```
from sqlalchemy import create_engine, Column,
Integer, String
from sqlalchemy.ext.declarative import
declarative_base
from sqlalchemy.orm import sessionmaker

Base = declarative_base()
engine = create_engine('sqlite:///example.db')

class User(Base):
 __tablename__ = 'users'
 id = Column(Integer, primary_key=True)
 name = Column(String)
 email = Column(String)

Session = sessionmaker(bind=engine)
session = Session()
users = session.query(User).filter(User.id > 5).all()
for user in users:
 print(user.name, user.email)
```

Database Best Practices

Connection Pooling

Reuse database connections instead of creating new ones for each operation. Most ORMs and many drivers support connection pooling.

```
from sqlalchemy import create_engine
engine =
create_engine('postgresql://user:pass@localhost/mydb',
                pool_size=5,
                max_overflow=10)
```

Proper Error Handling

Catch database-specific exceptions, not just generic ones. This allows more granular recovery strategies.

```
import psycopg2
try:
    # Database operation
except psycopg2.errors.UniqueViolation:
    # Handle duplicate key
except psycopg2.errors.ForeignKeyViolation:
    # Handle referential integrity issues
except psycopg2.Error as e:
    # Handle other database errors
```

Database Migrations

Use tools like Alembic (for SQLAlchemy) or Django Migrations to manage database schema changes over time.

```
# With Alembic
$ alembic revision --autogenerate -m "Add user table"
$ alembic upgrade head
```

Query Optimization

Monitor and optimize slow queries. Use EXPLAIN to understand query execution plans and add appropriate indexes.

Introduction to Web APIs in Python

Web APIs (Application Programming Interfaces) enable machines to communicate over HTTP. They're the backbone of modern web applications and microservices.

REST Architecture

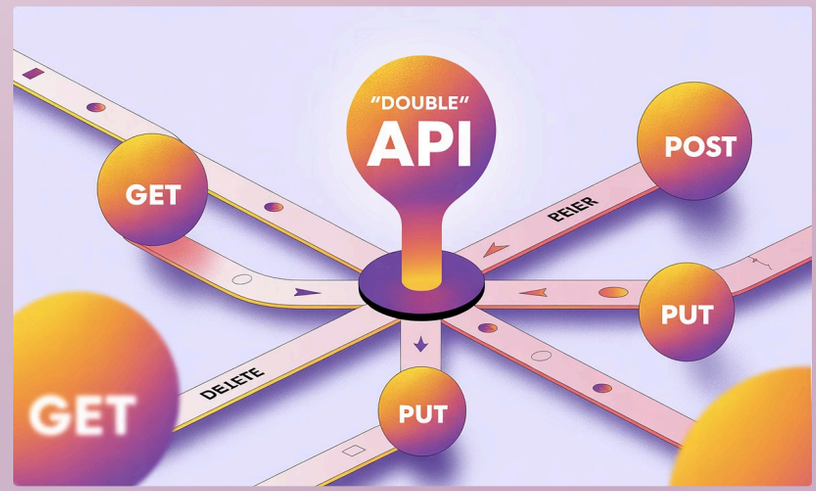
- Resources identified by URLs
- Standard HTTP methods (GET, POST, PUT, DELETE)
- Stateless communication
- Typically JSON or XML responses

Python Web Frameworks

- Flask: Lightweight microframework
- FastAPI: Modern, fast, async-capable
- Django REST Framework: Full-featured
- Falcon: Minimalist and high-performance

Key Concepts

- Endpoints/Routes
- Request/Response cycle
- Serialization (Python objects ↔️ JSON)
- Authentication/Authorization



Flask: Micro Web Framework

Flask is a lightweight WSGI web application framework. It's designed to make getting started quick and easy, with the ability to scale up to complex applications.

Flask is considered a microframework because it doesn't require particular tools or libraries. It has no database abstraction layer, form validation, or any other components where pre-existing third-party libraries provide common functions.

Installation:

```
pip install flask
```

Key Flask features:

- Built-in development server and debugger
- Integrated unit testing support
- RESTful request dispatching
- Jinja2 templating
- Support for secure cookies (client-side sessions)
- WSGI 1.0 compliant
- Unicode-based

Building a Basic Flask API

```
from flask import Flask, jsonify, request

# Create Flask application
app = Flask(__name__)

# In-memory data store (would be a database in production)
books = [
 {"id": 1, "title": "The Hobbit", "author": "J.R.R. Tolkien"},
 {"id": 2, "title": "Dune", "author": "Frank Herbert"}
]

# GET endpoint - Retrieve all books
@app.route('/api/books', methods=['GET'])
def get_books():
 return jsonify(books)

# GET endpoint - Retrieve a specific book
@app.route('/api/books/', methods=['GET'])
def get_book(book_id):
 book = next((b for b in books if b["id"] == book_id), None)
 if book:
  return jsonify(book)
  return jsonify({"error": "Book not found"}), 404

# POST endpoint - Create a new book
@app.route('/api/books', methods=['POST'])
def create_book():
 if not request.json or 'title' not in request.json:
  return jsonify({"error": "Invalid book data"}), 400

 new_book = {
 "id": max(b["id"] for b in books) + 1,
 "title": request.json["title"],
 "author": request.json.get("author", "Unknown")
 }
 books.append(new_book)
 return jsonify(new_book), 201

if __name__ == '__main__':
 app.run(debug=True)
```

Flask Routing and Request Handling

URL Routing

Flask uses decorators to bind functions to URLs:

```
@app.route('/path/', methods=['GET', 'POST'])
```

Variable parts can be typed:

- **string**: (default) accepts any text without a slash
- **int**: accepts positive integers
- **float**: accepts positive floating point values
- **path**: like string but also accepts slashes
- **uuid**: accepts UUID strings

Request Object

Access incoming request data:

```
from flask import request

@app.route('/search')
def search():
 query = request.args.get('q', '')
 return f"Searching for: {query}"

@app.route('/user', methods=['POST'])
def create_user():
 # Form data
 username = request.form.get('username')

 # JSON data
 data = request.json

 # Files
 if 'avatar' in request.files:
  file = request.files['avatar']
  file.save('/path/to/uploads/' + \
            file.filename)
```

Flask Response Types

JSON Responses

Most common for APIs, convert Python objects to JSON:

```
from flask import jsonify

@app.route('/api/user')
def get_user():
    user = {"name": "Alice", "email": "alice@example.com"}
    return jsonify(user)
```

HTML Responses

Using templates for web pages:

```
from flask import render_template

@app.route('/profile/')
def profile(username):
    user = get_user_from_database(username)
    return render_template('profile.html', user=user)
```

Custom Responses

Control status codes, headers, etc:

```
from flask import make_response

@app.route('/download')
def download():
    response = make_response("File content")
    response.headers['Content-Type'] = 'text/plain'
    response.headers['Content-Disposition'] = 'attachment;
filename=data.txt'
    return response
```

Error Handling

Define custom error handlers:

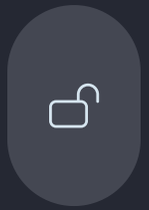
```
@app.errorhandler(404)
def not_found(error):
 return jsonify({"error": "Resource not found"}), 404
```

Flask Extensions

Flask's philosophy is to provide a simple core with the ability to extend functionality through extensions. Some popular extensions include:

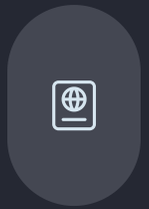
Flask-SQLAlchemy

Integrates SQLAlchemy ORM with Flask for simplified database access



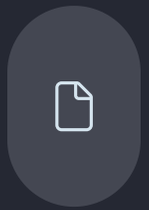
Flask-Login

User session management for authentication



Flask-JWT-Extended

JSON Web Token (JWT) authentication support



Flask-RESTful

Simplifies building REST APIs with resource-based routing

Flask-Migrate

Database migrations with Alembic

Introduction to FastAPI

FastAPI is a modern, high-performance web framework for building APIs with Python 3.6+ based on standard Python type hints.

Installation:

```
pip install fastapi uvicorn
```

FastAPI requires an ASGI server like Uvicorn or Hypercorn to run.

- Based on Starlette for web functionality
- Uses Pydantic for data validation
- Automatic API documentation with Swagger UI and ReDoc
- Built-in support for asynchronous code

Key benefits:

- Fast: On par with NodeJS and Go
- Intuitive: Great editor support with autocompletion
- Easy: Designed to be easy to use and learn
- Robust: Get production-ready code with automatic interactive documentation
- Standards-based: Based on OpenAPI and JSON Schema

Building a FastAPI Application

```
from fastapi import FastAPI, HTTPException, Path, Query
from pydantic import BaseModel
from typing import Optional, List

# Create FastAPI application
app = FastAPI(title="Book API", description="API for managing books", version="1.0.0")

# Define data model with validation
class Book(BaseModel):
 id: Optional[int] = None
 title: str
 author: str
 pages: Optional[int] = None
 class Config:
  schema_extra = {
   "example": {
    "title": "The Hitchhiker's Guide to the Galaxy",
    "author": "Douglas Adams",
    "pages": 224
   }
  }

# In-memory database
books_db = [
 Book(id=1, title="The Hobbit", author="J.R.R. Tolkien", pages=295),
 Book(id=2, title="Dune", author="Frank Herbert", pages=412)
]

# GET all books
@app.get("/books/", response_model=List[Book], summary="Get all books")
async def get_books(skip: int = 0, limit: int = 10):
 return books_db[skip : skip + limit]

# GET single book
@app.get("/books/{book_id}", response_model=Book, summary="Get a book by ID")
async def get_book(book_id: int = Path(..., title="The ID of the book to get", ge=1)):
 for book in books_db:
  if book.id == book_id:
   return book
  raise HTTPException(status_code=404, detail="Book not found")

# POST new book
@app.post("/books/", response_model=Book, status_code=201, summary="Create a new book")
async def create_book(book: Book):
 # Generate new ID
 book.id = max(b.id for b in books_db) + 1 if books_db else 1
 books_db.append(book)
 return book
```

FastAPI Path and Query Parameters

Path Parameters

Values extracted from the URL path:

```
@app.get("/items/{item_id}")
async def read_item(
    item_id: int = Path(
        ...,  # ... means required
        title="The ID of the item",
        description="Must be a positive integer",
        ge=1  # greater than or equal to 1
    )
):
    return {"item_id": item_id}
```

Query Parameters

Values extracted from the URL query string:

```
@app.get("/users/")
async def read_users(
 skip: int = Query(
 0, # default value
 title="Skip records",
 description="Number of records to skip",
 ge=0
 ),
 limit: int = Query(
 10,
 title="Limit records",
 description="Max number of records to return",
 le=100
 ),
 search: Optional[str] = Query(
 None,
 title="Search string",
 min_length=3,
 max_length=50
 )
):
 results = get_users_from_db(skip=skip, limit=limit)
 if search:
 results = filter_by_search(results, search)
 return results
```

FastAPI Request Body Validation

FastAPI uses Pydantic models to validate request bodies, providing automatic validation, serialization, and documentation:

```
from fastapi import FastAPI, HTTPException
from pydantic import BaseModel, Field, EmailStr, validator
from typing import Optional, List
from datetime import date

app = FastAPI()

class Address(BaseModel):
 street: str
 city: str
 post_code: str
 country: str

class User(BaseModel):
 username: str = Field(..., min_length=3, max_length=50)
 email: EmailStr
 full_name: Optional[str] = None
 birth_date: Optional[date] = None
 addresses: List[Address] = []

 @validator('username')
 def username_alphanumeric(cls, v):
  if not v.isalnum():
     raise ValueError('Username must be alphanumeric')
  return v

@app.post("/users/", response_model=User)
async def create_user(user: User):
 # FastAPI validates all input data against the User model
 # If validation fails, it returns a 422 Unprocessable Entity
 # with detailed error information

 # Store in database...
 return user
```

FastAPI Dependency Injection

FastAPI's dependency injection system helps implement shared logic, enforce security, and handle database connections:

```
from fastapi import FastAPI, Depends, HTTPException, status
from fastapi.security import OAuth2PasswordBearer
from sqlalchemy.orm import Session

app = FastAPI()

# Authentication dependency
oauth2_scheme = OAuth2PasswordBearer(tokenUrl="token")

def get_current_user(token: str = Depends(oauth2_scheme)):
    user = decode_token(token)
    if not user:
        raise HTTPException(
            status_code=status.HTTP_401_UNAUTHORIZED,
            detail="Invalid authentication credentials",
            headers={"WWW-Authenticate": "Bearer"},
        )
    return user

# Database dependency
def get_db():
    db = SessionLocal()
    try:
        yield db
    finally:
        db.close()

@app.get("/users/me")
async def read_users_me(
    current_user: User = Depends(get_current_user),
    db: Session = Depends(get_db)
):
    # Use db to query data
    # Use current_user for authorization
    return current_user
```

Dependencies can depend on other dependencies, creating a dependency graph that FastAPI resolves automatically.

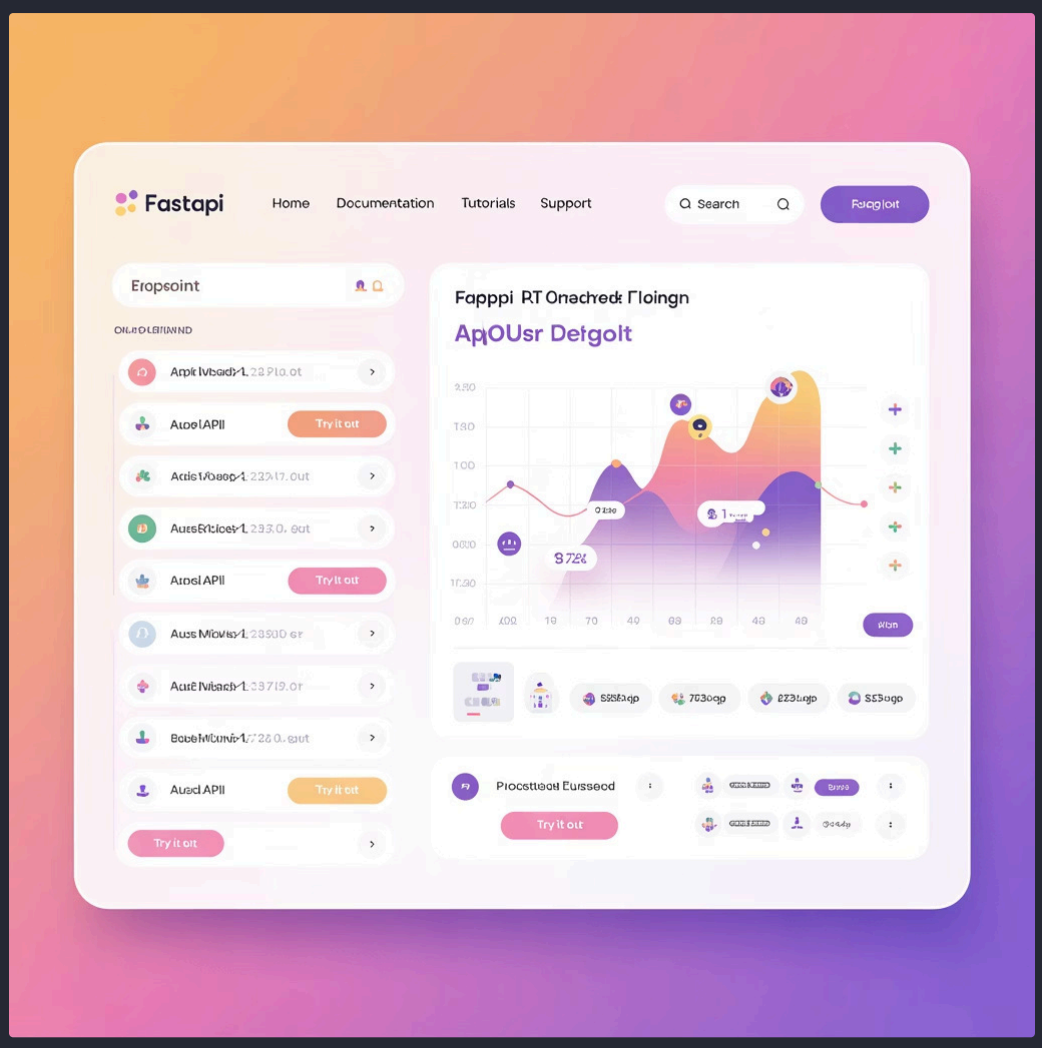
API Documentation with FastAPI

FastAPI automatically generates interactive API documentation based on your code, including:

- Your function parameters converted to API parameters
- Your Pydantic models converted to JSON Schema
- All validations documented
- Interactive API documentation UI

Two documentation interfaces are included:

- Swagger UI at /docs
- ReDoc at /redoc



Enhance your documentation with docstrings and metadata:

```
@app.get(
 "/items/{item_id}",
 summary="Get a specific item",
 description="Retrieve a specific item by its ID from the database.",
 response_description="The item details",
 responses={
 404: {"description": "Item not found"},
 200: {
 "content": {
 "application/json": {
 "example": {"id": 1, "name": "Example item"}
 }
 }
 }
 }
)
async def read_item(item_id: int):
 """
 Get an item by ID.

 This will fetch the item from the database.
 If the item doesn't exist, a 404 error will be returned.
 """
 # Function implementation
```

Web API Deployment Strategies

Development Server

For local testing only, not for production use

- Flask: `app.run(debug=True)`
- FastAPI: `uvicorn main:app --reload`

WSGI/ASGI Server

Production-grade application servers

- Gunicorn (WSGI): `gunicorn -w 4 -b 0.0.0.0:8000 app:app`
- Uvicorn (ASGI): `uvicorn app:app --host 0.0.0.0 --port 8000 --workers 4`

Reverse Proxy

Handles SSL, caching, static files

- Nginx: Industry standard reverse proxy
- Caddy: Simpler configuration with automatic HTTPS

Deployment Platform

Hosting environments

- Containerization: Docker, Kubernetes
- Cloud: AWS, GCP, Azure, Heroku, Render

NumPy: Numerical Python

NumPy is the fundamental package for scientific computing in Python. It provides:

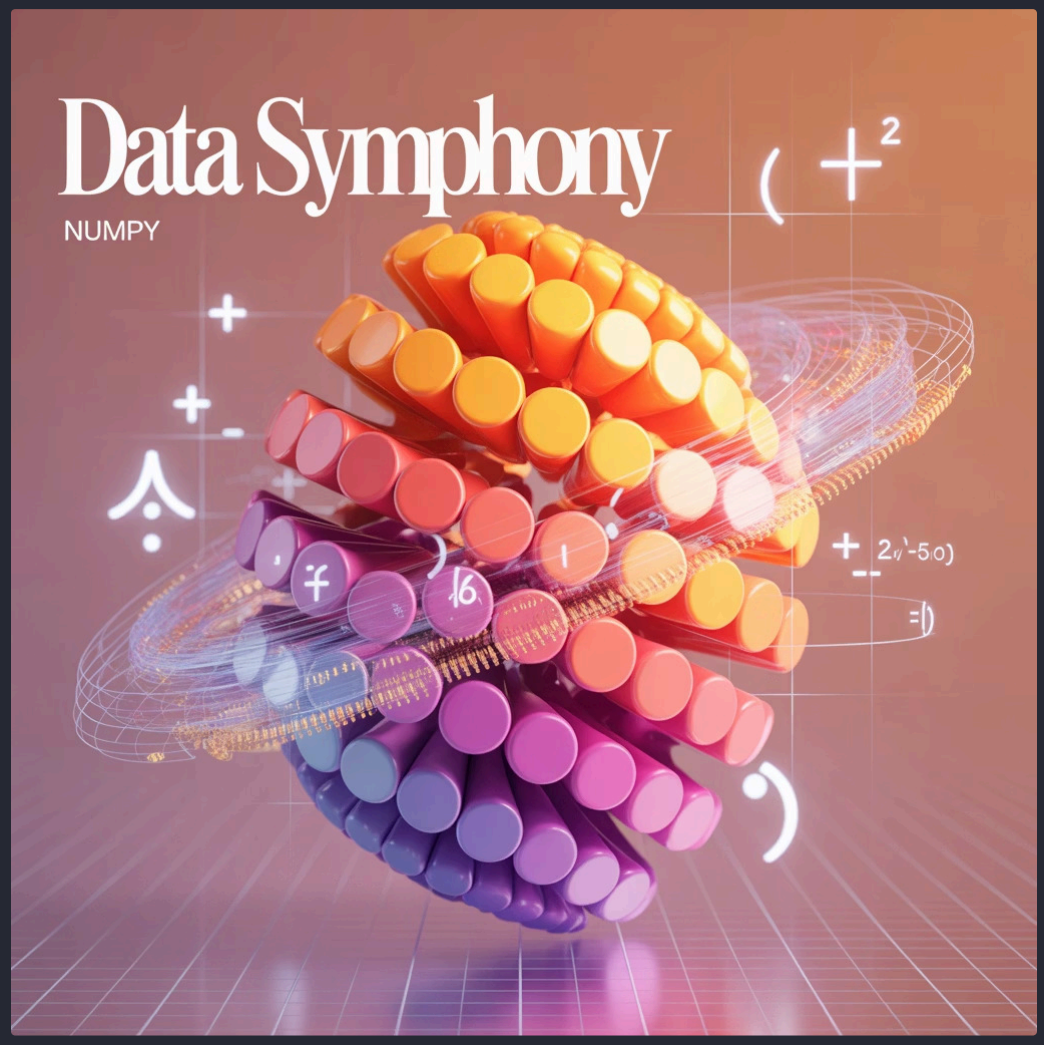
- A powerful N-dimensional array object
- Sophisticated broadcasting functions
- Tools for integrating C/C++ and Fortran code
- Linear algebra, Fourier transform, and random number capabilities

Installation:

```
pip install numpy
```

Import convention:

```
import numpy as np
```



Why use NumPy instead of Python lists?

- ## Performance

  NumPy operations are implemented in C, making them much faster than Python loops

- ## Memory Efficiency

  NumPy arrays are stored contiguously in memory unlike Python lists

- ## Convenience

  NumPy provides vectorized operations and a large library of mathematical functions

Creating NumPy Arrays

```
import numpy as np

# From Python lists
arr1 = np.array([1, 2, 3, 4, 5])
arr2 = np.array([[1, 2, 3], [4, 5, 6]]) # 2D array

# Array properties
print(arr2.shape) # Output: (2, 3)
print(arr2.ndim) # Output: 2
print(arr2.dtype) # Output: int64
print(arr2.size) # Output: 6

# Creating arrays with specific values
zeros = np.zeros((3, 4)) # 3x4 array of zeros
ones = np.ones((2, 3, 4)) # 2x3x4 array of ones
empty = np.empty((2, 3)) # Uninitialized array (random values)
full = np.full((2, 2), 7) # 2x2 array filled with 7s
eye = np.eye(3) # 3x3 identity matrix

# Creating sequences
arange = np.arange(0, 10, 2) # [0, 2, 4, 6, 8]
linspace = np.linspace(0, 1, 5) # 5 evenly spaced values from 0 to 1: [0, 0.25, 0.5, 0.75, 1]

# Creating arrays with random values
rand = np.random.rand(3, 2) # Uniform distribution [0, 1)
randn = np.random.randn(3, 2) # Standard normal distribution
randint = np.random.randint(1, 10, (3, 3)) # Random integers 1-9
```

NumPy Array Indexing and Slicing

Basic Indexing

```
import numpy as np

arr = np.array([1, 2, 3, 4, 5])
print(arr[0])    # 1
print(arr[-1])   # 5

arr2d = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
print(arr2d[0, 0])  # 1
print(arr2d[2, 1])  # 8
```

Slicing

```
print(arr[1:4])   # [2, 3, 4]
print(arr[::2])   # [1, 3, 5]

# Rows 0 and 1, columns 1 and 2
print(arr2d[0:2, 1:3])  # [[2, 3], [5, 6]]

# All rows, column 1
print(arr2d[:, 1])  # [2, 5, 8]
```

Boolean Indexing

```
arr = np.array([1, 2, 3, 4, 5])
mask = arr > 3
print(mask)  # [False, False, False, True, True]
print(arr[mask])  # [4, 5]

# One-liner
print(arr[arr > 3])  # [4, 5]

# Combined conditions
arr2d = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
print(arr2d[(arr2d > 2) & (arr2d < 8)])
# [3, 4, 5, 6, 7]
```

Fancy Indexing

```
arr = np.array([10, 20, 30, 40, 50])
indices = [0, 2, 4]
print(arr[indices]) # [10, 30, 50]

arr2d = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
print(arr2d[[0, 2], [1, 2]]) # [2, 9]
# Selects arr2d[0,1] and arr2d[2,2]
```

NumPy Vectorized Operations

One of NumPy's key features is vectorization—the ability to perform operations on entire arrays without explicit loops.

Arithmetic Operations

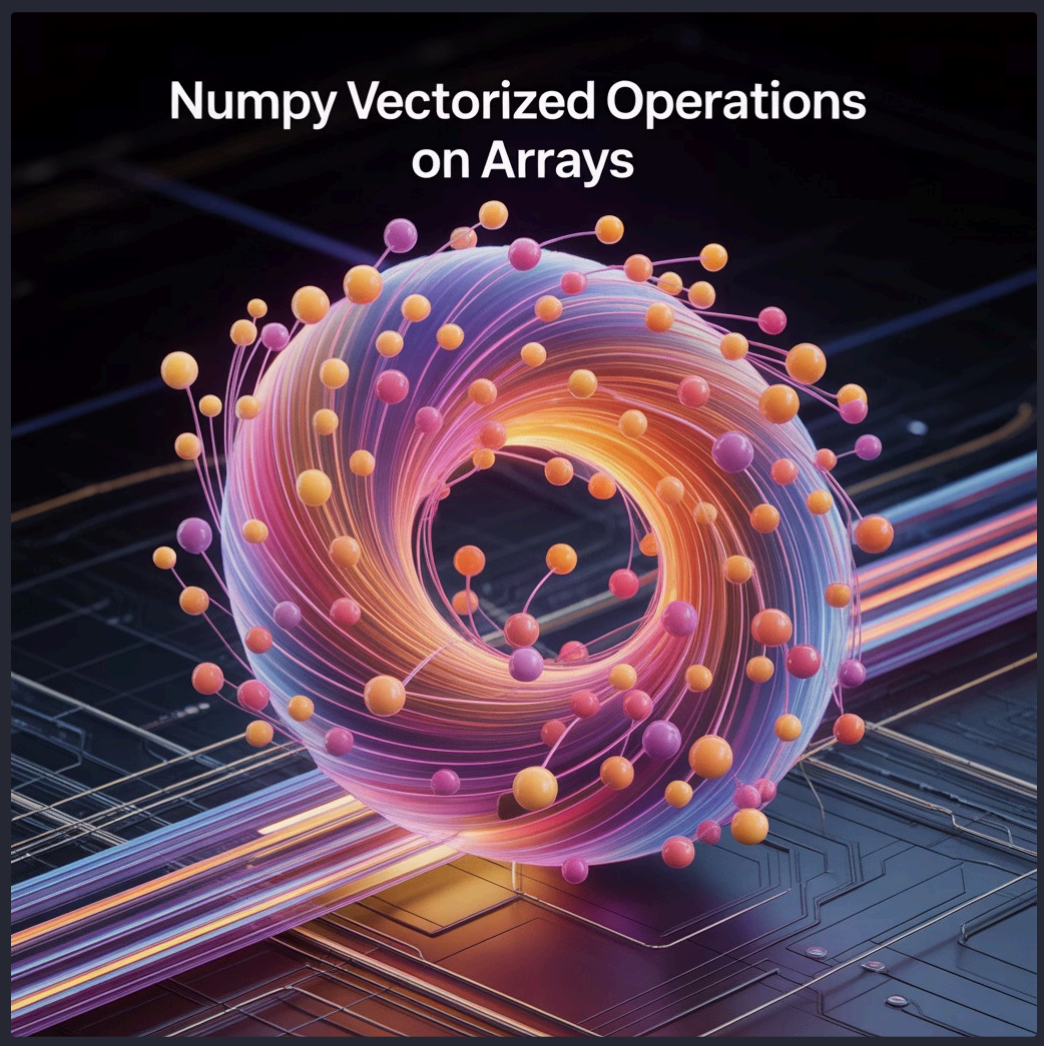
```
import numpy as np

arr1 = np.array([1, 2, 3, 4])
arr2 = np.array([10, 20, 30, 40])

# Element-wise operations
print(arr1 + arr2)    # [11, 22, 33, 44]
print(arr1 * arr2)    # [10, 40, 90, 160]
print(arr1 / arr2)    # [0.1, 0.1, 0.1, 0.1]
print(arr1 ** 2)      # [1, 4, 9, 16]
```

Comparison Operations

```
print(arr1 > 2)        # [False, False, True, True]
print(arr1 == arr2)    # [False, False, False, False]
print(np.array_equal(arr1, arr2))  # False
```

Mathematical Functions

```
# Universal functions (ufuncs)
print(np.sqrt(arr1))    # [1., 1.41421356, 1.73205081, 2.]
print(np.exp(arr1))     # [2.71828183, 7.3890561, 20.08553692,
54.59815003]
print(np.log(arr1))     # [0., 0.69314718, 1.09861229,
1.38629436]
print(np.sin(arr1))     # [0.84147098, 0.90929743, 0.14112001,
-0.7568025]
```



Vectorized operations are much faster than Python loops:

```
# Slow way (with loop)
result = []
for i in range(len(arr1)):
 result.append(arr1[i] * arr2[i])

# Fast way (vectorized)
result = arr1 * arr2
```

NumPy Broadcasting

Broadcasting allows NumPy to perform operations on arrays of different shapes. The smaller array is "broadcast" to match the shape of the larger array.

Broadcasting Rules

1. If arrays don't have the same number of dimensions, prepend the shape of the smaller array with 1s until both shapes have the same length
2. Two dimensions are compatible when:
   - They are equal, or
   - One of them is 1

Examples

```
import numpy as np

# Scalar and array
arr = np.array([1, 2, 3, 4])
print(arr + 10)  # [11, 12, 13, 14]

# 1D and 2D arrays
arr1d = np.array([1, 2, 3])
arr2d = np.array([[10], [20], [30]])
# arr1d shape: (3,)
# arr2d shape: (3, 1)
print(arr1d + arr2d)
# Result shape: (3, 3)
# [[11, 12, 13],
#  [21, 22, 23],
#  [31, 32, 33]]
```

Practical Uses

- Adding a constant to every element
- Adding a row vector to each row of a matrix
- Adding a column vector to each column of a matrix
- Normalizing data by subtracting mean and dividing by standard deviation

```
# Normalizing data
data = np.random.randn(5, 3)
means = data.mean(axis=0) # Column means
stds = data.std(axis=0) # Column standard deviations
normalized = (data - means) / stds
```

NumPy Array Reshaping and Transposing

Reshaping Arrays

Change array dimensions without changing the data:

```
import numpy as np

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

# Reshape to 3x4 matrix
arr_2d = arr.reshape(3, 4)
print(arr_2d)
# [[ 0  1  2  3]
#  [ 4  5  6  7]
#  [ 8  9 10 11]]

# Use -1 to automatically calculate one dimension
arr_2d_alt = arr.reshape(4, -1)  # 4 rows, automatically 3 columns
print(arr_2d_alt)
# [[ 0  1  2]
#  [ 3  4  5]
#  [ 6  7  8]
#  [ 9 10 11]]
```

Transposing Arrays

Swap rows and columns:

```
arr_2d = np.array([[1, 2, 3],
                   [4, 5, 6]])  # 2x3 matrix

# Transpose
arr_2d_T = arr_2d.T  # or np.transpose(arr_2d)
print(arr_2d_T)
# [[1 4]
#  [2 5]
#  [3 6]]  # Now 3x2 matrix
```

Flattening and Raveling

```
# Flatten returns a copy
flat_arr = arr_2d.flatten()
print(flat_arr) # [1 2 3 4 5 6]

# Ravel returns a view (when possible)
ravel_arr = arr_2d.ravel()
print(ravel_arr) # [1 2 3 4 5 6]

# Changing ravel_arr may change arr_2d
ravel_arr[0] = 99
print(arr_2d) # [[99 2 3], [4 5 6]]
```

NumPy Aggregate Functions

NumPy provides functions to compute statistics across entire arrays or along specific axes.

```
import numpy as np

arr2d = np.array([[1, 2, 3],
 [4, 5, 6],
 [7, 8, 9]])

# Statistics on the entire array
print(np.sum(arr2d)) # 45
print(np.mean(arr2d)) # 5.0
print(np.median(arr2d)) # 5.0
print(np.min(arr2d)) # 1
print(np.max(arr2d)) # 9
print(np.std(arr2d)) # Standard deviation ≈ 2.58
print(np.var(arr2d)) # Variance ≈ 6.67
print(np.prod(arr2d)) # Product of all elements = 362880

# Operations along axes
print(np.sum(arr2d, axis=0)) # Column sums: [12, 15, 18]
print(np.sum(arr2d, axis=1)) # Row sums: [6, 15, 24]

# Cumulative operations
print(np.cumsum(arr2d)) # Cumulative sum: [1, 3, 6, 10, 15, 21, 28, 36, 45]
print(np.cumprod(arr2d)) # Cumulative product: [1, 2, 6, 24, 120, 720, 5040, 40320, 362880]

# Finding indices
print(np.argmin(arr2d)) # Index of minimum value: 0
print(np.argmax(arr2d)) # Index of maximum value: 8
print(np.argmax(arr2d, axis=0)) # Index of max in each column: [2, 2, 2]
```

Pandas: Introduction

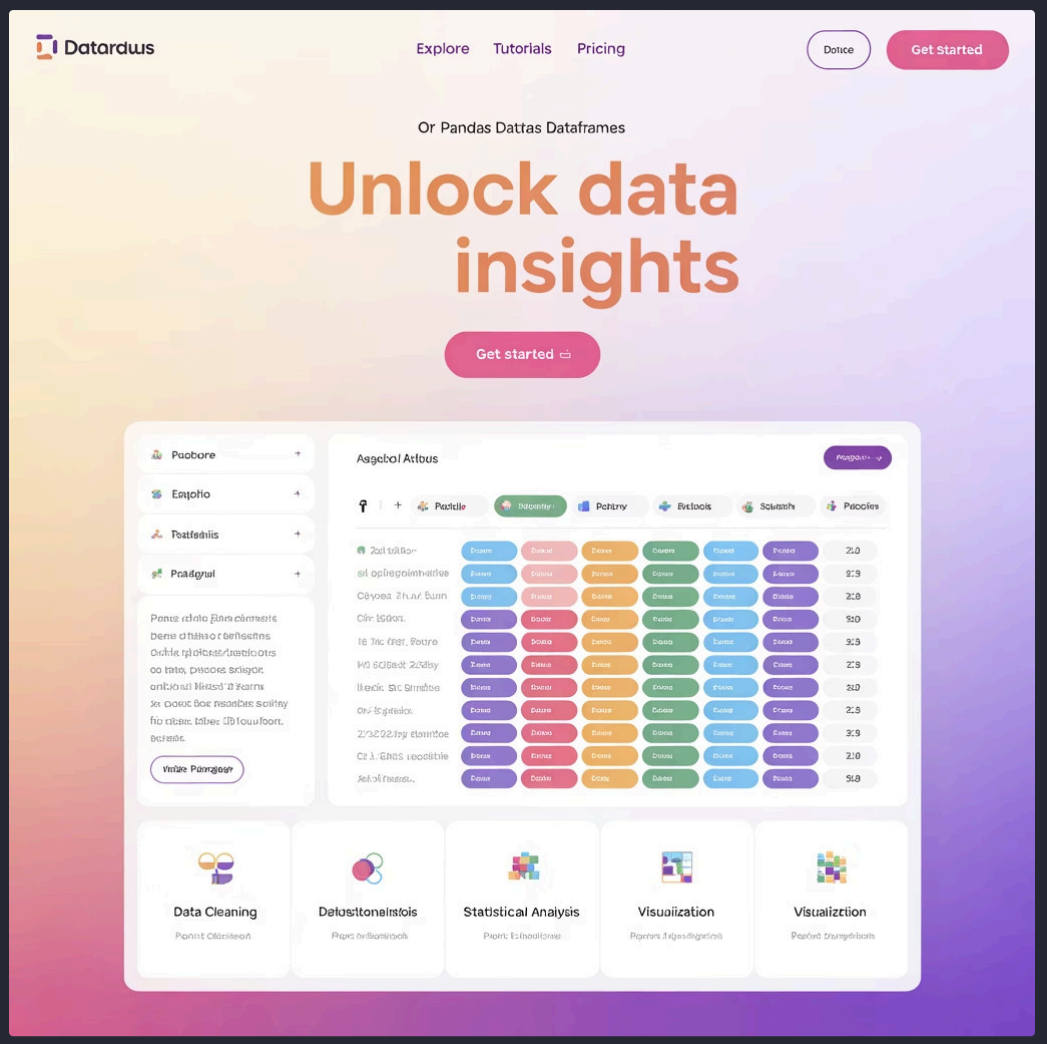
Pandas is a powerful Python data analysis toolkit built on NumPy. It provides data structures and functions needed to efficiently manipulate structured data.

Key features:

- DataFrame object for data manipulation with labeled axes
- Tools for reading and writing data between in-memory data structures and file formats
- Data alignment and integrated handling of missing data
- Reshaping and pivoting of datasets
- Intelligent label-based slicing, fancy indexing, and subsetting
- Group by functionality for aggregation and transformations

Installation:

```
pip install pandas
```



Import convention:

```
import pandas as pd
import numpy as np
```

Pandas is built on NumPy and is designed to work well with it and other libraries in the PyData ecosystem, including matplotlib, scikit-learn, and statsmodels.

Primary data structures:

- **Series**: 1D labeled homogeneously-typed array
- **DataFrame**: 2D labeled, size-mutable tabular structure with potentially heterogeneously-typed columns

Pandas Series

A Series is a one-dimensional labeled array capable of holding any data type. The axis labels are collectively called the index.

```
import pandas as pd
import numpy as np

# Creating a Series from a list
s = pd.Series([1, 3, 5, np.nan, 6, 8])
print(s)
# 0    1.0
# 1    3.0
# 2    5.0
# 3    NaN
# 4    6.0
# 5    8.0
# dtype: float64

# Creating a Series with custom index
s = pd.Series([1, 2, 3, 4], index=['a', 'b', 'c', 'd'])
print(s)
# a    1
# b    2
# c    3
# d    4
# dtype: int64
```

```
# Creating a Series from a dictionary
d = {'a': 1, 'b': 2, 'c': 3}
s = pd.Series(d)
print(s)
# a    1
# b    2
# c    3
# dtype: int64

# Series attributes
print(s.index)    # Index(['a', 'b', 'c'], dtype='object')
print(s.values)   # [1 2 3]
print(s.dtype)    # int64

# Series operations
print(s + 5)      # Add 5 to each element
print(s * 2)      # Multiply each element by 2
print(s[s > 1])   # Filter elements > 1

# NumPy functions work on Series
print(np.sqrt(s))  # Square root of each element
```

Series combine properties of both NumPy arrays and Python dictionaries, making them ideal for labeled data.

Pandas DataFrame

A DataFrame is a 2-dimensional labeled data structure with columns of potentially different types. It's like a spreadsheet or SQL table.

```
import pandas as pd
import numpy as np

# Creating a DataFrame from a dictionary of Series
d = {
    'Name': pd.Series(['Alice', 'Bob', 'Charlie']),
    'Age': pd.Series([25, 30, 35]),
    'Rating': pd.Series([4.5, 3.2, 4.8])
}
df = pd.DataFrame(d)
print(df)
#       Name  Age  Rating
# 0    Alice   25     4.5
# 1      Bob   30     3.2
# 2  Charlie   35     4.8
```

A DataFrame has both row and column indices, providing a convenient way to access and manipulate data by labels.

```
# Creating a DataFrame from a dictionary of lists
data = {
 'Name': ['Alice', 'Bob', 'Charlie', 'David'],
 'Age': [25, 30, 35, 40],
 'City': ['New York', 'Paris', 'London', 'Tokyo']
}
df = pd.DataFrame(data)
print(df)
# Name Age City
# 0 Alice 25 New York
# 1 Bob 30 Paris
# 2 Charlie 35 London
# 3 David 40 Tokyo

# DataFrame attributes and methods
print(df.shape) # (4, 3) - 4 rows, 3 columns
print(df.dtypes) # Data types of each column
print(df.columns) # Index(['Name', 'Age', 'City'], dtype='object')
print(df.index) # RangeIndex(start=0, stop=4, step=1)
print(df.head(2)) # First 2 rows
print(df.tail(2)) # Last 2 rows
print(df.describe()) # Statistical summary of numeric columns
```

Creating DataFrames

There are multiple ways to create DataFrames depending on your data source:

```
import pandas as pd
import numpy as np

# From a NumPy array
arr = np.random.randn(3, 3)
df1 = pd.DataFrame(arr, columns=['A', 'B', 'C'])

# From a list of dictionaries
data = [
 {'name': 'Alice', 'age': 25, 'city': 'London'},
 {'name': 'Bob', 'age': 30, 'city': 'Paris'},
 {'name': 'Charlie', 'age': 35} # Note missing 'city'
]
df2 = pd.DataFrame(data)

# From a dictionary of lists
data = {
 'name': ['Alice', 'Bob', 'Charlie'],
 'age': [25, 30, 35],
 'city': ['London', 'Paris', None] # Note None value
}
df3 = pd.DataFrame(data)

# With explicit indices
df4 = pd.DataFrame(data, index=['person1', 'person2', 'person3'])

# From a CSV file
df5 = pd.read_csv('data.csv')

# From a SQL query
import sqlite3
conn = sqlite3.connect('database.db')
df6 = pd.read_sql_query("SELECT * FROM users", conn)

# From Excel
df7 = pd.read_excel('data.xlsx', sheet_name='Sheet1')
```

Pandas Data Input/Output

CSV Files

```
# Reading
df = pd.read_csv('data.csv',
                 index_col=0,    # Use first column as index
                 parse_dates=['Date'],  # Parse date columns
                 encoding='utf-8')      # Specify encoding

# Writing
df.to_csv('output.csv',
          index=False,  # Don't write row indices
          float_format='%.2f')  # Format floats
```

Excel Files

```
# Reading
df = pd.read_excel('data.xlsx',
                   sheet_name='Sheet1',
                   header=1)  # Use row 1 as header

# Writing
df.to_excel('output.xlsx',
            sheet_name='Data',
            engine='openpyxl',
            freeze_panes=(1, 0))  # Freeze header row
```

SQL Databases

```
# Reading (SQLite example)
import sqlite3
conn = sqlite3.connect('database.db')
df = pd.read_sql('SELECT * FROM users', conn)

# Writing
df.to_sql('users',
          conn,
          if_exists='replace',  # 'fail', 'replace', 'append'
          index=False)
```

JSON/HTML/Parquet

```
# JSON
df = pd.read_json('data.json')
df.to_json('output.json', orient='records')

# HTML
df = pd.read_html('table.html')[0] # Returns list of
DataFrames
df.to_html('output.html')

# Parquet (requires pyarrow or fastparquet)
df = pd.read_parquet('data.parquet')
df.to_parquet('output.parquet', compression='snappy')
```

Pandas: Indexing and Selection

Selecting Columns

```
import pandas as pd

df = pd.DataFrame({
    'name': ['Alice', 'Bob', 'Charlie'],
    'age': [25, 30, 35],
    'city': ['London', 'Paris', 'Berlin']
})

# Single column (returns Series)
print(df['name'])
print(df.name)  # Attribute access (if column name is valid identifier)

# Multiple columns (returns DataFrame)
print(df[['name', 'age']])
```

Selecting Rows by Label (loc)

```
df = df.set_index('name')  # Set 'name' as index

# Single row by label
print(df.loc['Alice'])

# Multiple rows by label
print(df.loc[['Alice', 'Charlie']])

# Row and column selection
print(df.loc['Bob', 'city'])  # Single value
print(df.loc['Alice':'Charlie', 'age':'city'])  # Slice
```

Selecting Rows by Position (iloc)

```
# Single row by position
print(df.iloc[0])

# Multiple rows by position
print(df.iloc[[0, 2]])

# Row and column by position
print(df.iloc[1, 1])  # Second row, second column
print(df.iloc[0:2, 1:3])  # First two rows, second and third columns
```

Boolean Indexing

```
# Create a boolean mask
mask = df['age'] > 28
print(mask)
# 0 False
# 1 True
# 2 True

# Filter rows using the mask
print(df[mask])
# or more directly:
print(df[df['age'] > 28])

# Multiple conditions
print(df[(df['age'] > 28) & (df['city'] == 'Paris')])
print(df[(df['age'] < 30) | (df['city'] == 'Berlin')])
```

Pandas: Assigning Data

Adding New Columns

```
import pandas as pd

df = pd.DataFrame({
    'name': ['Alice', 'Bob', 'Charlie'],
    'age': [25, 30, 35]
})

# Add a single new column
df['height'] = [165, 180, 175]
print(df)
#       name  age  height
# 0    Alice   25     165
# 1      Bob   30     180
# 2  Charlie   35     175

# Add a column based on existing columns
df['age_months'] = df['age'] * 12
print(df)

# Add a column with scalar value
df['status'] = 'active'
print(df)
```

Modifying Existing Data

```
# Modify a single value
df.loc[1, 'age'] = 31
print(df)

# Modify based on condition
df.loc[df['age'] > 30, 'status'] = 'senior'
print(df)

# Modify multiple columns
df.loc[0, ['age', 'height']] = [26, 166]
print(df)

# Using assign method (returns new DataFrame)
df2 = df.assign(
    age_years = lambda x: x['age'],
    height_m = lambda x: x['height'] / 100
)
print(df2)
```

The `assign()` method creates a new DataFrame without modifying the original, which is useful for method chaining.

Pandas: Data Cleaning and Preparation

Handling Missing Data

```
import pandas as pd
import numpy as np

df = pd.DataFrame({
    'A': [1, 2, np.nan, 4],
    'B': [5, np.nan, np.nan, 8],
    'C': [9, 10, 11, 12]
})

# Check for missing values
print(df.isna())        # Boolean mask for NaN values
print(df.isna().sum())   # Count NaN values in each column

# Drop rows with any missing values
print(df.dropna())

# Drop rows where all values are NaN
print(df.dropna(how='all'))

# Drop columns with at least 2 NaN values
print(df.dropna(axis=1, thresh=3))

# Fill missing values
print(df.fillna(0))              # Fill with specific value
print(df.fillna(df.mean()))         # Fill with column means
print(df.fillna(method='ffill'))     # Forward fill (propagate last valid value)
print(df.fillna(method='bfill'))     # Back fill (use next valid value)
```

Removing Duplicates

```
df = pd.DataFrame({
 'A': [1, 1, 2, 3],
 'B': [1, 1, 2, 3]
})

# Check for duplicates
print(df.duplicated()) # Boolean Series marking duplicates
print(df.duplicated().sum()) # Count of duplicates

# Drop duplicate rows
print(df.drop_duplicates())

# Drop duplicates based only on column 'A'
print(df.drop_duplicates(subset=['A']))

# Keep last occurrence instead of first
print(df.drop_duplicates(keep='last'))
```

Pandas: Data Transformations

Data Type Conversion

```
import pandas as pd

df = pd.DataFrame({
    'A': ['1', '2', '3'],
    'B': [4, 5, 6]
})

# Check data types
print(df.dtypes)

# Convert column to numeric
df['A'] = pd.to_numeric(df['A'])

# Convert entire DataFrame
df = df.astype({'A': 'int64', 'B': 'float64'})

# Convert to datetime
df['date'] = pd.to_datetime(['2023-01-01', '2023-01-02',
'2023-01-03'])
```

Mapping Values

```
# Replace specific values
df['A'] = df['A'].replace({1: 100, 2: 200})

# Map values using dictionary
status_map = {1: 'Active', 2: 'Inactive', 3: 'Pending'}
df['status'] = df['A'].map(status_map)

# Apply custom function
def double(x):
    return x * 2

df['B_doubled'] = df['B'].apply(double)

# Apply function to multiple columns
df[['A', 'B']] = df[['A', 'B']].apply(lambda x: x + 1)
```

Sorting and Ranking

```
# Sort by values
df_sorted = df.sort_values(by='A')
df_sorted = df.sort_values(by=['A', 'B'],
                           ascending=[True, False])

# Sort by index
df_sorted = df.sort_index()

# Rank values
df['A_rank'] = df['A'].rank()  # Default: average
df['B_rank'] = df['B'].rank(method='dense')  # No gaps
```

Random Sampling

```
# Random sample of rows
sample = df.sample(n=2) # 2 random rows
sample = df.sample(frac=0.5) # 50% of data
sample = df.sample(n=2, random_state=42) # Reproducible
```

Pandas Summary Functions

Descriptive Statistics

```
import pandas as pd
import numpy as np

df = pd.DataFrame({
    'A': [1, 2, 3, 4, 5],
    'B': [10, 20, 30, 40, 50],
    'C': ['a', 'b', 'c', 'd', 'e']
})

# Summary statistics for numeric columns
print(df.describe())

# Include all columns (categorical too)
print(df.describe(include='all'))

# Individual statistics
print(df.mean())      # Mean of each numeric column
print(df.median())    # Median of each numeric column
print(df.min())       # Minimum of each column
print(df.max())       # Maximum of each column
print(df.std())       # Standard deviation
print(df.var())       # Variance
print(df.count())     # Count of non-NA values
```

Aggregate Functions

```
# Sum values
print(df.sum())        # Sum of each column
print(df.sum(axis=1))   # Sum of each row (for numeric)

# Custom aggregation
print(df.agg(['min', 'max', 'mean', 'median']))

# Different aggregations per column
print(df.agg({
    'A': ['min', 'max', 'mean'],
    'B': ['sum', 'std']
}))
```

Value Counts and Unique Values

```
# Count of unique values
print(df['C'].value_counts())

# List of unique values
print(df['C'].unique())

# Number of unique values
print(df['C'].nunique())
```

Pandas: apply(), map(), and applymap()

Series.map()

Maps values of a Series according to an input dictionary or function:

```
import pandas as pd

# Create a Series
s = pd.Series(['A', 'B', 'C', 'A', 'B'])

# Using a dictionary
mapping = {'A': 1, 'B': 2, 'C': 3}
print(s.map(mapping))
# 0    1
# 1    2
# 2    3
# 3    1
# 4    2

# Using a function
print(s.map(lambda x: x.lower()))
# 0    a
# 1    b
# 2    c
# 3    a
# 4    b

# NaN for values not in the mapping
print(s.map({'A': 1, 'B': 2}))
# 0    1
# 1    2
# 2  NaN
# 3    1
# 4    2
```

DataFrame.apply()

Apply a function along an axis (row or column):

```
df = pd.DataFrame({
    'A': [1, 2, 3],
    'B': [4, 5, 6],
    'C': [7, 8, 9]
})

# Apply to each column (default axis=0)
print(df.apply(sum))
# A     6
# B    15
# C    24

# Apply to each row
print(df.apply(sum, axis=1))
# 0    12
# 1    15
# 2    18

# Custom function returning multiple values
def stats(x):
    return pd.Series([x.min(), x.max(), x.mean()],
                     index=['min', 'max', 'mean'])

print(df.apply(stats))
```

DataFrame.applymap()

Apply a function to every element:

```
print(df.applymap(lambda x: x**2))
```

Pandas: Grouping Data

GroupBy operations involve splitting data, applying a function, and combining results—a common pattern for data analysis.

```
import pandas as pd
import numpy as np

# Sample data: sales by region and product
df = pd.DataFrame({
 'region': ['North', 'South', 'East', 'West', 'North', 'South', 'East', 'West'],
 'product': ['A', 'A', 'A', 'A', 'B', 'B', 'B', 'B'],
 'sales': [100, 80, 90, 110, 120, 95, 105, 130],
 'units': [10, 8, 9, 11, 8, 7, 9, 12]
})

# Group by one column
by_region = df.groupby('region')

# Basic aggregations
print(by_region.mean()) # Mean of numeric columns for each region
print(by_region.sum()) # Sum of numeric columns for each region
print(by_region.size()) # Count of rows in each group
print(by_region.count()) # Count of non-NA values in each group

# Group by multiple columns
by_region_product = df.groupby(['region', 'product'])
print(by_region_product.sum())

# Custom aggregations
print(by_region.agg({'sales': 'sum', 'units': 'mean'}))

# Multiple aggregations per column
print(by_region.agg({
 'sales': ['sum', 'mean', 'std'],
 'units': ['min', 'max', 'count']
}))

# Named aggregations (Pandas 0.25+)
print(df.groupby('region').agg(
 total_sales=('sales', 'sum'),
 avg_sales=('sales', 'mean'),
 total_units=('units', 'sum')
))
```

Pandas: Pivot Tables and Cross-Tabulation

Pivot Tables

Create a spreadsheet-style pivot table from DataFrame data:

```
import pandas as pd
import numpy as np

# Sample data
df = pd.DataFrame({
    'date': pd.date_range('2023-01-01', periods=8),
    'region': ['North', 'South', 'East', 'West', 'North', 'South',
'East', 'West'],
    'product': ['A', 'A', 'A', 'A', 'B', 'B', 'B', 'B'],
    'sales': [100, 80, 90, 110, 120, 95, 105, 130],
    'units': [10, 8, 9, 11, 8, 7, 9, 12]
})

# Basic pivot table
pivot = pd.pivot_table(
    df,
    values='sales',        # Values to aggregate
    index='region',        # Rows
    columns='product',     # Columns
    aggfunc='sum'          # Aggregation function
)
print(pivot)
```

```
# Multiple values and aggregation functions
pivot = pd.pivot_table(
    df,
    values=['sales', 'units'],
    index=['region'],
    columns=['product'],
    aggfunc={'sales': 'sum', 'units': 'mean'},
    fill_value=0,          # Replace NaN
    margins=True           # Include row/column totals
)
print(pivot)
```

Cross-Tabulation

Compute a simple cross-tabulation of two factors:

```
# Cross-tabulation
ct = pd.crosstab(
    df['region'],
    df['product'],
    values=df['sales'],
    aggfunc='sum',
    normalize=True  # Show proportions instead of counts
)
print(ct)
```

Cross-tabulation is a special case of pivot tables focused on counting occurrences of factor combinations.

Pandas: Merging and Joining DataFrames

Merge (SQL-style Joins)

```
import pandas as pd

# Sample DataFrames
df1 = pd.DataFrame({
    'key': ['A', 'B', 'C', 'D'],
    'value1': [1, 2, 3, 4]
})

df2 = pd.DataFrame({
    'key': ['B', 'D', 'E', 'F'],
    'value2': [5, 6, 7, 8]
})

# Inner join (only matching keys)
inner = pd.merge(df1, df2, on='key',
how='inner')
print(inner)
#   key  value1  value2
# 0   B       2       5
# 1   D       4       6

# Left join (all from left, matching
from right)
left = pd.merge(df1, df2, on='key',
how='left')
print(left)
#   key  value1  value2
# 0   A       1     NaN
# 1   B       2     5.0
# 2   C       3     NaN
# 3   D       4     6.0
```

Right and Outer Joins

```
# Right join (all from right,
matching from left)
right = pd.merge(df1, df2, on='key',
how='right')
print(right)
#   key  value1  value2
# 0   B     2.0       5
# 1   D     4.0       6
# 2   E     NaN       7
# 3   F     NaN       8

# Outer join (all rows from both)
outer = pd.merge(df1, df2, on='key',
how='outer')
print(outer)
#   key  value1  value2
# 0   A     1.0     NaN
# 1   B     2.0     5.0
# 2   C     3.0     NaN
# 3   D     4.0     6.0
# 4   E     NaN     7.0
# 5   F     NaN     8.0
```

Concatenation

```
# Vertical concatenation (stack)
df3 = pd.DataFrame({
 'key': ['G', 'H'],
 'value1': [9, 10]
})

vertical = pd.concat([df1, df3])
print(vertical)
# key value1
# 0 A 1
# 1 B 2
# 2 C 3
# 3 D 4
# 0 G 9
# 1 H 10

# Horizontal concatenation
(column-wise)
df4 = pd.DataFrame({
 'value3': [11, 12, 13, 14]
})

horizontal = pd.concat([df1, df4],
axis=1)
print(horizontal)
# key value1 value3
# 0 A 1 11
# 1 B 2 12
# 2 C 3 13
# 3 D 4 14
```

Pandas: Time Series Analysis

Pandas has extensive capabilities for working with time series data:

```
import pandas as pd
import numpy as np

# Create a date range
dates = pd.date_range('2023-01-01', periods=6)
print(dates)
# DatetimeIndex(['2023-01-01', '2023-01-02', '2023-01-03',
#                '2023-01-04', '2023-01-05', '2023-01-06'],
#               dtype='datetime64[ns]', freq='D')

# Create a time series
ts = pd.Series(np.random.randn(6), index=dates)
print(ts)
# 2023-01-01    0.557323
# 2023-01-02   -0.431163
# ...

# Date ranges with different frequencies
monthly = pd.date_range('2023-01-01', '2023-12-31', freq='M')
print(monthly)  # Month end dates

business_days = pd.date_range('2023-01-01', '2023-01-31',
                              freq='B')  # Business days
print(business_days)
```

Time Series Operations

```
# Resampling (changing frequency)
daily = pd.Series(np.random.randn(30),
                  index=pd.date_range('2023-01-01', periods=30))

# Downsample to weekly frequency
weekly = daily.resample('W').mean()
print(weekly)

# Upsample to hourly and forward-fill
hourly = daily.resample('H').ffill()

# Time shifting
print(daily.shift(1))  # Shift values forward 1 day
print(daily.shift(-1))  # Shift values backward 1 day

# Date/time components
df = pd.DataFrame(daily)
df['year'] = df.index.year
df['month'] = df.index.month
df['day'] = df.index.day
df['weekday'] = df.index.weekday
print(df.head())
```

Time series capabilities are crucial for financial data, sensor readings, event logs, and other temporal data.

Pandas: Text Data Operations

Pandas provides vectorized string operations through the `str` accessor:

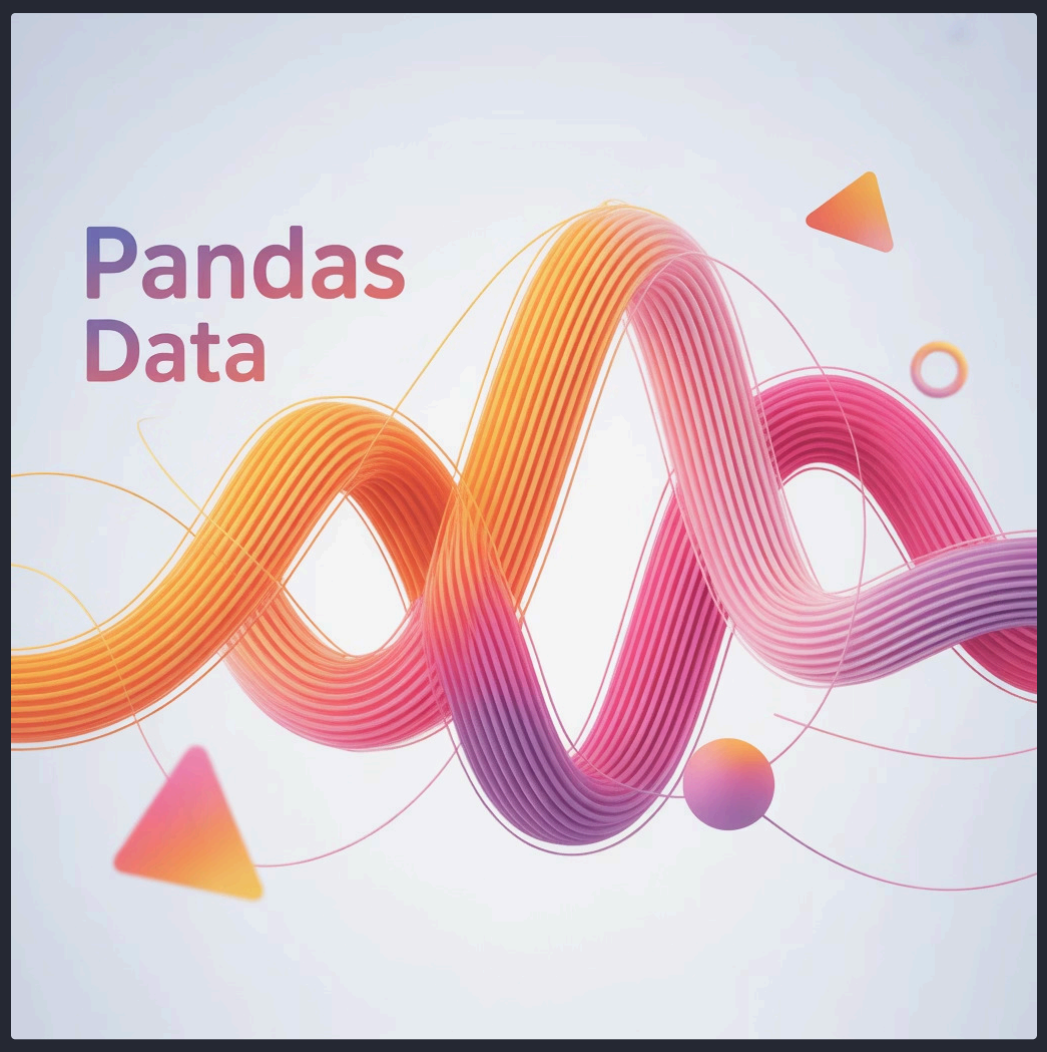
```
import pandas as pd

# Sample text data
df = pd.DataFrame({
    'text': ['Python 3.9', 'pandas 1.3.4', 'NumPy 1.21.0', 'SciPy 1.7.1'],
    'tags': ['language,programming', 'data,analysis', 'numerical,computing', 'scientific,computing'],
    'url': ['http://python.org', 'https://pandas.pydata.org', 'https://numpy.org', 'https://scipy.org']
})

# String operations
print(df['text'].str.lower())             # Lowercase
print(df['text'].str.upper())             # Uppercase
print(df['text'].str.len())               # String length
print(df['text'].str.replace(' ', '_'))   # Replace
print(df['text'].str.contains('py'))      # Check if contains substring
print(df['text'].str.startswith('Python'))  # Check if starts with
print(df['text'].str.endswith('.0'))      # Check if ends with

# Extracting patterns
print(df['text'].str.extract(r'(\d+\.\d+)'))  # Extract version numbers
print(df['text'].str.extract(r'(\w+) (\d+\.\d+)'))  # Extract name and version

# Splitting
print(df['tags'].str.split(','))          # Split into lists
print(df['tags'].str.split(',', expand=True))  # Split into columns

# URL parsing
print(df['url'].str.extract(r'https?://([^/]+)'))  # Extract domain
print(df['url'].str.count('/'))
```

These string methods are vectorized and apply to each element in the Series, making them much faster than iterating through the Series with a Python loop.

Integrating Pandas with Matplotlib

Pandas integrates well with Matplotlib for plotting data directly from DataFrames:

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt

# Sample data
dates = pd.date_range('2023-01-01', periods=30)
df = pd.DataFrame({
    'A': np.random.randn(30).cumsum(),
    'B': np.random.randn(30).cumsum(),
    'C': np.random.randn(30).cumsum()
}, index=dates)

# Set up Matplotlib for better display in notebooks
plt.rcParams['figure.figsize'] = (10, 6)
plt.rcParams['figure.dpi'] = 100

# Basic line plot
df.plot()
plt.title('Cumulative Random Walk')
plt.xlabel('Date')
plt.ylabel('Value')
plt.grid(True)
plt.savefig('line_plot.png')
plt.close()
```



Different plot types:

```
# Bar chart
df.iloc[::3].plot.bar()  # Sample every 3rd row
plt.title('Bar Chart')
plt.savefig('bar_chart.png')
plt.close()

# Histogram
df['A'].plot.hist(bins=15, alpha=0.7)
plt.title('Histogram')
plt.savefig('histogram.png')
plt.close()

# Scatter plot
df.plot.scatter(x='A', y='B', c='C', cmap='viridis', s=50)
plt.title('Scatter Plot')
plt.savefig('scatter.png')
plt.close()

# Box plot
df.plot.box()
plt.title('Box Plot')
plt.savefig('box.png')
plt.close()
```

Pandas uses Matplotlib internally but provides a simpler interface for common plots through the `plot` accessor.

Real-World Case Study: Combining Database, API, and Analysis

Let's build a simple application that demonstrates how to use databases, APIs, and data analysis together:

```
import sqlite3
import pandas as pd
import numpy as np
import requests
from flask import Flask, jsonify, request
from datetime import datetime, timedelta

# 1. Database Setup
conn = sqlite3.connect('sales.db', check_same_thread=False)
cursor = conn.cursor()

# Create table
cursor.execute('''
CREATE TABLE IF NOT EXISTS sales (
 id INTEGER PRIMARY KEY,
 product_id INTEGER,
 quantity INTEGER,
 price REAL,
 sale_date TEXT,
 region TEXT
)
''')

# Sample data
if not pd.read_sql("SELECT * FROM sales LIMIT 1", conn).shape[0]:
 # Generate some sample data
 products = range(1, 6) # 5 products
 regions = ['North', 'South', 'East', 'West']
 start_date = datetime.now() - timedelta(days=90)

 sales_data = []
 for i in range(1000): # 1000 sales records
  product_id = np.random.choice(products)
  quantity = np.random.randint(1, 10)
  price = np.round(np.random.uniform(10, 100), 2)
  days_ago = np.random.randint(0, 90)
  sale_date = (start_date + timedelta(days=days_ago)).strftime('%Y-%m-%d')
  region = np.random.choice(regions)

  sales_data.append((product_id, quantity, price, sale_date, region))

 cursor.executemany(
  "INSERT INTO sales (product_id, quantity, price, sale_date, region) VALUES (?, ?, ?, ?, ?)",
  sales_data
 )
 conn.commit()

# 2. Create Flask API
app = Flask(__name__)

@app.route('/api/sales', methods=['GET'])
def get_sales():
 region = request.args.get('region', None)
 start_date = request.args.get('start_date', None)

 query = "SELECT * FROM sales"
 params = []

 if region or start_date:
  query += " WHERE "
  conditions = []

 if region:
  conditions.append("region = ?")
  params.append(region)

 if start_date:
  conditions.append("sale_date >= ?")
  params.append(start_date)

 query += " AND ".join(conditions)

 df = pd.read_sql(query, conn, params=params)

 # Calculate total revenue
 df['revenue'] = df['quantity'] * df['price']

 return jsonify({
  'records': df.to_dict(orient='records'),
  'summary': {
   'total_sales': len(df),
   'total_revenue': float(df['revenue'].sum()),
   'avg_order_value': float(df['revenue'].mean())
  }
 })

@app.route('/api/analysis', methods=['GET'])
def get_analysis():
 # Read all sales data
 df = pd.read_sql("SELECT * FROM sales", conn)
 df['revenue'] = df['quantity'] * df['price']
 df['sale_date'] = pd.to_datetime(df['sale_date'])

 # Weekly sales trend
 weekly = df.resample('W', on='sale_date')['revenue'].sum().reset_index()
 weekly_data = weekly.to_dict(orient='records')

 # Sales by region
 region_data = df.groupby('region')['revenue'].sum().reset_index()
 region_dict = region_data.to_dict(orient='records')

 # Product performance
 product_data = df.groupby('product_id').agg({
  'quantity': 'sum',
  'revenue': 'sum'
 }).reset_index()
 product_dict = product_data.to_dict(orient='records')

 return jsonify({
  'weekly_trend': weekly_data,
  'by_region': region_dict,
  'by_product': product_dict
 })

# Run the app
if __name__ == '__main__':
 app.run(debug=True)
```

Key Takeaways

Database Connectivity

- Use parameterised queries to prevent SQL injection
- Wrap database operations in transactions for data consistency
- Consider SQLAlchemy for complex applications requiring an ORM
- Use context managers to ensure connections are properly closed

Web APIs

- Flask provides a lightweight framework for building REST APIs
- FastAPI offers modern features like automatic validation and documentation
- Use WSGI/ASGI servers and reverse proxies for production deployment
- Structure APIs around resources and follow REST conventions

NumPy

- Provides high-performance multi-dimensional arrays
- Vectorized operations eliminate explicit loops for better performance
- Broadcasting enables operations on arrays of different shapes
- Rich set of mathematical functions for numerical computing

Pandas

- Powerful data structures for working with tabular and time series data
- Comprehensive tools for data cleaning, transformation and analysis
- GroupBy operations for aggregating and summarising data
- Integration with various file formats and visualization libraries

Python's ecosystem for data handling provides a complete workflow: storing data in databases, exposing it through APIs, and analysing it with NumPy and Pandas.