Exception Handling, Some Python Packages - Numpy, Pandas, Matplotlib, File Handling, Time-Space Complexity

Martin Wafula
Multimedia University of Kenya

April 23, 2025

Outline

Exception Handling

NumPy

Pandas

Matplotlib

File Handling

GUI Development

Machine Learning

Exception Handling

Definition

Exceptions are Python's mechanism for handling runtime errors and exceptional conditions in a controlled way.

Key Characteristics:

- Disrupt normal program flow
- Are objects (instances of classes)
- Follow inheritance hierarchy
- Can be caught and handled

Common Exception Types:

- `ValueError`
- `TypeError`
- `IndexError`
- `KeyError`
- `FileNotFoundError`

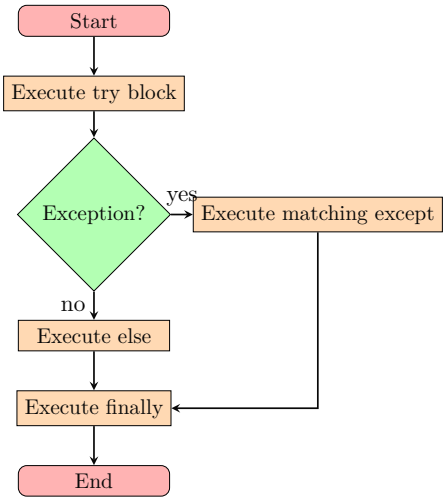
Python Philosophy

EAFP (Easier to Ask for Forgiveness than Permission) style is preferred over LBYL (Look Before You Leap).

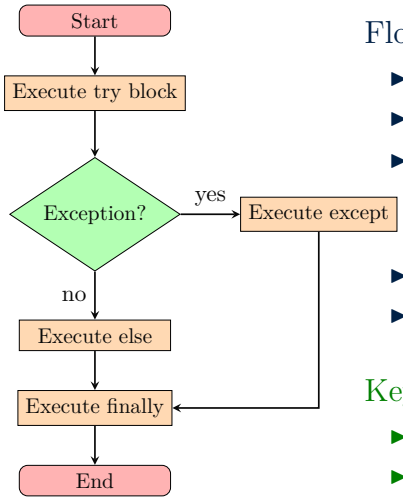
Basic Exception Handling Syntax

```python
# Basic try-except block
try:
    # Code that might raise exceptions
    result = 10 / denominator
    file = open("data.txt")
    value = int(file.read())
except ZeroDivisionError:
    print("Cannot divide by zero!")
except FileNotFoundError:
    print("File not found!")
except ValueError as ve:
    print(f"Invalid value: {ve}")
except Exception as e:
    print(f"Unexpected error: {type(e).__name__}")
else:
    print("Operation successful!")
finally:
    file.close()  # Always executes
    print("Cleanup complete")
```

Exception Handling Flow



Exception Handling Flow



Flow Explanation

- **Start**: Begin exception handling block
- **Try**: Code that might raise exceptions
- **Exception?**: Check if any exception occurred
  - Yes: Execute matching `except` block
  - No: Execute `else` block (optional)
- **Finally**: Always executes (cleanup code)
- **End**: Continue with normal program flow

Key Points

- `else` runs only if no exceptions occur
- `finally` runs *always* (even if exceptions propagate)
- Order matters: Specific exceptions first

Custom Exceptions

```python
class InvalidEmailError(ValueError):
    """Raised when email format is invalid"""
    def __init__(self, email):
        self.email = email
        super().__init__(f"Invalid email: {email}")
class WeakPasswordError(Exception):
    """Raised when password doesn't meet requirements"""
    def __init__(self, reason):
        self.reason = reason
        super().__init__(f"Weak password: {reason}")
    def validate_user(email, password):
        if "@" not in email:
            raise InvalidEmailError(email)
        if len(password) < 8:
            raise WeakPasswordError("too short")
        if not any(c.isdigit() for c in password):
            raise WeakPasswordError("no digits")
        return True
```

Exception Chaining

```python
def process_file(filename):
    try:
        with open(filename) as f:
            return int(f.read())
    except FileNotFoundError as fnf_error:
        raise RuntimeError("Processing failed") from fnf_error

def main():
    try:
        result = process_file("missing.txt")
    except RuntimeError as err:
        print(f"Error: {err}")
        print(f"Caused by: {err.__cause__}")
        # Access original traceback
        import traceback
        traceback.print_tb(err.__traceback__)

```

Key Attributes

- `__cause__`: Direct cause of exception
- `__context__`: Context for implicit chaining
- `__traceback__`: Traceback object

Context Managers for Resource Handling

```python
class DatabaseConnection:
    def __init__(self, dbname):
        self.dbname = dbname
        self.connection = None

    def __enter__(self):
        print(f"Connecting to {self.dbname}")
        self.connection = sqlite3.connect(self.dbname)
        return self.connection

    def __exit__(self, exc_type, exc_val, exc_tb):
        print("Closing connection")
        if self.connection:
            self.connection.close()
        if exc_type is not None:
            print(f"Error occurred: {exc_val}")
        return True  # Suppress exception

# Usage
```

```python
import logging

logging.basicConfig(
    filename='app.log',
    level=logging.ERROR,
    format='%(asctime)s - %(levelname)s - %(message)s'
)

def process_data(data):
    try:
        result = complex_operation(data)
    except ValueError as e:
        logging.error(f"Value error in processing: {e}", exc_info=True)
        raise
    except Exception as e:
        logging.critical(f"Unexpected error: {e}", exc_info=True)
        raise
    else:
        logging.info("Processing completed successfully")
```

Exception Best Practices

Do's

- Catch specific exceptions first
- Use custom exceptions for domain errors
- Clean up resources in `finally` blocks
- Log exceptions with context
- Preserve stack traces when re-raising

Don'ts

- Don't catch `Exception` unless necessary
- Don't ignore exceptions silently
- Don't use exceptions for flow control
- Don't expose sensitive information

Real-World Example: Web API Client


```python
import requests
from requests.exceptions import RequestException
class APIClient:
    def __init__(self, base_url):
        self.base_url = base_url
    def get_data(self, endpoint, retries=3):
        url = f"{self.base_url}/{endpoint}"
        for attempt in range(retries):
            try:
                response = requests.get(url, timeout=5)
                response.raise_for_status()
                return response.json()
            except requests.HTTPError as http_err:
                if response.status_code == 404:
                    raise ValueError("Endpoint not found") from http_err
                if attempt == retries - 1:
                    raise
            except RequestException as req_err:
                if attempt == retries - 1:
```

Database Transaction Example


```python
import sqlite3
from contextlib import contextmanager

@contextmanager
def transaction(db_path):
    conn = sqlite3.connect(db_path)
    cursor = conn.cursor()
    try:
        yield cursor
        conn.commit()
    except Exception as e:
        conn.rollback()
        raise e
    finally:
        conn.close()
# Usage
with transaction("app.db") as cursor:
    cursor.execute("INSERT INTO users VALUES (?, ?)", ("alice", 25))
    cursor.execute("UPDATE accounts SET balance = balance - 100 WHERE
```

Exception Handling in Threads

```
import threading
import queue

def worker(task_queue, result_queue):
    while True:
        try:
            task = task_queue.get_nowait()
            result = process_task(task)
            result_queue.put(result)
        except queue.Empty:
            break
        except Exception as e:
            result_queue.put(e)
            break

def main():
    tasks = [1, 2, 3, "invalid", 5]
    task_queue = queue.Queue()
    result_queue = queue.Queue()
```

Handling Keyboard Interrupt


```python
import time

def long_running_process():
    try:
        for i in range(10):
            print(f"Processing step {i}")
            time.sleep(1)
    except KeyboardInterrupt:
        print("\nReceived interrupt, cleaning up...")
        # Perform cleanup
        time.sleep(1)  # Simulate cleanup
        print("Cleanup complete")
        raise  # Re-raise to exit

def main():
    try:
        long_running_process()
    except KeyboardInterrupt:
        print("Program terminated by user")
```

Exception Groups (Python 3.11+)


```python
def validate_user(user):
    errors = []
    if len(user.name) < 3:
        errors.append(ValueError("Name too short"))
    if not user.email or "@" not in user.email:
        errors.append(ValueError("Invalid email"))
    if len(user.password) < 8:
        errors.append(ValueError("Password too short"))
    if errors:
        raise ExceptionGroup("User validation failed", errors)

try:
    validate_user(user)
except* ValueError as eg:
    for error in eg.exceptions:
        print(f"Validation error: {error}")

```

Numpy

Core Features

- Homogeneous N-dimensional arrays
- Vectorized operations
- Memory-efficient contiguous storage
- Broadcasting capabilities

Memory Layout:

- Contiguous memory block
- Fixed-type elements
- Strides for indexing
- Views vs copies

Performance Benefits:

- Avoid Python interpreter overhead
- Optimized C/Fortran code
- Cache-friendly access
- Parallelizable operations

Array Creation Methods


```
import numpy as np
# From Python sequences
arr1 = np.array([1, 2, 3])          # 1D array
arr2 = np.array([[1, 2], [3, 4]])   # 2D array
# Special arrays
zeros = np.zeros((3, 4))            # 3x4 zeros
ones = np.ones((2, 2, 2))           # 2x2x2 ones
identity = np.eye(3)                # Identity matrix
empty = np.empty((2, 3))            # Uninitialized
# Ranges and sequences
range_arr = np.arange(0, 10, 0.5)   # With step
linspace = np.linspace(0, 1, 5)     # 5 points in [0,1]
# Random arrays
random_arr = np.random.rand(2, 3)    # Uniform [0,1)
normal_arr = np.random.normal(0, 1, (2, 2))  # Normal dist

```

Array Attributes and Methods


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

# Basic attributes
print(arr.shape)     # (2, 3)
print(arr.ndim)      # 2
print(arr.size)      # 6
print(arr.dtype)     # int64
print(arr.itemsize)  # 8 (bytes per element)

# Array methods
print(arr.sum())              # 21
print(arr.mean(axis=0))       # [2.5 3.5 4.5] (column means)
print(arr.max(axis=1))        # [3 6] (row maximums)
print(arr.reshape(3, 2))      # Reshaped array
print(arr.flatten())          # [1 2 3 4 5 6]
print(arr.T)                  # Transpose

```

Indexing and Slicing

```
arr = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
# Basic indexing
print(arr[0, 1])      # 2 (row 0, column 1)
print(arr[:, 1])      # [2 5 8] (all rows, column 1)
print(arr[1:3, :2])   # [[4 5], [7 8]] (rows 1-2, cols 0-1)
# Boolean indexing
mask = arr > 5
print(mask)           # [[False False False], [False False True], [True
    True True]]
print(arr[mask])      # [6 7 8 9]
# Fancy indexing
print(arr[[0, 2], [1, 0]])  # [2 7] (elements (0,1) and (2,0))
# Views vs copies
arr_view = arr[:2, :2]      # View (shares memory)
arr_copy = arr[:2, :2].copy() # Copy (new memory)

```

```
a = np.array([1, 2, 3])
b = np.array([4, 5, 6])
# Element-wise operations
print(a + b)          # [5 7 9]
print(a * 2)          # [2 4 6]
print(np.sqrt(a))     # [1. 1.414 1.732]
print(np.exp(a))      # [2.718 7.389 20.085]
# Comparison operations
print(a == b)         # [False False False]
print(a > 1)          # [False True True]
# Matrix operations
mat = np.array([[1, 2], [3, 4]])
print(mat @ mat)      # Matrix multiplication
print(np.linalg.inv(mat))  # Matrix inverse

```

Broadcasting Definition

NumPy's ability to work with arrays of different shapes during arithmetic operations

```
# Broadcasting examples
a = np.array([1, 2, 3])
b = np.array([[10], [20]])

print(a + b)  # [[11 12 13], [21 22 23]]

# Rules:
# 1. Align array shapes from right
# 2. Dimensions must match or be 1
# 3. Missing dimensions are treated as 1

# Example shapes:
# (3,) + (2,1) -> (1,3) + (2,1) -> (2,3)

```

What are ufuncs?

Fast element-wise operations that support broadcasting, type casting, and other features

```
# Built-in ufuncs
arr = np.array([0, np.pi/2, np.pi])
print(np.sin(arr))    # [0. 1. 0.]
print(np.add(arr, 1)) # [1. 2.571 4.142]
# Creating custom ufuncs
def my_func(x, y):
    return x**2 + y**2
my_ufunc = np.frompyfunc(my_func, 2, 1)
print(my_ufunc(arr, 2)) # [4. 6.467 13.87]

```

Key Features

- Optimized C implementations
- Support all array methods

```
data = np.array([(1, 'Alice', 25.5),
                 (2, 'Bob', 30.2)],
                dtype=[('id', 'i4'), ('name', 'U10'), ('age', 'f4')])
# Accessing fields
print(data['name'])   # ['Alice' 'Bob']
print(data[0]['age']) # 25.5
# Record arrays (alternative syntax)
recarr = data.view(np.recarray)
print(recarr.age)     # [25.5 30.2]
# Sorting
data_sorted = np.sort(data, order='age')

```

Use Cases

- Tabular data with mixed types
- Memory-efficient storage
- Interoperability with other systems

```python
# Saving and loading arrays
arr = np.random.rand(3, 3)
# Text files
np.savetxt('array.txt', arr, delimiter=',')
loaded = np.loadtxt('array.txt', delimiter=',')
# Binary files (more efficient)
np.save('array.npy', arr)           # Single array
np.savez('arrays.npz', a=arr, b=arr*2)  # Multiple arrays
loaded = np.load('array.npy')
arrays = np.load('arrays.npz')
print(arrays['a'])  # Access by name
# Memory mapping
large_arr = np.memmap('large.dat', dtype='float32',
                      mode='w+', shape=(1000, 1000))

```

Performance Optimization

```python
import numpy as np
import numexpr as ne
# Vectorized vs loop
def slow_dot(a, b):
    result = 0
    for x, y in zip(a, b):
        result += x * y
    return result
a = np.random.rand(1000000)
b = np.random.rand(1000000)
%timeit slow_dot(a, b)  # ~500ms
%timeit np.dot(a, b)    # ~1ms
# Using numexpr for complex expressions
expr = ne.evaluate('sin(a) + log(b) * sqrt(a/b)')

```

Optimization Tips

- Avoid Python loops
- Use built-in functions
- Consider memory layout
- Use specialized libraries

Do's

- Preallocate arrays when possible
- Use vectorized operations
- Specify dtypes explicitly
- Use views instead of copies
- Leverage broadcasting

Don'ts

- Don't treat NumPy arrays like Python lists
- Don't ignore array shapes
- Don't mix Python types in arrays
- Don't forget about memory usage

Real-World Example: Image Processing


```
from scipy.misc import face
import matplotlib.pyplot as plt
# Load sample image (raccoon)
img = face()
# Convert to grayscale
gray = img.mean(axis=2)
# Apply filters
blurred = np.zeros_like(gray, dtype=float)
blurred[1:-1,1:-1] = (gray[:-2,:-2] + gray[:-2,1:-1] + gray[:-2,2:] +
                      gray[1:-1,:-2] + gray[1:-1,1:-1] + gray[1:-1,2:] +
                      gray[2:,:-2] + gray[2:,1:-1] + gray[2:,2:]) / 9
# Thresholding
binary = (gray > gray.mean()).astype(int)
# Display
plt.imshow(binary, cmap='gray')
plt.show()

```

Pandas

Series:

- 1D labeled array
- Homogeneous data
- Index alignment
- Vectorized operations

Series Creation

```
s = pd.Series([1, 2, 3],
              index=['a', 'b', 'c'],
              dtype='float32',
              name='values')

```

DataFrame:

- 2D labeled structure
- Columns can be different types
- Row and column indexes
- SQL-like operations

DataFrame Creation

```
data = {'A': [1, 2], 'B': ['x', 'y']}
df = pd.DataFrame(data,
                  index=['row1', 'row2'])

```

```python
# Create sample DataFrame
df = pd.DataFrame({
    'Name': ['Alice', 'Bob', 'Charlie'],
    'Age': [25, 30, 35],
    'City': ['NY', 'LA', 'Chicago']
}, index=['a', 'b', 'c'])
# Selection methods
df['Name']         # Column selection
df.loc['a']        # Row by label
df.iloc[0]         # Row by position
df[df['Age'] > 28] # Boolean indexing
df.query('Age > 28 & City == "LA"') # Query method
# Multi-indexing
df.set_index(['City', 'Name']).sort_index()
df.xs('NY', level='City')  # Cross-section
# Adding/removing columns
df['Senior'] = df['Age'] > 30
df.drop(columns=['Senior'], inplace=True)

```

Handling Missing Data

```python
# Create DataFrame with missing values
df = pd.DataFrame({
    'A': [1, 2, np.nan],
    'B': [5, np.nan, np.nan],
    'C': ['a', 'b', None]
})
# Detection
df.isna()
df.notna()
# Handling
df.dropna()        # Drop rows with any NA
df.fillna(0)       # Fill NA with 0
df.fillna(df.mean()) # Fill with mean
df.interpolate()  # Interpolate values
# Advanced
df['A'].fillna(df['A'].mean(), inplace=True)
df['B'].fillna(method='ffill', inplace=True)

```

```python
# Example messy data
data = {
    'date': ['2022-01-01', '2022/02/01', '01-03-2022'],
    'price': ['$10.50', '15.99', '20'],
    'quantity': [1, 'two', 3]
}
df = pd.DataFrame(data)
# Cleaning steps
df['date'] = pd.to_datetime(df['date'], errors='coerce')
df['price'] = df['price'].str.replace('$', '').astype(float)
df['quantity'] = pd.to_numeric(df['quantity'], errors='coerce')
# Outlier handling
df = df[(df['price'] > 0) & (df['price'] < 100)]
df = df.dropna()
# String operations
df['price_category'] = np.where(df['price'] < 15, 'low', 'high')

```

GroupBy Operations

```python
# Sample sales data
sales = pd.DataFrame({
    'region': ['East', 'West', 'East', 'West', 'East'],
    'product': ['A', 'B', 'A', 'B', 'A'],
    'revenue': [100, 150, 200, 50, 300],
    'cost': [80, 120, 150, 40, 250]
})

# Basic grouping
grouped = sales.groupby('region')
grouped.sum()
grouped.mean()

# Multiple aggregations
sales.groupby('region').agg({
    'revenue': ['sum', 'mean'],
    'cost': ['sum', 'mean']
})
```

```python
# Sample DataFrames
customers = pd.DataFrame({
    'id': [1, 2, 3],
    'name': ['Alice', 'Bob', 'Charlie']
})

orders = pd.DataFrame({
    'order_id': [101, 102, 103],
    'customer_id': [1, 3, 4],
    'amount': [50, 100, 75]
})

# Merge options
pd.merge(customers, orders, left_on='id', right_on='customer_id')
pd.merge(customers, orders, left_on='id', right_on='customer_id', how='left')
pd.merge(customers, orders, left_on='id', right_on='customer_id', how='outer')
```

```python
# Create time series
date_rng = pd.date_range(start='1/1/2022', end='1/10/2022', freq='D')
ts = pd.Series(np.random.randn(len(date_rng)), index=date_rng)

# Time-based operations
ts.resample('3D').mean()  # Resample to 3-day periods
ts.rolling(window=3).mean()  # Moving average

# Timezone handling
ts = ts.tz_localize('UTC')
ts = ts.tz_convert('US/Eastern')

# Shifting
ts.shift(1)    # Shift forward
ts.shift(-1)   # Shift backward

# Time deltas
ts.index + pd.Timedelta(days=2)

```

```python
# Reading from files
df_csv = pd.read_csv('data.csv')
df_excel = pd.read_excel('data.xlsx', sheet_name='Sheet1')
df_json = pd.read_json('data.json')
df_sql = pd.read_sql('SELECT * FROM table', con)

# Writing to files
df.to_csv('output.csv', index=False)
df.to_excel('output.xlsx', sheet_name='Results')
df.to_json('output.json', orient='records')
df.to_sql('table', con, if_exists='replace')

# Performance tips
pd.read_csv('large.csv', chunksize=10000)  # Process in chunks
pd.read_csv('data.csv', usecols=['col1', 'col2'])  # Read only needed
    columns

```

```python
# Pivot tables
sales.pivot_table(index='region', columns='product',
                  values='revenue', aggfunc='sum')

# Melt
pd.melt(df, id_vars=['Name'], value_vars=['Math', 'Science'])

# Apply functions
df['Age'].apply(lambda x: x + 1)
df.apply(np.mean, axis=0)  # Column-wise
df.apply(np.mean, axis=1)  # Row-wise

# Vectorized operations
df['revenue'] = df['price'] * df['quantity']
df['profit'] = df['revenue'] - df['cost']

```

```python
# Creating categories
df['grade'] = pd.Categorical(
    ['A', 'B', 'C', 'A', 'B'],
    categories=['A', 'B', 'C', 'D'],
    ordered=True
)

# Operations
df['grade'] > 'B'  # Comparison works with ordered
df['grade'].cat.codes  # Numeric codes
df['grade'].cat.categories  # View categories

# Memory efficiency
df['large_category'] = df['large_category'].astype('category')

# Grouping benefits
df.groupby('grade').size()
```

```python
# Data types matter
df['id'] = df['id'].astype('int32')  # Save memory
# Avoid iterrows()
# Instead of:
for index, row in df.iterrows():
    process(row)

# Use:
df.apply(process, axis=1)
# Or vectorized operations:
df['new_col'] = df['col1'] + df['col2']
# Query optimization
df.query('age > 30 & salary < 50000')  # Often faster than boolean
    indexing

# Use eval() for complex expressions
df.eval('revenue = price * quantity - cost', inplace=True)
```

Real-World Example: Sales Analysis


```python
# Load and clean data
sales = pd.read_csv('sales.csv', parse_dates=['date'])
sales['revenue'] = sales['price'] * sales['quantity']
# Monthly analysis
monthly = sales.resample('M', on='date').agg({
    'revenue': 'sum',
    'quantity': 'sum',
    'product': 'nunique'
})
# Top products
top_products = sales.groupby('product')['revenue'].sum().nlargest(5)
# Customer segmentation
sales['customer_type'] = pd.cut(
    sales['revenue'],
    bins=[0, 100, 500, float('inf')],
    labels=['small', 'medium', 'large']
)
# Visualization
monthly['revenue'].plot(kind='bar')
```

Pandas Best Practices

Do's

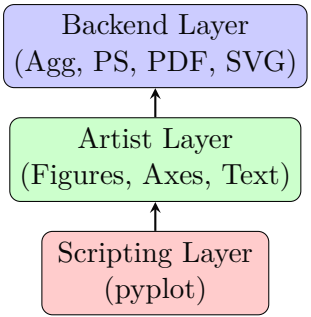
- Use vectorized operations
- Specify data types
- Use categoricals for strings
- Leverage built-in methods
- Use query() for complex filtering

Don'ts

- Don't use iterrows()/itertuples() unless necessary
- Don't ignore chained assignment warnings
- Don't modify data during iteration
- Don't forget to set indexes when appropriate

```python
# Chunk processing
chunk_size = 10000
results = []
for chunk in pd.read_csv('huge.csv', chunksize=chunk_size):
    results.append(process_chunk(chunk))
final = pd.concat(results)

# Dask alternative
import dask.dataframe as dd
ddf = dd.read_csv('huge_*.csv')
result = ddf.groupby('category').size().compute()

# Memory optimization
df = pd.read_csv('data.csv', dtype={'id': 'int32', 'price': 'float32'})
df = df.select_dtypes(include=['number'])  # Keep only numeric columns

# Sparse data
df = df.to_sparse()  # For data with many NaN/zeros

```

Matplotlib



Three Layer Architecture:

1. Scripting: Quick plotting (MATLAB-style)
2. Artist: Object-oriented interface
3. Backend: Actual rendering

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

# Create data
x = np.linspace(0, 2*np.pi, 100)
y1 = np.sin(x)
y2 = np.cos(x)

# Create figure and axes
fig, ax = plt.subplots(figsize=(10, 5))

# Plot data
ax.plot(x, y1, label='sin(x)', color='blue', linestyle='-')
ax.plot(x, y2, label='cos(x)', color='red', linestyle='--')

# Customize
ax.set_title("Trigonometric Functions")
ax.set_xlabel("x (radians)")
ax.set_ylabel("Function Value")
```

```
# Create figure with multiple subplots
fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(12, 4))

# First subplot
ax1.plot(x, np.sin(x))
ax1.set_title('Sine Wave')
ax1.set_facecolor('#f0f0f0')  # Light gray background
ax1.spines['top'].set_visible(False)  # Remove top spine

# Second subplot
ax2.plot(x, np.cos(x), 'r--')
ax2.set_title('Cosine Wave')
ax2.set_ylim(-1.5, 1.5)
ax2.grid(True, linestyle=':')

# Figure-level settings
fig.suptitle('Trigonometric Functions', fontsize=16)
fig.tight_layout()  # Adjust spacing

```

```
# Line plot
plt.plot(x, y)
# Scatter plot
plt.scatter(x, y, c=colors, s=sizes, alpha=0.5)
# Bar chart
plt.bar(categories, values)
# Histogram
plt.hist(data, bins=20)
# Box plot
plt.boxplot([data1, data2])
# Pie chart
plt.pie(sizes, labels=labels, autopct='%1.1f%%')
# Heatmap
plt.imshow(matrix, cmap='viridis')
plt.colorbar()
# Contour plot
plt.contour(X, Y, Z, levels=20)
```

```
# Create figure
fig = plt.figure(figsize=(10, 6))
ax = fig.add_subplot(111, projection='polar')  # Polar plot
# Plot data
theta = np.linspace(0, 2*np.pi, 100)
r = np.abs(np.sin(5*theta))
ax.plot(theta, r, linewidth=3)
# Customize
ax.set_theta_offset(np.pi/2)
ax.set_theta_direction(-1)
ax.set_rgrids([0.2, 0.4, 0.6, 0.8, 1.0])
# Annotations
ax.annotate('Peak', xy=(np.pi/2, 1), xytext=(0.5, 1.1),
            arrowprops=dict(facecolor='black', shrink=0.05))

# Text and arrows
ax.text(0, 0, 'Center', ha='center')

```

```
# Available styles
print(plt.style.available)
# ['ggplot', 'seaborn', 'dark_background', 'bmh', ...]
# Using styles
plt.style.use('ggplot')
# Custom style dictionary
mystyle = {
    'axes.facecolor': 'white',
    'axes.grid': True,
    'grid.color': '.8',
    'xtick.color': '.15',
    'ytick.color': '.15'
}
plt.style.use(mystyle)
# Context manager for temporary style
with plt.style.context('dark_background'):
    plt.plot(x, np.sin(x))
    plt.show()
```

```python
# Basic saving
plt.savefig('plot.png')  # Default format from extension

# Quality and size control
plt.savefig('high_res.png', dpi=300, bbox_inches='tight')

# Different formats
plt.savefig('plot.pdf')  # Vector format
plt.savefig('plot.svg')  # Scalable Vector Graphics

# Transparent background
plt.savefig('transparent.png', transparent=True)
# Multiple plots to PDF
from matplotlib.backends.backend_pdf import PdfPages
with PdfPages('multipage.pdf') as pdf:
    plt.plot(x, y1)
    pdf.savefig()
    plt.plot(x, y2)
    pdf.savefig()
```

```python
# Interactive mode
plt.ion()  # Turn on interactive mode

# Create plot
fig, ax = plt.subplots()
line, = ax.plot(x, np.sin(x))

# Update plot in loop
for phase in np.linspace(0, 2*np.pi, 20):
    line.set_ydata(np.sin(x + phase))
    fig.canvas.draw()
    fig.canvas.flush_events()
    plt.pause(0.1)

# Widgets
from matplotlib.widgets import Slider
ax_slider = plt.axes([0.2, 0.1, 0.6, 0.03])
slider = Slider(ax_slider, 'Phase', 0, 2*np.pi)

```

```
from mpl_toolkits.mplot3d import Axes3D

# Create 3D axes
fig = plt.figure(figsize=(8, 6))
ax = fig.add_subplot(111, projection='3d')

# Generate data
theta = np.linspace(0, 2*np.pi, 100)
z = np.linspace(0, 10, 100)
r = z
x = r * np.sin(theta)
y = r * np.cos(theta)


```

```
from mpl_toolkits.mplot3d import Axes3D

# Plot
ax.plot(x, y, z, label='3D spiral')
ax.set_xlabel('X')
ax.set_ylabel('Y')
ax.set_zlabel('Z')
ax.legend()

# Surface plot
X, Y = np.meshgrid(np.linspace(-5, 5, 100), np.linspace(-5, 5, 100))
Z = np.sin(np.sqrt(X**2 + Y**2))
ax.plot_surface(X, Y, Z, cmap='viridis')

```

Real-World Example: Stock Analysis


```
import pandas as pd
import matplotlib.dates as mdates

# Load stock data
df = pd.read_csv('stock.csv', parse_dates=['Date'])
df = df.set_index('Date')

# Create figure
fig, (ax1, ax2) = plt.subplots(2, 1, figsize=(12, 8))

# Price plot
ax1.plot(df.index, df['Close'], label='Closing Price')
ax1.set_ylabel('Price ($)')
ax1.xaxis.set_major_formatter(mdates.DateFormatter('%Y-%m'))
ax1.grid(True)
```

```
1
2  # Volume plot
3  ax2.bar(df.index, df['Volume'], width=5, label='Volume')
4  ax2.set_ylabel('Volume')
5  ax2.xaxis.set_major_formatter(mdates.DateFormatter('%Y-%m'))
6
7  # Formatting
8  fig.autofmt_xdate()
9  fig.suptitle('Stock Performance')
10 plt.tight_layout()
11
```

Do's

- Use the OO interface for complex plots
- Set figure size early
- Use tight_layout() to prevent overlaps
- Label your axes and add titles
- Choose appropriate color maps

Don'ts

- Don't use the pyplot interface for complex figures
- Don't forget to close figures in scripts
- Don't use jet colormap
- Don't overload your plots

Seaborn Integration

```python
import seaborn as sns

# Set style
sns.set_style("whitegrid")
# Create complex plots easily
tips = sns.load_dataset("tips")
sns.relplot(x="total_bill", y="tip", hue="day",
            col="time", data=tips)
# Statistical plots
sns.boxplot(x="day", y="total_bill", data=tips)
sns.violinplot(x="day", y="total_bill", data=tips)
# Regression plots
sns.lmplot(x="total_bill", y="tip", data=tips)
# Matrix plots
flights = sns.load_dataset("flights").pivot("month", "year", "passengers")
sns.heatmap(flights, annot=True, fmt="d")

```

File Handling

```python
# Writing to a file
with open('example.txt', 'w', encoding='utf-8') as f:
    f.write('Hello World!\n')
    f.write('Second line\n')
# Reading from a file
with open('example.txt', 'r') as f:
    content = f.read()    # Entire content
    lines = f.readlines() # List of lines
# Appending to a file
with open('example.txt', 'a') as f:
    f.write('Appended line\n')

```

Best Practice

Always use `with` statement for automatic file closing

File Modes and Encodings

```python
# Common file modes
with open('file.txt', 'r') as f:
    # Read (default)
    content = f.read()
with open('output.txt', 'w') as f:
    # Write (truncate)
    f.write("Hello\nWorld")
with open('data.bin', 'wb') as f:
    # Binary write
    f.write(bytes([65, 66, 67]))
with open('log.txt', 'a') as f:
    # Append
    f.write("New entry\n")
with open('data.txt', 'r+') as f:
    # Read/write
    content = f.read()
    f.seek(0)
    f.write("Updated")
```

Common Encodings

- utf-8 (recommended)
- ascii (basic English)
- latin-1 (Western Europe)
- utf-16 (alternative Unicode)

Encoding Issues

Always specify encoding when opening text files to avoid platform-dependent behavior

```python
import struct
# Packing binary data

data = struct.pack('3i2f', 1, 2, 3, 4.5, 5.5)

# 3 ints, 2 floats
with open('data.bin', 'wb') as f:
    f.write(data)

# Unpacking binary data
with open('data.bin', 'rb') as f:
    data = f.read()
    a, b, c, d, e = struct.unpack('3i2f', data)
    print(f"Values: {a}, {b}, {c}, {d}, {e}")

# Working with bytes
bytes_data = b'Hello\x00World'
with open('binary.txt', 'wb') as f:
    f.write(bytes_data)
```

Struct Format Characters

- `i` integer
- `f` float
- `d` double
- `s` char[]

```python
import csv
# Reading CSV
with open('data.csv', 'r') as f:
    reader = csv.reader(f)
    header = next(reader)  # Skip header
    for row in reader:
        print(row[0], float(row[1]))  # Access columns
# Writing CSV
with open('output.csv', 'w', newline='') as f:
    writer = csv.writer(f)
    writer.writerow(['Name', 'Age', 'City'])  # Header
    writer.writerow(['Alice', 25, 'NY'])
    writer.writerow(['Bob', 30, 'LA'])
# DictReader/DictWriter
with open('data.csv', 'r') as f:
    reader = csv.DictReader(f)
    for row in reader:
        print(row['Name'], row['Age'])

```

```python
import json
# Writing JSON
data = {
    'name': 'Alice',
    'age': 25,
    'courses': ['Math', 'Physics']
}
with open('data.json', 'w') as f:
    json.dump(data, f, indent=2)
# Reading JSON
with open('data.json', 'r') as f:
    loaded = json.load(f)
    print(loaded['name'])
# Custom serialization
class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age
def person_encoder(obj):
```

```python
import xml.etree.ElementTree as ET
# Parsing XML
tree = ET.parse('data.xml')
root = tree.getroot()
# Accessing elements
for child in root:
    print(child.tag, child.attrib)
    for subchild in child:
        print(subchild.text)
# Creating XML
root = ET.Element('catalog')
book = ET.SubElement(root, 'book', id='1')
title = ET.SubElement(book, 'title')
title.text = 'Python Programming'
ET.dump(root)
# Writing to file
tree = ET.ElementTree(root)
tree.write('output.xml', encoding='utf-8', xml_declaration=True)

```

```python
import openpyxl
# Reading Excel
wb = openpyxl.load_workbook('data.xlsx')
sheet = wb['Sheet1']

for row in sheet.iter_rows(values_only=True):
    print(row[0], row[1])
# Writing Excel
wb = openpyxl.Workbook()
sheet = wb.active
sheet['A1'] = 'Name'
sheet['B1'] = 'Age'
sheet.append(['Alice', 25])
wb.save('output.xlsx')
# Using pandas
df = pd.read_excel('data.xlsx', sheet_name='Sheet1')
df.to_excel('output.xlsx', sheet_name='Results', index=False)

```

```python
import os
import shutil

# Directory operations
os.mkdir('new_dir')
os.rmdir('empty_dir')
shutil.rmtree('dir_with_content')  # Recursive delete

# File operations
os.rename('old.txt', 'new.txt')
os.remove('file.txt')
shutil.copy('source.txt', 'dest.txt')
shutil.move('source.txt', 'new_location/')
```

File System Operations [cont...]

```python
import os
import shutil


# Path operations
path = os.path.join('dir', 'subdir', 'file.txt')
if os.path.exists(path):
    print(f"Size: {os.path.getsize(path)} bytes")
    print(f"Modified: {os.path.getmtime(path)}")

# Walking directory tree
for root, dirs, files in os.walk('.'):
    for file in files:
        print(os.path.join(root, file))

```

```python
import zipfile

# Creating ZIP archive
with zipfile.ZipFile('archive.zip', 'w') as zipf:
    zipf.write('file1.txt')
    zipf.write('file2.txt', arcname='renamed.txt')
# Extracting ZIP
with zipfile.ZipFile('archive.zip', 'r') as zipf:
    zipf.extractall('extracted')
# Reading ZIP contents
with zipfile.ZipFile('archive.zip', 'r') as zipf:
    print(zipf.namelist())
    with zipf.open('file1.txt') as f:
        content = f.read().decode('utf-8')
# Adding compression
with zipfile.ZipFile('compressed.zip', 'w', zipfile.ZIP_DEFLATED) as zipf
    :
    zipf.write('large_file.txt')

```

```python
import configparser
# Writing config
config = configparser.ConfigParser()
config['DEFAULT'] = {'debug': 'True', 'log_level': 'INFO'}
config['DATABASE'] = {'host': 'localhost', 'port': '5432'}

with open('config.ini', 'w') as f:
    config.write(f)
# Reading config
config = configparser.ConfigParser()
config.read('config.ini')

debug = config['DEFAULT'].getboolean('debug')
db_host = config['DATABASE']['host']
port = config['DATABASE'].getint('port')
# Using YAML (requires PyYAML)
import yaml
with open('config.yaml') as f:
    config = yaml.safe_load(f)
```

File Handling Best Practices

Do's

- Use `with` statements for file handling
- Specify encoding for text files
- Handle exceptions (FileNotFound, PermissionError)
- Use `os.path` for path operations
- Close files explicitly when not using `with`

Don'ts

- Don't assume files exist - always check
- Don't use string concatenation for paths
- Don't ignore file closing
- Don't load huge files into memory all at once

Real-World Example: Log Parser

```python
import re
from collections import defaultdict

log_pattern = re.compile(r'(\d{4}-\d{2}-\d{2}) (\d{2}:\d{2}:\d{2}) (\w+) (.*)')

def parse_log_file(filename):
    stats = defaultdict(int)
    errors = []

    with open(filename, 'r') as f:
        for line in f:
            match = log_pattern.match(line)
            if match:
                date, time, level, message = match.groups()
                stats[level] += 1
                if level == 'ERROR':
                    errors.append((date, time, message))
```

Graphical User Interface (GUI) Development

```python
import tkinter as tk
from tkinter import ttk

# Create main window
root = tk.Tk()
root.title("Simple GUI")
root.geometry("300x200")

# Add widgets
label = ttk.Label(root, text="Enter your name:")
entry = ttk.Entry(root)
button = ttk.Button(root, text="Greet",
                    command=lambda: print(f"Hello, {entry.get()}!"))

# Layout widgets
label.pack(pady=10)
entry.pack(pady=5)
button.pack(pady=10)

```

```python
# Label
label = tk.Label(root, text="Information", font=('Arial', 12))

# Button
button = tk.Button(root, text="Click", command=callback)

# Entry
entry = tk.Entry(root, show="*")  # For passwords

# Text (multi-line)
text = tk.Text(root, height=5, width=30)

# Checkbutton
check = tk.Checkbutton(root, text="Agree", variable=tk.BooleanVar())

# Radiobutton
radio1 = tk.Radiobutton(root, text="Option 1", variable=var, value=1)
radio2 = tk.Radiobutton(root, text="Option 2", variable=var, value=2)

```

pack():

```
# Simple vertical layout
label.pack(side="top", fill="x",
    padx=5, pady=5)
button.pack(side="bottom")

```

grid():

```
# Table-like layout
label.grid(row=0, column=0, sticky="
    ew")
entry.grid(row=0, column=1)
button.grid(row=1, columnspan=2)

```

place():

```
# Absolute positioning
label.place(x=10, y=10)
button.place(relx=0.5, rely=0.5,
    anchor="center")

```

Best Practices

- Use `grid` for forms
- Use `pack` for simple layouts
- Avoid mixing managers in same container

```python
# Button command
button = tk.Button(root, text="Click", command=callback)

# Binding events
def on_key(event):
    print(f"Key pressed: {event.char}")

root.bind('<Key>', on_key)

# Event types:
# <Button-1> - Left mouse click
# <Return> - Enter key
# <Motion> - Mouse movement
# <Configure> - Window resize

# Event object attributes:
# widget - The widget that received the event
# x, y - Mouse coordinates
# char - Character for keyboard events
```

```python
from tkinter import messagebox, filedialog, simpledialog

# Message boxes
messagebox.showinfo("Title", "Information")
messagebox.showwarning("Warning", "Be careful!")
messagebox.showerror("Error", "Something went wrong")
answer = messagebox.askyesno("Question", "Continue?")

# File dialogs
filename = filedialog.askopenfilename()
directory = filedialog.askdirectory()
save_as = filedialog.asksaveasfilename()

# Input dialogs
name = simpledialog.askstring("Input", "Enter your name")
age = simpledialog.askinteger("Input", "Enter your age")

```

```python
class ValidatedEntry(ttk.Entry):
    def __init__(self, parent, *args, **kwargs):
        super().__init__(parent, *args, **kwargs)
        self.configure(validate='key')
        self.configure(validatecommand=(
            self.register(self.validate_input), '%P'))

    def validate_input(self, proposed):
        # Only allow digits
        return proposed.isdigit() or proposed == ""

# Usage
entry = ValidatedEntry(root)
entry.pack()

# Custom composite widget
class LabelledEntry(ttk.Frame):
    def __init__(self, parent, text):
        super().__init__(parent)
```

```python
from tkinter import ttk

# Using themed widgets
style = ttk.Style()
style.configure('TButton', font=('Arial', 12))
style.map('TButton',
          foreground=[('pressed', 'red'), ('active', 'blue')],
          background=[('pressed', '!disabled', 'black'), ('active', 'white
    ')])

# Available themes
print(style.theme_names())  # ('clam', 'alt', 'default', 'classic')
style.theme_use('clam')

# Custom style
style.configure('My.TButton', padding=10, relief="flat")
button = ttk.Button(root, text="Styled", style='My.TButton')

```

```
# Create menu bar
menubar = tk.Menu(root)
root.config(menu=menubar)

# File menu
file_menu = tk.Menu(menubar, tearoff=0)
file_menu.add_command(label="Open", command=open_file)
file_menu.add_command(label="Save", command=save_file)
file_menu.add_separator()
file_menu.add_command(label="Exit", command=root.quit)
menubar.add_cascade(label="File", menu=file_menu)

# Edit menu
edit_menu = tk.Menu(menubar, tearoff=0)
edit_menu.add_command(label="Cut")
edit_menu.add_command(label="Copy")
menubar.add_cascade(label="Edit", menu=edit_menu)

# Context menu
```

```
canvas = tk.Canvas(root, width=400, height=300)
canvas.pack()

# Drawing shapes
canvas.create_line(10, 10, 100, 100, fill="red", width=2)
canvas.create_rectangle(50, 50, 150, 150, fill="blue")
canvas.create_oval(100, 100, 200, 200, fill="green")
canvas.create_text(200, 50, text="Hello", font=('Arial', 14))

# Interactive drawing
def start_draw(event):
    global last_x, last_y
    last_x, last_y = event.x, event.y

def draw(event):
    global last_x, last_y
    canvas.create_line(last_x, last_y, event.x, event.y, width=2)
    last_x, last_y = event.x, event.y

```

```python
class TextEditor(tk.Tk):
    def __init__(self):
        super().__init__()
        self.title("Simple Text Editor")
        self.geometry("600x400")

        # Create widgets
        self.text = tk.Text(self, wrap="word")
        scrollbar = ttk.Scrollbar(self, command=self.text.yview)
        self.text.config(yscrollcommand=scrollbar.set)

        # Menu
        menubar = tk.Menu(self)
        file_menu = tk.Menu(menubar, tearoff=0)
        file_menu.add_command(label="Open", command=self.open_file)
        file_menu.add_command(label="Save", command=self.save_file)
        menubar.add_cascade(label="File", menu=file_menu)
        self.config(menu=menubar)

```

Do's

- Use `ttk` widgets for native look
- Organize code with classes
- Use geometry managers consistently
- Separate UI and business logic
- Use resource files for strings

Don'ts

- Don't use `place()` unless necessary
- Don't mix `tk` and `ttk` widgets arbitrarily
- Don't block the main thread
- Don't create widgets in a loop

```
import threading

class LongRunningTask:
    def __init__(self, root):
        self.root = root
        self.progress = ttk.Progressbar(root, mode='indeterminate')
        self.progress.pack()

    def start(self):
        self.progress.start()
        thread = threading.Thread(target=self.run_task)
        thread.start()
        self.check_thread(thread)

    def run_task(self):
        # Simulate long task
        import time
        time.sleep(5)

```

Introduction to Machine Learning

Definition

Machine learning is the study of algorithms that improve automatically through experience, building models from sample data.

Types of ML:

- Supervised Learning
  - Classification
  - Regression
- Unsupervised Learning
  - Clustering
  - Dimensionality Reduction

Typical Workflow:

1. Data collection
2. Preprocessing Training
3. Evaluation
4. Deployment

Python Ecosystem

Scikit-learn, TensorFlow, PyTorch, Keras, XGBoost, LightGBM

```python
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import accuracy_score

# Load dataset
iris = load_iris()
X, y = iris.data, iris.target

# Split data
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.2, random_state=42)

# Preprocessing
scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)

```

```python
from sklearn.preprocessing import (StandardScaler, MinMaxScaler,
                                   OneHotEncoder, LabelEncoder)
from sklearn.impute import SimpleImputer
from sklearn.pipeline import Pipeline
from sklearn.compose import ColumnTransformer

# Numeric features
numeric_features = ['age', 'income']
numeric_transformer = Pipeline(steps=[
    ('imputer', SimpleImputer(strategy='median')),
    ('scaler', StandardScaler())])

# Categorical features
categorical_features = ['gender', 'country']
categorical_transformer = Pipeline(steps=[
    ('imputer', SimpleImputer(strategy='constant', fill_value='missing'))
    ,
    ('onehot', OneHotEncoder(handle_unknown='ignore'))])
```

```
from sklearn.metrics import (accuracy_score, precision_score,
                             recall_score, f1_score,
                             confusion_matrix, classification_report)

# For classification
y_pred = model.predict(X_test)
print("Accuracy:", accuracy_score(y_test, y_pred))
print("Precision:", precision_score(y_test, y_pred, average='macro'))
print("Recall:", recall_score(y_test, y_pred, average='macro'))
print("F1:", f1_score(y_test, y_pred, average='macro'))
print(confusion_matrix(y_test, y_pred))
print(classification_report(y_test, y_pred))
```

```
from sklearn.metrics import (accuracy_score, precision_score,
                             recall_score, f1_score,
                             confusion_matrix, classification_report)


# For regression
from sklearn.metrics import mean_squared_error, r2_score
print("MSE:", mean_squared_error(y_test, y_pred))
print("R2:", r2_score(y_test, y_pred))

# Cross-validation
from sklearn.model_selection import cross_val_score
scores = cross_val_score(model, X, y, cv=5)
print("CV Accuracy: %0.2f (+/- %0.2f)" % (scores.mean(), scores.std() *
    2))

```

```python
from sklearn.model_selection import GridSearchCV
# Define parameter grid
param_grid = {
    'n_estimators': [50, 100, 200],
    'max_depth': [None, 10, 20],
    'min_samples_split': [2, 5, 10]}
# Create grid search
grid_search = GridSearchCV(
    estimator=RandomForestClassifier(),
    param_grid=param_grid,
    cv=5,
    n_jobs=-1, verbose=2)
# Fit grid search
grid_search.fit(X_train, y_train)
# Best parameters
print("Best params:", grid_search.best_params_)
print("Best score:", grid_search.best_score_)
# Use best model
best_model = grid_search.best_estimator_
```

```python
from sklearn.feature_selection import SelectKBest, chi2
from sklearn.decomposition import PCA
from sklearn.feature_extraction.text import TfidfVectorizer

# Feature selection
selector = SelectKBest(chi2, k=10)
X_new = selector.fit_transform(X, y)

# Dimensionality reduction
pca = PCA(n_components=2)
X_pca = pca.fit_transform(X)

# Text feature extraction
corpus = ["This is good", "This is bad", "Excellent"]
vectorizer = TfidfVectorizer()
X_text = vectorizer.fit_transform(corpus)
```

Feature Engineering [cont...]

```
from sklearn.feature_selection import SelectKBest, chi2
from sklearn.decomposition import PCA
from sklearn.feature_extraction.text import TfidfVectorizer



# Custom feature engineering
from sklearn.base import BaseEstimator, TransformerMixin

class CustomTransformer(BaseEstimator, TransformerMixin):
    def fit(self, X, y=None):
        return self
    def transform(self, X):
        return np.c_[X, X[:, 0]**2]  # Add squared feature

```

```python
import joblib
import pickle

# Save model with joblib (preferred)
joblib.dump(model, 'model.joblib')
loaded_model = joblib.load('model.joblib')

# Save model with pickle
with open('model.pkl', 'wb') as f:
    pickle.dump(model, f)
with open('model.pkl', 'rb') as f:
    loaded_model = pickle.load(f)

# Save entire pipeline
from sklearn.pipeline import Pipeline
pipe = Pipeline([
    ('preprocessor', preprocessor),
    ('classifier', model)
])
```

```python
import pandas as pd
from sklearn.ensemble import RandomForestRegressor
from sklearn.model_selection import train_test_split
from sklearn.metrics import mean_absolute_error

# Load data
data = pd.read_csv('housing.csv')
X = data.drop('price', axis=1)
y = data['price']

# Split data
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.2, random_state=42)

# Preprocessing (simplified)
X_train = pd.get_dummies(X_train)
X_test = pd.get_dummies(X_test)

# Align columns
```

Do's

- Split data before any processing
- Use pipelines for reproducibility
- Track experiments and parameters
- Monitor model performance over time
- Start with simple models

Don'ts

- Don't test on training data
- Don't ignore feature scaling
- Don't leak information from test set
- Don't use accuracy for imbalanced data

```python
# Deep Learning with Keras
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
model = Sequential([
    Dense(64, activation='relu', input_shape=(10,)),
    Dense(64, activation='relu'),
    Dense(1)
])
model.compile(optimizer='adam', loss='mse')
model.fit(X_train, y_train, epochs=10)
# Ensemble Methods
from sklearn.ensemble import VotingClassifier
from sklearn.linear_model import LogisticRegression
from sklearn.svm import SVC
ensemble = VotingClassifier(estimators=[
    ('lr', LogisticRegression()),
    ('svm', SVC()),
    ('rf', RandomForestClassifier())
])
```

```python
# Feature importance

importances = model.feature_importances_
indices = np.argsort(importances)[::-1]
for f in range(X_train.shape[1]):
    print(f"{X_train.columns[indices[f]]}: {importances[indices[f]]}")

# SHAP values
import shap

explainer = shap.TreeExplainer(model)
shap_values = explainer.shap_values(X_test)
shap.summary_plot(shap_values, X_test)


```

```
1
2  # Partial dependence plots
3  from sklearn.inspection import plot_partial_dependence
4  plot_partial_dependence(model, X_train, features=[0, 1])
5
6  # LIME for local explanations
7  import lime
8  import lime.lime_tabular
9
10 explainer = lime.lime_tabular.LimeTabularExplainer(
11     X_train.values, feature_names=X_train.columns)
12 exp = explainer.explain_instance(X_test.iloc[0], model.predict_proba)
13 exp.show_in_notebook()
14
```

```python
# Resampling techniques
from imblearn.over_sampling import SMOTE
from imblearn.under_sampling import RandomUnderSampler
from imblearn.pipeline import Pipeline

over = SMOTE(sampling_strategy=0.1)
under = RandomUnderSampler(sampling_strategy=0.5)
steps = [('o', over), ('u', under)]
pipeline = Pipeline(steps=steps)
X_resampled, y_resampled = pipeline.fit_resample(X, y)

# Class weights
model = RandomForestClassifier(class_weight='balanced')

# Appropriate metrics
from sklearn.metrics import balanced_accuracy_score, roc_auc_score
print("Balanced accuracy:", balanced_accuracy_score(y_test, y_pred))
print("ROC AUC:", roc_auc_score(y_test, y_pred_proba))

```

Time Complexity

Key Concepts

- **Time Complexity**: How runtime grows with input size
- **Space Complexity**: How memory usage grows with input size
- **Big-O Notation**: Describes upper bound of growth
- **Best/Average/Worst Case**: Different scenarios

Why It Matters

Input Size	$O(n^2)$ Time
10	100 operations
100	10,000 operations
1000	1,000,000 operations

Practical Impact

A $O(n^2)$ algorithm with 1M items could take days vs seconds for O(n log n)

O(1) - Constant:

```
def get_first(items):
    return items[0]

```

O(log n) - Logarithmic:

```
def binary_search(sorted_list, target):
    low, high = 0, len(sorted_list) -1
    while low <= high:
        mid = (low + high) // 2
        if sorted_list[mid] == target:
            return mid
        elif sorted_list[mid] < target:
            low = mid + 1
        else:
            high = mid - 1
    return -1

```

O(n) - Linear:

```
def find_max(items):
    max_val = items[0]
    for item in items:
        if item > max_val:
            max_val = item
    return max_val

```

O(n²) - Quadratic:

```
def bubble_sort(items):
    n = len(items)
    for i in range(n):
        for j in range(n-i-1):
            if items[j] > items[j
    +1]:
                items[j], items[j+1]
     = items[j+1], items[j]

```

Python Operations Complexity

List Operations

Operation	Time Complexity
Indexing (lst[i])	O(1)
Append	O(1)
Insert	O(n)
Delete	O(n)
Search (x in lst)	O(n)
Sort	O(n log n)

Dict Operations

Operation	Time Complexity
Get item	O(1)
Set item	O(1)
Delete item	O(1)
Search (key in d)	O(1)

```python
# O(n) - Linear
def linear(items):
    total = 0
    for item in items:          # n iterations
        total += item           # O(1)
    return total                # Total: O(n)

# O(n²) - Quadratic
def quadratic(items):
    pairs = []
    for x in items:             # n iterations
        for y in items:         # n iterations
            pairs.append((x, y))  # O(1)
    return pairs                # Total: O(n²)


```

```
1
2 # O(n log n) - Linearithmic
3 def linearithmic(items):
4     items.sort()                # O(n log n)
5     result = []
6     for item in items:          # O(n)
7         result.append(item*2) # O(1)
8     return result               # Total: O(n log n)
9
```

```python
# O(1) - Constant space
def constant_space(n):
    total = 0
    for i in range(n):
        total += i
    return total

# O(n) - Linear space
def linear_space(n):
    numbers = []
    for i in range(n):
        numbers.append(i)
    return numbers
```

```
1
2
3  # O(n²) - Quadratic space
4  def quadratic_space(n):
5      matrix = []
6      for i in range(n):
7          row = []
8          for j in range(n):
9              row.append(i*j)
10         matrix.append(row)
11     return matrix
12
```

Before:

```python
def contains_duplicate(nums):
    for i in range(len(nums)):
        for j in range(i+1, len(nums)):
            if nums[i] == nums[j]:
                return True
    return False
# O(n²) time, O(1) space

```

After:

```python
def contains_duplicate(nums):
    seen = set()
    for num in nums:
        if num in seen:
            return True
        seen.add(num)
    return False
# O(n) time, O(n) space

```

Tradeoffs

- Time vs space
- Readability vs performance
- Best case vs worst case

```python
import timeit
import random
from functools import lru_cache
# Time measurement
def test_func():
    return sum(range(1000000))

time = timeit.timeit(test_func, number=100)
print(f"Average time: {time/100:.6f} seconds")
# Profiling
import cProfile
cProfile.run('test_func()')
# Memoization
@lru_cache(maxsize=None)
def fibonacci(n):
    if n < 2:
        return n
    return fibonacci(n-1) + fibonacci(n-2)
# Reduces from O(2^n) to O(n)
```

Techniques

- Memoization (caching)
- Using more efficient data structures
- Early termination
- Precomputing values
- Vectorization (NumPy)

```
# Vectorized vs loop
import numpy as np

def slow_dot(a, b):
    result = 0
    for x, y in zip(a, b):
        result += x * y
    return result

```

Real-World Example: Two Sum

Naive Approach:

```python
def two_sum(nums, target):
    for i in range(len(nums)):
        for j in range(i+1, len(nums)):
            if nums[i] + nums[j] == target:
                return [i, j]
    return []
# O(n²) time, O(1) space

```

Optimized:

```python
def two_sum(nums, target):
    seen = {}
    for i, num in enumerate(nums):
        complement = target - num
        if complement in seen:
            return [seen[complement], i]
        seen[num] = i
    return []
# O(n) time, O(n) space

```

Performance

For n=10,000:

- Naive: 50 million operations
- Optimized: 10,000 operations

Common Algorithms

Algorithm	Time	Space
Linear Search	O(n)	O(1)
Binary Search	O(log n)	O(1)
Bubble Sort	$O(n^2)$	O(1)
Merge Sort	O(n log n)	O(n)
Quick Sort	O(n log n)	O(log n)
Dijkstra's	O((V+E) log V)	O(V)

Data Structures

Operation	List	Dict
Access	O(1)	O(1)
Search	O(n)	O(1)
Insert	O(n)	O(1)
Delete	O(n)	O(1)

Optimization Guidelines

- First make it work, then make it fast
- Profile before optimizing
- Focus on bottlenecks (90/10 rule)
- Consider readability/maintainability

```
# Example: Choosing data structures
# Need fast lookups? Use dict/set
# Need ordered data? Use list
# Need both? Consider OrderedDict
from collections import OrderedDict, defaultdict
# Example: Using built-ins
# Instead of:
result = []
for item in items:
    result.append(func(item))
# Use:
```

```python
# Common pitfalls
# 1. Hidden loops
def contains_duplicate(nums):
    return len(nums) != len(set(nums))  # O(n)

# 2. String concatenation
result = ""
for s in strings:  # O(n²) - strings are immutable
    result += s
# Better:
result = "".join(strings)  # O(n)
```

Big-O in Practice [cont...]

```
1
2
3  # 3. Unnecessary computations
4  def sum_of_squares(n):
5      total = 0
6      for i in range(n):
7          for j in range(n):  # O(n²) when O(n) possible
8              total += i*j
9      return total
10 # Better:
11 def sum_of_squares(n):
12     sum_i = sum(range(n))
13     return sum_i * sum_i  # O(n)
14
```

Conclusion

Covered Topics:

- Exception handling
- NumPy arrays
- Pandas data analysis
- Matplotlib visualization
- File I/O operations
- GUI development
- ML fundamentals
- Time complexity

Key Skills:

- Writing robust Python code
- Efficient data processing
- Creating visualizations
- Building applications
- Algorithm analysis

Further Learning

- Advanced Python features
- Web frameworks (Django, Flask)
- Parallel programming
- Cloud deployment

Thank You!

Questions and Discussion