Python for Beginners

Object-Oriented Programming & Statistical Properties of Data

Dr. Sushil Sharma

(2025/2026)

Part 1: Object-Oriented Programming

What is Object-Oriented Programming?

A Programming Paradigm

OOP organizes code around 'objects' rather than functions and logic. Think of it like organizing a laboratory - each instrument (object) has its own properties and functions.

Real-World Analogy

Consider a microscope: it has properties (magnification, model) and actions (focus, adjust). In OOP, we model such real-world entities in code.

Why Use OOP?

Helps organize complex programs, enables code reuse, makes programs easier to maintain and extend.

Four Pillars of OOP

1. Encapsulation

Bundling data (attributes) and methods that operate on the data within a single unit (class). Like a sealed reaction vessel containing all necessary components.

2. Abstraction

Hiding complex implementation details, showing only essential features. Like using a spectrophotometer without knowing its internal electronics.

3. Inheritance

Creating new classes based on existing classes. Like how all imaging devices share common properties but each has unique features.

4. Polymorphism

Same interface for different underlying forms. Like how 'analyze()' can work on different sample types.

Classes and Objects: The Building Blocks

Class: The Blueprint

A class is a template or blueprint that defines what an object will contain. Like a molecular formula that describes a compound.

Object: The Instance

An object is a specific instance of a class. Like a specific sample of a compound - it follows the formula but is a real, usable thing.

Attributes: Properties

Data stored in an object. For a patient record: name, age, diagnosis.

Methods: Actions

Functions that belong to an object. For a patient record: update_diagnosis(), get_history().

Creating Your First Class

```
# Define a class for a laboratory sample
class Sample:
    def __init__(self, name, concentration, volume):
        # __init__ is the constructor - runs when creating an object
        self.name = name              # Attribute: sample name
        self.concentration = concentration  # Attribute: in mol/L
        self.volume = volume          # Attribute: in mL

    def get_moles(self):
        # Method: calculate number of moles
        return self.concentration * (self.volume / 1000)

    def dilute(self, factor):
        # Method: dilute the sample
        self.concentration = self.concentration / factor
        print(f"New concentration: {self.concentration} mol/L")

# Create objects (instances) of the Sample class
sample1 = Sample("Glucose", 0.5, 100)   # 0.5 M, 100 mL
sample2 = Sample("NaCl", 1.0, 50)       # 1.0 M, 50 mL

print(sample1.get_moles())  # Output: 0.05 moles
sample1.dilute(2)           # Dilute by factor of 2
```

Understanding 'self' in Python Classes

What is 'self'?

A reference to the current instance of the class. It allows accessing attributes and methods of the object.

Why is it needed?

Python needs to know WHICH object's data to use. If you have 10 samples, 'self' tells Python which specific sample to work with.

Always the first parameter

Every method in a class must have 'self' as its first parameter. Python passes it automatically when you call a method.

The __init__ Constructor Method

```
# The __init__ method initializes a new object
class Patient:
    def __init__(self, patient_id, name, age):
        # These run every time a new Patient is created
        self.patient_id = patient_id
        self.name = name
        self.age = age
        self.diagnoses = []  # Start with empty list
        self.treatments = []
        print(f"New patient record created for {name}")

    def add_diagnosis(self, diagnosis, date):
        self.diagnoses.append({"diagnosis": diagnosis, "date": date})

    def get_summary(self):
        return f"Patient: {self.name}, Age: {self.age}, Diagnoses: {len(self.diagnoses)}"

# Create patient objects
patient1 = Patient("P001", "John Doe", 45)
patient2 = Patient("P002", "Jane Smith", 32)

patient1.add_diagnosis("Hypertension", "2024-01-15")
print(patient1.get_summary())
```

Inheritance: Building on Existing Code

Parent Class (Base Class)

The original class that provides attributes and methods to be inherited. Like a general 'Instrument' class.

Child Class (Derived Class)

A new class that inherits from the parent. Like 'Spectrometer' inheriting from 'Instrument'.

Benefits of Inheritance

Reuse code without rewriting it. Extend functionality while keeping base behavior. Create organized class hierarchies.

The 'super()' Function

Calls the parent class methods. Essential for extending __init__ while keeping parent's initialization.

Inheritance in Practice

```
# Parent class (base class)
class Instrument:
    def __init__(self, name, manufacturer):
        self.name = name
        self.manufacturer = manufacturer
        self.is_calibrated = False

    def calibrate(self):
        self.is_calibrated = True
        print(f"{self.name} is now calibrated")

# Child class (derived class) - inherits from Instrument
class Spectrometer(Instrument):
    def __init__(self, name, manufacturer, wavelength_range):
        super().__init__(name, manufacturer)  # Call parent's __init__
        self.wavelength_range = wavelength_range  # New attribute
        self.measurements = []

    def measure(self, sample_name):
        if not self.is_calibrated:
            print("Please calibrate first!")
            return None
        result = {"sample": sample_name, "absorbance": 0.45}
        self.measurements.append(result)
        return result
```

Inheritance in Practice

```
# Parent class (base class)
class Instrument:
    def __init__(self, name, manufacturer):
        self.name = name
        self.manufacturer = manufacturer
        self.is_calibrated = False

    def calibrate(self):
        self.is_calibrated = True
        print(f"{self.name} is now calibrated")

# Child class (derived class) - inherits from Instrument
class Spectrometer(Instrument):
    def __init__(self, name, manufacturer, wavelength_range):
        super().__init__(name, manufacturer)  # Call parent's __init__
        self.wavelength_range = wavelength_range  # New attribute
        self.measurements = []

    def measure(self, sample_name):
        if not self.is_calibrated:
            print("Please calibrate first!")
            return None
        result = {"sample": sample_name, "absorbance": 0.45}
        self.measurements.append(result)
        return result
```

```
# Usage
spec = Spectrometer("UV-Vis 2000", "LabCorp", (200, 800))
spec.calibrate()       # Method from parent class
spec.measure("Sample1") # Method from child class
```

Exercise: Object-Oriented Programming

1. Create a class called 'Molecule' with attributes: name, formula, molecular_weight
2. Add a method 'calculate_moles(mass)' that returns moles given mass in grams
3. Create a child class 'Protein' that inherits from Molecule and adds an 'amino_acids' attribute
4. Create two instances: one Molecule (water) and one Protein (hemoglobin)
5. Test all methods and print results

Part 2: Statistical Properties of Data

What is Statistics?

Definition

The science of collecting, organizing, analyzing, and interpreting data to make informed decisions.

Why Statistics for Scientists?

Every experiment generates data. Statistics helps us understand what the data means, whether results are significant, and how confident we can be.

Two Main Branches

Descriptive Statistics (summarizing data) and Inferential Statistics (making predictions from samples).

Two Branches of Statistics

Descriptive Statistics

- Summarizes and describes data
- Tells us what happened
- Uses measures like mean, median, mode
- Measures spread: range, variance, std dev
- Creates visual summaries (charts, graphs)
- No predictions beyond the data

Inferential Statistics

- Makes predictions from samples
- Draws conclusions about populations
- Uses hypothesis testing
- Calculates confidence intervals
- Estimates population parameters
- Quantifies uncertainty

Measures of Central Tendency

Mean (Average)

Sum of all values divided by count. Sensitive to outliers. Use when data is symmetric without extreme values.

Median (Middle Value)

The middle value when data is sorted. Robust to outliers. Use when data has extreme values or is skewed.

Mode (Most Frequent)

Value that appears most often. Can have multiple modes. Use for categorical data or finding common values.

Calculating Central Tendency with Python

```
import numpy as np
from scipy import stats

# Sample data: patient ages in a clinical trial
ages = [25, 28, 32, 35, 35, 38, 42, 45, 48, 52, 55, 58, 35, 62]

# Mean - average value
mean_age = np.mean(ages)
print(f"Mean age: {mean_age:.2f}")        # Output: 42.14

# Median - middle value (robust to outliers)
median_age = np.median(ages)
print(f"Median age: {median_age:.2f}")    # Output: 40.00

# Mode - most frequent value
mode_result = stats.mode(ages, keepdims=True)
print(f"Mode: {mode_result.mode[0]}")     # Output: 35

# Compare with an outlier
ages_with_outlier = ages + [150]  # Add outlier
print(f"Mean with outlier: {np.mean(ages_with_outlier):.2f}")    # 49.47
print(f"Median with outlier: {np.median(ages_with_outlier):.2f}") # 42.00
```

Measures of Variability (Spread)

Range

Difference between maximum and minimum values. Simple but heavily affected by outliers.

Variance (σ^2)

Average of squared differences from the mean. Gives weight to extreme deviations. Units are squared.

Standard Deviation (σ)

Square root of variance. Same units as original data. Most commonly used measure of spread.

Interquartile Range (IQR)

Difference between 75th and 25th percentiles. Robust to outliers, shows where middle 50% of data lies.

Calculating Variability with Python

```
import numpy as np

# Measurement data: radiation doses in Gray
doses = [2.1, 2.3, 2.2, 2.5, 2.0, 2.4, 2.1, 2.6, 2.3, 2.2]

# Range: max - min
data_range = np.max(doses) - np.min(doses)
print(f"Range: {data_range:.2f} Gy")           # 0.60 Gy

# Variance: average squared deviation from mean
variance = np.var(doses, ddof=1)  # ddof=1 for sample variance
print(f"Variance: {variance:.4f} Gy²")         # 0.0349 Gy²

# Standard Deviation: square root of variance
std_dev = np.std(doses, ddof=1)
print(f"Standard Deviation: {std_dev:.4f} Gy") # 0.1867 Gy

# Interquartile Range (IQR)
q75, q25 = np.percentile(doses, [75, 25])
iqr = q75 - q25
print(f"IQR: {iqr:.2f} Gy")                    # 0.25 Gy
print(f"Q1: {q25:.2f}, Q3: {q75:.2f}")
```

Inferential Statistics: Key Concepts

Population vs Sample

Population: all possible subjects (often impossible to measure all). Sample: subset we actually measure.

Null Hypothesis (H_o)

Assumption of 'no effect' or 'no difference'. We try to reject this with evidence.

Alternative Hypothesis (H_1)

What we're trying to prove - there IS an effect or difference.

P-value

Probability of seeing our results (or more extreme) if null hypothesis were true. Small p-value = evidence against H_o.

Probability Distributions: What Are They?

Definition

A mathematical function describing the likelihood of different outcomes. Shows how data is spread across possible values.

Why They Matter

Many statistical tests assume your data follows a specific distribution. Understanding distributions helps you choose the right analysis.

Parameters

Each distribution has parameters (like mean and standard deviation for normal) that define its exact shape.

Common Probability Distributions

Continuous Distributions

- Normal (Gaussian): bell curve, most common
- Exponential: time between events
- Uniform: equal probability everywhere
- Chi-square: variance testing, goodness of fit
- Student's t: small samples, unknown variance

Discrete Distributions

- Binomial: yes/no outcomes, fixed trials
- Poisson: count of rare events
- Bernoulli: single yes/no trial
- Geometric: trials until first success
- Negative binomial: trials until n successes

Normal Distribution (Gaussian)

The Bell Curve

Symmetric, bell-shaped. Defined by mean (μ) and standard deviation (σ). Most natural phenomena approximately follow this.

Key Properties

68% of data within 1σ of mean, 95% within 2σ, 99.7% within 3σ (the empirical rule or 68-95-99.7 rule).

When to Use

Measurement errors, heights, blood pressure, many biological variables. Central Limit Theorem: sample means are normally distributed.

Python Function

np.random.normal(mean, std_dev, size) or scipy.stats.norm

Other Important Distributions

Poisson Distribution

Models count of rare events in fixed time/space. Example: number of mutations per DNA region, radioactive decays per minute. Parameter: λ (lambda) = average rate.

Binomial Distribution

Models number of successes in n independent trials. Example: number of patients responding to treatment out of 100. Parameters: n (trials), p (probability of success).

Exponential Distribution

Models time between events in a Poisson process. Example: time between radioactive decays, patient arrival times. Parameter: λ (rate).

Uniform Distribution

Equal probability for all values in range. Example: random number generators, rounding errors. Parameters: min and max values.

Comparing Distributions with Python

```
import numpy as np
import matplotlib.pyplot as plt
from scipy import stats

# Generate samples from different distributions
np.random.seed(42)
n_samples = 10000

# Normal Distribution (μ=0, σ=1)
normal_data = np.random.normal(loc=0, scale=1, size=n_samples)

# Poisson Distribution (λ=5)
poisson_data = np.random.poisson(lam=5, size=n_samples)

# Exponential Distribution (λ=1)
exponential_data = np.random.exponential(scale=1, size=n_samples)

# Compare means and standard deviations
print(f"Normal:      Mean={np.mean(normal_data):.3f}, Std={np.std(normal_data):.3f}")
print(f"Poisson:     Mean={np.mean(poisson_data):.3f}, Std={np.std(poisson_data):.3f}")
print(f"Exponential: Mean={np.mean(exponential_data):.3f}, Std={np.std(exponential_data):.3f}")
```

Exercise: Statistical Analysis

1. Generate 1000 random samples from a normal distribution with mean=50 and std=10
2. Calculate and print the mean, median, and standard deviation
3. Generate 1000 samples from a Poisson distribution with $\lambda=50$
4. Compare the means and standard deviations of both distributions
5. Create histograms of both distributions side by side

Hypothesis Testing: A Practical Framework

Step 1: State Hypotheses

H_0: No effect/difference (e.g., drug has no effect). H_1: There is an effect (drug works).

Step 2: Choose Significance Level

Usually $\alpha = 0.05$. This is your threshold for 'unlikely enough to reject H_0'.

Step 3: Calculate Test Statistic

Compare your sample to what you'd expect under H_0. Get a p-value.

Step 4: Make Decision

If $p < \alpha$, reject H_0 (evidence supports H_1). If $p \geq \alpha$, fail to reject H_0 (not enough evidence).

Hypothesis Testing: T-test Example

```
from scipy import stats
import numpy as np

# Scenario: Testing if a new drug reduces blood pressure
# Control group vs Treatment group (blood pressure reduction in mmHg)
np.random.seed(42)

control = [5, 3, 4, 6, 2, 4, 3, 5, 4, 3]     # Placebo group
treatment = [8, 9, 7, 10, 8, 9, 11, 8, 9, 10] # Drug group

# Independent samples t-test
# H₀: No difference between groups (μ_treatment = μ_control)
# H₁: Treatment is different from control (μ_treatment ≠ μ_control)

t_statistic, p_value = stats.ttest_ind(treatment, control)

print(f"Control mean: {np.mean(control):.2f} mmHg")
print(f"Treatment mean: {np.mean(treatment):.2f} mmHg")
print(f"T-statistic: {t_statistic:.3f}")
print(f"P-value: {p_value:.6f}")

# Decision (α = 0.05)
alpha = 0.05
if p_value < alpha:
    print("Reject H₀: Significant difference between groups!")
else:
    print("Fail to reject H₀: No significant difference")
```

Exercise: Comprehensive Statistics Exercise

1. Load or create a dataset of measurement values (at least 30 samples)
2. Calculate all descriptive statistics: mean, median, mode, std, variance, range, IQR
3. Create a histogram and identify what distribution your data resembles
4. Perform a normality test using scipy.stats.normaltest()
5. If comparing two groups, perform a t-test and interpret the results

Key Takeaways

Object-Oriented Programming

Classes are blueprints, objects are instances. Use inheritance for code reuse. Encapsulation keeps data organized.

Descriptive Statistics

Mean, median, mode describe center. Standard deviation, variance, IQR describe spread. Choose based on data characteristics.

Probability Distributions

Normal for continuous measurements, Poisson for counts, Binomial for success/failure. Know your data's distribution.

Inferential Statistics

Use samples to make claims about populations. Hypothesis testing with p-values. Always consider practical significance.

Course syllabus: -

Introduction (an overview to course)

- ✓ Getting started “Hello World”
- ✓ Keywords and identifiers
- ✓ Comments and Statements
- ✓ Variables and assignments
- ✓ Data types Flow control
- ✓ Methods and Functions
- ✓ Reading and Writing files
- ✓ Modules and import
- ✓ Object Oriented Programming

Data types

- ✓ Numbers
- ✓ List
- ✓ Tuple
- ✓ String
- ✓ Dictionary
- ✓ Set

Program Flow Control in Python

- ✓ *If* statement
- ✓ Elif statement
- ✓ More on if, elif and else
- ✓ *For* loop
- ✓ *While* loop
- ✓ Useful operators

Methods and Functions

- ✓ Defining a function
- ✓ Flow when calling a function
- ✓ Parameters and arguments
- ✓ Global/local
- ✓ Functions calling functions

Modules and Import

- ✓ Standard Python library
- ✓ Datetime module
- ✓ Math and Random module
- ✓ Generators and decorators
- ✓ NumPy, Pandas, Matplotlib (basic uses)

Object Oriented Programming (OOP)

- ✓ Introduction to OOPs
- ✓ Attributes and Class keywords
- ✓ Inheritance and Polymorphism

Reading/Writing Files

- ✓ Files and directories in Python
- ✓ Reading from a file
- ✓ Parsing data
- ✓ Printing data to external file

Stastistical analysis of data with Python

Thank You!

Exam will be on February 6, 2026