Advanced Python Programming for Data Science

CH – 01

Advanced Python Concepts and Best Practices

Baikuntha Acharya (baikunth2a@gmail.com)

Senior Lecturer, Sagarmatha Engineering College, Sanepa, Lalitpur

Classroom Rules

- ✓ Enter class on time
- ✓ **Important: Keep your class notes updated**
  - they contribute to your assignment mark and extra activities mark.
- ✓ Maintain respectful behavior
  - avoid side conversations and remain silent
- ✓ Don't hesitate to ask questions
- ✓ Don't use phone during class
  - Use devices only when allowed
- ✓ Required Skills
  - Comprehension
  - Summarization
  - Note-taking

Syllabus

1 Advanced Python Concepts and Best Practices (7 hours)

- 1.1 Review of Python essentials and coding conventions
- 1.2 Advanced data structures: Collections, iterators, generators, and decorators
- 1.3 Functions and lambda expressions
- 1.4 Object-Oriented Programming for data science applications
- 1.5 Exception handling, debugging, and logging
- 1.6 Working with modules and packages

2 Data Sources and APIs (7 hours)

- 2.1 Reading and writing structured/unstructured data (CSV, JSON, Excel, text)
- 2.2 Database access with relational database and non-relational database
- 2.3 Accessing and processing data from APIs (REST, SOAP)
- 2.4 Web scraping using requests and BeautifulSoup
- 2.5 Handling large datasets with chunking and lazy evaluation

3 Advanced Data Wrangling and Transformation (9 hours)

- 3.1 Advanced Pandas operations: Merging, joining, reshaping, pivoting
- 3.2 Handling missing, categorical, and time-series data
- 3.3 Feature transformation, scaling, and encoding
- 3.4 Memory optimization and efficient data processing
- 3.5 Building a reusable data-cleaning pipeline
- 3.6 Introduction to data pipeline components (Ingestion, transformation, storage)

Syllabus (Cont..)

4 Applied Statistics and Exploratory Analysis (7 hours)

- 4.1 Statistical measures: Correlation, covariance, skewness, kurtosis
- 4.2 Probability review, sampling, and hypothesis testing
- 4.3 Regression and trend analysis using stats models
- 4.4 Exploratory data analysis (EDA) using descriptive and inferential methods
- 4.5 Automation of EDA workflows using Python

5 Data Visualization and Storytelling (7 hours)

- 5.1 Principles of effective visualization and dashboard design
- 5.2 Visualization with Matplotlib: Line, bar, histogram, scatter, subplots
- 5.3 Seaborn for statistical visualization: Box plot, pair plot, heat map
- 5.4 Interactive visualization using Plotly
- 5.5 Visualization driven insight generation
- 5.6 Case study: End-to-end visualization and reporting project

6 Data Engineering and Automation (8 hours)

- 6.1 Overview of data engineering in applied data science
- 6.2 Designing and implementing ETL pipelines
- 6.3 Automating workflows with schedulers (CRON, schedule)
- 6.4 Logging, monitoring, and error handling in pipelines
- 6.5 Data storage and retrieval strategies for pipelines
- 6.6 Automated report generation (Excel, HTML, PDF)
- 6.7 Case study: End-to-end automated analytics pipeline

Python Essentials

- ✓ Python is a high-level, interpreted programming language known for its simplicity, readability, and versatility.
- ✓ It's used everywhere – from web apps and automation tools to data science and artificial intelligence.
  - **It's easy to learn** – its clear and readable syntax helps you focus on problem-solving, not on memorizing complex rules.
  - **It's fast to use** – you can build and test ideas quickly, whether it's a simple script or a full application.
  - **It's free and open** – Python is open source and runs on any major platform – Windows, macOS, Linux, or even Raspberry Pi.

Programming Fundamentals

✓ Variables and Data Types

- Python uses dynamic typing - No need to declare variable types explicitly.
- Common built-in data types:
  - int – integers
  - float – decimal numbers
  - str – strings
  - bool – True/False values

✓ Operators

- Arithmetic operations:
  - + (addition),
  - - (subtraction)
  - * (multiplication)
  - / (division)
  - // (floor division)
  - % (modulo)

Python Essentials (Contd..)

- Comparison operators
  - ==, !=, >, <
- Logical Operators
  - and, or, not

✓ Control Flow

- Conditional statements
  - if, elif , else
- Loops
  - for, while
- Loop control statements
  - break- exits loop
  - continue – skips current iterations
  - pass- placeholder statement

Python Essentials (Contd..)

✓ Functions

- Used to organize reusable code blocks
- Defined using the **def** keyword
- Key concepts:
  - Function arguments:
    - Positional arguments
    - Keyword arguments
    - Default arguments
    - Variable-length (*args, **kwargs)
  - Returns values using **return**
  - Variable scope:
    - Local variable
    - Global variable

```
def stats(x):
    return min(x), max(x)

mn, mx = stats([2, 5, 1])
```

```
def student_info(*args, **kwargs):
    print("Subjects:", args)
    print("Details:", kwargs)

student_info("Math", "AI", name="Sita", age=21)
```

`*args` → collects variable positional arguments as a tuple

`**kwargs` → collects variable keyword arguments as a dictionary

Core Data Structures

Python provides built-in data structures for efficient data organization and manipulation.

✓ Lists

- Ordered collection of items
- Mutable (*Elements can be changed*)
- Created using square brackets [ ]
- Example: ['apple' , 'banana']

```
numbers = [10, 20, 30, 40]
mixed = [1, "data", 3.14, True]
```

✓ Tuples

- Ordered collection of items
- Immutable
- Created using parentheses ( )
- Often used for fixed records
- Example: ('red' , 'green')

```
coordinates = (10, 20)
person = ("Ram", 25, "Engineer")
```

Python Essentials (Contd..)

✓ Dictionaries

- Store data as key-value pairs
- Keys must be immutable and hashable.
- Mutable data structure
- Enables fast lookups
- Example: { 'name' : 'John' , 'age' : 30}

Mutable objects can be changed after they are created.

```
student = {
    "name": "Sita",
    "age": 21,
    "department": "Data Science"
}
```

✓ Sets

- Unordered collection of unique elements
- A set does not support indexing or slicing (*Accessed via loops or membership testing*)
- Does not allow duplicate values
- Automatically removes duplicates
- Useful for mathematical set operation – Union, Intersection
- Example: {1, 2, 3}

```
unique_ids = {101, 102, 103}
```

Python Essentials (Contd..)

✓ String Manipulation

- Strings are immutable
- Any “change” creates a new string
- It supports:
  - Indexing
  - Slicing
  - Built-in string methods

```
s = "data"
s[0] = "D"      # ✗ TypeError
```

```
s = "data"
s = s + " science"
```

Python Essentials (Contd..)

Collections – Basic Operations

```
# LIST: ordered, mutable
lst = [1, 2, 3]          # create
lst.append(4)            # add
lst[0] = 10              # update
lst.remove(2)            # remove
x = lst[1]               # access
lst.extend([5, 6])       # extend
sub = lst[1:3]           # slicing
```

```
# SET: unordered, unique
st = {1, 2, 3}           # create
st.add(4)                # add
st.remove(2)             # remove
x = 3 in st              # membership test
st2 = st | {5, 6}        # union
inter = st & {3, 4}      # intersection
```

```
# TUPLE: ordered, immutable
tpl = (1, 2, 3)          # create
x = tpl[0]               # access
tpl2 = tpl + (4,)        # concatenate
a, b, c = tpl            # unpack
cnt = tpl.count(2)       # count value
idx = tpl.index(3)       # find index
```

```
# DICTIONARY: key-value mapping
dct = {"a": 1, "b": 2} # create
dct["c"] = 3             # add / update
x = dct["a"]             # access by key
del dct["b"]             # remove
v = dct.get("x", 0)      # safe access
item = dct.pop("a")      # remove & return
```

Advanced Topics

✓ Object Oriented Programming (OOP)

- Structure code using:
  - Classes (blueprints)
  - Objects (instances)
- Core OOP principles:
  - **Inheritance** – reuse functionality
  - **Polymorphism** – same interference, different behavior
  - **Encapsulation** – hide implementation details
  - **Abstraction** – hide implementation details

✓ File Handling

- Used to read from and write to files
- Functions and techniques: **open ()** function, **with** statement

Python Essentials (Contd..)

✓ Modules and Packages

- Used to organize code across multiple files
- Accessed using the **import** statement
- **Types:**
  - Built-in modules (e.g., math, datetime, os)
  - Third-party packages installed via **pip**

✓ Exception Handling

- Handles runtime errors gracefully
- Prevents program crashes
- Keywords: **try, except, else, finally**

```
try:
    file = open("data.txt", "r")
    content = file.read()
except FileNotFoundError:
    print("File not found")
finally:
    file.close()
    print("File closed")
```

Coding Conventions in Python

- ✓ Guidelines for writing clear, readable, and maintainable code.
- ✓ Widely accepted and followed coding convention in Python is **PEP8** (Python Enhancement Proposal 8).

Key aspects:

- ✓ **Indentation:**
  - Always use **4 spaces** for indentation; avoid tabs (PEP 8 standard).

```
# Good
def my_function():
    if x:
        print("Hello, world!")

# Bad (using tabs)
def my_function():
 if x:
  print("Hello, world!")
```

Coding Conventions in Python (Contd..)

✓ Maximum Line Length:

- Keep lines within **79–88 characters** to maintain readability.

```
# Good
def long_function_name(parameter1, parameter2, parameter3,
                       parameter4, parameter5):
    # Code goes here...

# Bad (exceeds line length limit)
def long_function_name(parameter1, parameter2, parameter3, parameter4, parameter5):
    # Code goes here...
```

✓ Whitespace in Expressions

- Avoid unnecessary whitespace around brackets, commas, and function calls.

```
# Good
spam(ham[1], {eggs: 2})

# Bad (extraneous whitespace)
spam( ham[ 1 ], { eggs: 2 } )
```

Coding Conventions in Python (Contd..)

✓ Import:

- One import per line
- Organize imports in a clear order: **standard** → **third-party** → **local**, one per line.

```python
# Good
import os
import sys

from my_module import my_function

# Bad (unorganized or multiple imports on one line)
import os, sys
from my_module import *
```

✓ Comments:

- Comment only to **explain why** code exists, **not what** it does.
- Use complete sentences
- **Docstrings**: Use triple quotes (""") to describe modules, classes, and functions.

```python
# Good
def calculate_total(price, tax_rate):
    """Calculate the total cost given a price and tax rate."""
    return price * (1 + tax_rate / 100)

# Bad (excessive or unclear comments)
def calculate_total(price, tax_rate):
    # This function calculates the total cost
    # Based on the given price and tax rate
    return price * (1 + tax_rate / 100)
```

Coding Conventions in Python (Contd..)

✓ Naming Conventions:

- Follow PEP 8 naming rules for variables, functions, classes, and modules.
  - Use lowercase_with_underscores for variable and function names (e.g., my_variable, calculate_total), module names (e.g., my_module.py).
  - Use CapitalizedWords for class names (e.g., MyClass).
  - Constants: UPPERCASE_WITH_UNDERSCORES

```
def compute_mean(): ...
class DataLoader: ...
MAX_ITER = 100
```

✓ Whitespace Between Operators and Variables:

- Use a single space around operators (e.g., assignment, comparison, arithmetic) but not when defining keyword arguments or default parameter values.

```
# Good
x = 5
y = x + 2
result = function_name(arg1, arg2)

# Bad (no space around assignment)
x=5
y =x+2
result = function_name (arg1, arg2)
```

```
# Good PEP8 whitespace usage
x = 5 + 3          # single space aroun
y = x * 2          # single space aroun
is_equal = (y == 16)  # single space

def greet(name="Alice", age=25):  # n
    return f"Hello {name}, age {age}"

print(is_equal, greet())
```

Coding Conventions in Python (Contd..)

✓ Avoid Unnecessary Parentheses:

- Avoid using parentheses when not necessary , but can be used when required for clarity or logic.

```
# Good
if x > 5:
    print("x is greater than 5")

# Bad (unnecessary parentheses)
if (x > 5):
    print("x is greater than 5")
```

✓ Consistency

- Maintain a consistent coding style throughout the entire project.

✓ Document Your Code:

- Use docstrings to document your code. This helps others understand how to use your code and what it does.

Collections

- Collections are container objects that hold and organize data.
- Python provides built-in collections (lists, tuples, dictionaries, sets) and specialized ones in the standard **collections** module.

✓ Namedtuple

- A **namedtuple** is like a regular tuple but with named fields, making data more readable and accessible.

```
>>> from collections import namedtuple                    # Import extension type
>>> Rec = namedtuple('Rec', ['name', 'age', 'jobs'])      # Make a generated class
>>> bob = Rec('Bob', age=40.5, jobs=['dev', 'mgr'])       # A named-tuple record
>>> bob
Rec(name='Bob', age=40.5, jobs=['dev', 'mgr'])

>>> bob[0], bob[2]                                        # Access by position
('Bob', ['dev', 'mgr'])
>>> bob.name, bob.jobs                                    # Access by attribute
('Bob', ['dev', 'mgr'])
```

Advanced Data Structures (Contd..)

✓ Deque (Doubly Ended Queue)

- Deque is the optimized list for quicker append and pop operations from both sides of the container.
- It provides **O(1)** time complexity for append and pop operations as compared to list with **O(n)** time complexity.

Example 2-23. Working with a deque

```
>>> from collections import deque
>>> dq = deque(range(10), maxlen=10)  ❶
>>> dq
deque([0, 1, 2, 3, 4, 5, 6, 7, 8, 9], maxlen=10)
>>> dq.rotate(3)  ❷
>>> dq
deque([7, 8, 9, 0, 1, 2, 3, 4, 5, 6], maxlen=10)
>>> dq.rotate(-4)
>>> dq
deque([1, 2, 3, 4, 5, 6, 7, 8, 9, 0], maxlen=10)
>>> dq.appendleft(-1)  ❸
>>> dq
deque([-1, 1, 2, 3, 4, 5, 6, 7, 8, 9], maxlen=10)
>>> dq.extend([11, 22, 33])  ❹
>>> dq
deque([3, 4, 5, 6, 7, 8, 9, 11, 22, 33], maxlen=10)
>>> dq.extendleft([10, 20, 30, 40])  ❺
>>> dq
deque([40, 30, 20, 10, 3, 4, 5, 6, 7, 8], maxlen=10)
```

✓ OrderedDict

- An OrderedDict is a dictionary that preserves the order in which keys are inserted
- Python 3.7+: dictionaries **preserve insertion order**

```
od = OrderedDict(a=1, b=2, c=3)
od.move_to_end('b')
print(list(od.keys()))  # ['a', 'c', 'b']
```

- Regular dict cannot do this without rebuilding the dict.

```
from collections import OrderedDict

d = OrderedDict()
d['foo'] = 1
d['bar'] = 2
d['spam'] = 3
d['grok'] = 4

# Outputs "foo 1", "bar 2", "spam 3", "grok 4"
for key in d:
    print(key, d[key])
```

Advanced Data Structures (Contd..)

✓ Counter

- Counter is a subclass of **dictionary** that counts elements in an iterable, where each **key** is an element and its **value** is the number of occurrences.

```
words = [
    'look', 'into', 'my', 'eyes', 'look', 'into', 'my', 'eyes',
    'the', 'eyes', 'the', 'eyes', 'the', 'eyes', 'not', 'around', 'the',
    'eyes', "don't", 'look', 'around', 'the', 'eyes', 'look', 'into',
    'my', 'eyes', "you're", 'under'
]

from collections import Counter
word_counts = Counter(words)
top_three = word_counts.most_common(3)
print(top_three)
# Outputs [('eyes', 8), ('the', 5), ('look', 4)]
```

✓ DefaultDict

- A DefaultDict is also a sub-class to **dictionary**.
- It is used to provide some default values for the key that does not exist and never raises a KeyError.

```
from collections import defaultdict

d = defaultdict(int)
d["apple"] += 1
```

Step by step:

1. `"apple"` does **not exist** in `d` → it is a **missing key**.
2. `defaultdict` sees the missing key and calls the **default factory** `int()` → returns `0`.
3. `"apple"` is automatically created in the dictionary with value `0`.
4. Then `+= 1` updates it to `1`.

```
count = defaultdict(int)
fruits = defaultdict(list)
colors = defaultdict(set)

count["apple"] += 1
fruits["banana"].append("yellow")
colors["cherry"].add("red")
```

Example: Counting items (*like a frequency dictionary*)

```
words = ["cat", "dog", "cat", "bird"]
freq = defaultdict(int)
for w in words:
    freq[w] += 1
print(freq)   # {'cat': 2, 'dog': 1, 'bird': 1}
```

Advanced Data Structures (Contd..)

✓ ChainMap

- A ChainMap encapsulates many dictionaries into a single unit and returns a list of dictionaries.
- The ChainMap instance does not copy the input mappings, but holds references to them.
- **First occurrence wins** — if the same key exists in multiple dictionaries, the **value from the first dictionary** is returned.
- Updates or insertions to a ChainMap only affect the first input mapping.
- Useful for combining **multiple contexts** (e.g., configuration, defaults, and user input).

```
>>> d1 = dict(a=1, b=3)
>>> d2 = dict(a=2, b=4, c=6)
>>> from collections import ChainMap
>>> chain = ChainMap(d1, d2)
>>> chain['a']
1
>>> chain['c']
6
```

```
>>> chain['c'] = -1
>>> d1
{'a': 1, 'b': 3, 'c': -1}
>>> d2
{'a': 2, 'b': 4, 'c': 6}
```

Advanced Data Structures (Contd..)

Iterators

- An **iterator** is an object that enables sequential access to its elements, allowing traversal through all values.
- To make an object iterable, it must implement two special methods:
  - **__iter__()**: Returns the iterator object itself and is used by for loops and in statements.
  - **__next__()**: Returns the next value in the sequence and raises a **StopIteration** exception to indicate that the iteration has completed.

```
# A list (iterable)
my_list = [4, 7, 0]

# Create an iterator
iterator = iter(my_list)

print(next(iterator))  # 4
print(next(iterator))  # 7
print(next(iterator))  # 0
# next(iterator) would now raise StopIteration
```

```
# Defining a custom iterator class
class SquaredNumbers:
    def __init__(self, start, end):
        self.start = start
        self.end = end

    def __iter__(self):
        self.current = self.start
        return self

    def __next__(self):
        if self.current >= self.end:
            raise StopIteration
        else:
            self.current += 1
            return self.current ** 2

# Using the custom iterator
squared_numbers = SquaredNumbers(1, 5)
for num in squared_numbers:
    print(num)
```

Output

```
4
9
16
25
```

Advanced Data Structures (Contd..)

Generators

- A **generator** is a special type of iterator
- Generator functions act just like regular functions with just one difference they use the Python **yield** keyword instead of **return**. It returns an iterator.
- Generator objects are used either by calling the **next method** on the generator object or using the generator object in a "**for in**" loop.

•return: Ends the function immediately and sends back a value.
•yield: Pauses the function, sends back a value, but keeps the function's state so it can resume later.

✓ Why Use Generators?

- **Memory efficiency**: They don't store all values in memory, only generate them when needed.
- **Performance**: Useful for large datasets or infinite sequences.
- **Readable code**: yield makes complex iteration logic easier to express.

```
# Defining a custom generator function
def squared_numbers(start, end):
    current = start
    while current < end:
        yield current ** 2
        current += 1

# Using the custom generator function
for num in squared_numbers(1, 5):
    print(num)
```

Output

```
1
4
9
16
```

Decorators

- A decorator is a special type of function that wraps another function, adding additional functionality to it.
- Decorators can be used to modify the behaviour of a function, class, or method. They are a powerful tool for enhancing the functionality of your code.

```
def decorator(func):
    def wrapper():
        print("Before function call")
        func()
        print("After function call")
    return wrapper

@decorator
def say_hello():
    print("Hello!")

say_hello()
```

```
Before function call
Hello!
After function call
```

```
# Defining a custom decorator
def square_numbers(func):
    def wrapper(*args, **kwargs):
        result = func(*args, **kwargs)
        return result ** 2
    return wrapper

# Using the custom decorator
@square_numbers
def add_numbers(a, b):
    return a + b

result = add_numbers(1, 2)
print(result) # 9
```

Output: 9

Functions and Lambda Expressions

- ✓ Functions serve two primary development roles:
  - **Maximizing code reuse and minimizing redundancy :** Functions allow logic to be written once and reused multiple times, reducing code duplication and making programs easier to maintain.
  - **Procedural decomposition:** Functions help break complex tasks into smaller, well-defined steps, making systems easier to design, understand, and implement.

```python
# Function to calculate factorial
def factorial(n):
    """Return factorial of n"""
    result = 1
    for i in range(1, n+1):
        result *= i
    return result

# Reuse: call multiple times (maximizing code reuse)
print(factorial(5))   # 120
print(factorial(3))   # 6

# Procedural decomposition: break task into smaller steps
def combinations(n, r):
    """Return nCr using factorial function"""
    return factorial(n) // (factorial(r) * factorial(n - r))

print(combinations(5, 2))  # 10
```

Functions and Lambda Expressions (Contd..)

Main concepts behind Python functions:

- ✓ **def** is executable code
  - The **def** statement is executed at runtime, creating a function object and binding it to a name; the function body itself executes only when the function is later called, and **def** may appear inside other blocks.
- ✓ lambda creates an object but returns it as a result
  - The **lambda** expression creates a function object and returns it directly for inline usage where **def** is not suitable.

```
square = lambda x: x ** 2
print(square(4))
```

- ✓ return sends a result object back to the caller
  - The **return** statement ends function execution and sends a value back to the caller, or **None** if no value is specified.

Functions and Lambda Expressions (Contd..)

- ✓ Yield sends a result object back to the caller, but remembers where it left off:
  - The **yield** statement produces a value and pauses function execution, allowing it to resume later and generate multiple results over time.
- ✓ Global declares module-level variables that are to be assigned
  - The **global** statement allows a function to assign values to variables defined at the module level.
- ✓ Nonlocal declares enclosing function variables that are to be assigned
  - The **nonlocal** statement enables a function to modify variables defined in an enclosing function scope.

```
def count_up():
    yield 1
    yield 2
    yield 3
```

```
g = count_up()
print(next(g))  # 1
print(next(g))  # 2
print(next(g))  # 3
```

```
x = 10  # module-level variable

def update():
    global x
    x = 20
```

```
update()
print(x)
```

```
def outer():
    x = 5

    def inner():
        nonlocal x
        x = 10

    inner()
    return x
```

Functions and Lambda Expressions (Contd..)

- ✓ Arguments are passed by assignment (object reference)
  - Function arguments are passed as object references, so modifying mutable objects inside a function affects the shared object.

```
def add_item(lst):
    lst.append(4)

my_list = [1, 2, 3]
add_item(my_list)
print(my_list)  # O
```

Python passes references to objects. Modifying a **mutable object** inside a function affects the original object.

```
def increment(x):
    x += 1

a = 5
increment(a)
print(a)  # Output
```

Immutable objects (like int, str) won't be affected.

- ✓ Arguments are passed by position, unless you say otherwise
  - Arguments are matched by position by default, with support for keyword arguments and variable-length arguments using ***args** and ****kwargs**.

- ✓ Arguments, return values, and variables are not declared.
  - Python functions are dynamically typed, allowing arguments, return values, and variables to be of any type.

```
def multiply(x, y):
    return x * y

print(multiply(2, 3))     # int: 6
print(multiply(2.5, 4))   # float: 10.0
```

Functions and Lambda Expressions (Contd..)

Anonymous (Lambda) Functions

- Lambda functions are a way of defining functions that consist of a single expression, whose result is returned automatically.
- They are defined using the lambda keyword, which simply means **"we are declaring an anonymous function"**.

Example: Suppose you wanted to sort a collection of strings by the number of distinct letters in each string:

```
add = lambda x, y: x + y


print(add(3, 5))  # Output: 8
```

```
nums = [1, 2, 3, 4]
squared = list(map(lambda x: x**2, nums))
print(squared)  # Output: [1, 4, 9, 16]
```

```
In [177]: strings = ['foo', 'card', 'bar', 'aaaa', 'abab']
```

Here we could pass a lambda function to the list's `sort` method:

```
In [178]: strings.sort(key=lambda x: len(set(list(x))))

In [179]: strings
Out[179]: ['aaaa', 'foo', 'abab', 'bar', 'card']
```

Object-Oriented Programming for data science applications

- ✓ Data science involves data storage, manipulation, analysis, and visualization.
- ✓ Simple tasks can be handled procedurally, but complex workflows require structure.
- ✓ Object-Oriented Programming (OOP) provides scalability, organization, and maintainability.
- ✓ OOP is especially useful for large data pipelines and machine learning systems

- ✓ **Core Principles of OOP:**
  - Encapsulation
  - Abstraction
  - Inheritance
  - Polymorphism

OOP Basics

- ✓ **1. Class**
  - A blueprint for creating objects.
- ✓ **2. Object**
  - An instance of a class.
- ✓ **3. Attributes**
  - Variables associated with an object.
- ✓ **4. Methods**
  - Functions inside a class that operate on objects.

```python
class Dataset:
    def __init__(self, data):
        self.data = data

    def mean(self):              # Method
        return sum(self.data)/len(self.data)

data1 = Dataset([10, 20, 30])
print(data1.mean())  # Output: 20.0
```

OOP Basics

✓ 5. Inheritance

- A class can inherit properties and methods from another class.

✓ 6. Encapsulation

- Hiding internal details using _ or __.

✓ 7. Polymorphism

- Same interface, different behavior.

```
class ExtendedDataset(Dataset):
    def max_value(self):
        return max(self.get_data())

data2 = ExtendedDataset([5, 15, 25])
print(data2.mean())      # 15.0
print(data2.max_value()) # 25
```

```
def summarize(data):
    if isinstance(data, list):
        print("Mean of list:", sum(data)/len(data))
    elif isinstance(data, dict):
        print("Keys in dict:", list(data.keys()))
    else:
        print("Unsupported data type")

# Using same function with different types
summarize([10, 20, 30])        # Output: Mean of list: 20.0
summarize({'a': 1, 'b': 2})    # Output: Keys in dict: ['a', 'b']
```

Encapsulation

- ✓ Encapsulation bundles data and related methods together
- ✓ Prevents unintended data manipulation
- ✓ Improves code readability and organization
- ✓ Common use cases:
  - Dataset handling
  - Model training and evaluation
  - Preprocessing pipelines

Modifier	Syntax	Meaning
Public	variable	Accessible from anywhere
Protected	_variable	Accessible within class and subclasses (convention)
Private	__variable	Accessible only inside the class (name mangling)

```python
class FeatureStats:
    def __init__(self, data):
        self.data = data        # Public
        self._n = len(data)     # Protected
        self.__mean = sum(data) / self._n   # Private

    def get_mean(self):         # Public method
        return self.__mean
```

```python
x = [2, 4, 6, 8]

stats = FeatureStats(x)

print(stats.data)        # Public → OK
print(stats._n)          # Protected → OK (not recommended)
print(stats.get_mean())  # Private accessed via public method

print(stats.__mean)      # ✗ Error
```

✓ Basic Linear Regression model

```python
from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_squared_error
import numpy as np

class MyLinearRegression:
    def __init__(self):
        self.model = LinearRegression()

    def train(self, X, y):
        self.model.fit(X, y)
        self.coefficients = self.model.coef_

    def predict(self, X):
        return self.model.predict(X)

    def evaluate(self, X, y):
        predictions = self.predict(X)
        mse = mean_squared_error(y, predictions)
        return mse

# Usage
regressor = MyLinearRegression()

# Assume we have some data in X_train, y_train, X_test, y_test
X_train = np.array([[1], [2], [3], [4], [5]])
y_train = np.array([2, 3, 4, 5, 6])

X_test = np.array([[6], [7]])
y_test = np.array([7, 8])

regressor.train(X_train, y_train)
predictions = regressor.predict(X_test)
mse = regressor.evaluate(X_test, y_test)

print(f"Predictions: {predictions}")
print(f"Mean Squared Error: {mse}")
```

✓ Abstraction

- Abstraction hides implementation details.
- Provides simple, high-level interfaces.
- Common libraries:
  - Pandas for data manipulation
  - NumPy for numerical computation
  - Scikit-learn for machine learning

```
import pandas as pd

# Load the data from a CSV file. We don't need to know how pandas reads the file and parses the data.
df = pd.read_csv('data.csv')

# Calculate the mean of a column. We don't need to know how pandas iterates over the column and calculates the mean.
mean_value = df['column_name'].mean()

# Drop missing values. We don't need to know how pandas identifies and removes the rows with missing values.
df_clean = df.dropna()

print(mean_value)
print(df_clean)
```

✓ Inheritance

- Creating a new class from an existing class
- The child class reuses properties and methods of the parent class
- Represents an “is-a” relationship between classes
- Allows method overriding to modify inherited behavior
- Promotes code reuse and hierarchical design
- Makes programs easier to maintain and extend

```
class Animal:
    def speak(self):
        return "Some sound"

class Dog(Animal):
    def speak(self):          # overriding
        return "Bark"

d = Dog()
print(d.speak())
```

✓ Polymorphism

- One interface or method name, multiple implementations
- The same method call behaves differently for different objects
- Behavior depends on the object's actual type, not the variable
- Method selection happens at runtime (dynamic binding)

```python
from sklearn.linear_model import LogisticRegression
from sklearn.tree import DecisionTreeClassifier
from sklearn.model_selection import train_test_split
from sklearn.datasets import load_iris

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

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

# Define models
models = [
    LogisticRegression(),
    DecisionTreeClassifier()
]

# We can use the same method calls for different models, thanks to polymorphism.
for model in models:
    model.fit(X_train, y_train)
    predictions = model.predict(X_test)
    print(f"Predictions from {model.__class__.__name__}: {predictions}")
```

Exception handling, debugging, and logging

- ✓ Errors are inevitable in real-world programs
- ✓ Python provides structured mechanisms to:
  - Handle runtime errors (Exceptions)
  - Identify and fix bugs (Debugging)
  - Record program behavior (Logging)

```
try:
    x = 10 / 0
except ZeroDivisionError:
    print("Cannot divide by zero")
```

- ✓ Essential for building robust and maintainable applications.
- ✓ What is an Exception?
  - An **exception** is a runtime error that disrupts normal program flow.
  - Examples: division by zero, file not found, invalid type operations.
- ✓ Why Exception Handling?
  - Prevents program crashes
  - Handles errors **gracefully**
  - Improves reliability and user experience

`try` → code that may raise an exception
`except` → handles the exception
`else` → runs if no exception occurs
`finally` → always executes (cleanup code)

Exception handling, debugging, and logging (Contd..)

✓ Exception

- An exception is a runtime event that disrupts normal program execution
- Raised when Python encounters an error condition
- Without handling, exceptions cause program termination
- Exception handling allows programs to recover gracefully

```
try:
    result = 10 / 0
except ZeroDivisionError as e:
    print(f"An error occurred: {e}")
```

- **try-expect mechanism**
  - **try** block contains code that may fail
  - **except** block defines how to handle specific exceptions
  - Prevents abrupt program crashes
  - Enables controlled error handling and user feedback

✓ Common Built-in Exceptions in Python

- ValueError → Invalid data type or value
- FileNotFoundError → Missing file during I/O
- IndexError → Accessing out-of-range indices
- Each exception represents a specific failure category

```
# ValueError
try:
    number = int("not_a_number")
except ValueError as e:
    print(f"ValueError: {e}")

# FileNotFoundError
try:
    with open("non_existent_file.txt") as file:
        content = file.read()
except FileNotFoundError as e:
    print(f"FileNotFoundError: {e}")

# IndexError
try:
    my_list = [1, 2, 3]
    print(my_list[10])
except IndexError as e:
    print(f"IndexError: {e}")
```

Best Practices for Exception Handling

✓ Be Specific with Exceptions

- Catch specific exceptions, not generic ones (*Eg. FileNotFoundError not Exception*)
- Avoid masking errors with **except Exception** unnecessarily
- Never silence exceptions using **except: pass**
- Clear exception handling improves debuggability

```
# Specific Exception Handling
try:
    with open("non_existent_file.txt") as file:
        content = file.read()
except FileNotFoundError as e:
    print(f"File not found: {e}")

# General Exception Handling
try:
    with open("non_existent_file.txt") as file:
        content = file.read()
except Exception as e:
    print(f"An error occurred: {e}")
```

```
try:
    with open("non_existent_file.txt") as file:
        content = file.read()
except:
    pass
```

Exception handling, debugging, and logging (Contd..)

✓ Use else and finally Clauses

- **else** block executes when no exception occurs
- **finally** block executes regardless of exceptions
- Useful for:
  - Cleanup operations
  - Resource management
- Improves clarity of success vs failure logic

```
try:
    result = 10 / 2
except ZeroDivisionError as e:
    print(f"An error occurred: {e}")
else:
    print("No errors occurred, result:", result)
finally:
    print("This will always execute, regardless of whether an error occurred.")
```

Exception handling, debugging, and logging (Contd..)

✓ Raising Exceptions

- Use **raise** to explicitly trigger an exception
- Enforces certain conditions or indicate that something went wrong.
- Prevents invalid data from propagating
- Makes error conditions explicit and predictable

```
def validate_age(age):
    if age < 0:
        raise ValueError("Age cannot be negative.")
    return age

try:
    validate_age(-5)
except ValueError as e:
    print(f"Validation Error: {e}")
```

Exception handling, debugging, and logging (Contd..)

✓ Custom Exceptions

- Built-in exceptions may not capture application-specific errors
- Custom exceptions improve code clarity and intent
- Inherit from Python's Exception class
- Useful for domain-specific validation

```
class InvalidAgeError(Exception):
    pass

def validate_age(age):
    if age < 0:
        raise InvalidAgeError("Age cannot be negative.")
    return age

try:
    validate_age(-5)
except InvalidAgeError as e:
    print(f"Custom Exception Caught: {e}")
```

Exception handling, debugging, and logging (Contd..)

✓ Debugging Techniques

- Debugging is the process of identifying and fixing local and runtime errors.

✓ Common Sources of Bugs

- Logical errors
- Incorrect assumptions
- Edge (extreme) cases not handled
- Wrong data types or values

✓ Common techniques:

- **Print statements** for quick inspection
- **Python Debugger (pdb)** for step-by-step execution
- **IDE debugging tools** for visual inspection

Command	Description
n	Execute **next line**
c	**Continue** until next breakpoint
s	**Step into** a function call
l	**List source code** around current line
p var	**Print** the value of a variable
q	Quit debugger

```
def calculate_total(price, tax_rate):
    print(f"Debug: price={price}, tax_rate={tax_rate}")
    total = price + (price * tax_rate)
    print(f"Debug: total={total}")
    return total

calculate_total(100, 0.1)
```

```
import pdb

def calculate_total(price, tax_rate):
    pdb.set_trace()  # Set a breakpoint here
    total = price + (price * tax_rate)
    return total

calculate_total(100, 0.1)
```

Exception handling, debugging, and logging (Contd..)

✓ What is Logging?

- Recording **program events** during execution
- Used instead of excessive print() statements
- Helps in debugging and monitoring production systems

When to Use Logging vs Exceptions

- **Exceptions** → handle abnormal situations
- **Logging** → record what happened
- Often used **together**

✓ Logging for Debugging and Monitoring

- Logging records program behavior over time
- Supports multiple severity levels:
  - DEBUG, INFO, WARNING, ERROR, CRITICAL
- More scalable and flexible than print statements
- Essential for production systems

Logging Levels

- `DEBUG` → detailed diagnostic info
- `INFO` → general program flow
- `WARNING` → potential issues
- `ERROR` → serious problems
- `CRITICAL` → system failure

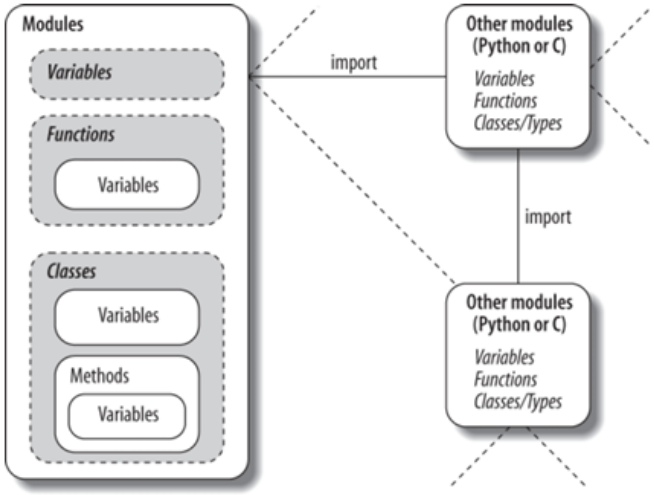
```python
import logging

logging.basicConfig(level=logging.INFO)
logging.info("Program started")
logging.error("An error occurred")
```

```python
import logging

logging.basicConfig(level=logging.DEBUG)

def calculate_total(price, tax_rate):
    logging.debug(f"price={price}, tax_rate={tax_rate}")
    total = price + (price * tax_rate)
    logging.debug(f"total={total}")
    return total

calculate_total(100, 0.1)
```

Working with modules and packages

✓ Module

- A **module** is a single Python file (.py).
- It contains Python code such as:
  - Variables
  - Functions
  - Classes
- Used to organize code into logical, reusable pieces.
- Example: math.py, utils.py




```python
# math_utils.py
def add(a, b):
    return a + b

def multiply(a, b):
    return a * b
```

Basic import

```python
import module_name
```

Import specific function/class

```python
from math_utils import add
print(add(2, 3))
```

Import with alias

```python
import math_utils as mu
print(mu.multiply(2, 4))
```

Working with modules and packages (Contd..)

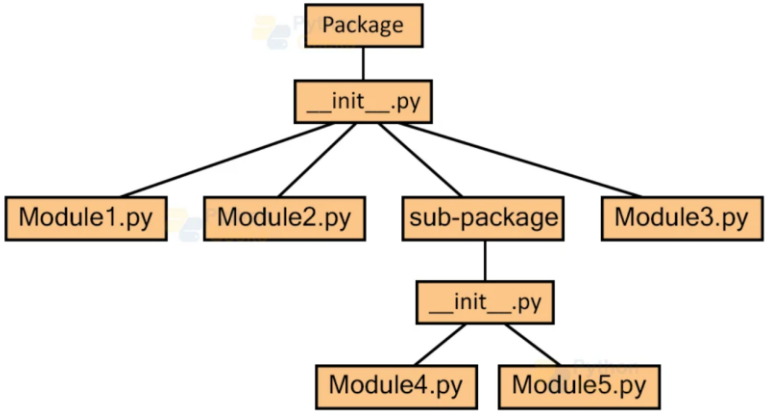
✓ Package

- A **package** is a directory that groups related modules together.
- **package** is a **collection of Python modules in a folder**.
- It can contain:
  - Multiple modules
  - Sub-packages (nested directories)
  - An __init__.py file (marks the directory as a package)
- Helps organize large projects in a structured way.

```
mypackage/
│
├── __init__.py
├── arithmetic.py
└── geometry.py
```

```
from mypackage import arithmetic
from mypackage.geometry import area_circle
```

Structure of Packages




```
project/
│
├── main.py
├── utils/
│   ├── __init__.py
│   ├── math_utils.py
│   └── string_utils.py
```

```
from utils.math_utils import add
from utils.string_utils import capitalize_text
```

Working with modules and packages (Contd..)

Organizing code into package consisting of hierarchical collection of modules.

- Make sure that every directory defines an __init__.py file.
- The __init__.py file is used to define a package and can contain optional initialization code.

```
graphics/
    __init__.py
    primitive/
        __init__.py
        line.py
        fill.py
        text.py
    formats/
        __init__.py
        png.py
        jpg.py
```

- Performing various import statements as follow:

```
import graphics.primitive.line
from graphics.primitive import line
import graphics.formats.jpg as jpg
```

Working with modules and packages (Contd..)

✓ Importing Package Submodules Using Relative Names

```
mypackage/
    __init__.py
    A/
        __init__.py
        spam.py
        grok.py
    B/
        __init__.py
        bar.py
```

- To import a module from the **same directory**:

```
# mypackage/A/spam.py

from . import grok
```

- To import a module from a **parent package's subdirectory**:

```
# mypackage/A/spam.py

from ..B import bar
```

- The **.** refers to the current package level.
- The **..** refers to the parent package level.
- Relative imports do **not** include the top-level package name.

Working with modules and packages (Contd..)

✓ Importing Package Submodules Using Relative Names

```
mypackage/
    __init__.py
    A/
        __init__.py
        spam.py
        grok.py
    B/
        __init__.py
        bar.py
```

```
# mypackage/A/spam.py

from mypackage.A import grok        # OK
from . import grok                  # OK
import grok                         # Error (not found)
```

- Relative imports work **only inside proper packages**.
  - They do **not work in top-level scripts**.
- They fail if a package module is run **directly as a script**.
- Example where relative imports fail:
  - **python3 mypackage/A/spam.py**
- Relative imports work when the module is run using the **-m option**:
  - **python3 -m mypackage.A.spam**
- **-m** tells Python to treat the module as **part of a package**.

Working with modules and packages (Contd..)

Reloading Modules

- Used when you want to reload a module after changing its source code.

```
>>> import spam
>>> import imp
>>> imp.reload(spam)
<module 'spam' from './spam.py'>
>>>
```

- **reload()** does not update definitions that have been imported using stat ments such as **from module import name.**

```
# spam.py

def bar():
    print('bar')

def grok():
    print('grok')
```

```
>>> import spam
>>> from spam import grok
>>> spam.bar()
bar

>>> grok()
grok
>>>
```

Working with modules and packages (Contd..)

- Without quitting Python, go edit the source code to spam.py so that the function grok() looks like this:

```
def grok():
    print('New grok')
```

- Now go back to the interactive session, perform a reload, and try this experiment:

```
>>> import imp
>>> imp.reload(spam)
<module 'spam' from './spam.py'>
>>> spam.bar()
bar
>>> grok()              # Notice old output
grok
>>> spam.grok()         # Notice new output
New grok
>>>
```

  - This results in **two versions of the same function** in memory
  - Reloading does **not** fix existing objects, classes, or instances
  - Mixing old and new code leads to subtle, hard-to-debug bugs
  - Avoid reloading modules in production
  - Use reload only for interactive/debug sessions
  - Prefer **import spam** over **from spam import ...** when experimenting
  - Restarting the interpreter is the safest solution

Working with modules and packages (Contd..)

Making a Directory or Zip File Runnable As a Main Script

- If your application program has grown into multiple files, you can put it into its own directory and add a **__main__.py** file.

```
myapplication/
        spam.py
        bar.py
        grok.py
        __main__.py
```

- Run the Python interpreter on the top-level directory.

```
bash % python3 myapplication
```

- The interpreter will execute the __main__.py file as the main program.\
- This technique also works if you package all of your code up into a zip file.

```
bash % ls
spam.py    bar.py   grok.py   __main__.py
bash % zip -r myapp.zip *.py
bash % python3 myapp.zip
... output from __main__.py ...
```

Working with modules and packages (Contd..)

Installing Packages

- Python has a per-user installation directory that's typically located in a directory such as ~/.local/lib/python3.3/site-packages.
- To force packages to install in this directory, give the --user option to the installation command.

```
python3 setup.py install --user
```

or

```
pip install --user packagename
```

Working with modules and packages (Contd..)

✓ Creating a New Python Environment

- Virtual environments are isolated environments that allow you to manage dependencies for different projects without conflicts.
- **Creating a Virtual Environment**
```
python -m venv myenv
```
- **Activating the Virtual Environment**
  - **On Windows**
```
myenv\Scripts\activate
```
  - **On macOS and Linux**
```
source myenv/bin/activate
```
- **Installing Dependencies in the Virtual Environment:**
```
pip install -r requirements.txt
```
- **Deactivating the Virtual Environment:**
```
deactivate
```

Virtual environments help maintain clean and isolated development spaces, ensuring that dependencies for one project do not interfere with those of another.

Working with modules and packages (Contd..)

✓ Distributing Packages

- Typical library package

```
projectname/
      README.txt
      Doc/
          documentation.txt
      projectname/
         __init__.py
         foo.py
         bar.py
         utils/
              __init__.py
              spam.py
              grok.py
      examples/
         helloworld.py
         ...
```

- To make the package something that you can distribute, first write a setup.py file that looks like this:

```
# setup.py
from distutils.core import setup

setup(name='projectname',
      version='1.0',
      author='Your Name',
      author_email='you@youraddress.com',
      url='http://www.you.com/projectname',
      packages=['projectname', 'projectname.utils'],
)
```

Working with modules and packages (Contd..)

- Next, make a file MANIFEST.in that lists various nonsource files that you want to include in your package:

```
# MANIFEST.in
include *.txt
recursive-include examples *
recursive-include Doc *
```

- Make sure the setup.py and MANIFEST.in files appear in the top-level directory of your package.
- Once you have done this, you should be able to make a source distribution by typing a command such as this:

```
% bash python3 setup.py sdist
```

- This will create a file such as projectname-1.0.zip or projectname-1.0.tar.gz, depending on the platform.