

MODULE-4

Modules

In Python, a **module** is essentially a file that contains Python code—this could be **functions**, **classes**, **variables**, or even runnable statements—that you can **reuse across different programs**. Think of a module as a **toolbox**: it keeps related tools together so you can pick and use only what you need without rewriting code.

Using modules has several advantages:

1. **Code organization:** Large programs can be broken into smaller, manageable pieces.
2. **Reusability:** Once a module is written, it can be reused in multiple programs.
3. **Namespace separation:** Each module maintains its own **namespace**, preventing variable name conflicts.

In Python, the filename of the module is also its **namespace identifier**, meaning that when you import it, Python uses the filename (without the .py extension) as the reference for all its variables and functions.

Python Standard Library Modules

Python comes with a rich **standard library**—a collection of modules included with Python that provide ready-made functionality, saving you from reinventing the wheel.

1. **math**
 - Provides common mathematical functions and constants such as `sqrt()`, `sin()`, `cos()`, `pi`, and `e`.
 - Pure functions—calculating `sqrt(2)` doesn't depend on any previous calculations.
2. **random**
 - Generates pseudo-random numbers for simulations, games, testing, or security purposes.
 - Includes functions like `randint()`, `randrange()`, `random()`, `shuffle()`, and `sample()`.
3. **time**
 - Provides tools for handling time, including `sleep()` for pausing execution and `perf_counter()` for high-resolution timing.
 - Useful for measuring code performance or creating delays.

4. **turtle**

- A module for simple graphics and drawing.
- Ideal for beginners to understand coordinates, loops, and animation.

5. **string**

- Contains useful constants and helper functions for string operations, like `string.ascii_letters` and `string.punctuation`.

Example – Using a Standard Module:

```
import math
radius = 5
area = math.pi * radius**2
print(f'Area of the circle: {area}')
```

Creating Your Own Module

You can create a **custom module** to keep your functions organized. Steps:

1. **Save a Python file with a .py extension.**

Example: `seqtools.py`

```
# seqtools.py
def remove_at(index, string):
    return string[:index] + string[index+1:]
```

2. **Import your module in another program:**

```
import seqtools
result = seqtools.remove_at(2, "Hello")
print(result) # Output: 'Helo'
```

Explanation:

- `seqtools` is the module namespace.
- `remove_at` is a function inside the module.

- By using `seqtools.remove_at()`, you **fully qualify** the function call, preventing naming conflicts with functions of the same name in other modules.

Module Organization Best Practices

1. Group related functions together

- For example, functions for string manipulation go in a single module, functions for math operations in another.

2. Use meaningful names for modules

- Avoid generic names like `module1.py`. Use descriptive names like `file_utils.py` or `string_ops.py`.

3. Avoid running code at import

- Modules can contain executable statements, but usually, it's better to only define functions/classes.
- Use the `if __name__ == "__main__":` guard for test code that should only run when the module is executed directly.

```
# seqtools.py
def remove_at(index, string):
    return string[:index] + string[index+1:]
if __name__ == "__main__":
    print(remove_at(2, "Hello")) # Runs only if executed directly
```

Namespaces and Avoiding Conflicts

Every module has its **own namespace**, which means:

- Variables inside moduleA don't clash with variables in moduleB.
- You can have functions or variables with the same name in different modules safely.

Example:

```
# module1.py
question = "What is Python?"

# module2.py
question = "Explain modules."

import module1
import module2

print(module1.question) # What is Python?
print(module2.question) # Explain modules.
```

Even though both modules have a variable called question, their namespaces keep them **isolated**, avoiding collisions.

Random Numbers

Purpose

Random numbers are widely used in programming for tasks such as:

1. **Games and simulations:** To simulate dice rolls, shuffle cards, or generate random positions in a game.
2. **Statistical simulations:** For example, estimating rainfall patterns or Monte Carlo simulations in science and finance.
3. **Testing and debugging:** Generating randomized inputs to test the behavior of algorithms.
4. **Security and cryptography:** Random numbers help in generating secure passwords, session tokens, and cryptographic keys.

Python's random module does not generate truly random numbers but **pseudo-random numbers**. These are **deterministic sequences based on an initial seed value**, which can be repeated if needed.

Functions in the random Module

1. randrange(start, stop[, step])

Generates a random integer within a specified range, similar to Python's range():

```
import random

# Random number between 1 and 10 (10 excluded)
num = random.randrange(1, 10)

print(num)
```

```
# Random odd number less than 100
odd_num = random.randrange(1, 100, 2)

print(odd_num)
```

Explanation:

- The lower bound is included, the upper bound is excluded.
- Optional step lets you skip numbers, e.g., randrange(1, 100, 2) picks only odd numbers.

2. random.random()

Generates a **floating-point number** in the range [0.0, 1.0):

```
r = random.random()

print(r) # Example output: 0.6394267984578837
```

- Use this function when you need **fractions**, probabilities, or normalized random numbers.
- To scale to a different range [a, b), use:

```
scaled = a + (b - a) * random.random()
```

3. random.shuffle(list)

Shuffles a list **in-place**, randomly rearranging its elements:

```
cards = ["A", "K", "Q", "J"]
```

```
random.shuffle(cards)
```

```
print(cards) # Example output: ['J', 'A', 'K', 'Q']
```

- Useful in **games** (shuffling cards) or **simulations**.
- **Important:** Works **in-place**, so the original list is modified.

4. random.sample(population, k)

Picks **k unique elements** from a sequence or set:

```
lottery_numbers = random.sample(range(1, 50), 6)
```

```
print(lottery_numbers) # Example: [7, 23, 1, 42, 15, 36]
```

- Unlike randrange, sample ensures **no duplicates**.
- Perfect for lotteries, random draws, or selecting unique test samples.

Seeding for Repeatability

Random number generators are **deterministic**. The seed() function initializes the generator so that it produces the **same sequence of numbers every time**:

```
import random
```

```
random.seed(42) # Initialize with a known seed
```

```
print(random.randint(1, 100)) # Output: 82
```

```
print(random.randint(1, 100)) # Output: 15
```

- **Useful for debugging and unit testing:** You can reproduce exact scenarios to identify errors.
- For true randomness, you can omit the seed or use system time: random.seed().

Examples

1. Simulating Dice Rolls

```
dice_rolls = [random.randint(1, 6) for _ in range(5)]
```

```
print(dice_rolls) # Could contain duplicates
```

- Duplicates are allowed because rolling a die multiple times can give the same result.

2. Selecting Lottery Numbers Without Repetition

```
lottery = random.sample(range(1, 50), 6)
print(lottery) # Always 6 unique numbers
```

3. Shuffling a Deck of Cards

```
deck = ['A', '2', '3', '4', '5', '6', '7', '8', '9', '10', 'J', 'Q', 'K']
random.shuffle(deck)
print(deck)
```

4. Random Choices for Simulation

```
enemy_positions = [(random.randint(0, 100), random.randint(0, 100)) for _ in range(10)]
print(enemy_positions) # Positions of 10 enemies in a game grid
```

Time Module

Purpose

The Python time module provides tools for handling **time-related tasks** in programs. It is commonly used for:

1. Measuring program execution time:

- Essential for analyzing the **performance of algorithms**.
- Allows comparison between custom solutions and optimized or built-in functions.
- Useful in profiling code for optimization.

2. Creating delays or pauses in program execution:

- Important in **simulations, games, animations, or timed processes**.
- Enables throttling loops or network requests to prevent overwhelming resources.

3. Scheduling tasks and measuring intervals:

- Provides high-resolution timers for precise measurement in scientific and real-time applications.

1. time.perf_counter()

- Returns a **floating-point number** representing the **current value of a high-resolution timer** in seconds.
- Highly precise, including **time elapsed during sleep**.
- Ideal for **benchmarking code**, as it provides more accurate measurements than time.time().

Example – Timing a Code Block:

```
import time

start = time.perf_counter() # Start timer
sum_val = sum(range(1000000)) # Code to measure
end = time.perf_counter() # End timer
print(f"Elapsed time: {end - start} seconds")
```

Explanation:

- The timer measures the **real elapsed time**, including any system or OS delays.
- The difference end - start represents how long the code took to execute.
- Useful for comparing **custom implementations** (loops, functions) with **Python built-ins** like sum().

Practical Example

```
import time

def custom_sum(n):
    total = 0
    for i in range(n):
        total += i
    return total

# Built-in sum
start = time.perf_counter()
sum(range(1000000))
```

```
end = time.perf_counter()
print(f"Built-in sum: {end - start} seconds")

# Custom sum
start = time.perf_counter()
custom_sum(1000000)
end = time.perf_counter()
print(f"Custom sum: {end - start} seconds")
```

- This demonstrates how **profiling helps identify faster methods**.
- In this example, the built-in sum function is typically much faster than a manual loop.

2. time.sleep(seconds)

- Suspends program execution for the **given number of seconds**.
- Accepts **floating-point numbers**, e.g., time.sleep(0.5) for half a second.
- Useful for **pausing loops, controlling timing, or delaying output**.

Example – Countdown Timer:

```
import time
print("Countdown starts:")
for i in range(5, 0, -1):
    print(i)
    time.sleep(1) # Wait 1 second
print("Go!")
```

Explanation:

- sleep is especially useful for **animations or event-driven programs** where you want a delay between steps.
- Can be used in **network scripts** to respect API rate limits.

Math Module

Purpose

The Python math module is a built-in library that provides a **comprehensive set of mathematical functions and constants**. It is designed to:

1. Perform **numerical calculations** efficiently and accurately.
2. Provide access to **mathematical constants** like π (pi) and e (e).
3. Support **trigonometric, logarithmic, and exponential calculations**.
4. Enable **angle conversions** between degrees and radians.
5. Offer **pure functions**, meaning the output depends only on the input and not on any internal state or history of previous calls.

Use Case Examples:

- Engineering calculations (mechanical, electrical, or civil).
- Physics simulations (angles, forces, motion).
- Financial modeling (exponentials, logarithms).
- Data science and machine learning (distances, trigonometric features, normalization).

Key Constants

The math module provides several widely used constants:

1. **math.pi** – Represents the value of π (~3.141592653589793).
2. **math.e** – Represents Euler's number (~2.718281828459045).

Example:

```
import math  
  
print("Value of pi:", math.pi)  
  
print("Value of e:", math.e)
```

- Constants are **predefined and precise**, avoiding the need to manually type them.
- Useful in **circle calculations, exponential growth models, and scientific formulas**.

Functions

1. Square Root

- `math.sqrt(x)` → Returns the square root of x.

```
print(math.sqrt(25)) # Output: 5.0
```

2. Trigonometric Functions

- `math.sin(x)`, `math.cos(x)`, `math.tan(x)`
- **Input must be in radians**, not degrees.

```
angle_degrees = 90
```

```
angle_radians = math.radians(angle_degrees) # Convert degrees to radians
```

```
print(math.sin(angle_radians)) # Output: 1.0
```

Explanation:

- $\text{Radians} = (\pi / 180) \times \text{Degrees}$
- Conversion functions:
 - `math.radians(degrees)` → Convert degrees to radians
 - `math.degrees(radians)` → Convert radians to degrees

Practical Use Case:

- Calculating angles in physics simulations or computer graphics.

3. Logarithmic and Exponential Functions

- `math.log(x)` → Natural logarithm (base e)
- `math.log10(x)` → Logarithm base 10
- `math.exp(x)` → Exponential e^x

```
print(math.log(math.e)) # Output: 1.0
```

```
print(math.log10(100)) # Output: 2.0
```

```
print(math.exp(2)) # Output: 7.38905609893065
```

Applications:

- Modeling growth/decay processes in biology, finance, or physics.

4. Rounding Functions

- `math.ceil(x)` → Rounds up to the nearest integer
- `math.floor(x)` → Rounds down to the nearest integer

```
print(math.ceil(4.2)) # Output: 5
```

```
print(math.floor(4.8)) # Output: 4
```

Pure Functions

- All functions in the `math` module are **pure**:
 - They produce the same output for the same input.
 - They do **not depend on internal state** (unlike `random` or `turtle`).
- Advantage: **predictable results**, safe for repeated calculations or multithreading environments.

Example:

```
x = 16
```

```
print(math.sqrt(x)) # Always outputs 4.0
```

Namespaces

Definition

A **namespace** in Python is a **container that holds a mapping of names to objects**. Essentially, it keeps track of all the variable names, function names, class names, and module names in a particular context and what objects they refer to.

- Think of a namespace as a **directory of names**, where each name points to a specific object in memory.
- Python uses namespaces to **avoid conflicts** when multiple parts of a program use the same names.

Types of Namespaces

Namespaces are **created at different times and have different lifetimes**:

1. Module Namespace

Each Python module has its own namespace.

Variables, functions, and classes defined in a module are stored in that module's namespace.

Example:

```
import module1
import module2

print(module1.question) # Access question from module1
print(module2.question) # Access question from module2
```

Here, both module1 and module2 can have a variable named question without conflict because they belong to **different module namespaces**.

2. Function Namespace

Every function creates a new local namespace.

Variables defined inside a function are **local to that function** and cannot be accessed from outside.

Example:

```
def greet():
    message = "Hello"
    print(message)

greet()

# print(message) # This would raise an error because 'message' is local to greet()
```

3. Class Namespace

Each class has its own namespace.

Attributes and methods belong to the class namespace.

Example:

```
class Dog:
```

```
species = "Canine" # Class attribute

def __init__(self, name):
    self.name = name # Instance attribute

print(Dog.species) # Access class attribute

fido = Dog("Fido")

print(fido.name) # Access instance attribute
```

4. Built-in Namespace

- Python provides a built-in namespace containing names like len, print, range, etc.
- These names are always available unless overridden locally.

Why Namespaces Are Important

1. Avoid naming collisions

- Multiple modules or functions can use the same variable names without overwriting each other's values.
- Example:
- `module1.answer = 42`
- `module2.answer = "Python"`
- `print(module1.answer) # 42`
- `print(module2.answer) # Python`

2. Organize code logically

- Namespaces help separate different parts of a program, improving readability and maintainability.

3. Support modular programming

- Modules, classes, and functions each have isolated namespaces, making large projects manageable.

Scope and Lookup Rules (LEGB)

Python uses namespaces to **determine the scope of a variable**. The lookup order follows the **LEGB rule**:

1. **Local (L):** Names defined inside the current function.
2. **Enclosing (E):** Names in enclosing functions (nested functions).
3. **Global (G):** Names defined at the top-level of the module.
4. **Built-in (B):** Names pre-defined in Python (like len, range).

Example – Scope Lookup:

```
x = 10 # Global

def outer():
    x = 20 # Enclosing
    def inner():
        x = 30 # Local
        print(x) # Prints 30 (local takes precedence)
    inner()
outer()
print(x) # Prints 10 (global)
```

Attributes and the Dot Operator

- The **dot operator (.)** is used to access variables, functions, and classes within a namespace.
- Example:

```
import math
print(math.pi)    # Access pi in math module
print(math.sqrt(16)) # Access sqrt function
```

- Fully-qualified names clearly indicate which namespace the name belongs to, avoiding ambiguity.

Practical Examples of Namespaces

1. Multiple modules with same variable names

```
# module1.py
value = 100

# module2.py
value = 200

# main.py
import module1, module2
print(module1.value) # 100
print(module2.value) # 200
```

2. Functions and variable isolation

```
def add_numbers():
    total = 0
    for i in range(5):
        total += i
    return total

# total is not accessible here; it is local to add_numbers()
```

3. Classes and instance vs class attributes

```
class Circle:
    pi = 3.14159 # Class attribute
    def __init__(self, radius):
        self.radius = radius # Instance attribute

c1 = Circle(5)
print(c1.radius) # 5
print(Circle.pi) # 3.14159
```

Mutable vs Immutable Objects and Aliasing

Mutable Objects

- Mutable objects can **change their contents after creation**.
- Examples in Python:
 - **List**: You can add, remove, or modify elements.
 - **Dictionary**: You can add, remove, or update key-value pairs.
 - **Set**: You can add or remove elements.
- Mutability is useful for **dynamic data structures**, but it requires care to avoid unintended side effects.

Immutable Objects

- Immutable objects **cannot be changed after creation**.
- Examples in Python:
 - **Tuple**: Fixed-size, ordered collection.
 - **String**: Text cannot be changed; methods return new strings.
 - **Frozenset**: Immutable version of a set.
- Immutability provides **safety** and **predictability**—you can use them as dictionary keys or set elements because their value cannot change.

Aliasing

Aliasing occurs when **two or more variables reference the same object in memory**. Changes made through one variable affect all other aliases pointing to the same object.

Example: List Aliasing

```
list_one = [1, 2, 3]
list_two = list_one # Both refer to the same list
list_two.append(4)
```

```
print(list_one) # Output: [1, 2, 3, 4]
```

```
print(list_two) # Output: [1, 2, 3, 4]
```

- Here, `list_one` is **modified** even though we used `list_two` to append an element.
- This is because **both variables point to the same memory location**.

3. Solutions to Avoid Unintended Aliasing

Shallow Copy

- Creates a **new object** but the elements inside are still references to the original elements.
- Works for **flat lists**, but not nested structures.

```
list_one = [1, 2, 3]
```

```
list_two = list_one.copy() # Shallow copy
```

```
list_two.append(4)
```

```
print(list_one) # [1, 2, 3]
```

```
print(list_two) # [1, 2, 3, 4]
```

Deep Copy

- For **nested structures** (lists of lists, dictionaries containing lists), use `copy.deepcopy()`.
- This creates a **completely independent copy**, including all nested objects.

```
import copy
```

```
nested_list = [[1, 2], [3, 4]]
```

```
nested_copy = copy.deepcopy(nested_list)
```

```
nested_copy[0].append(99)
```

```
print(nested_list) # [[1, 2], [3, 4]]
```

```
print(nested_copy) # [[1, 2, 99], [3, 4]]
```

Sets and Frozensets in Python

Sets

- A **set** is an **unordered collection of unique elements**.
- It is **mutable**, which means you can **add or remove elements** after creation.
- Sets **do not maintain order**; the elements may appear in any order when printed.
- Duplicate elements are automatically **discarded**.
- Syntax:

```
my_set = {1, 2, 3} # Curly braces
```

```
another_set = set([4, 5, 6]) # Using set() constructor
```

Frozensets

- A **frozenset** is like a set but **immutable**.
- Once created, its contents **cannot be changed** (no adding or removing elements).
- Useful for using as **keys in dictionaries** or elements of other sets because it is hashable.
- Syntax:

```
frozen = frozenset([1, 2, 3])
```

Operations on Sets

- **Membership Testing**

Check if an element exists in a set:

```
my_set = {1, 2, 3}
```

```
print(2 in my_set) # True
```

```
print(5 in my_set) # False
```

- **Adding Elements**

Add a single element:

```
my_set.add(4)
```

```
print(my_set) # {1, 2, 3, 4}
```

- **Removing Elements**

Remove an element; raises `KeyError` if not present:

```
my_set.remove(3)
```

```
print(my_set) # {1, 2, 4}
```

- Alternatively, `discard()` removes the element **without error** if it doesn't exist:

```
my_set.discard(10) # No error
```

Set Operations

Sets support **mathematical operations**:

1. **Union** – combines all unique elements:

```
a = {1, 2, 3}
```

```
b = {3, 4, 5}
```

```
print(a | b) # {1, 2, 3, 4, 5}
```

2. **Intersection** – elements common to both sets:

```
print(a & b) # {3}
```

3. **Difference** – elements in first set but not in second:

```
print(a - b) # {1, 2}
```

4. **Symmetric Difference** – elements in either set, but not in both:

```
print(a ^ b) # {1, 2, 4, 5}
```

Frozenset Operations

- Frozensets support most **set operations** like union, intersection, difference:

```
fs1 = frozenset([1, 2, 3])
```

```
fs2 = frozenset([3, 4, 5])
```

```
print(fs1 | fs2) # {1, 2, 3, 4, 5}
```

- **Cannot modify** frozensets:

```
fs1.add(6) # AttributeError: 'frozenset' object has no attribute 'add'
```



Object-Oriented Programming (OOP)

- **Definition:** Python is an **object-oriented programming language**, which means it supports **object-oriented programming (OOP)**.
- **History:**
 - OOP originated in the 1960s.
 - Became widely used in the 1980s to manage **large and complex software systems**.
 - Purpose: Simplify the design, modification, and maintenance of complex programs.
- **Procedural vs. Object-Oriented Programming:**
 - **Procedural Programming:** Focuses on functions or procedures that operate on data.
 - **Object-Oriented Programming:** Focuses on **objects** which contain both **data and methods** (functions) together.
 - Examples in Python: Turtle, str, random — these are objects we have already worked with.
- **Real-world analogy:**
 - Each object represents a real-world entity or concept.
 - Functions (methods) define how the object behaves or interacts with other objects.

User-Defined Compound Data Types

- Python has built-in classes like int, float, str, and Turtle.
- You can also create your **own classes**.

Example: Representing a Point

- A **point** in 2D space has two coordinates: (x, y).
- Operations we might want on points:
 - Distance from origin or another point
 - Midpoint between points
 - Check if a point lies within a shape (rectangle, circle, etc.)

Ways to Represent a Point in Python

1. **Tuple:** (x, y) — simple, quick, but limited functionality.
2. **Custom Class:** Provides **attributes** and **methods**, more flexible and organized.

Defining a Point Class

- **Syntax:**

class Point:

```
    """A simple class to represent a point in 2D space"""
```

```
    def __init__(self):
```

```
        self.x = 0
```

```
        self.y = 0
```

Explanation:

- class Point: → starts the class definition.
- Docstring """...""" → describes the class.
- __init__(self) → **initializer method**, called automatically when a new object is created.
 - self → refers to the **new object being created**.
 - Attributes x and y initialized to 0.

Creating Objects from a Class

```
p = Point() # Create a new Point object
```

```
q = Point() # Create another Point object
```

```
print(p.x, p.y) # Output: 0 0
```

```
print(q.x, q.y) # Output: 0 0
```

Explanation:

- Point() → constructor function that creates a new object.
- Each object (p and q) has its **own attributes** (x and y).

- The class itself is **like a factory**:
 - It is not an object itself.
 - It provides machinery to **produce objects**.
 - Each time the constructor is called, a new object is made and initialized.

Attributes in Python Classes

What Are Attributes?

- **Attributes** are variables that belong to an object (instance of a class).
- They represent the **state or properties** of the object, much like characteristics of a real-world object:
 - Example: A Point object has x and y coordinates.
 - A Car object might have color, model, and speed.

Accessing and Modifying Attributes

- **Dot Notation:** object.attribute
- Example:

```
class Point:
```

```
    def __init__(self):
```

```
        self.x = 0
```

```
        self.y = 0
```

```
p = Point()
```

```
print(p.x, p.y) # Output: 0 0
```

```
# Modifying attributes
```

```
p.x = 5
```

```
p.y = 7
```

```
print(p.x, p.y) # Output: 5 7
```

Explanation:

- `self.x` and `self.y` are attributes of the object `p`.
- `p.x` refers **specifically** to the `x` attribute of the object `p`, avoiding conflicts with other variables in your program.
- You can **use attributes in expressions**:

```
distance_squared = p.x**2 + p.y**2  
print(distance_squared) # Output: 74
```

Namespaces and Dot Notation

- Each object has its **own namespace**: the set of attribute names unique to it.
- Dot notation (`p.x`) tells Python exactly which namespace to look in.

Example:

```
x = 10    # Global variable  
p = Point()  
p.x = 3   # Attribute of object p  
print(x)  # Output: 10  
print(p.x) # Output: 3
```

- **Key Idea:** The attribute `p.x` and the global variable `x` can coexist without conflict.

Improving the Initializer (`__init__`)

Why Improve the Initializer?

- Without parameters, you need multiple steps to set the object's state:

```
p = Point()  
p.x = 7  
p.y = 6
```

- Better approach: allow the constructor to **accept initial values** for attributes.

Parameterized Constructor

class Point:

```
def __init__(self, x=0, y=0):  
    """Initialize a Point with coordinates x and y"""  
    self.x = x  
    self.y = y
```

Creating points

```
p1 = Point()    # Defaults to (0,0)  
p2 = Point(7, 6) # Coordinates set to (7,6)  
print(p1.x, p1.y) # Output: 0 0  
print(p2.x, p2.y) # Output: 7 6
```

Explanation:

- $x=0$ and $y=0$ are **default values**.
- If arguments are provided, they **override the defaults**.
- Initialization happens automatically when the object is created.

Technical Note: The `__init__` method doesn't create the object—it just **initializes it** after creation. Creation and initialization usually happen together when you call the class like a function.

Adding Methods to the Class

Why Methods?

- **Methods** are functions defined inside a class that operate on the object.
- They allow objects to **perform operations relevant to them**.

Example: Distance from Origin

```
import math  
class Point:  
    def __init__(self, x=0, y=0):
```

```
self.x = x
self.y = y
def distance_from_origin(self):
    """Calculate distance from the origin (0,0)"""
    return math.sqrt(self.x**2 + self.y**2)
p = Point(3, 4)
print(p.distance_from_origin()) # Output: 5.0
```

Explanation:

- distance_from_origin() accesses the object's x and y via self.
- Provides **behavior specific to the object**.
- This is not possible with a simple tuple (3, 4).

Other Useful Methods for Points

You can define multiple methods in the class:

1. Move the point:

```
def move(self, dx, dy):
    """Move point by dx and dy"""
    self.x += dx
    self.y += dy
```

2. Find midpoint with another point:

```
def midpoint(self, other):
    """Return midpoint between self and another Point"""
    return Point((self.x + other.x)/2, (self.y + other.y)/2)
```

3. Display point:

```
def display(self):
    print(f"Point({self.x}, {self.y})")
```

Complete Enhanced Point Class

```
import math

class Point:
    def __init__(self, x=0, y=0):
        """Initialize a Point at coordinates (x, y)"""
        self.x = x
        self.y = y
    def distance_from_origin(self):
        return math.sqrt(self.x**2 + self.y**2)
    def move(self, dx, dy):
        self.x += dx
        self.y += dy
    def midpoint(self, other):
        return Point((self.x + other.x)/2, (self.y + other.y)/2)
    def display(self):
        print(f'Point({self.x}, {self.y})')

# Example usage
p1 = Point(3, 4)
p2 = Point(7, 6)
p1.display()           # Point(3, 4)
print(p1.distance_from_origin())  # 5.0
p1.move(2, 3)
p1.display()           # Point(5, 7)
mid = p1.midpoint(p2)
mid.display()           # Point(6.0, 6.5)
```

