Complete Python Classes Tutorial

Table of Contents

Introduction to Classes

What is a Class? A class is a blueprint or template for creating objects. Think of it as a cookie cutter - you can use it to make many cookies (objects) with the same shape and properties.

What is an Object? An object is an instance of a class. It's a specific entity created from the class blueprint with actual values.

Real-world analogy:

- Class = Car blueprint
- Object = Actual car (Toyota Camry, Honda Civic, etc.)

Creating Your First Class

python

```python
# Basic class definition
class Dog:
    pass  # Empty class for now

# Creating objects (instances) of the class
dog1 = Dog()
dog2 = Dog()

print(type(dog1))  # <class '__main__.Dog'>
print(type(dog2))  # <class '__main__.Dog'>
```

Class Naming Convention:

- Use PascalCase (first letter of each word capitalized)
- Examples: `Dog`, `BankAccount`, `StudentRecord`

Constructor Method (init)

The `__init__` method is automatically called when an object is created. It's used to initialize the object's attributes.

python

```python
class Dog:
    def __init__(self, name, breed, age):
        self.name = name      # Instance variable
        self.breed = breed    # Instance variable
        self.age = age        # Instance variable

# Creating objects with initial values
dog1 = Dog("Buddy", "Golden Retriever", 3)
dog2 = Dog("Max", "German Shepherd", 5)

print(f"{dog1.name} is a {dog1.breed}, {dog1.age} years old")
print(f"{dog2.name} is a {dog2.breed}, {dog2.age} years old")
```

Output:

```
Buddy is a Golden Retriever, 3 years old
Max is a German Shepherd, 5 years old
```

Instance Variables vs Class Variables

python

```python
class Dog:
    # Class variable (shared by all instances)
    species = "Canis lupus"
    total_dogs = 0

    def __init__(self, name, breed, age):
        # Instance variables (unique to each instance)
        self.name = name
        self.breed = breed
        self.age = age
        Dog.total_dogs += 1  # Increment class variable

# Creating objects
dog1 = Dog("Buddy", "Golden Retriever", 3)
dog2 = Dog("Max", "German Shepherd", 5)

# Accessing class variables
print(f"Species: {Dog.species}")
print(f"Total dogs created: {Dog.total_dogs}")

# Accessing instance variables
print(f"Dog 1: {dog1.name}")
print(f"Dog 2: {dog2.name}")
```

Key Differences:

- **Instance variables**: Unique to each object, defined with `self.variable_name`
- **Class variables**: Shared by all objects of the class, defined directly in the class

---

Methods in Classes

Methods are functions defined inside a class that operate on the object's data.

python

```python
class Dog:
    def __init__(self, name, breed, age):
        self.name = name
        self.breed = breed
        self.age = age
        self.energy = 100

    def bark(self):
        return f"{self.name} says Woof!"

    def play(self):
        if self.energy > 20:
            self.energy -= 20
            return f"{self.name} is playing! Energy: {self.energy}"
        else:
            return f"{self.name} is too tired to play."

    def sleep(self):
        self.energy = 100
        return f"{self.name} is sleeping and restored energy to {self.energy}"

# Using methods
dog = Dog("Buddy", "Golden Retriever", 3)
print(dog.bark())
print(dog.play())
print(dog.play())
print(dog.sleep())
```

The self Parameter

`self` refers to the current instance of the class. It's used to access instance variables and methods.

python

```python
class Person:
    def __init__(self, name, age):
        self.name = name  # self.name refers to this instance's name
        self.age = age    # self.age refers to this instance's age

    def introduce(self):
        # self allows us to access this instance's attributes
        return f"Hi, I'm {self.name} and I'm {self.age} years old"

    def have_birthday(self):
        self.age += 1  # Modify this instance's age
        return f"Happy birthday {self.name}! Now {self.age} years old"

person1 = Person("Alice", 25)
person2 = Person("Bob", 30)

print(person1.introduce())  # Uses person1's data
print(person2.introduce())  # Uses person2's data
```

Important: You don't pass `self` when calling methods - Python does it automatically!

Encapsulation and Access Modifiers

Encapsulation means hiding internal details and providing controlled access to data.

python

```python
class BankAccount:
    def __init__(self, account_number, initial_balance):
        self.account_number = account_number    # Public
        self._balance = initial_balance         # Protected (convention)
        self.__pin = "1234"                    # Private (name mangling)

    def deposit(self, amount):
        if amount > 0:
            self._balance += amount
            return f"Deposited ${amount}. New balance: ${self._balance}"
        return "Invalid deposit amount"

    def withdraw(self, amount):
        if 0 < amount <= self._balance:
            self._balance -= amount
            return f"Withdrew ${amount}. New balance: ${self._balance}"
        return "Insufficient funds or invalid amount"

    def get_balance(self):
        return self._balance

    def __str__(self):
        return f"Account {self.account_number}: ${self._balance}"

# Usage
account = BankAccount("12345", 1000)
print(account.deposit(500))
print(account.withdraw(200))
print(account.get_balance())

# Direct access (not recommended for protected/private)
print(account._balance)           # Accessible but not recommended
# print(account.__pin)            # This would cause an AttributeError
print(account._BankAccount__pin) # Name mangling - still accessible but discouraged
```

Access Levels:

- **Public**: `variable` - Accessible from anywhere
- **Protected**: `_variable` - Should only be accessed within class and subclasses
- **Private**: `__variable` - Name mangled, harder to access from outside

---

Inheritance

Inheritance allows a class to inherit attributes and methods from another class.

python

```python
# Parent class (Base class)
class Animal:
    def __init__(self, name, species):
        self.name = name
        self.species = species

    def make_sound(self):
        return f"{self.name} makes a sound"

    def info(self):
        return f"{self.name} is a {self.species}"

# Child class (Derived class)
class Dog(Animal):
    def __init__(self, name, breed):
        super().__init__(name, "Dog")  # Call parent constructor
        self.breed = breed

    def make_sound(self):  # Override parent method
        return f"{self.name} barks: Woof!"

    def fetch(self):  # New method specific to Dog
        return f"{self.name} is fetching the ball!"

class Cat(Animal):
    def __init__(self, name, breed):
        super().__init__(name, "Cat")
        self.breed = breed

    def make_sound(self):  # Override parent method
        return f"{self.name} meows: Meow!"

    def climb(self):  # New method specific to Cat
        return f"{self.name} is climbing a tree!"

# Usage
dog = Dog("Buddy", "Golden Retriever")
cat = Cat("Whiskers", "Persian")

print(dog.info())        # Inherited method
print(dog.make_sound())  # Overridden method
print(dog.fetch())       # Dog-specific method
print(cat.info())        # Inherited method
print(cat.make_sound())  # Overridden method
print(cat.climb())       # Cat-specific method
```

Method Overriding

Method overriding allows a child class to provide a specific implementation of a method that's already defined in the parent class.

python

```python
class Vehicle:
    def __init__(self, make, model):
        self.make = make
        self.model = model

    def start(self):
        return "Vehicle is starting..."

    def stop(self):
        return "Vehicle is stopping..."

class Car(Vehicle):
    def start(self):  # Override parent method
        return f"{self.make} {self.model} engine is starting with a key"

class ElectricCar(Vehicle):
    def start(self):  # Override parent method
        return f"{self.make} {self.model} is starting silently (electric)"

# Usage
regular_car = Car("Toyota", "Camry")
electric_car = ElectricCar("Tesla", "Model 3")

print(regular_car.start())   # Uses Car's version
print(electric_car.start())  # Uses ElectricCar's version
print(regular_car.stop())    # Uses inherited method
```

Super() Function

`super()` is used to call methods from the parent class.

```python
class Rectangle:
    def __init__(self, width, height):
        self.width = width
        self.height = height

    def area(self):
        return self.width * self.height

    def perimeter(self):
        return 2 * (self.width + self.height)

class Square(Rectangle):
    def __init__(self, side):
        super().__init__(side, side)  # Call parent constructor
        self.side = side

    def area(self):
        # We could calculate differently, but let's use parent method
        return super().area()

    def diagonal(self):
        return (self.side * 2) ** 0.5

# Usage
square = Square(5)
print(f"Area: {square.area()}")
print(f"Perimeter: {square.perimeter()}")
print(f"Diagonal: {square.diagonal():.2f}")
```

Multiple Inheritance

A class can inherit from multiple parent classes.

```python
class Flyable:
    def fly(self):
        return "Flying through the air!"

class Swimmable:
    def swim(self):
        return "Swimming through water!"

class Duck(Flyable, Swimmable):
    def __init__(self, name):
        self.name = name

    def quack(self):
        return f"{self.name} says: Quack!"

# Usage
duck = Duck("Donald")
print(duck.quack())
print(duck.fly())    # From Flyable
print(duck.swim())   # From Swimmable

# Check inheritance
print(Duck.__mro__)  # Method Resolution Order
```

Method Resolution Order (MRO): Determines which method is called when there are multiple inheritance paths.

Polymorphism

Polymorphism allows objects of different classes to be treated as objects of a common base class.

python

```python
class Shape:
    def area(self):
        pass

    def perimeter(self):
        pass

class Circle(Shape):
    def __init__(self, radius):
        self.radius = radius

    def area(self):
        return 3.14159 * self.radius ** 2

    def perimeter(self):
        return 2 * 3.14159 * self.radius

class Rectangle(Shape):
    def __init__(self, width, height):
        self.width = width
        self.height = height

    def area(self):
        return self.width * self.height

    def perimeter(self):
        return 2 * (self.width + self.height)

# Polymorphic function
def print_shape_info(shape):
    print(f"Area: {shape.area():.2f}")
    print(f"Perimeter: {shape.perimeter():.2f}")
    print("-" * 20)

# Usage - same function works with different object types
shapes = [
    Circle(5),
    Rectangle(4, 6),
    Circle(3)
]
```

```
for shape in shapes:
    print_shape_info(shape)  # Polymorphism in action!
```

Special Methods (Magic Methods)

Special methods (dunder methods) allow you to define how objects behave with built-in functions and operators.

python

```python
class Book:
    def __init__(self, title, author, pages):
        self.title = title
        self.author = author
        self.pages = pages

    def __str__(self):  # Called by str() and print()
        return f"'{self.title}' by {self.author}"

    def __repr__(self):  # Developer representation
        return f"Book('{self.title}', '{self.author}', {self.pages})"

    def __len__(self):  # Called by len()
        return self.pages

    def __eq__(self, other):  # Called by ==
        if isinstance(other, Book):
            return self.title == other.title and self.author == other.author
        return False

    def __lt__(self, other):  # Called by <
        if isinstance(other, Book):
            return self.pages < other.pages
        return NotImplemented

    def __add__(self, other):  # Called by +
        if isinstance(other, Book):
            combined_title = f"{self.title} & {other.title}"
            combined_author = f"{self.author} & {other.author}"
            combined_pages = self.pages + other.pages
            return Book(combined_title, combined_author, combined_pages)
        return NotImplemented

# Usage
book1 = Book("Python Basics", "John Doe", 300)
book2 = Book("Advanced Python", "Jane Smith", 450)
book3 = Book("Python Basics", "John Doe", 300)

print(book1)                # Uses __str__
print(repr(book1))          # Uses __repr__
print(len(book1))           # Uses __len__
print(book1 == book3)       # Uses __eq__
print(book1 < book2)        # Uses __lt__
```

```
combined_book = book1 + book2   # Uses __add__
print(combined_book)
```

Common Magic Methods:

- `__init__`: Constructor
- `__str__`: String representation for users
- `__repr__`: String representation for developers
- `__len__`: Length of object
- `__eq__`, `__lt__`, `__gt__`: Comparison operators
- `__add__`, `__sub__`, `__mul__`: Arithmetic operators

Class Methods and Static Methods

python

```python
class Student:
    school_name = "Python Academy"  # Class variable
    total_students = 0

    def __init__(self, name, grade):
        self.name = name
        self.grade = grade
        Student.total_students += 1

    # Instance method (regular method)
    def study(self, subject):
        return f"{self.name} is studying {subject}"

    # Class method - works with class, not instance
    @classmethod
    def get_school_info(cls):
        return f"School: {cls.school_name}, Total Students: {cls.total_students}"

    # Class method as alternative constructor
    @classmethod
    def from_string(cls, student_string):
        name, grade = student_string.split("-")
        return cls(name, int(grade))  # Create new instance

    # Static method - doesn't need class or instance
    @staticmethod
    def is_passing_grade(grade):
        return grade >= 60

# Usage
# Regular instance creation
student1 = Student("Alice", 85)
student2 = Student("Bob", 92)

# Using class method
print(Student.get_school_info())

# Using class method as alternative constructor
student3 = Student.from_string("Charlie-78")
print(f"{student3.name}: {student3.grade}")

# Using static method
print(f"Is 75 passing? {Student.is_passing_grade(75)}")
print(f"Is 45 passing? {Student.is_passing_grade(45)}")

# Static methods can be called on instances too
print(f"Is Alice passing? {student1.is_passing_grade(student1.grade)}")
```

Method Types:

- **Instance Method**: Takes `self`, works with instance data
- **Class Method**: Takes `cls`, works with class data, can create instances
- **Static Method**: Takes neither, utility function related to the class

---

Properties and Getters/Setters

Properties provide a way to customize access to instance attributes.

python

```python
class Temperature:
    def __init__(self, celsius=0):
        self._celsius = celsius

    @property
    def celsius(self):
        return self._celsius

    @celsius.setter
    def celsius(self, value):
        if value < -273.15:
            raise ValueError("Temperature cannot be below absolute zero")
        self._celsius = value

    @property
    def fahrenheit(self):
        return (self._celsius * 9/5) + 32

    @fahrenheit.setter
    def fahrenheit(self, value):
        self.celsius = (value - 32) * 5/9  # Uses celsius setter for validation

    @property
    def kelvin(self):
        return self._celsius + 273.15

    @kelvin.setter
    def kelvin(self, value):
        self.celsius = value - 273.15  # Uses celsius setter for validation

# Usage
temp = Temperature(25)
print(f"Celsius: {temp.celsius}")
print(f"Fahrenheit: {temp.fahrenheit}")
print(f"Kelvin: {temp.kelvin}")

# Setting temperature using different scales
temp.fahrenheit = 100
print(f"After setting to 100°F: {temp.celsius}°C")

temp.kelvin = 300
print(f"After setting to 300K: {temp.celsius}°C")
```

```python
# Validation in action
try:
    temp.celsius = -300  # This will raise an error
except ValueError as e:
    print(f"Error: {e}")
```

Abstract Classes

Abstract classes cannot be instantiated and are meant to be inherited by other classes.

python

```python
from abc import ABC, abstractmethod

class Animal(ABC):  # Abstract base class
    def __init__(self, name):
        self.name = name

    @abstractmethod
    def make_sound(self):  # Must be implemented by subclasses
        pass

    @abstractmethod
    def move(self):  # Must be implemented by subclasses
        pass

    def sleep(self):  # Regular method - can be inherited as-is
        return f"{self.name} is sleeping"

class Dog(Animal):
    def make_sound(self):  # Must implement this
        return f"{self.name} barks: Woof!"

    def move(self):  # Must implement this
        return f"{self.name} runs on four legs"

class Bird(Animal):
    def make_sound(self):  # Must implement this
        return f"{self.name} chirps: Tweet!"

    def move(self):  # Must implement this
        return f"{self.name} flies with wings"

# Usage
dog = Dog("Buddy")
bird = Bird("Tweety")

print(dog.make_sound())
print(dog.move())
print(dog.sleep())  # Inherited method

print(bird.make_sound())
print(bird.move())
print(bird.sleep())  # Inherited method
```

```python
# This would cause an error:
# animal = Animal("Generic")  # TypeError: Can't instantiate abstract class
```

Practical Examples

Example 1: Library Management System

python

```python
from datetime import datetime, timedelta

class Book:
    def __init__(self, isbn, title, author, total_copies):
        self.isbn = isbn
        self.title = title
        self.author = author
        self.total_copies = total_copies
        self.available_copies = total_copies
        self.borrowed_by = []  # List of (member_id, due_date) tuples

    def is_available(self):
        return self.available_copies > 0

    def borrow(self, member_id):
        if self.is_available():
            self.available_copies -= 1
            due_date = datetime.now() + timedelta(days=14)
            self.borrowed_by.append((member_id, due_date))
            return due_date
        return None

    def return_book(self, member_id):
        for i, (borrower_id, due_date) in enumerate(self.borrowed_by):
            if borrower_id == member_id:
                self.available_copies += 1
                self.borrowed_by.pop(i)
                return True
        return False

class Member:
    def __init__(self, member_id, name, email):
        self.member_id = member_id
        self.name = name
        self.email = email
        self.borrowed_books = []  # List of ISBN numbers

    def borrow_book(self, book):
        if book.is_available():
            due_date = book.borrow(self.member_id)
            if due_date:
                self.borrowed_books.append(book.isbn)
                return f"Book borrowed successfully. Due date: {due_date.strftime('%Y-%m-%d')}"
        return "Book not available"

    def return_book(self, book):
        if book.isbn in self.borrowed_books:
            if book.return_book(self.member_id):
                self.borrowed_books.remove(book.isbn)
                return "Book returned successfully"
        return "Book not borrowed by this member"

class Library:
    def __init__(self, name):
        self.name = name
        self.books = {}  # ISBN -> Book object
        self.members = {}  # member_id -> Member object

    def add_book(self, book):
        self.books[book.isbn] = book

    def add_member(self, member):
        self.members[member.member_id] = member

    def find_book(self, isbn):
        return self.books.get(isbn)

    def find_member(self, member_id):
        return self.members.get(member_id)

# Usage
library = Library("City Library")

# Add books
book1 = Book("978-0134685991", "Effective Python", "Brett Slatkin", 3)
book2 = Book("978-0596009259", "Head First Design Patterns", "Freeman & Robson", 2)

library.add_book(book1)
library.add_book(book2)

# Add members
member1 = Member("M001", "Alice Johnson", "alice@email.com")
member2 = Member("M002", "Bob Smith", "bob@email.com")

library.add_member(member1)
library.add_member(member2)
```

```python
# Borrow books
print(member1.borrow_book(book1))
print(member2.borrow_book(book1))

# Check availability
print(f"Available copies of '{book1.title}': {book1.available_copies}")
```

Example 2: Simple Game Character System

```
python
```

```python
import random

class Character:
    def __init__(self, name, health, attack_power):
        self.name = name
        self.max_health = health
        self.health = health
        self.attack_power = attack_power
        self.level = 1

    def attack(self, target):
        damage = random.randint(self.attack_power - 5, self.attack_power + 5)
        damage = max(1, damage)  # Minimum 1 damage
        target.take_damage(damage)
        return f"{self.name} attacks {target.name} for {damage} damage!"

    def take_damage(self, damage):
        self.health -= damage
        self.health = max(0, self.health)  # Health can't go below 0

    def heal(self, amount):
        self.health += amount
        self.health = min(self.max_health, self.health)  # Can't exceed max health

    def is_alive(self):
        return self.health > 0

    def level_up(self):
        self.level += 1
        self.max_health += 20
        self.health = self.max_health
        self.attack_power += 5
        return f"{self.name} leveled up to level {self.level}!"

class Warrior(Character):
    def __init__(self, name):
        super().__init__(name, health=120, attack_power=25)
        self.armor = 10

    def take_damage(self, damage):
        # Warriors have armor that reduces damage
        reduced_damage = max(1, damage - self.armor)
        super().take_damage(reduced_damage)
    def shield_bash(self, target):
        # Special warrior ability
        damage = self.attack_power // 2
        target.take_damage(damage)
        return f"{self.name} shield bashes {target.name} for {damage} damage!"

class Mage(Character):
    def __init__(self, name):
        super().__init__(name, health=80, attack_power=30)
        self.mana = 100

    def fireball(self, target):
        # Special mage ability
        if self.mana >= 20:
            self.mana -= 20
            damage = self.attack_power * 2
            target.take_damage(damage)
            return f"{self.name} casts fireball on {target.name} for {damage} damage!"
        return f"{self.name} doesn't have enough mana!"

    def heal_spell(self):
        if self.mana >= 15:
            self.mana -= 15
            heal_amount = 30
            self.heal(heal_amount)
            return f"{self.name} heals for {heal_amount} health!"
        return f"{self.name} doesn't have enough mana!"

# Usage
warrior = Warrior("Sir Lancelot")
mage = Mage("Gandalf")

print(f"{warrior.name}: Health={warrior.health}, Attack={warrior.attack_power}")
print(f"{mage.name}: Health={mage.health}, Attack={mage.attack_power}")

# Combat simulation
print("\n--- Battle Begin ---")
print(warrior.attack(mage))
print(f"{mage.name} health: {mage.health}")

print(mage.fireball(warrior))
print(f"{warrior.name} health: {warrior.health}")
print(warrior.shield_bash(mage))
print(f"{mage.name} health: {mage.health}")

print(mage.heal_spell())
print(f"{mage.name} health: {mage.health}")
```

Best Practices

1. Class Design Principles

python

```
# Good: Single Responsibility Principle
class EmailSender:
    def send_email(self, to, subject, body):
        # Only responsible for sending emails
        pass

class EmailValidator:
    def validate_email(self, email):
        # Only responsible for validating emails
        pass

# Bad: Class doing too many things
class EmailManager:
    def send_email(self, to, subject, body):
        pass

    def validate_email(self, email):
        pass

    def log_email(self, email_data):
        pass

    def encrypt_email(self, email):
        pass
```

2. Naming Conventions

python

```python
# Good naming
class BankAccount:
    def __init__(self, account_number, initial_balance):
        self.account_number = account_number
        self._balance = initial_balance  # Protected
        self.__pin = None                # Private

    def deposit_money(self, amount):
        pass

    def withdraw_money(self, amount):
        pass

    def get_current_balance(self):
        return self._balance

# Use descriptive names
class CustomerOrderProcessor:
    def process_online_order(self, order):
        pass
```

3. Documentation and Type Hints

python

```python
from typing import List, Optional

class Student:
    """
    Represents a student in the school system.

    Attributes:
        student_id (str): Unique identifier for the student
        name (str): Full name of the student
        grades (List[float]): List of grades for the student
    """

    def __init__(self, student_id: str, name: str) -> None:
        """
        Initialize a new student.

        Args:
            student_id: Unique identifier for the student
            name: Full name of the student
        """
        self.student_id = student_id
        self.name = name
        self.grades: List[float] = []

    def add_grade(self, grade: float) -> None:
        """Add a grade to the student's record."""
        if 0 <= grade <= 100:
            self.grades.append(grade)
        else:
            raise ValueError("Grade must be between 0 and 100")

    def get_average_grade(self) -> Optional[float]:
        """
        Calculate the average grade.

        Returns:
            The average grade, or None if no grades exist.
        """
        if not self.grades:
            return None
        return sum(self.grades) / len(self.grades)
```

4. Error Handling

python

```python
class BankAccount:
    def __init__(self, account_number: str, initial_balance: float):
        if initial_balance < 0:
            raise ValueError("Initial balance cannot be negative")

        self.account_number = account_number
        self._balance = initial_balance

    def withdraw(self, amount: float) -> str:
        if amount <= 0:
            raise ValueError("Withdrawal amount must be positive")

        if amount > self._balance:
            raise ValueError("Insufficient funds")

        self._balance -= amount
        return f"Withdrew ${amount}. New balance: ${self._balance}"

    def deposit(self, amount: float) -> str:
        if amount <= 0:
            raise ValueError("Deposit amount must be positive")

        self._balance += amount
```