Advanced Python Programming

LESSON 4: Metaprogramming & Decorators

Learning Objectives:

By the end of this lesson, participants will be able to:

- Understand functions and classes as first-class objects in Python.
- Write function decorators with and without parameters.
- Create class decorators for modifying class behavior.
- Use functools.wraps and inspect module for metadata preservation and reflection.

Lesson Outline:

I. Functions and Classes as First-Class Objects (10 min)

In Python, functions and classes are first-class objects, meaning they can be passed around, stored in variables, and manipulated like any other object.

Functions as objects:

- Can be assigned to variables
- Can be passed as arguments to other functions
- Can be returned from functions
- Can be stored in data structures
- Have attributes and methods

Basic function manipulation:

```python
def greet(name):
    """A simple greeting function."""
    return f"Hello, {name}!"

def shout(name):
    """A loud greeting function."""
    return f"HELLO, {name.upper()}!!!"

# Functions are objects - they can be assigned to variables
my_func = greet
print(my_func("Alice"))  # Hello, Alice!

# Functions have attributes
print(greet.__name__)      # greet
print(greet.__doc__)       # A simple greeting function.
print(greet.__module__)    # __main__

# Functions can be stored in data structures
greetings = {
    'polite': greet,
```

```
    'loud': shout
}

for style, func in greetings.items():
    print(f"{style}: {func('Bob')}")
```

Higher-order functions:

```python
def apply_operation(func, value):
    """Apply a function to a value."""
    return func(value)

def double(x):
    return x * 2

def square(x):
    return x ** 2

# Functions as arguments
result1 = apply_operation(double, 5)  # 10
result2 = apply_operation(square, 5)  # 25

print(f"Double: {result1}, Square: {result2}")

# Functions returning functions
def make_multiplier(factor):
    """Return a function that multiplies by factor."""
    def multiplier(value):
        return value * factor
    return multiplier

times_three = make_multiplier(3)
times_ten = make_multiplier(10)

print(f"3 * 7 = {times_three(7)}")    # 21
print(f"10 * 7 = {times_ten(7)}")     # 70
```

Commentary:

Understanding functions as first-class objects is fundamental to metaprogramming. This property enables decorators, callbacks, and functional programming patterns.

II. Function Decorators: Basics and Patterns (15 min)

Decorators are a powerful way to modify or extend function behavior without changing the function's code.

Basic decorator pattern:

```python
import time
import functools

def timer(func):
    """Decorator that times function execution."""
    @functools.wraps(func)
    def wrapper(*args, **kwargs):
        start_time = time.time()
        result = func(*args, **kwargs)
        end_time = time.time()
        print(f"{func.__name__} took {end_time - start_time:.4f} seconds")
        return result
    return wrapper

@timer
def slow_function():
    """A function that takes some time to execute."""
    time.sleep(1)
    return "Done!"

# Usage
result = slow_function()  # Prints timing information
print(result)

# Without decorator syntax (equivalent)
def another_slow_function():
    time.sleep(0.5)
    return "Also done!"

timed_function = timer(another_slow_function)
result2 = timed_function()
```

Multiple decorators and execution order:

```python
def bold(func):
    """Wrap result in bold tags."""
    @functools.wraps(func)
    def wrapper(*args, **kwargs):
        result = func(*args, **kwargs)
        return f"<b>{result}</b>"
    return wrapper

def italic(func):
    """Wrap result in italic tags."""
    @functools.wraps(func)
    def wrapper(*args, **kwargs):
        result = func(*args, **kwargs)
        return f"<i>{result}</i>"
    return wrapper
```

```
@bold
@italic
def get_message():
    return "Hello, World!"

# Execution order: bold(italic(get_message))
print(get_message())  # <b><i>Hello, World!</i></b>

# Demonstrates the decorator chain
print(f"Function name: {get_message.__name__}")  # get_message (preserved
by @wraps)
```

Decorators with state:

```
def counter(func):
    """Decorator that counts function calls."""
    @functools.wraps(func)
    def wrapper(*args, **kwargs):
        wrapper.calls += 1
        print(f"{func.__name__} has been called {wrapper.calls} times")
        return func(*args, **kwargs)

    wrapper.calls = 0
    return wrapper

@counter
def say_hello(name):
    return f"Hello, {name}!"

# Each call increments the counter
say_hello("Alice")  # say_hello has been called 1 times
say_hello("Bob")    # say_hello has been called 2 times
say_hello("Charlie") # say_hello has been called 3 times

print(f"Total calls: {say_hello.calls}")
```

Commentary:

Decorators provide a clean way to add cross-cutting concerns like logging, timing, validation, and caching without cluttering the main function logic.

III. Parameterized Decorators (10 min)

Parameterized decorators take arguments and return a decorator, providing more flexibility.

Basic parameterized decorator:

```
def retry(max_attempts=3, delay=1):
    """Decorator that retries a function on failure."""
    def decorator(func):
        @functools.wraps(func)
        def wrapper(*args, **kwargs):
            last_exception = None

            for attempt in range(max_attempts):
                try:
                    return func(*args, **kwargs)
                except Exception as e:
                    last_exception = e
                    if attempt < max_attempts - 1:
                        print(f"Attempt {attempt + 1} failed: {e}.
Retrying in {delay}s...")
                        time.sleep(delay)
                    else:
                        print(f"All {max_attempts} attempts failed.")

            raise last_exception
        return wrapper
    return decorator

# Usage with parameters
@retry(max_attempts=3, delay=0.5)
def unreliable_function():
    """Function that fails randomly."""
    import random
    if random.random() < 0.7:  # 70% chance of failure
        raise ConnectionError("Network error")
    return "Success!"

# try:
#     result = unreliable_function()
#     print(result)
# except Exception as e:
#     print(f"Final failure: {e}")
```

Validation decorator with parameters:

```
def validate(*validators):
    """Decorator that validates function arguments."""
    def decorator(func):
        @functools.wraps(func)
        def wrapper(*args, **kwargs):
            # Validate positional arguments
            for i, (arg, validator) in enumerate(zip(args, validators)):
                if not validator(arg):
                    raise ValueError(f"Argument {i} failed validation:
{arg}")
```

```
            return func(*args, **kwargs)
        return wrapper
    return decorator

# Validator functions
def is_positive(x):
    return isinstance(x, (int, float)) and x > 0

def is_string(x):
    return isinstance(x, str) and len(x) > 0

@validate(is_positive, is_string)
def create_user(age, name):
    """Create a user with validated input."""
    return f"User {name}, age {age}"

# Valid usage
user1 = create_user(25, "Alice")
print(user1)  # User Alice, age 25

# Invalid usage would raise ValueError
# user2 = create_user(-5, "Bob")    # Negative age
# user3 = create_user(30, "")       # Empty name
```

Caching decorator with TTL:

```
import time
from collections import defaultdict

def cache_with_ttl(ttl_seconds=60):
    """Decorator that caches results with time-to-live."""
    def decorator(func):
        cache = {}
        timestamps = {}

        @functools.wraps(func)
        def wrapper(*args, **kwargs):
            # Create cache key
            key = str(args) + str(sorted(kwargs.items()))
            current_time = time.time()

            # Check if cached result is still valid
            if (key in cache and
                key in timestamps and
                current_time - timestamps[key] < ttl_seconds):
                print(f"Cache hit for {func.__name__}")
                return cache[key]

            # Compute and cache result
            print(f"Cache miss for {func.__name__}")
```

```python
            result = func(*args, **kwargs)
            cache[key] = result
            timestamps[key] = current_time

            return result

        # Add cache inspection methods
        wrapper.cache_info = lambda: {
            'size': len(cache),
            'keys': list(cache.keys())
        }
        wrapper.cache_clear = lambda: (cache.clear(), timestamps.clear())

        return wrapper
    return decorator

@cache_with_ttl(ttl_seconds=2)
def expensive_computation(x, y):
    """Simulate an expensive computation."""
    time.sleep(1)  # Simulate work
    return x ** y

# Usage
print(expensive_computation(2, 10))  # Cache miss (takes ~1s)
print(expensive_computation(2, 10))  # Cache hit (immediate)
time.sleep(3)  # Wait for TTL to expire
print(expensive_computation(2, 10))  # Cache miss again
```

Commentary:

Parameterized decorators provide reusable patterns for common functionality like retries, validation, and caching with configurable behavior.

IV. Class Decorators (10 min)

Class decorators modify or enhance entire classes, providing a powerful way to add functionality across all methods or modify class behavior.

Basic class decorator:

```python
def add_repr(cls):
    """Class decorator that adds a __repr__ method."""
    def __repr__(self):
        class_name = self.__class__.__name__
        attrs = ', '.join(f"{k}={v!r}" for k, v in self.__dict__.items())
        return f"{class_name}({attrs})"

    cls.__repr__ = __repr__
    return cls
```

```python
@add_repr
class Point:
    def __init__(self, x, y):
        self.x = x
        self.y = y

@add_repr
class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age

# Usage
p1 = Point(10, 20)
p2 = Person("Alice", 30)

print(p1)  # Point(x=10, y=20)
print(p2)  # Person(name='Alice', age=30)
```

Method decoration class decorator:

```python
def log_methods(cls):
    """Class decorator that adds logging to all methods."""
    for attr_name in dir(cls):
        attr = getattr(cls, attr_name)

        # Only decorate callable methods (not special methods)
        if (callable(attr) and
            not attr_name.startswith('__') and
            not attr_name.endswith('__')):

            def make_logged_method(method, name):
                @functools.wraps(method)
                def logged_method(self, *args, **kwargs):
                    print(f"Calling {cls.__name__}.{name} with args=
{args}, kwargs={kwargs}")
                    result = method(self, *args, **kwargs)
                    print(f"{cls.__name__}.{name} returned {result}")
                    return result
                return logged_method

            logged_method = make_logged_method(attr, attr_name)
            setattr(cls, attr_name, logged_method)

    return cls

@log_methods
class Calculator:
    def add(self, a, b):
        return a + b
```

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

# Usage
calc = Calculator()
result1 = calc.add(5, 3)      # Logs method call and return
result2 = calc.multiply(4, 7) # Logs method call and return
```

Singleton class decorator:

```python
def singleton(cls):
    """Class decorator that makes a class a singleton."""
    instances = {}

    @functools.wraps(cls)
    def get_instance(*args, **kwargs):
        if cls not in instances:
            instances[cls] = cls(*args, **kwargs)
        return instances[cls]

    return get_instance

@singleton
class DatabaseConnection:
    def __init__(self, host="localhost"):
        self.host = host
        self.connected = False
        print(f"Creating database connection to {host}")

    def connect(self):
        self.connected = True
        print(f"Connected to {self.host}")

# Usage
db1 = DatabaseConnection("server1")  # Creates instance
db2 = DatabaseConnection("server2")  # Returns same instance

print(f"Same instance: {db1 is db2}")  # True
print(f"Host: {db1.host}")             # server1 (from first creation)
```

Commentary:

Class decorators provide a clean way to modify class behavior without inheritance, making them ideal for adding cross-cutting functionality.

V. Advanced Metaprogramming with inspect (10 min)

The inspect module provides powerful tools for examining live objects and extracting metadata.

Function signature inspection:

```
import inspect

def analyze_function(func):
    """Analyze and display function metadata."""
    print(f"Function: {func.__name__}")
    print(f"Module: {func.__module__}")
    print(f"Doc: {func.__doc__}")

    # Get function signature
    sig = inspect.signature(func)
    print(f"Signature: {sig}")

    # Analyze parameters
    for name, param in sig.parameters.items():
        print(f"  Parameter '{name}':")
        print(f"    Kind: {param.kind}")
        print(f"    Default: {param.default}")
        print(f"    Annotation: {param.annotation}")

    # Return annotation
    if sig.return_annotation != inspect.Signature.empty:
        print(f"Return annotation: {sig.return_annotation}")

def sample_function(a: int, b: str = "default", *args, **kwargs) -> str:
    """A sample function for inspection."""
    return f"{a}: {b}"

analyze_function(sample_function)
```

Dynamic function call with signature validation:

```
def call_with_validation(func, *args, **kwargs):
    """Call function with signature validation."""
    sig = inspect.signature(func)

    try:
        # Bind arguments to parameters
        bound_args = sig.bind(*args, **kwargs)
        bound_args.apply_defaults()

        print(f"Calling {func.__name__} with validated arguments:")
        for name, value in bound_args.arguments.items():
            print(f"  {name} = {value}")

        return func(*bound_args.args, **bound_args.kwargs)
```

```python
    except TypeError as e:
        print(f"Signature validation failed: {e}")
        return None

def greet_user(name: str, age: int, greeting: str = "Hello") -> str:
    return f"{greeting}, {name}! You are {age} years old."

# Valid call
result1 = call_with_validation(greet_user, "Alice", 25)
print(f"Result: {result1}")

# Invalid call (missing required argument)
result2 = call_with_validation(greet_user, "Bob")  # Missing age
```

Class inspection and dynamic method creation:

```python
def auto_property_class(cls):
    """Class decorator that creates properties for all attributes."""

    # Find all attributes that don't start with underscore
    for attr_name in dir(cls):
        if not attr_name.startswith('_') and not callable(getattr(cls, 
attr_name)):
            # Create getter and setter for this attribute
            def make_property(name):
                private_name = f"_{name}"

                def getter(self):
                    return getattr(self, private_name, None)

                def setter(self, value):
                    print(f"Setting {name} to {value}")
                    setattr(self, private_name, value)

                return property(getter, setter)

            # Replace attribute with property
            setattr(cls, attr_name, make_property(attr_name))

    return cls

@auto_property_class
class User:
    def __init__(self, name, email):
        self.name = name
        self.email = email

# Usage
user = User("Alice", "alice@example.com")
print(f"Name: {user.name}")  # Calls getter
```

```python
user.email = "newemail@example.com"  # Calls setter with logging
print(f"Email: {user.email}")
```

Stack frame inspection for debugging:

```python
def debug_trace():
    """Print debug information about the current call stack."""
    frame = inspect.currentframe()

    try:
        # Get the caller's frame
        caller_frame = frame.f_back

        print("Debug trace:")
        print(f"  Function: {caller_frame.f_code.co_name}")
        print(f"  File: {caller_frame.f_code.co_filename}")
        print(f"  Line: {caller_frame.f_lineno}")
        print(f"  Local variables: {caller_frame.f_locals}")

        # Walk up the call stack
        current_frame = caller_frame
        level = 1

        while current_frame.f_back and level < 3:
            current_frame = current_frame.f_back
            print(f"  Caller {level}: {current_frame.f_code.co_name} "
                  f"at line {current_frame.f_lineno}")
            level += 1

    finally:
        del frame  # Prevent reference cycles

def business_logic(x, y):
    """Some business logic that might need debugging."""
    intermediate = x * 2
    debug_trace()  # Call our debug function
    return intermediate + y

def main():
    result = business_logic(5, 3)
    print(f"Result: {result}")

# main()  # Uncomment to see debug trace
```

Commentary:

The inspect module enables powerful metaprogramming patterns for debugging, validation, and dynamic code generation while maintaining type safety and clear documentation.

VI. Real-World Metaprogramming Applications (10 min)

Let's explore practical applications of metaprogramming in production systems.

API endpoint decorator with automatic documentation:

```
from typing import Dict, Any
import json

class APIRegistry:
    """Registry for API endpoints with automatic documentation."""

    def __init__(self):
        self.endpoints = {}

    def endpoint(self, path: str, method: str = "GET"):
        """Decorator for registering API endpoints."""
        def decorator(func):
            # Extract function metadata
            sig = inspect.signature(func)

            endpoint_info = {
                'function': func,
                'path': path,
                'method': method.upper(),
                'parameters': {},
                'return_type': sig.return_annotation,
                'docstring': func.__doc__
            }

            # Analyze parameters
            for name, param in sig.parameters.items():
                endpoint_info['parameters'][name] = {
                    'type': param.annotation,
                    'default': param.default if param.default != 
inspect.Parameter.empty else None,
                    'required': param.default == inspect.Parameter.empty
                }

            self.endpoints[path] = endpoint_info

            @functools.wraps(func)
            def wrapper(*args, **kwargs):
                print(f"Calling API endpoint: {method} {path}")
                return func(*args, **kwargs)

            return wrapper
        return decorator

    def generate_docs(self) -> str:
        """Generate API documentation."""
```

```
        docs = ["API Documentation", "=" * 18, ""]

        for path, info in self.endpoints.items():
            docs.append(f"{info['method']} {path}")
            docs.append("-" * (len(path) + len(info['method']) + 1))

            if info['docstring']:
                docs.append(f"Description: {info['docstring']}")

            docs.append("Parameters:")
            for param_name, param_info in info['parameters'].items():
                required = "required" if param_info['required'] else "optional"
                docs.append(f"  - {param_name} ({param_info['type']}) - {required}")
                if param_info['default'] is not None:
                    docs.append(f"    Default: {param_info['default']}")

            docs.append("")

        return "\n".join(docs)

# Usage
api = APIRegistry()

@api.endpoint("/users", "GET")
def get_users(limit: int = 10, offset: int = 0) -> Dict[str, Any]:
    """Retrieve a list of users."""
    return {"users": [], "total": 0, "limit": limit, "offset": offset}

@api.endpoint("/users", "POST")
def create_user(name: str, email: str, age: int = None) -> Dict[str, Any]:
    """Create a new user."""
    return {"id": 123, "name": name, "email": email, "age": age}

# Generate documentation
print(api.generate_docs())
```

ORM-style model decorator:

```
class ModelRegistry:
    """Simple ORM-style model registry."""

    models = {}

    @classmethod
    def model(cls, table_name: str):
        """Class decorator for registering models."""
        def decorator(model_cls):
            # Add model metadata
            model_cls._table_name = table_name
```

```
            model_cls._fields = {}

            # Analyze class annotations for fields
            for field_name, field_type in getattr(model_cls,
'__annotations__', {}).items():
                if not field_name.startswith('_'):
                    model_cls._fields[field_name] = field_type

            # Add ORM methods
            def save(self):
                print(f"Saving {model_cls.__name__} to table
'{table_name}'")
                for field, value in self.__dict__.items():
                    if field in model_cls._fields:
                        print(f"  {field}: {value}")
                return self

            def to_dict(self):
                return {field: getattr(self, field, None)
                       for field in model_cls._fields}

            model_cls.save = save
            model_cls.to_dict = to_dict

            # Register model
            cls.models[table_name] = model_cls

            return model_cls
        return decorator

# Usage
@ModelRegistry.model("users")
class User:
    name: str
    email: str
    age: int

    def __init__(self, name, email, age):
        self.name = name
        self.email = email
        self.age = age

# Create and save a user
user = User("Alice", "alice@example.com", 30)
user.save()
print(f"User data: {user.to_dict()}")

# Check registered models
print(f"Registered models: {list(ModelRegistry.models.keys())}")
```

Commentary:

These examples demonstrate how metaprogramming enables elegant solutions for common patterns like API registration, ORM functionality, and automatic documentation generation.

VII. Recap & Best Practices (5 min)

Key Takeaways:

- Functions and classes are first-class objects that can be manipulated programmatically.
- Decorators provide a clean way to add cross-cutting functionality.
- Parameterized decorators offer configurable behavior for reusable patterns.
- Class decorators can modify entire classes without inheritance.
- The inspect module enables powerful runtime introspection and validation.

Best Practices:

- Use `@functools.wraps` to preserve function metadata in decorators.
- Keep decorators simple and focused on single responsibilities.
- Document decorator behavior clearly, especially for parameterized decorators.
- Use type hints with inspect for better validation and documentation.
- Consider performance implications of runtime introspection.
- Prefer explicit over implicit behavior in metaprogramming.

Common Pitfalls:

- Forgetting to use `@functools.wraps` in decorators.
- Creating overly complex parameterized decorators.
- Not handling edge cases in dynamic code generation.
- Overusing metaprogramming where simple solutions would suffice.

When to Use Metaprogramming:

- Cross-cutting concerns (logging, timing, validation).
- Framework and library development.
- Automatic code generation and documentation.
- When you need to modify behavior of many functions/classes consistently.

Final Multiple-Choice Question:

What is the PRIMARY purpose of using `@functools.wraps` in a decorator?

A. To make the decorator run faster B. To preserve the original function's metadata (**name**, **doc**, etc.) C. To allow the decorator to accept parameters D. To enable the decorator to be used on classes

(Answer: B. To preserve the original function's metadata (**name**, **doc**, etc.) - this ensures that introspection and debugging tools work correctly with decorated functions.)