Uppala Sanjay Kumar
244161008

Unlocking Python's Power: Lambda Functions, Decorators, and Dunder Methods

A DEEP DIVE INTO PYTHON'S FUNCTIONAL AND OBJECT-ORIENTED FEATURES

Overview of topics

- **Introduction** to Python's functional and object-oriented features.
- **Lambda Functions**: Basics, advanced usage, and best practices.
- **Decorators**: Patterns, real-world applications, and customizations.
- **Dunder (Magic) Methods**: Customizing object behavior.

Introduction to Python's Functional and Object-Oriented Magic

- The Dual Nature of Python : functional and object-oriented

➢ **Lambda Functions**: Write compact, throwaway functions for simple tasks.

➢ **Decorators**: Add reusable functionality to functions or classes.

➢ **Dunder Methods**: Customize object behavior and interaction with Python syntax.

Takeaway:
"These tools let us write powerful, Pythonic code with simplicity and control."

Lambda Functions

- One-line anonymous functions useful for short, throwaway functions.

```
lambda arguments: expression
```

  - Basic example used for quick inline transformation

```
add = lambda x, y: x + y
print(add(2, 3))  # Output: 5
```

  - **Common Use Cases**: commonly employed in scenarios requiring quick computations such as sorting, filtering or applying functions to data collections

  - Promotes ***concise*** coding making code cleaner and improving readability and maintainability

Lambda Functions

```python
@log_action
def sort_tasks(self, key=lambda task: task.priority):
    """
    Sorts the tasks based on a given key function.
    :param key: A lambda function that defines the sorting criteria (default is by priority).
    """
    self.tasks.sort(key=key)

@log_action
def filter_tasks(self, condition=lambda task: not task.completed):
    """
    Filters tasks based on a given condition.
    :param condition: A lambda function that defines the filtering criteria (default is to find incomplete tasks).
    :return: A list of tasks that meet the condition.
    """
    return list(filter(condition, self.tasks))
```

- **Limitations:** Lack of multiline support, hard to debug.
- **Alternatives:** ***functools.partial*** and named functions for complex logic
- **Best Practice Tip:** Use lambda functions only for short, simple tasks for readability; For complex cases define a named function

Decorators

- Functions that add behavior to other functions or classes ***enhancing the functions*** on the fly

```
def my_decorator(func):
    def wrapper():
        # Do something before calling the function
        func()
        # Do something after calling the function
    return wrapper

@my_decorator
def my_function():
    # Function code here
    pass
```

- **Basic Example: @*time_it*** decorator to measure run time of a function
- In built decorators are ***@staticmethod, @classmethod,@property***

Decorators Examples

```python
@log_action
@validate_task
def mark_completed(self):
    """Mark the task as completed."""
    self.completed = True
    return f"Task '{self.name}' marked as completed."
```

```python
def log_action(func):
    """Decorator to log actions performed on tasks."""
    def wrapper(*args, **kwargs):
        result = func(*args, **kwargs)
        print(f"Action performed: {func.__name__}, Result: {result}")
        return result
    return wrapper


def validate_task(func):
    """Decorator to validate task input."""
    def wrapper(self, *args, **kwargs):
        if not args or not isinstance(args[0], str) or not args[0].strip():
            raise ValueError("Task name must be a non-empty string.")
        return func(self, *args, **kwargs)
    return wrapper
```

- ***Common decorator patterns:*** Logging, Authentication, Caching etc.
- ***Added Features:*** Stacked decorators, decorators with parameters

Decorators for Classes

```python
def my_class_decorator(cls):
    # Modify the class or add new attributes/methods
    cls.new_attribute = "This is a new attribute"
    return cls

@my_class_decorator
class MyClass:
    def __init__(self, value):
        self.value = value

    def display_value(self):
        print(f"Value: {self.value}")

# Using the decorated class
obj = MyClass(10)
print(obj.new_attribute)  # Output: This is a new attribute
obj.display_value()       # Output: Value: 10
```

- We can use the decorators for classes the same we use them for functions

Dunder(Magic) Methods

- Makes your classes behaves like built-in classes
- Special methods with ***double underscores*** that enables Python's Magic
- Predefined methods that you can override to customize the behavior of your classes
- Commonly used dunder methods
  - *__init__*
  - *__str__*
  - *__len__*
  - *__getitem__*
  - *__add__*
  - *__call_*
  - *__lt__*

Dunder Methods

```python
@log_action
def __add__(self, other):
    """Combine dependencies of two tasks."""
    if isinstance(other, Task):
        combined_dependencies = list(
            set(self.dependencies) | set(other.dependencies))
        return Task(name=f"{self.name} & {other.name}", dependencies=combined_dependencies)
    return NotImplemented
```

```python
def __iter__(self):
    """Return an iterator over the dependencies."""
    self._iter_index = 0  # Initialize the index for iteration
    return self

def __next__(self):
    if self._iter_index < len(self.dependencies):
        dependency = self.dependencies[self._iter_index]
        self._iter_index += 1
        return dependency

    else:
        raise StopIteration # Signal that the iteration is complete
```

```python
def __lt__(self, other):
    """Less than comparison based on priority and due date."""

    if self.PRIORITY_LEVELS[self.priority] == self.PRIORITY_LEVELS[other.priority]:
        return self.due_date < other.due_date
    return (
        self.PRIORITY_LEVELS[self.priority] < self.PRIORITY_LEVELS[other.priority]
    )
```

- Implementing specific dunder methods makes classes feel more intuitive and Pythonic
- Facilitates operator overloading, enhancing versatility

References for further reading

- Python documentation for dunder methods https://docs.python.org/3/reference/datamodel.html#special-method-names
- Primer on Python decorators https://realpython.com/primer-on-python-decorators/
- Understanding Python lambda functions https://realpython.com/python-lambda/

Thank You

ANY QUESTIONS??