Advanced Python Concepts

Here's a comprehensive guide to advanced Python topics, including everything. You can use this as a reference for your studies.

This guide covers a range of advanced Python concepts including `decorators`, `context managers`, `callable objects`, `property decorators`, `abstract base classes`, `metaclasses`, `descriptors`, `data classes`, `function annotations`, `memoization`, and the use of `__slots__`.

1. Decorators

Function Decorators

Decorators modify the behavior of a function or method.

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

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

say_hello()
```

Class Decorators

Class decorators modify classes.

```python
def class_decorator(cls):
    cls.extra_attribute = 'This is an extra attribute'
    return cls

@class_decorator
class MyClass:
    def greet(self):
        return "Hello!"

obj = MyClass()
print(obj.extra_attribute)  # Output: This is an extra attribute
```

2. Generators

Generators yield values one at a time, conserving memory.

```python
def count_up_to(n):
    count = 1
    while count <= n:
        yield count
        count += 1

for number in count_up_to(5):
    print(number)
```

3. Context Managers

Custom Context Managers

Use the `with` statement to manage resources.

```
class MyContext:
    def __enter__(self):
        print("Entering the context.")
        return self

    def __exit__(self, exc_type, exc_val, exc_tb):
        print("Exiting the context.")

with MyContext() as ctx:
    print("Inside the context.")
```

Context Managers with `contextlib`

Simplify context manager creation.

```
from contextlib import contextmanager

@contextmanager
def managed_resource():
    print("Acquiring resource")
    yield "Resource"
    print("Releasing resource")

with managed_resource() as resource:
    print(f"Using {resource}")
```

4. Callable Objects

Classes that implement `__call__` can be invoked like functions.

```
class CallableClass:
    def __init__(self, value):
        self.value = value

    def __call__(self, new_value):
        self.value = new_value
        return f'Value updated to {self.value}'

obj = CallableClass(10)
print(obj(20))  # Output: Value updated to 20
```

5. Property Decorators

Manage attribute access using properties.

```
class MyClass:
    def __init__(self, value):
        self._value = value

    @property
    def value(self):
        return self._value

    @value.setter
    def value(self, new_value):
        if new_value < 0:
            raise ValueError("Value cannot be negative.")
        self._value = new_value

obj = MyClass(10)
print(obj.value)  # Output: 10
obj.value = 15    # Updates value
```

6. Abstract Base Classes (ABCs)

Define abstract methods that must be implemented in derived classes.

```python
from abc import ABC, abstractmethod

class Shape(ABC):
    @abstractmethod
    def area(self):
        pass

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

    def area(self):
        return 3.14 * (self.radius ** 2)

class Square(Shape):
    def __init__(self, side):
        self.side = side

    def area(self):
        return self.side ** 2

shapes = [Circle(5), Square(4)]
for shape in shapes:
    print(f'Area: {shape.area()}')
```

7. Metaclasses

Metaclasses modify class creation.

Basic Example

```
class Meta(type):
    def __new__(cls, name, bases, attrs):
        attrs['greeting'] = 'Hello'
        return super().__new__(cls, name, bases, attrs)

class MyClass(metaclass=Meta):
    pass

obj = MyClass()
print(obj.greeting)  # Output: Hello
```

Enforcing Naming Conventions

```
class NameEnforcer(type):
    def __new__(cls, name, bases, attrs):
        if not name.startswith("My"):
            raise ValueError("Class name must start with 'My'.")
        return super().__new__(cls, name, bases, attrs)

class MyClass(metaclass=NameEnforcer):
    pass

# class AnotherClass(metaclass=NameEnforcer):  # Raises error
#     pass

obj = MyClass()  # Works
```

8. Descriptors

Customize attribute access with descriptors.

```python
class Descriptor:
    def __get__(self, instance, owner):
        return 'This is a descriptor value.'

    def __set__(self, instance, value):
        print(f'Setting value to {value}.')

class MyClass:
    my_attr = Descriptor()

obj = MyClass()
print(obj.my_attr)  # Output: This is a descriptor value.
obj.my_attr = 'New value'  # Output: Setting value to New value.
```

9. Data Classes

Simplify class definitions for data storage.

```python
from dataclasses import dataclass

@dataclass
class Point:
    x: int
    y: int

p1 = Point(1, 2)
print(p1)           # Output: Point(x=1, y=2)
```

10. Function Annotations

Add metadata to function parameters and return values.

```python
def add(a: int, b: int) -> int:
    return a + b

print(add(2, 3))  # Output: 5
print(add.__annotations__)  # Output: {'a': <class 'int'>, 'b': <class 'int'>, 'return': <class
```

11. Using `functools` for Memoization

Cache function results for efficiency.

```python
from functools import lru_cache

@lru_cache(maxsize=None)
def fibonacci(n):
    if n < 2:
        return n
    return fibonacci(n - 1) + fibonacci(n - 2)

print(fibonacci(10))  # Output: 55
```

12. `__slots__` {#12--slots }

Optimize memory usage by limiting class attributes.

```python
class MyClass:
    __slots__ = ['attr1', 'attr2']  # Only these attributes can be used

    def __init__(self, attr1, attr2):
        self.attr1 = attr1
        self.attr2 = attr2

obj = MyClass(1, 2)
print(obj.attr1)  # Output: 1
# obj.attr3 = 3  # Raises an AttributeError
```

Author: Laksh Kumar Sisodiya

github: https://github.com/thefcraft