Python y la Metaprogramación

Seminario 8

Programación por Contratos, Decoradores y Metaclases

18 de junio de 2025

Índice

1. Introducción

La metaprogramación es una técnica de programación que permite a los programas tratar a otros programas como datos. En Python, esta capacidad es especialmente poderosa debido a la naturaleza dinámica del lenguaje. Este informe explora tres conceptos fundamentales de la metaprogramación en Python:

- **Programación por Contratos**: Un paradigma que establece condiciones formales para el correcto funcionamiento del software.
- **Decoradores**: Funciones que modifican el comportamiento de otras funciones de manera declarativa.
- **Metaclases**: Clases cuyas instancias son otras clases, permitiendo controlar la creación y comportamiento de las clases.

2. Programación por Contratos

2.1. Fundamentos Teóricos

La programación por contratos, introducida por Bertrand Meyer en el lenguaje Eiffel, es un enfoque de diseño de software que define formalmente las obligaciones y beneficios de los componentes del software. Se basa en tres tipos de aserciones:

1. **Precondiciones**: Condiciones que deben cumplirse antes de ejecutar un método.
2. **Postcondiciones**: Condiciones que deben cumplirse después de ejecutar un método.
3. **Invariantes**: Condiciones que deben mantenerse durante toda la vida del objeto.

2.2. Ejemplo en C# con CodeContracts

CodeContracts es una biblioteca de Microsoft que permite implementar programación por contratos en C#. Veamos un ejemplo práctico:

```
1  using System.Diagnostics.Contracts;
2
3  public class CuentaBancaria
4  {
5      private decimal saldo;
6
7      public decimal Saldo
8      {
9          get { return saldo; }
10     }
11
12     public CuentaBancaria(decimal saldoInicial)
13     {
14         // Precondicion: el saldo inicial debe ser no negativo
15         Contract.Requires(saldoInicial >= 0);
16
```

```
17          saldo = saldoInicial;
18
19          // Postcondicion: el saldo debe ser igual al saldo inicial
20          Contract.Ensures(saldo == saldoInicial);
21      }
22
23      public void Depositar(decimal cantidad)
24      {
25          // Precondicion: la cantidad debe ser positiva
26          Contract.Requires(cantidad > 0);
27
28          // Capturamos el saldo anterior para la postcondicion
29          decimal saldoAnterior = saldo;
30
31          saldo += cantidad;
32
33          // Postcondicion: el nuevo saldo debe ser mayor que el anterior
34          Contract.Ensures(saldo == saldoAnterior + cantidad);
35      }
36
37      public void Retirar(decimal cantidad)
38      {
39          // Precondiciones
40          Contract.Requires(cantidad > 0);
41          Contract.Requires(cantidad <= saldo);
42
43          decimal saldoAnterior = saldo;
44
45          saldo -= cantidad;
46
47          // Postcondicion
48          Contract.Ensures(saldo == saldoAnterior - cantidad);
49      }
50
51      // Invariante de la clase
52      [ContractInvariantMethod]
53      private void InvarianteDeCuenta()
54      {
55          // El saldo nunca debe ser negativo
56          Contract.Invariant(saldo >= 0);
57      }
58  }
```

Listing 1: Ejemplo de CodeContracts en C#

2.3. Ventajas de la Programación por Contratos

- **Documentación ejecutable**: Los contratos sirven como documentación que se verifica en tiempo de ejecución.
- **Detección temprana de errores**: Los errores se detectan en el punto donde ocurren, no donde se manifiestan.
- **Diseño más robusto**: Fuerza a pensar en las condiciones de contorno y casos

especiales.

- **Facilita el testing**: Los contratos actúan como casos de prueba automáticos.

3. Implementación de Contratos con Decoradores en Python

3.1. Conceptos de Decoradores

Los decoradores en Python son funciones que toman otra función como argumento y extienden su comportamiento sin modificarla explícitamente. Son una forma elegante de aplicar metaprogramación.

3.1.1. Sintaxis Básica

Un decorador es esencialmente una función que recibe una función y retorna una función modificada:

```
def mi_decorador(funcion):
    def funcion_envolvente(*args, **kwargs):
        # C digo antes de ejecutar la funci n
        resultado = funcion(*args, **kwargs)
        # C digo despu s de ejecutar la funci n
        return resultado
    return funcion_envolvente

# Uso del decorador
@mi_decorador
def mi_funcion():
    print("Funci n original")

# Equivalente a: mi_funcion = mi_decorador(mi_funcion)
```

Listing 2: Estructura básica de un decorador

3.2. Ejemplos de Decoradores Simples

3.2.1. Decorador de Tiempo de Ejecución

```
import time
from functools import wraps

def medir_tiempo(func):
    """Decorador que mide el tiempo de ejecuci n de una funci n"""
    @wraps(func)
    def wrapper(*args, **kwargs):
        inicio = time.time()
        resultado = func(*args, **kwargs)
        fin = time.time()
        tiempo_total = fin - inicio
        print(f"{func.__name__} tard  {tiempo_total:.4f} segundos")
```

```
        return resultado
    return wrapper

# Ejemplo de uso
@medir_tiempo
def operacion_costosa():
    """Simula una operaci n que toma tiempo"""
    suma = 0
    for i in range(1000000):
        suma += i
    return suma

resultado = operacion_costosa()
# Output: operacion_costosa tard  0.0532 segundos
```

Listing 3: Decorador para medir tiempo de ejecución

3.2.2. Decorador de Logging

```
from functools import wraps
import logging

# Configurar logging
logging.basicConfig(level=logging.INFO)

def log_llamadas(func):
    """Decorador que registra las llamadas a una funci n"""
    @wraps(func)
    def wrapper(*args, **kwargs):
        # Registrar argumentos
        args_repr = [repr(a) for a in args]
        kwargs_repr = [f"{k}={v!r}" for k, v in kwargs.items()]
        signature = ", ".join(args_repr + kwargs_repr)

        logging.info(f"Llamando {func.__name__}({signature})")

        try:
            resultado = func(*args, **kwargs)
            logging.info(f"{func.__name__} retorn  {resultado!r}")
            return resultado
        except Exception as e:
            logging.error(f"{func.__name__} lanz  excepci n: {e}")
            raise

    return wrapper

# Ejemplo de uso
@log_llamadas
def dividir(a, b):
    return a / b

resultado = dividir(10, 2)
# INFO:root:Llamando dividir(10, 2)
# INFO:root:dividir retorn  5.0

```

```
try:
    dividir(10, 0)
except ZeroDivisionError:
    pass
# INFO:root:Llamando dividir(10, 0)
# ERROR:root:dividir lanz  excepci n: division by zero
```

Listing 4: Decorador para registro de llamadas

3.2.3. Decorador de Cache (Memoización)

```
from functools import wraps

def cache(func):
    """Decorador que cachea los resultados de una funci n"""
    cache_dict = {}

    @wraps(func)
    def wrapper(*args, **kwargs):
        # Crear una clave  nica  para los argumentos
        key = str(args) + str(kwargs)

        if key not in cache_dict:
            # Calcular y almacenar el resultado
            cache_dict[key] = func(*args, **kwargs)
            print(f"Calculando {func.__name__}{args}")
        else:
            print(f"Retornando desde cache {func.__name__}{args}")

        return cache_dict[key]

    # A adir  m todo para limpiar cache
    wrapper.clear_cache = lambda: cache_dict.clear()

    return wrapper

# Ejemplo con fibonacci
@cache
def fibonacci(n):
    if n < 2:
        return n
    return fibonacci(n-1) + fibonacci(n-2)

print(fibonacci(5))  # Calcula valores nuevos
print(fibonacci(5))  # Retorna desde cache
# Output:
# Calculando fibonacci(0)
# Calculando fibonacci(1)
# Calculando fibonacci(2)
# Calculando fibonacci(3)
# Calculando fibonacci(4)
# Calculando fibonacci(5)
# 5
# Retornando desde cache fibonacci(5)
# 5
```

Listing 5: Decorador para cachear resultados

3.2.4. Decorador de Validación de Tipos

```python
from functools import wraps

def validar_tipos(**tipos_esperados):
    """Decorador que valida los tipos de los argumentos"""
    def decorador(func):
        @wraps(func)
        def wrapper(*args, **kwargs):
            # Obtener nombres de par metros
            import inspect
            sig = inspect.signature(func)
            bound = sig.bind(*args, **kwargs)
            bound.apply_defaults()

            # Validar cada argumento
            for nombre, valor in bound.arguments.items():
                if nombre in tipos_esperados:
                    tipo_esperado = tipos_esperados[nombre]
                    if not isinstance(valor, tipo_esperado):
                        raise TypeError(
                            f"Argumento '{nombre}' debe ser {tipo_esperado.__name__}, "
                            f"pero se recibi  {type(valor).__name__}"
                        )

            return func(*args, **kwargs)

        return wrapper
    return decorador

# Ejemplo de uso
@validar_tipos(x=int, y=int)
def sumar(x, y):
    return x + y

print(sumar(5, 3))  # 8

try:
    sumar(5, "3")  # TypeError
except TypeError as e:
    print(f"Error: {e}")
# Error: Argumento 'y' debe ser int, pero se recibi  str
```

Listing 6: Decorador para validar tipos de argumentos

3.2.5. Decorador con Parámetros

```python
from functools import wraps
import time
```

```python

def reintentar(intentos=3, delay=1, excepciones=(Exception,)):
    """
    Decorador que reintenta una funci n si falla

    Args:
        intentos: N mero m ximo de intentos
        delay: Segundos de espera entre intentos
        excepciones: Tupla de excepciones a capturar
    """
    def decorador(func):
        @wraps(func)
        def wrapper(*args, **kwargs):
            for intento in range(intentos):
                try:
                    return func(*args, **kwargs)
                except excepciones as e:
                    if intento == intentos - 1:
                        print(f"Fallo despu s de {intentos} intentos")
                        raise
                    print(f"Intento {intento + 1} fall : {e}")
                    print(f"Reintentando en {delay} segundos...")
                    time.sleep(delay)

        return wrapper
    return decorador

# Ejemplo de uso
import random

@reintentar(intentos=3, delay=0.5, excepciones=(ValueError,))
def operacion_inestable():
    """Simula una operaci n que puede fallar"""
    if random.random() < 0.7:  # 70% de probabilidad de fallo
        raise ValueError("Operaci n fall  aleatoriamente")
    return " xito !"

try:
    resultado = operacion_inestable()
    print(f"Resultado: {resultado}")
except ValueError:
    print("La operaci n fall  definitivamente")
```

Listing 7: Decorador parametrizable para reintentos

3.3. Diseño del Sistema de Contratos

Implementaremos un sistema de contratos similar a CodeContracts usando decoradores. El diseño debe considerar:

1. Validación de precondiciones antes de ejecutar la función.
2. Validación de postcondiciones después de ejecutar la función.
3. Manejo de excepciones personalizadas.

4. Validación de compatibilidad de signaturas.

3.4. Implementación Completa

```python
import inspect
from functools import wraps
from typing import Callable, Any

class ContractException(Exception):
    """Excepcion base para violaciones de contratos"""
    pass

class PreconditionViolation(ContractException):
    """Excepcion lanzada cuando se viola una precondicion"""
    pass

class PostconditionViolation(ContractException):
    """Excepcion lanzada cuando se viola una postcondicion"""
    pass

class SignatureIncompatibility(ContractException):
    """Excepcion lanzada cuando las signaturas no son compatibles"""
    pass

def validate_ensure_signature(ensure_func):
    """
    Valida que la funcion ensure tenga una signatura valida.
    Solo debe tener un argumento posicional obligatorio (result).
    """
    sig = inspect.signature(ensure_func)
    params = list(sig.parameters.values())

    # Contar parametros obligatorios
    required_positional = 0
    required_keyword_only = 0

    for param in params:
        if param.default == inspect.Parameter.empty:
            if param.kind in (inspect.Parameter.POSITIONAL_ONLY,
                              inspect.Parameter.POSITIONAL_OR_KEYWORD):
                required_positional += 1
            elif param.kind == inspect.Parameter.KEYWORD_ONLY:
                required_keyword_only += 1

    # Validar restricciones
    if required_positional > 1:
        raise SignatureIncompatibility(
            f"La funcion ensure '{ensure_func.__name__}' tiene "
            f"{required_positional} argumentos posicionales obligatorios, "
            "pero solo puede tener 1 (el resultado)"
        )

    if required_keyword_only > 0:
        raise SignatureIncompatibility(
```

```
51              f"La funcion ensure '{ensure_func.__name__}' tiene "
52              f"{required_keyword_only} argumentos keyword-only
    obligatorios, "
53              "pero no puede tener ninguno"
54          )
55
56  def validate_require_signature(require_func, target_func):
57      """
58      Valida que la funcion require sea compatible con la funcion objetivo
    .
59      """
60      require_sig = inspect.signature(require_func)
61      target_sig = inspect.signature(target_func)
62
63      try:
64          # Intentar bind de los parametros de target a require
65          target_params = list(target_sig.parameters.values())
66          require_params = list(require_sig.parameters.values())
67
68          # Crear argumentos de prueba basados en target
69          test_args = []
70          test_kwargs = {}
71
72          for param in target_params:
73              if param.kind == inspect.Parameter.VAR_POSITIONAL:
74                  test_args.extend([None] * 3)  # Simular *args
75              elif param.kind == inspect.Parameter.VAR_KEYWORD:
76                  test_kwargs.update({'key1': None, 'key2': None})  #
    Simular **kwargs
77              elif param.kind == inspect.Parameter.KEYWORD_ONLY:
78                  test_kwargs[param.name] = None
79              else:
80                  test_args.append(None)
81
82          # Intentar bind
83          require_sig.bind(*test_args, **test_kwargs)
84
85      except TypeError as e:
86          raise SignatureIncompatibility(
87              f"La signatura de require '{require_func.__name__}' "
88              f"no es compatible con la funcion '{target_func.__name__}':
    {str(e)}"
89          )
90
91  def contract(require=None, ensure=None):
92      """
93      Decorador para implementar programacion por contratos.
94
95      Args:
96          require: Funcion que verifica precondiciones
97          ensure: Funcion que verifica postcondiciones
98      """
99      def decorator(func):
100         # Validar signaturas si se proporcionan funciones
101         if require is not None:
102             validate_require_signature(require, func)
```

```
103
104         if ensure is not None:
105             validate_ensure_signature(ensure)
106
107         @wraps(func)
108         def wrapper(*args, **kwargs):
109             # Verificar precondiciones
110             if require is not None:
111                 try:
112                     if not require(*args, **kwargs):
113                         raise PreconditionViolation(
114                             f"Precondicion violada en '{func.__name__}'"
115                         )
116                 except Exception as e:
117                     if isinstance(e, PreconditionViolation):
118                         raise
119                     raise PreconditionViolation(
120                         f"Error al evaluar precondicion en '{func.
    __name__}': {str(e)}"
121                     )
122
123             # Ejecutar la funcion
124             result = func(*args, **kwargs)
125
126             # Verificar postcondiciones
127             if ensure is not None:
128                 try:
129                     if not ensure(result):
130                         raise PostconditionViolation(
131                             f"Postcondicion violada en '{func.__name__}'
    "
132                         )
133                 except Exception as e:
134                     if isinstance(e, PostconditionViolation):
135                         raise
136                     raise PostconditionViolation(
137                         f"Error al evaluar postcondicion en '{func.
    __name__}': {str(e)}"
138                     )
139
140             return result
141
142         return wrapper
143
144     return decorator
```

Listing 8: Implementación del decorador de contratos

3.5. Ejemplos de Uso

```
1 # Ejemplo 1: Division segura
2 @contract(
3     require=lambda x, y: y != 0,
4     ensure=lambda result: result >= 0
```

```
)
def division_positiva(x, y):
    return abs(x / y)

# Ejemplo 2: Calculo de factorial
@contract(
    require=lambda n: n >= 0 and isinstance(n, int),
    ensure=lambda result: result >= 1
)
def factorial(n):
    if n == 0:
        return 1
    return n * factorial(n - 1)

# Ejemplo 3: Procesamiento de lista
@contract(
    require=lambda lst: len(lst) > 0 and all(isinstance(x, (int, float))
    for x in lst),
    ensure=lambda result: 0 <= result <= 1
)
def normalizar_maximo(lst):
    max_val = max(lst)
    if max_val == 0:
        return 0
    return max(lst) / max(abs(x) for x in lst)

# Pruebas
try:
    print(division_positiva(10, 2))  # 5.0
    print(division_positiva(10, 0))  # PreconditionViolation
except PreconditionViolation as e:
    print(f"Error: {e}")

try:
    print(factorial(5))  # 120
    print(factorial(-1))  # PreconditionViolation
except PreconditionViolation as e:
    print(f"Error: {e}")
```

Listing 9: Ejemplos de uso del decorador contract

4. Metaclases en Python

4.1. Fundamentos de Metaclases

Las metaclases son uno de los conceptos más avanzados en Python. Una metaclase es una clase cuyas instancias son clases. En otras palabras, así como una clase define cómo se comportan sus instancias, una metaclase define cómo se comportan las clases.

4.1.1. La Jerarquía de Tipos en Python

```
# Todo en Python es un objeto
x = 5
```

```python
print(type(x))  # <class 'int'>
print(type(int))  # <class 'type'>
print(type(type))  # <class 'type'>

# Las clases son instancias de metaclases
class MiClase:
    pass

print(type(MiClase))  # <class 'type'>
print(isinstance(MiClase, type))  # True
```

Listing 10: Jerarquía de tipos

4.2. Implementación del Patrón Singleton

El patrón Singleton garantiza que una clase tenga solo una instancia y proporciona un punto de acceso global a ella.

```python
class SingletonMeta(type):
    """
    Metaclase que implementa el patron Singleton.
    Mantiene un diccionario de instancias por clase.
    """
    _instances = {}

    def __call__(cls, *args, **kwargs):
        """
        Controla la creacion de instancias.
        Si ya existe una instancia, la retorna.
        Si no, crea una nueva y la almacena.
        """
        if cls not in cls._instances:
            # Crear la instancia usando el __call__ de la clase padre
            instance = super(SingletonMeta, cls).__call__(*args, **
    kwargs)
            cls._instances[cls] = instance
        return cls._instances[cls]

class Singleton(metaclass=SingletonMeta):
    """
    Clase base para implementar el patron Singleton.
    Cualquier clase que herede de esta sera un Singleton.
    """
    pass

# Ejemplo de uso
class ConfiguracionGlobal(Singleton):
    def __init__(self):
        self.configuracion = {}
        print("Creando instancia de ConfiguracionGlobal")

    def set_config(self, key, value):
        self.configuracion[key] = value

    def get_config(self, key):
```

```
        return self.configuracion.get(key)

# Pruebas
config1 = ConfiguracionGlobal()  # Imprime: Creando instancia...
config2 = ConfiguracionGlobal()  # No imprime nada

print(config1 is config2)  # True

config1.set_config("debug", True)
print(config2.get_config("debug"))  # True

# Herencia del comportamiento Singleton
class ConfiguracionEspecifica(ConfiguracionGlobal):
    def __init__(self):
        super().__init__()
        self.especifica = "valor especifico"

esp1 = ConfiguracionEspecifica()
esp2 = ConfiguracionEspecifica()
print(esp1 is esp2)  # True
```

Listing 11: Implementación de Singleton con metaclases

4.3. Implementación de Objetos Inmutables

Los objetos inmutables no pueden ser modificados después de su creación. Esto es útil para garantizar la integridad de los datos.

```
class MetaInmutable(type):
    """
    Metaclase que hace que las instancias de las clases sean inmutables.
    """
    def __call__(cls, *args, **kwargs):
        """
        Crea la instancia y la marca como inicializada.
        """
        instance = super(MetaInmutable, cls).__call__(*args, **kwargs)
        # Marcar la instancia como inicializada
        object.__setattr__(instance, '_inicializado', True)
        return instance

class ObjetoInmutable(metaclass=MetaInmutable):
    """
    Clase base para objetos inmutables.
    Una vez creados, no se pueden modificar sus atributos.
    """
    def __setattr__(self, name, value):
        """
        Previene la modificacion de atributos despues de la
   inicializacion.
        """
        if hasattr(self, '_inicializado') and self._inicializado:
            raise AttributeError(
                f"No se puede modificar el atributo '{name}' "
```

```
26                  f"en un objeto inmutable de tipo '{type(self).__name__}'
    "
27              )
28          super().__setattr__(name, value)
29
30      def __delattr__(self, name):
31          """
32          Previene la eliminacion de atributos.
33          """
34          if hasattr(self, '_inicializado') and self._inicializado:
35              raise AttributeError(
36                  f"No se puede eliminar el atributo '{name}' "
37                  f"en un objeto inmutable de tipo '{type(self).__name__}'
    "
38              )
39          super().__delattr__(name)
40
41  # Ejemplo de uso
42  class Punto(ObjetoInmutable):
43      def __init__(self, x, y):
44          self.x = x
45          self.y = y
46
47      def __str__(self):
48          return f"Punto({self.x}, {self.y})"
49
50  class Vector(ObjetoInmutable):
51      def __init__(self, componentes):
52          self.componentes = list(componentes)  # Copia para evitar
    referencias
53
54      def __str__(self):
55          return f"Vector({self.componentes})"
56
57  # Pruebas
58  p1 = Punto(3, 4)
59  print(p1)  # Punto(3, 4)
60
61  try:
62      p1.x = 5  # AttributeError
63  except AttributeError as e:
64      print(f"Error esperado: {e}")
65
66  try:
67      p1.z = 10  # AttributeError
68  except AttributeError as e:
69      print(f"Error esperado: {e}")
70
71  try:
72      del p1.y  # AttributeError
73  except AttributeError as e:
74      print(f"Error esperado: {e}")
75
76  # La inmutabilidad se hereda
77  v1 = Vector([1, 2, 3])
78  print(v1)  # Vector([1, 2, 3])
```

```
79
80 try:
81     v1.componentes = [4, 5, 6]  # AttributeError
82 except AttributeError as e:
83     print(f"Error esperado: {e}")
```

Listing 12: Implementación de ObjetoInmutable con metaclases

4.4. Ventajas y Consideraciones de las Metaclases

4.4.1. Ventajas

- **Control total**: Permiten controlar completamente el proceso de creación de clases.
- **Reutilización**: Los comportamientos definidos en metaclases se heredan automáticamente.
- **Validación**: Pueden validar la definición de clases en tiempo de creación.
- **Patrones avanzados**: Facilitan la implementación de patrones de diseño complejos.

4.4.2. Consideraciones

- **Complejidad**: Las metaclases añaden complejidad al código.
- **Debugging**: Pueden hacer el debugging más difícil.
- **Alternativas**: Muchos casos de uso pueden resolverse con decoradores o herencia simple.
- **Principio**: "Las metaclases son magia profunda que el 99 % de los usuarios no deberían preocuparse Tim Peters

5. Integración de Conceptos

5.1. Ejemplo Completo: Sistema de Configuración Inmutable con Contratos

Combinemos todos los conceptos aprendidos en un ejemplo práctico:

```
1  class ConfiguracionInmutable(Singleton, ObjetoInmutable):
2      """
3      Sistema de configuracion que es singleton e inmutable.
4      """
5      @contract(
6          require=lambda self, config_dict: isinstance(config_dict, dict),
7          ensure=lambda result: result is None
8      )
9      def __init__(self, config_dict):
10         # Copiar configuracion para evitar referencias externas
11         self._config = dict(config_dict)
12
```

```
    @contract(
        require=lambda self, key: isinstance(key, str),
        ensure=lambda result: result is not None
    )
    def get(self, key):
        if key not in self._config:
            raise KeyError(f"Clave '{key}' no encontrada en la
    configuracion")
        return self._config[key]

    def __str__(self):
        return f"ConfiguracionInmutable({self._config})"

# Uso del sistema
try:
    # Primera instancia
    config = ConfiguracionInmutable({
        "host": "localhost",
        "port": 8080,
        "debug": True
    })

    print(config.get("host"))  # localhost

    # Segunda instancia - retorna la misma
    config2 = ConfiguracionInmutable({"otro": "valor"})
    print(config is config2)  # True
    print(config2.get("host"))  # localhost (misma configuracion)

    # Intentar modificar - falla
    config._config["nuevo"] = "valor"  # AttributeError

except (AttributeError, ContractException) as e:
    print(f"Error: {e}")
```

Listing 13: Sistema integrado de configuración

6. Conclusiones

La metaprogramación en Python ofrece herramientas poderosas para crear código más expresivo, robusto y reutilizable:

1. **Programación por Contratos**: Proporciona una forma formal de especificar y verificar el comportamiento del código, mejorando la confiabilidad y documentación.

2. **Decoradores**: Ofrecen una sintaxis elegante para modificar el comportamiento de funciones y métodos, permitiendo separar concerns transversales del código principal.

3. **Metaclases**: Proporcionan control total sobre la creación y comportamiento de las clases, permitiendo implementar patrones de diseño avanzados de manera elegante.

6.1. Recomendaciones de Uso

- Use contratos para APIs críticas donde la validación de entrada/salida es esencial.
- Prefiera decoradores sobre metaclases cuando sea posible, ya que son más simples.
- Reserve las metaclases para casos donde realmente necesite controlar la creación de clases.
- Documente extensivamente el código que use metaprogramación.
- Considere el impacto en el rendimiento, especialmente con validaciones en runtime.

6.2. Recursos Adicionales

- **PEP 316**: Propuesta para programación por contratos en Python
- **PEP 318**: Decoradores para funciones y métodos
- **PEP 3115**: Metaclases en Python 3
- **Documentación oficial**: `https://docs.python.org/3/reference/datamodel.html`