Lenguajes de Programación
Seminario #5
Tema: Python Mágico

Equipo 5:

Osvaldo R. Moreno Prieto C311

Daniel Toledo Martínez C311

Daniel Machado Pérez C311

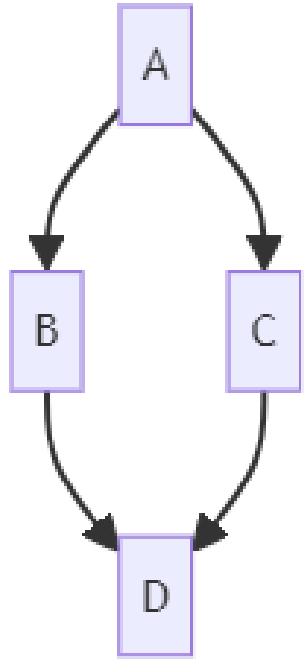
Resolución de miembros y métodos en Python

- Proceso de determinar a qué clase pertenece un miembro (un método o un atributo).
- Orden de Resolución de Métodos (MRO).
- Es posible acceder al MRO de la clase utilizando el atributo __mro__ o la función mro().
- La función super() en Python se utiliza para acceder al método de la clase base en lugar del método de la clase actual (en el orden del MRO).

Ejemplo:

```python
class A:
    def saludar(self):
        return "Hola desde A"

class B(A):
    def saludar(self):
        return "Hola desde B, " + super().saludar()

class C(A):
    def saludar(self):
        return "Hola desde C, " + super().saludar()

class D(B, C):
    pass

instancia_d = D()
print(instancia_d.saludar())
```

mro(D) = D,B,C,A



#Salida: Hola desde B, Hola desde C, Hola desde A

Método super()

- Si la Herencia es simple, llama a los métodos de la clase base en el orden de la herencia, pues ese es el orden del MRO.
- Si la Herencia es múltiple, su uso puede provocar confusiones.
- Es útil para llamar a la implementación de un método mágico de la clase base desde una subclase.

Ejemplo con el método mágico __init__:

```
class MiClaseBase:
        def __init__(self, valor):
            self.valor = valor

    class MiSubClase(MiClaseBase):
        def __init__(self, valor, otro_valor):
            super().__init__(valor)  # Llamando al __init__ de la clase base
            self.otro_valor = otro_valor

    # Creando una instancia de la subclase
    objeto = MiSubClase(10, 20)

    print(objeto.valor)        # Accediendo al atributo de la clase base
    print(objeto.otro_valor)  # Accediendo al atributo de la subclase
```

- #Salida
- #10
- #20

Métodos Mágicos

- Es el enfoque de Python para la sobrecarga de operadores.
- Permite que las clases definan su propio comportamiento con respecto a los operadores del lenguaje.
- Si una clase define un método llamado __getitem__(), y x es una instancia de esta clase, entonces x[i] es aproximadamente equivalente a type(x).__getitem__(x, i).
- Establecer un método especial en None u omitirlo, indica que la operación correspondiente no está disponible.

Métodos Mágicos más comunes

- __new__: Toma la clase como primer argumento, seguido de cualquier otro argumento que se pase al constructor . Su principal responsabilidad es crear y devolver la nueva instancia del objeto.
- __init__: es el segundo paso en la creación de un objeto, después de __new__. Toma la nueva instancia creada por __new__ y cualquier otro argumento que se pase al constructor , y lo utiliza para inicializar el objeto.

Ejemplo:

```
class MiClase:
    def __new__(cls, *args, **kwargs):
        print("MiClase.__new__ llamado")
        instancia = object.__new__(cls)
        return instancia

    def __init__(self, valor):
        print("MiClase.__init__ llamado")
        self.valor = valor

mi_instancia = MiClase("Hola, mundo")
#Salida:
#MiClase.__new__ llamado
#MiClase.__init__ llamado
```

Métodos Mágicos más comunes

- __repr__: Se utiliza para devolver la representación oficial de un objeto.
- __str__: Se utiliza para devolver una representación legible de un objeto. Es llamado con la función print() o str(). Si no se define __str__, Python llamará a __repr__ en su lugar.

Ejemplo:

```
class MiClase:
    def __init__(self, valor):
        self.valor = valor

    def __repr__(self):
        return f"MiClase({self.valor!r})"

    # def __str__(self):
    #     return f"Instancia de MiClase con el valor {self.valor!r}"

instancia = MiClase(10)
print("Si no se encuentra __str__, se imprime __repr__")
print(instancia)
```

#Salida:
#Si no se encuentra __str__, se imprime __repr__
#MiClase(10)

```
class MiClase:
    def __init__(self, valor):
        self.valor = valor

    def __repr__(self):
        return f"MiClase({self.valor!r})"

    def __str__(self):
        return f"Instancia de MiClase con el valor {self.valor!r}"

instancia = MiClase(10)
print("Imprimiendo con __str__")
print(instancia)
print("Imprimiendo la __repr__")
print(repr(instancia))
```

#Salida:
#Imprimiendo con __str__
#Instancia de MiClase con el valor 10
#Imprimiendo la __repr__
#MiClase(10)

Métodos de comparación y su correspondencia con los operadores de comparación:

```
object.__lt__(self, other) # self < other
object.__le__(self, other) # self <= other
object.__eq__(self, other) # self == other
object.__ne__(self, other) # self != other
object.__gt__(self, other) # self > other
object.__ge__(self, other) # self >= other
```

En ocasiones no es necesario tener explícitamente todas implementadas, ya que Python puede inferir el resultado de una operación de comparación a partir de otras. Por ejemplo, si __eq__y __ne__no están implementados, Python invocará __lt__y __gt__para determinar el resultado de __ne__.

Sobrecarga de operadores numéricos y su correspondencia con los operadores aritméticos:

```
object.__add__(self, other) # self + other
object.__sub__(self, other) # self - other
object.__mul__(self, other) # self * other
object.__matmul__(self, other) # self @ other
object.__truediv__(self, other) # self / other
object.__floordiv__(self, other) # self // other
object.__mod__(self, other) # self % other
object.__divmod__(self, other) # divmod(self, other)
object.__pow__(self, other[, modulo]) # self ** other or pow(self, other,
modulo)
object.__lshift__(self, other) # self << other
object.__rshift__(self, other) # self >> other
object.__and__(self, other) # self & other
object.__xor__(self, other) # self ^ other
object.__or__(self, other)  # self | other
```

Otros métodos

- __getitem__y __setitem__Son llamados con la sintaxis self[key] y self[key] = value respectivamente. Como su nombre indica, se utilizan para obtener y establecer valores.
- __len__Debe devolver la longitud del objeto.
- __iter__Debe devolver un iterador para el objeto. Debe poder ser usado en un bucle for .
- __next__Debe devolver el siguiente elemento en el iterador.
- __contains__Debe devolver True si el objeto contiene el valor especificado, False en caso contrario.

Ejemplo:

```
class MiClase:
    def __init__(self, lista):
        self.lista = lista

    def __getitem__(self, indice):
        return self.lista[indice]

    def __setitem__(self, indice, valor):
        self.lista[indice] = valor

    def __len__(self):
        return len(self.lista)

    def __iter__(self):
        for valor in self.lista:
            yield valor

mi_lista = MiClase([1, 2, 3, 4, 5])
```

```
print("modificando valores de la lista:")
print(mi_lista[0])  # Salida: 1
mi_lista[0] = 10
print(mi_lista[0])  # Salida: 10

print("obteniendo la longitud de la lista:")
print(len(mi_lista))  # Salida: 5


print("iterando sobre la lista:")
for valor in mi_lista:
    print(valor)
```

#Salida:
10 2 3 4 5

__call__Define un comportamiento para la instancia, cuando es llamada como función.

```
class MiClase:
    def __init__(self, valor):
        self.valor = valor

    def __call__(self, otro):
        return self.valor + otro
    def __str__(self):
        return str(self.valor)

a = MiClase(10)
b = a(20)
print(b)  # Salida: 30
```

Métodos built-in de Python

- Están disponibles en las clases por defecto, sin necesidad de importar ningún módulo adicional.
- Su sintaxis es en notación prefija, es decir metodo(objeto,argumentos) en lugar de objeto.metodo(argumentos).
- La mayoría de los métodos built-in son funciones que llaman a los métodos mágicos de las clases: len(), iter(), next(),…
- Algunos son funciones que realizan operaciones específicas:
  print() imprime un objeto en la consola,
  type() devuelve el tipo de un objeto,
  id() devuelve el identificador único de un objeto, entre otros.

Iteradores

- Son objetos que implementan los métodos __iter__() y __next__(), lo que les permite ser iterados (recorridos) a través de un bucle for.
- Proporcionan una forma eficiente y elegante de acceder a elementos de una secuencia, como listas, tuplas, diccionarios, conjuntos, y otros objetos iterables.
- Debe cumplir con dos requisitos principales:

1. __iter__(): Este método debe devolver el propio objeto iterador. Es llamado al iniciar la iteración y se espera que devuelva un objeto iterador .

2. __next__(): Este método debe devolver el próximo elemento en la secuencia. Cuando no hay más elementos para iterar, se espera que levante la excepción StopIteration.

Ejemplo:

```
class Contador:
    def __init__(self, limite):
        self.limite = limite
        self.valor = 0

    def __iter__(self):
        return self

    def __next__(self):
        if self.valor < self.limite:
            resultado = self.valor
            self.valor += 1
            return resultado
        else:
            raise StopIteration

# Crear un objeto iterador
contador_iterador = Contador(5)
# Utilizar el iterador en un bucle for
for numero in contador_iterador:
    print(numero)
```

#Salida

#0

#1

#2

#3

#4

Ejemplo de Iterador y método built-in:

```
class Iterable:
    _index = 0
    def __init__(self, *args):
        self.valores = args
    def __len__(self):
        return len(self.valores)
    def __iter__(self):
        for valor in self.valores:
            yield valor
    def __getitem__(self, index):
        return self.valores[index]

a = Iterable(1, 2, 3, 4, 5)
print(f"\nlenghth de a: {len(a)}")
print(list(a)) # funciona xq Iterable tiene el metodo __iter__
print(list(reversed(a))) # funciona xq Iterable tiene el metodo __getitem__
```

Método eval()

- Es un método built-in que evalúa una expresión en forma de cadena y devuelve el resultado.
- Puede ser útil en la construcción de AST o en la evaluación de expresiones dinámicas.
- eval() recibe un string y lo evalúa como una expresión de Python, también puede recibir un diccionario con variables locales y globales. Esto nos puede ayudar a evaluar expresiones dinámicas o mejorar la seguridad al usar eval.

Ejemplo:

```
expr = "print('Hola, mundo!')"
eval(expr) # Hola, mundo!

expr  = "x**2 + 2*x + 1"
x = 2
print(eval(expr, {'x': x})) # 9
```

Tema extra: Decoradores en Python

- Función que toma otra función o método y extiende o modifica su comportamiento sin modificar su código fuente.
- Forma elegante y poderosa de realizar acciones adicionales antes o después de la ejecución de una función.
- Pueden usarse para reutilizar y extender el comportamiento de funciones de manera modular .

Sintaxis básica de un Decorador

Un decorador se aplica a una función utilizando la sintaxis @decorador justo encima de la definición de la función.

```
@mi_decorador
def mi_funcion():
    # código de la función
```

En este caso, mi_funcion será pasada como argumento a la función mi_decorador.

Definición de un Decorador

Un decorador es simplemente una función en Python. Puede aceptar una función como argumento, realizar alguna acción, y luego devolver una función modificada o extender el comportamiento de la función original.

```
def mi_decorador(funcion_original):
    def funcion_modificada():
        print("Realizando acciones antes de llamar a la función original")
        funcion_original()
        print("Realizando acciones después de llamar a la función original")
    return funcion_modificada
```

Aplicación de un Decorador

```
@mi_decorador
def saludar():
    print("¡Hola, mundo!")
```

En este caso, cuando se llama a saludar(), en realidad se está llamando a la versión modificada de la función creada por el decorador .

Múltiples Decoradores

Se pueden aplicar múltiples decoradores a una función. Estos se aplican de abajo hacia arriba

```
@decorador1
@decorador2
def mi_funcion():
    # código de la función
```

En este caso, primero se aplica decorador2, y luego decorador1.

Decoradores incorporados

Python proporciona algunos decoradores incorporados, como @staticmethod, @classmethod, y @property. Estos son utilizados comúnmente en clases para definir métodos estáticos, de clase o propiedades.

```
class MiClase:
    @staticmethod
    def metodo_estatico():
        # código del método estático

    @classmethod
    def metodo_de_clase(cls):
        # código del método de clase

    @property
    def mi_propiedad(self):
        # código de la propiedad
```

Veamos un ejemplo sencillo y práctico de cómo se utiliza un decorador en Python.

Supongamos que queremos medir el tiempo de ejecución de una función. Podemos crear un decorador que haga esto sin modificar el código interno de la función.

```python
import time

# Decorador para medir el tiempo de ejecución de una función
def medir_tiempo(funcion):
    def wrapper(*args, **kwargs):
        inicio = time.time()
        resultado = funcion(*args, **kwargs)
        fin = time.time()
        print(f"{funcion.__name__} tomó {fin - inicio} segundos en
ejecutarse.")
        return resultado
    return wrapper

# Aplicar el decorador a una función
@medir_tiempo
def ejemplo_funcion():
    print("Iniciando...")
    time.sleep(2)
    print("Finalizando.")

# Llamando a la función decorada
ejemplo_funcion()
```

#Salida

#Iniciando…

#Finalizando

#ejemplo_funcion tomó 2.004222869873047 segundos en ejecutarse

Muchas Gracias