**Python Magic Methods Overview -**

python magic method examples and use case

In Python, \*\*magic methods\*\* (also called dunder methods, short for "double underscore") are special methods that allow you to define the behavior of objects for built-in operations. These methods are surrounded by double underscores (e.g., `\_\_init\_\_`, `\_\_str\_\_`, `\_\_add\_\_`). They enable you to customize how objects behave in various situations, such as arithmetic operations, comparisons, or type conversions.  
  
Here are some common magic methods, their use cases, and examples:  
  
---  
  
### 1. \*\*`\_\_init\_\_`\*\*: Object Initialization  
The `\_\_init\_\_` method is called when an object is created. It is used to initialize the object's attributes.  
  
#### Example:  
```python  
class Person:  
 def \_\_init\_\_(self, name, age):  
 self.name = name  
 self.age = age  
  
# Usage  
p = Person("Alice", 30)  
print(p.name) # Output: Alice  
print(p.age) # Output: 30  
```  
  
---  
  
### 2. \*\*`\_\_str\_\_`\*\*: String Representation  
The `\_\_str\_\_` method defines how an object is represented as a string (e.g., when using `print()`).  
  
#### Example:  
```python  
class Person:  
 def \_\_init\_\_(self, name, age):  
 self.name = name  
 self.age = age  
  
 def \_\_str\_\_(self):  
 return f"{self.name}, {self.age} years old"  
  
# Usage  
p = Person("Alice", 30)  
print(p) # Output: Alice, 30 years old  
```  
  
---  
  
### 3. \*\*`\_\_repr\_\_`\*\*: Official String Representation  
The `\_\_repr\_\_` method is used to provide an unambiguous string representation of an object, often for debugging.  
  
#### Example:  
```python  
class Person:  
 def \_\_init\_\_(self, name, age):  
 self.name = name  
 self.age = age  
  
 def \_\_repr\_\_(self):  
 return f"Person(name={self.name}, age={self.age})"  
  
# Usage  
p = Person("Alice", 30)  
print(repr(p)) # Output: Person(name=Alice, age=30)  
```  
  
---  
  
### 4. \*\*`\_\_add\_\_`\*\*: Overloading the `+` Operator  
The `\_\_add\_\_` method allows you to define the behavior of the `+` operator for your objects.  
  
#### Example:  
```python  
class Vector:  
 def \_\_init\_\_(self, x, y):  
 self.x = x  
 self.y = y  
  
 def \_\_add\_\_(self, other):  
 return Vector(self.x + other.x, self.y + other.y)  
  
 def \_\_str\_\_(self):  
 return f"Vector({self.x}, {self.y})"  
  
# Usage  
v1 = Vector(1, 2)  
v2 = Vector(3, 4)  
v3 = v1 + v2  
print(v3) # Output: Vector(4, 6)  
```  
  
---  
  
### 5. \*\*`\_\_len\_\_`\*\*: Length of an Object  
The `\_\_len\_\_` method is used to define the behavior of the `len()` function for your object.  
  
#### Example:  
```python  
class MyList:  
 def \_\_init\_\_(self, items):  
 self.items = items  
  
 def \_\_len\_\_(self):  
 return len(self.items)  
  
# Usage  
my\_list = MyList([1, 2, 3, 4])  
print(len(my\_list)) # Output: 4  
```  
  
---  
  
### 6. \*\*`\_\_getitem\_\_` and `\_\_setitem\_\_`\*\*: Indexing and Assignment  
These methods allow you to define how objects handle indexing and item assignment.  
  
#### Example:  
```python  
class MyList:  
 def \_\_init\_\_(self, items):  
 self.items = items  
  
 def \_\_getitem\_\_(self, index):  
 return self.items[index]  
  
 def \_\_setitem\_\_(self, index, value):  
 self.items[index] = value  
  
# Usage  
my\_list = MyList([1, 2, 3])  
print(my\_list[1]) # Output: 2  
my\_list[1] = 42  
print(my\_list[1]) # Output: 42  
```  
  
---  
  
### 7. \*\*`\_\_eq\_\_`\*\*: Equality Comparison  
The `\_\_eq\_\_` method is used to define the behavior of the `==` operator.  
  
#### Example:  
```python  
class Person:  
 def \_\_init\_\_(self, name, age):  
 self.name = name  
 self.age = age  
  
 def \_\_eq\_\_(self, other):  
 return self.name == other.name and self.age == other.age  
  
# Usage  
p1 = Person("Alice", 30)  
p2 = Person("Alice", 30)  
p3 = Person("Bob", 25)  
print(p1 == p2) # Output: True  
print(p1 == p3) # Output: False  
```  
  
---  
  
### 8. \*\*`\_\_call\_\_`\*\*: Callable Objects  
The `\_\_call\_\_` method allows an object to be called like a function.  
  
#### Example:  
```python  
class Adder:  
 def \_\_init\_\_(self, value):  
 self.value = value  
  
 def \_\_call\_\_(self, x):  
 return self.value + x  
  
# Usage  
add\_five = Adder(5)  
print(add\_five(10)) # Output: 15  
```  
  
---  
  
### 9. \*\*`\_\_iter\_\_` and `\_\_next\_\_`\*\*: Iteration  
These methods allow an object to be iterable (e.g., used in a `for` loop).  
  
#### Example:  
```python  
class Counter:  
 def \_\_init\_\_(self, start, end):  
 self.current = start  
 self.end = end  
  
 def \_\_iter\_\_(self):  
 return self  
  
 def \_\_next\_\_(self):  
 if self.current > self.end:  
 raise StopIteration  
 else:  
 self.current += 1  
 return self.current - 1  
  
# Usage  
counter = Counter(1, 5)  
for num in counter:  
 print(num) # Output: 1 2 3 4 5  
```  
  
---  
  
### 10. \*\*`\_\_del\_\_`\*\*: Object Destruction  
The `\_\_del\_\_` method is called when an object is about to be destroyed (e.g., when it goes out of scope).  
  
#### Example:  
```python  
class Person:  
 def \_\_init\_\_(self, name):  
 self.name = name  
  
 def \_\_del\_\_(self):  
 print(f"{self.name} is being deleted")  
  
# Usage  
p = Person("Alice")  
del p # Output: Alice is being deleted  
```  
  
---  
  
### Use Cases of Magic Methods  
1. \*\*Custom Data Structures\*\*: Implementing custom behavior for indexing, slicing, and iteration.  
2. \*\*Operator Overloading\*\*: Defining how objects interact with operators like `+`, `-`, `\*`, etc.  
3. \*\*Readable Debugging\*\*: Using `\_\_repr\_\_` and `\_\_str\_\_` for better debugging and logging.  
4. \*\*Custom Comparisons\*\*: Implementing comparison logic with `\_\_eq\_\_`, `\_\_lt\_\_`, etc.  
5. \*\*Function-like Objects\*\*: Using `\_\_call\_\_` to make objects behave like functions.  
  
Magic methods make Python objects highly flexible and allow you to create intuitive, Pythonic APIs for your classes.