Special Methods

In Python, special methods, also known as magic or dunder (double underscore) methods, are used to define how objects of a class behave with built-in functions and operators. These methods are surrounded by double underscores, and they provide a way to customize the behavior of your objects. Special methods are automatically called by the interpreter in response to specific operations on objects. Here are some common special methods:

1. `__init__(self, ...)`:
   - The constructor method is called when an object is created. It initializes the object's attributes.
   - Example:
     ```python
     class MyClass:
         def __init__(self, value):
             self.value = value

     obj = MyClass(42)
     ```

2. `__str__(self)`:
   - Defines the string representation of the object when `str()` is called. It is also used by the `print()` function.
   - Example:
     ```python
     class MyClass:
         def __init__(self, value):
             self.value = value

         def __str__(self):
             return f"MyClass instance with value: {self.value}"

     obj = MyClass(42)
     print(obj)  # Output: MyClass instance with value: 42
     ```

3. `__repr__(self)`:
   - Provides the unambiguous string representation of the object. It is used by the `repr()` function and is helpful for debugging.
   - Example:
     ```python
     class MyClass:
         def __init__(self, value):
             self.value = value

         def __repr__(self):
             return f"MyClass({self.value})"

     obj = MyClass(42)
     ```

    repr_obj = repr(obj)
    print(repr_obj) # Output: MyClass(42)

4. `__len__(self)`:
   - Defines the behavior of the `len()` function when applied to an object.
   - Example:
     ```python
     class MyList:
         def __init__(self, items):
             self.items = items

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

     my_list = MyList([1, 2, 3, 4])
     print(len(my_list))  # Output: 4
     ```

5. `__add__(self, other)`:
   - Defines the behavior of the `+` operator for objects of a class.
   - Example:
     ```python
     class Point:
         def __init__(self, x, y):
             self.x = x
             self.y = y

         def __add__(self, other):
             return Point(self.x + other.x, self.y + other.y)

     p1 = Point(1, 2)
     p2 = Point(3, 4)
     result = p1 + p2
     print(result.x, result.y)  # Output: 4 6
     ```

6. `__eq__(self, other)`:
   - Defines the behavior of the equality operator (`==`) for objects of a class.
   - Example:
     ```python
     class Point:
         def __init__(self, x, y):
             self.x = x
             self.y = y

         def __eq__(self, other):
             return self.x == other.x and self.y == other.y
     ```

```python
p1 = Point(1, 2)
p2 = Point(1, 2)
print(p1 == p2)  # Output: True
```

7. `__getitem__(self, key)`:
  - Enables object indexing using square brackets (`[]`).
  - Example:
```python
class MyList:
    def __init__(self, items):
        self.items = items

    def __getitem__(self, index):
        return self.items[index]

my_list = MyList([1, 2, 3, 4])
print(my_list[2])  # Output: 3
```

8. `__iter__(self)`:
  - Enables object iteration using the `iter()` function. Should return an iterator object.
  - Example:
```python
class MyRange:
    def __init__(self, start, end):
        self.start = start
        self.end = end

    def __iter__(self):
        return iter(range(self.start, self.end))

for num in MyRange(2, 5):
    print(num)
# Output:
# 2
# 3
# 4
```

These special methods allow you to customize how your objects interact with the Python language, making your classes more powerful and expressive. They are an integral part of creating Python classes that emulate built-in types or provide custom behaviors for specific operations.