Special and Magic Methods in Python

What Are Special Methods?

Special methods are predefined methods in Python that allow your objects to interact with Python's built in operations.

They usually have a name that starts and ends with **double underscores**:

`__method__`

Because of this, they are commonly called:

- **Special methods**
- **Magic methods**
- **Dunder methods**

"Dunder" comes from:

> **Double Underscore**

For example:

`__init__`
`__str__`
`__len__`
`__eq__`
`__add__`

You have already seen:

`__init__()`

It runs automatically when an object is initialized.

But `__init__()` is only one of many special methods available in Python.

Why Do Special Methods Exist?

Suppose you create your own class:

```
class Book:
    def __init__(self, title):
        self.title = title
```

You can create:

```
book = Book("Python OOP")
```

But what should happen when you write:

```
print(book)
```

Python needs to know how your object should be represented as text.

You can define that behavior with:

`__str__()`

Similarly, Python needs to know what these operations mean for your object:

```
len(book)
```

```
book1 == book2
book1 + book2
book[0]
```

Special methods allow you to define these behaviors.

The Basic Idea

Think of special methods as a way of telling Python:

> **"When Python performs this operation on my object, use this behavior."**

For example:

```
print(object)
     
__str__()

len(object)
     
__len__()

object1 == object2
     
__eq__()

object1 + object2
     
__add__()
```

```
object[index]
    ↓
__getitem__()
```

You don't normally call these methods manually.

Python calls them when the corresponding operation is used.

`__init__()`

You have already learned about `__init__()`.

It runs when an object is initialized.

```
class User:
    def __init__(self, name):
        self.name = name
```

Now:

```
user = User("Hafsa")
```

Python automatically calls:

```
user.__init__("Hafsa")
```

Conceptually, the important point is:

```
User(...)
   ↓
Object initialization
```


__init__()

You normally write:

```
user = User("Hafsa")
```

rather than:

```
user.__init__("Hafsa")
```

__str__()

__str__() defines the **human readable string representation** of an object.

Without it:

```
class User:
    def __init__(self, name):
        self.name = name


user = User("Hafsa")

print(user)
```

You may get output similar to:

```
<__main__.User object at 0x...>
```

That is not very useful to a person.

Add `__str__()`:

```
class User:
    def __init__(self, name):
        self.name = name

    def __str__(self):
        return f"User: {self.name}"
```

Now:

```
user = User("Hafsa")

print(user)
```

Output:

```
User: Hafsa
```

`__str__()` Must Return a String

This is correct:

```
def __str__(self):
    return self.name
```

This is incorrect:

```
def __str__(self):
    return 123
```

`__str__()` must return a string.

Otherwise Python raises an error when it tries to convert the object to text.

`__repr__()`

`__repr__()` provides a representation intended mainly for **developers and debugging**.

Example:

```
class User:
    def __init__(self, name, age):
        self.name = name
        self.age = age

    def __repr__(self):
        return f"User(name={self.name!r}, age={self.age!r})"
```

Now:

```
user = User("Hafsa", 25)

print(repr(user))
```

Output:

```
User(name='Hafsa', age=25)
```

`__str__()` vs `__repr__()`

The two methods have different purposes.

Method	Main Purpose
`__str__()`	Human friendly output
`__repr__()`	Developer/debug representation

Example:

```
class Product:
    def __init__(self, name, price):
        self.name = name
        self.price = price

    def __str__(self):
        return f"{self.name} - ${self.price}"

    def __repr__(self):
        return f"Product(name={self.name!r}, price={self.price!r})"
```

Then:

```
product = Product("Python Guide", 20)

print(product)
```

might produce:

`Python Guide - $20`

While:

```
print(repr(product))
```

produces:

```
Product(name='Python Guide', price=20)
```

Why `__repr__()` Matters

Suppose you have a list:

```
products = [
    Product("Python Guide", 20),
    Product("OOP Guide", 30)
]
```

A useful `__repr__()` makes debugging much easier:

```
print(products)
```

Instead of an unhelpful list of memory addresses, you can get meaningful object information.

For classes you create, a useful `__repr__()` is often valuable during development.

`__len__()`

`__len__()` defines what happens when you use:

```
len(object)
```

Example:

```
class Playlist:
    def __init__(self, songs):
        self.songs = songs

    def __len__(self):
        return len(self.songs)
```

Now:

```
playlist = Playlist([
    "Song A",
    "Song B",
    "Song C"
])

print(len(playlist))
```

Output:

```
3
```

Python effectively asks the object:

```
playlist.__len__()
```

when you call:

```
len(playlist)
```

__bool__()

__bool__() controls how an object behaves when Python needs to determine whether it is truthy or falsy.

Example:

```
class ShoppingCart:
    def __init__(self, items):
        self.items = items

    def __bool__(self):
        return len(self.items) > 0
```

Now:

```
cart = ShoppingCart([])

if cart:
    print("Cart has items")
else:
    print("Cart is empty")
```

Output:

```
Cart is empty
```

If we add an item:

```
cart.items.append("Python Guide")
```

```
if cart:
    print("Cart has items")
```

Output:

```
Cart has items
```

__eq__()

__eq__() defines what happens when you compare two objects using:

==

Consider:

```
class User:
    def __init__(self, name):
        self.name = name
```

Create:

```
user1 = User("Hafsa")
user2 = User("Hafsa")
```

These are two different objects.

By default:

```
print(user1 == user2)
```

does not mean that their attributes are automatically equal.

We can define equality ourselves:

```
class User:
    def __init__(self, name):
        self.name = name

    def __eq__(self, other):
        return self.name == other.name
```

Now:

```
user1 = User("Hafsa")
user2 = User("Hafsa")

print(user1 == user2)
```

Output:

```
True
```

Handling Different Types in `__eq__()`

A safer implementation checks the type.

```
class User:
    def __init__(self, name):
        self.name = name

    def __eq__(self, other):
        if not isinstance(other, User):
            return NotImplemented
```

```
        return self.name == other.name
```

This prevents your comparison logic from assuming that every object has a `name` attribute.

`__lt__()`, `__gt__()` and Comparison Methods

Python provides special methods for comparison operations.

Examples:

```
<  → __lt__()
<= → __le__()
>  → __gt__()
>= → __ge__()
== → __eq__()
!= → __ne__()
```

For example:

```
class Product:
    def __init__(self, price):
        self.price = price

    def __lt__(self, other):
        return self.price < other.price
```

Now:

```
product1 = Product(100)
product2 = Product(200)
```

```
print(product1 < product2)
```

Output:

```
True
```

__add__()

__add__() defines behavior for:

+

Example:

```
class Money:
    def __init__(self, amount):
        self.amount = amount

    def __add__(self, other):
        return Money(self.amount + other.amount)
```

Now:

```
money1 = Money(100)
money2 = Money(50)

total = money1 + money2

print(total.amount)
```

Output:

```
150
```

The `+` operator now has a meaning for `Money` objects.

Arithmetic Special Methods

Python provides special methods for many operators.

Operation	Special Method
`a + b`	`__add__()`
`a - b`	`__sub__()`
`a * b`	`__mul__()`
`a / b`	`__truediv__()`
`a // b`	`__floordiv__()`
`a % b`	`__mod__()`
`a ** b`	`__pow__()`

This lets your classes participate naturally in Python expressions.

`__getitem__()`

`__getitem__()` controls indexing and key access.

It allows objects to support:

```
object[index]
```

Example:

```
class Playlist:
    def __init__(self, songs):
        self.songs = songs

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

Now:

```
playlist = Playlist([
    "Song A",
    "Song B",
    "Song C"
])

print(playlist[0])
```

Output:

```
Song A
```

Python calls:

```
playlist.__getitem__(0)
```

Supporting Slicing

`__getitem__()` can also receive a slice.

```
class Playlist:
    def __init__(self, songs):
        self.songs = songs

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

Now:

```
print(playlist[0:2])
```

works because the slice is passed to `__getitem__()`.

This allows custom objects to behave like sequences.

`__setitem__()`

`__setitem__()` controls assignment through indexing.

Example:

```
class Playlist:
    def __init__(self, songs):
        self.songs = songs

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

    def __setitem__(self, index, value):
        self.songs[index] = value
```

Now:

```
playlist[0] = "New Song"
```

works.

Python calls:

```
playlist.__setitem__(0, "New Song")
```

`__contains__()`

`__contains__()` controls the `in` operator.

Example:

```
class Course:
    def __init__(self, topics):
        self.topics = topics
```

```
    def __contains__(self, topic):
        return topic in self.topics
```

Now:

```
course = Course([
    "Python",
    "OOP",
    "DSA"
])

print("OOP" in course)
```

Output:

```
True
```

__iter__()

__iter__() allows an object to be used in a for loop.

Example:

```
class Playlist:
    def __init__(self, songs):
        self.songs = songs

    def __iter__(self):
        return iter(self.songs)
```

Now:

```
playlist = Playlist([
    "Song A",
    "Song B",
    "Song C"
])

for song in playlist:
    print(song)
```

Output:

```
Song A
Song B
Song C
```

The object itself is now iterable.

__call__()

One of the more interesting special methods is:

```
__call__()
```

It allows an object to be called like a function.

Example:

```
class Greeter:
    def __call__(self, name):
        return f"Hello, {name}!"
```

Create an object:

```
greet = Greeter()
```

Normally, an object is called like this:

```
greet("Hafsa")
```

Because `Greeter` defines `__call__()`.

Output:

```
Hello, Hafsa!
```

Python effectively calls:

```
greet.__call__("Hafsa")
```

Why Would You Make an Object Callable?

This is useful when an object represents an operation or configurable behavior.

For example:

```
class Multiplier:
    def __init__(self, number):
        self.number = number

    def __call__(self, value):
        return value * self.number
```

Now:

```
double = Multiplier(2)

print(double(5))
```

Output:

```
10
```

The object behaves like a function while still being able to store state.

`__enter__()` and `__exit__()`

These methods are used to create **context managers**.

They work with:

```
with
```

For example:

```
class FileManager:
    def __enter__(self):
        print("Opening resource")
        return self

    def __exit__(self, exc_type, exc_value, traceback):
        print("Closing resource")
```

Now:

```
with FileManager():
    print("Working...")
```

Output:

```
Opening resource
Working...
Closing resource
```

This pattern is useful for resources that need setup and cleanup.

Examples include:

- files
- database connections
- locks
- network resources

`__del__()`

`__del__()` is associated with object cleanup.

Example:

```
class Example:
    def __del__(self):
        print("Object is being cleaned up")
```

However, you should **not rely on `__del__()` for important resource cleanup**.

Python's garbage collection and object lifetime behavior mean that `__del__()` is not the best place for critical cleanup logic.

For resources such as files, prefer:

`with`

and context managers.

`__hash__()`

`__hash__()` allows an object to provide a hash value.

Hashable objects can be used as:

- dictionary keys
- members of sets

For example, immutable built-in values such as strings and integers are hashable.

Custom classes can have hashing behavior too, but equality and hashing must be designed consistently.

If:

`a == b`

is `True`, then their hashes should also be equal.

`hash(a) == hash(b)`

This is an important rule when implementing `__eq__()` and `__hash__()` together.

`__new__()` vs `__init__()`

These two methods are often confused.

`__new__()`

Responsible for **creating the object**.

`__init__()`

Responsible for **initializing the object** after it has been created.

Conceptually:

```
Class(...)
   
__new__()
   
Object created
   
__init__()
   
Object initialized
```

Most everyday Python classes only need:

```
__init__()
```

You usually do not need to override `__new__()`.

Special Methods Are Usually Called by Python

Instead of:

```
object.__str__()
```

you normally write:

```
str(object)
```

Instead of:

```
object.__len__()
```

you write:

```
len(object)
```

Instead of:

```
object.__eq__(other)
```

you write:

```
object == other
```

Instead of:

```
object.__getitem__(0)
```

you write:

```
object[0]
```

The special method provides the implementation behind the normal Python syntax.

A Complete Example

Let's create a `ResourceCollection` similar to a collection of haas.dev resources.

```
class ResourceCollection:
    def __init__(self, resources):
        self.resources = resources

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

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

    def __contains__(self, resource):
        return resource in self.resources

    def __str__(self):
        return f"ResourceCollection({len(self.resources)} resources)"

    def __repr__(self):
        return f"ResourceCollection(resources={self.resources!r})"
```

Now:

```
resources = ResourceCollection([
    "Python OOP",
    "Python Exceptions",
    "Python Files"
])
```

You can use:

```
print(resources)
```

Output:

```
ResourceCollection(3 resources)
```

You can use:

```
print(len(resources))
```

Output:

```
3
```

You can use indexing:

```
print(resources[0])
```

Output:

```
Python OOP
```

You can use membership testing:

```
print("Python OOP" in resources)
```

Output:

```
True
```

One custom class now behaves naturally with several normal Python operations.

That is the power of special methods.

Special Methods and Python's Data Model

Python has a concept called the **data model**.

The data model defines how objects interact with Python's language features.

Special methods are a major part of that model.

For example:

```
Object behavior
    
Special methods
    
Python syntax and built-ins
```

Examples:

```
len(x)       x.__len__()
str(x)       x.__str__()
repr(x)      x.__repr__()
x + y        x.__add__(y)
x == y       x.__eq__(y)
x[i]         x.__getitem__(i)
item in x    x.__contains__(item)
for item in x  x.__iter__()
```

Once you understand this relationship, many Python features become easier to understand.

Special Methods and Polymorphism

Special methods are another example of polymorphic behavior.

Consider:

```
len("Python")
len([1, 2, 3])
len((10, 20))
```

The same operation:

```
len()
```

works with different objects.

Each type provides the appropriate behavior.

Your own class can participate in the same system by implementing:

```
__len__()
```

This allows your class to integrate with Python's existing syntax.

Common Mistake: Calling Special Methods Directly Everywhere

You technically can write:

```
user.__str__()
```

But normally you should use:

`str(user)`

Similarly:

`len(user)`

is preferable to:

`user.__len__()`

Special methods are primarily designed to support Python's normal operations.

Common Mistake: Returning the Wrong Type

Special methods often have expected return values.

For example:

`__str__()`

must return a string.

`__len__()`

must return an integer.

If the method returns an inappropriate value, Python can raise an error.

Always understand the contract of the special method you implement.

Common Mistake: Implementing Too Many Methods

You do not need to implement every special method.

Only implement the behavior your class actually needs.

For example, if your class does not need indexing:

`object[0]`

there is no reason to implement:

`__getitem__()`

Good class design focuses on meaningful behavior rather than adding special methods simply because they exist.

Common Mistake: Breaking Equality and Hashing

Be careful when implementing:

`__eq__()`

and:

`__hash__()`

Objects used as dictionary keys or set members need consistent hashing behavior.

If you change equality semantics without considering hashing, you can create confusing bugs.

For mutable objects, hashing also requires particular care.

When Should You Use Special Methods?

Use special methods when you want your custom class to naturally support Python features such as:

`print()`
`len()`
`==`
`!=`
`<`
`>`
`+`
`-`
`*`
`[]`
`in`
`for`
`with`
`()`

They are especially useful when your class represents something that naturally behaves like:

- a collection
- a number
- a resource
- a callable operation

- a value object
- a context-managed resource

When Should You Not Use Them?

Do not use special methods merely to make code look clever.

For example, if a class does not naturally represent a number, do not overload `+` just because you can.

Special methods should make the object's behavior **intuitive**.

A good rule:

> **If the Python syntax makes sense for the concept your class represents, a special method may be appropriate.**

Mini Project: Custom Shopping Cart

Create a `ShoppingCart` class that supports:

`len(cart)`

to return the number of products.

It should support:

`product in cart`

to check whether a product exists.

It should support:

```
print(cart)
```

to display a readable description.

It should support:

```
cart[0]
```

to access a product.

Suggested structure:

```
class ShoppingCart:
    def __init__(self):
        self.items = []

    def add(self, item):
        self.items.append(item)

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

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

    def __contains__(self, item):
        return item in self.items

    def __str__(self):
```

```
        return f"ShoppingCart with {len(self.items)} items"
```

Test it with:

```
cart = ShoppingCart()

cart.add("Python Guide")
cart.add("OOP Guide")

print(cart)
print(len(cart))
print(cart[0])
print("OOP Guide" in cart)
```

---

5 Minute Challenge

Create a `Book` class with:

```
title
author
pages
```

Implement:

```
__str__()
__repr__()
__len__()
__eq__()
```

Then test:

```
book1 = Book("Python Basics", "Hafsa", 200)
```

```
book2 = Book("Python Basics", "Hafsa", 200)

print(book1)
print(repr(book1))
print(len(book1))
print(book1 == book2)
```

Think about what each operation should mean for a `Book`.

Exercises

Exercise 1

Create a `Playlist` class that supports:

```
len(playlist)
playlist[0]
"Song" in playlist
```

Use:

```
__len__()
__getitem__()
__contains__()
```

Exercise 2

Create a `Money` class that supports:

```
money1 + money2
```

Implement:

`__add__()`

Exercise 3

Create a `Student` class.

Two students should be considered equal if they have the same student ID.

Implement:

`__eq__()`

Exercise 4

Create a `Multiplier` object that can be called like:

```
double = Multiplier(2)

print(double(10))
```

Implement:

`__call__()`

Exercise 5

Create a custom collection that supports:

```
for item in collection:
    print(item)
```

Implement:

`__iter__()`

Mini Quiz

1. What does "dunder" mean?

A. Double data
B. Double underscore
C. Dynamic number
D. Data under

Answer: B

---

2. Which method controls `str(object)`?

A. `__repr__()`
B. `__len__()`
C. `__str__()`
D. `__text__()`

Answer: C

---

3. Which method controls `len(object)`?

A. `__size__()`
B. `__len__()`
C. `__count__()`
D. `__length__()`

Answer: B

---

4. Which method controls `object1 + object2`?

A. `__plus__()`
B. `__sum__()`
C. `__add__()`
D. `__combine__()`

Answer: C

---

5. Which method allows an object to be called like a function?

A. `__run__()`
B. `__execute__()`
C. `__call__()`
D. `__function__()`

Answer: C

---

6. Which method controls indexing such as `object[0]`?

A. `__index__()`
B. `__getitem__()`
C. `__get__()`
D. `__position__()`

Answer: B

Interview Questions

Beginner

1. What are magic methods in Python?

Magic methods, more accurately called special or dunder methods, are predefined methods that allow custom objects to interact with Python's built-in operations and syntax.

2. What does `__init__()` do?

It initializes an object after it has been created.

3. What does `__str__()` do?

It provides a human-readable string representation of an object.

4. What does `__repr__()` do?

It provides a developer-oriented representation of an object, mainly useful for debugging.

Intermediate

5. What is the difference between `__str__()` and `__repr__()`?

`__str__()` focuses on readable output for users, while `__repr__()` focuses on an informative representation for developers.

6. How does `len()` work with a custom object?

If the object implements `__len__()`, Python uses that method when `len(object)` is called.

7. What is operator overloading?

Operator overloading means defining how operators such as `+`, `==`, or `<` behave for custom objects using special methods.

Advanced

8. What is `__call__()` used for?

It allows an object to be called using function-call syntax:

`object()`

9. What is the difference between `__new__()` and `__init__()`?

`__new__()` is responsible for creating the object, while `__init__()` initializes the object after creation.

10. Why should `__del__()` generally not be relied upon for resource cleanup?

Object destruction timing and garbage collection behavior mean it is not a reliable mechanism for critical cleanup. Context managers are generally preferred for managed resources.

Special Methods Cheat Sheet

Python Operation	Special Method
Object initialization	`__init__()`
`str(obj)`	`__str__()`
`repr(obj)`	`__repr__()`
`len(obj)`	`__len__()`
`bool(obj)`	`__bool__()`
`a == b`	`__eq__()`
`a != b`	`__ne__()`
`a < b`	`__lt__()`

a <= b	__le__()
a > b	__gt__()
a >= b	__ge__()
a + b	__add__()
a - b	__sub__()
a * b	__mul__()
a / b	__truediv__()
obj[index]	__getitem__()
obj[index] = value	__setitem__()
item in obj	__contains__()
for item in obj	__iter__()
obj(...)	__call__()
with obj	__enter__() / __exit__()

Object creation	`__new__()`

Key Takeaways

1. **Special methods are predefined methods that integrate custom objects with Python's language features.**
2. They usually use the `__name__` format and are called **dunder methods**.
3. `__init__()` initializes objects.
4. `__str__()` provides human-readable output.
5. `__repr__()` provides a developer-friendly representation.
6. `__len__()` makes `len(object)` work.
7. `__eq__()` defines equality.
8. `__add__()` defines addition for custom objects.
9. `__getitem__()` enables indexing.
10. `__iter__()` enables iteration.
11. `__call__()` makes objects callable.
12. `__enter__()` and `__exit__()` support context managers.
13. Special methods are the mechanism behind much of Python's object behavior.
14. Implement them when the corresponding Python operation naturally makes sense for your class.
15. Do not add special methods simply for the sake of using them.

The core mental model is:

```
Python syntax
    
Special method
    
Your object's behavior
```

For example:

```
len(cart)
  
cart.__len__()

cart[0]
  
cart.__getitem__(0)

cart1 + cart2
  
cart1.__add__(cart2)

print(cart)
  
cart.__str__()
```

Visit haas.dev for more resources and guides.

https://dev-roast-app.vercel.app