🧠 Master the 🔮 Python Magic Methods (`__dunder__` methods)

These are **predefined methods** in Python that **start and end with double underscores** like `__init__`, `__str__`, `__len__`, etc. They let you *customize the behavior* of your classes and make them behave like built-in types!

🏗 Object Construction & Initialization

Magic Method	Purpose	Example Use
`__new__`	Create a new instance (rarely used)	Metaclass or Singleton
`__init__`	Initialize object after creation	Setting attributes

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

b = Book("Python Mastery")
print(b.title)  # Python Mastery
```

🎭 String & Representation

Magic Method	Purpose	Example Use
`__str__`	Human-readable string (`str()`)	`print(obj)` output
`__repr__`	Debug/developer string (`repr()`)	Used in logging or debugging

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

    def __str__(self):
        return f"📘 {self.title}"

    def __repr__(self):
        return f"Book(title='{self.title}')"

b = Book("Magic Python")
print(str(b))    # 📘 Magic Python
print(repr(b))   # Book(title='Magic Python')
```

📐 Length, Size, Boolean

Magic Method	Purpose	Example Use
`__len__`	Length via `len()`	`len(obj)`
`__bool__`	Truth value via `bool()`	`if obj:`

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

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

    def __bool__(self):
        return bool(self.items)

b = Basket(['🍏', '🍌'])
print(len(b))      # 2
print(bool(b))     # True
```

➕ Arithmetic Operations

Magic Method	Operator Overloaded
`__add__`	`+`
`__sub__`	`-`
`__mul__`	`*`
`__truediv__`	`/`
`__floordiv__`	`//`
`__mod__`	`%`
`__pow__`	`**`

```python
class Price:
    def __init__(self, value):
        self.value = value

    def __add__(self, other):
        return Price(self.value + other.value)

    def __str__(self):
        return f"${self.value:.2f}"

p1 = Price(40)
```

```
p2 = Price(60)
print(p1 + p2)  # $100.00
```

📊 Comparison Operators

Magic Method	Operator
`__eq__`	`==`
`__ne__`	`!=`
`__lt__`	`<`
`__le__`	`<=`
`__gt__`	`>`
`__ge__`	`>=`

```
class Student:
    def __init__(self, marks):
        self.marks = marks

    def __eq__(self, other):
        return self.marks == other.marks

a = Student(85)
b = Student(85)
print(a == b)  # True
```

🎒 Collections & Indexing

Magic Method	Purpose
`__getitem__`	Access items `obj[i]`
`__setitem__`	Set items `obj[i] = x`
`__delitem__`	Delete items
`__contains__`	`in` operator
`__iter__`	Make iterable
`__next__`	For `next()` in loops

```
class MyList:
    def __init__(self):
        self.data = []
    def __getitem__(self, index):
        return self.data[index]

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

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

l = MyList()
l.data = [1, 2, 3]
print(l[1])     # 2
l[1] = 5
print(5 in l)   # True
```

🔄 Context Managers

Magic Method	Purpose
`__enter__`	Start of `with` block
`__exit__`	End of `with` block

```
class Demo:
    def __enter__(self):
        print("Entering...")
        return self

    def __exit__(self, exc_type, exc_val, exc_tb):
        print("Exiting...")

with Demo():
    print("Inside block")

# Output:
# Entering...
# Inside block
# Exiting...
```

🧙 Advanced Magic Methods

Method	Use Case
`__call__`	Make object callable like function
`__del__`	Called on object deletion

Method	Use Case
`__copy__`	Shallow copy of object
`__deepcopy__`	Deep copy
`__slots__`	Memory optimization

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

    def __call__(self, number):
        return self.factor * number

double = Multiplier(2)
print(double(5))  # 10
```

🌓 Meta & Dynamic Stuff

Magic Method	Description
`__getattr__`	Called if attribute not found normally
`__setattr__`	Triggered on attribute assignment
`__delattr__`	When attribute is deleted
`__dir__`	Customize `dir(obj)`

📚 Bonus: Dunder Methods You See Often

Method	Why You See It
`__main__`	Run file directly
`__name__`	Module name access
`__doc__`	String documentation

```
# In script file:
if __name__ == "__main__":
    print("This script is running directly")
```

🧩 Practice Tasks

1. Create a class with custom `__str__`, `__add__`, and `__len__`.

2. Implement a class that supports indexing with `__getitem__` and `__setitem__`.
3. Write a callable object with `__call__`.
4. Create a context manager using `__enter__` and `__exit__`.

🪄 Wrap Up

Dunder methods let you:

- 🧬 Make objects *feel like* built-ins
- 🪞 Customize operators and behavior
- 🧩 Add flexibility and magic to your classes

Mastering these will *level up your OOP wizardry* in Python! 🧠💫

🧙 The Ultimate Guide to Python `__dunder__` (Magic) Methods 💫

Magic methods, also known as **dunder (double underscore) methods**, let you define how your objects behave with built-in Python operations (like printing, adding, comparing, indexing, etc.).

🌟 Why Learn Magic Methods?

Feature	Benefit
`__str__`, `__repr__`	Custom string output for objects
`__add__`, `__sub__`, `__mul__`	Operator overloading
`__getitem__`, `__setitem__`	Make objects subscriptable
`__len__`, `__contains__`	Use with `len()` and `in`
`__enter__`, `__exit__`	Context manager support
`__call__`, `__iter__`	Make objects callable or iterable

🗂️ Categorized Dunder Methods

🔤 String Representation

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

    def __str__(self):
        return f"{self.title} by {self.author}"  # human-readable
    def __repr__(self):
        return f"Book('{self.title}', '{self.author}')"  # debug-friendly

b = Book("Python 101", "Darshan")
print(str(b))    # Python 101 by Darshan
print(repr(b))   # Book('Python 101', 'Darshan')
```

➕ Operator Overloading

```python
class Point:
    def __init__(self, x, y):
        self.x, self.y = x, y

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

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

    def __str__(self):
        return f"({self.x}, {self.y})"

p1 = Point(1, 2)
p2 = Point(3, 4)
print(p1 + p2)    # (4, 6)
print(p1 == Point(1, 2))  # True
```

📦 Object Construction

```python
class MyClass:
    def __new__(cls, *args, **kwargs):
        print("Creating instance")
        return super().__new__(cls)

    def __init__(self):
        print("Initializing instance")

obj = MyClass()
```

📞 Callable Objects

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

```python
greet = Greeter()
print(greet("Darshan"))  # Hello, Darshan!
```

🔁 Iteration Support

```python
class CountDown:
    def __init__(self, start):
        self.current = start

    def __iter__(self):
        return self

    def __next__(self):
        if self.current <= 0:
            raise StopIteration
        val = self.current
        self.current -= 1
        return val

for num in CountDown(5):
    print(num)
```

🧩 Container Behavior

```python
class MyList:
    def __init__(self, data):
        self.data = data

    def __getitem__(self, idx):
        return self.data[idx]

    def __setitem__(self, idx, value):
        self.data[idx] = value

    def __delitem__(self, idx):
        del self.data[idx]

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

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

ml = MyList([1, 2, 3])
print(ml[1])        # 2
ml[1] = 10
```

```
print(ml.data)      # [1, 10, 3]
print(3 in ml)      # True
```

🔐 Context Managers

```python
class OpenFile:
    def __init__(self, filename, mode):
        self.filename = filename
        self.mode = mode

    def __enter__(self):
        self.file = open(self.filename, self.mode)
        return self.file

    def __exit__(self, exc_type, exc_value, traceback):
        self.file.close()

with OpenFile("sample.txt", "w") as f:
    f.write("Hello from context manager!")
```

🧠 Custom Attribute Behavior

```python
class Demo:
    def __getattr__(self, name):
        return f"{name} not found!"

    def __setattr__(self, name, value):
        print(f"Setting {name} to {value}")
        self.__dict__[name] = value

    def __delattr__(self, name):
        print(f"Deleting {name}")
        del self.__dict__[name]

obj = Demo()
obj.x = 10         # Setting x to 10
print(obj.y)       # y not found!
```

🚀 Object Lifecycle

```python
class Person:
    def __init__(self, name):
        self.name = name
    def __del__(self):
        print(f"{self.name} has been deleted.")

p = Person("Darshan")
del p  # Darshan has been deleted.
```

🧾 Quick Reference Table

Method	Purpose
`__init__`	Object constructor
`__new__`	Object creation (before `__init__`)
`__str__`	Human-readable string
`__repr__`	Developer-readable string
`__add__`, `__sub__`, `__mul__`	Operator overloading
`__eq__`, `__lt__`, etc.	Comparison methods
`__getitem__`, `__setitem__`, `__delitem__`	Subscript support
`__len__`, `__contains__`	Use with `len()`, `in`
`__call__`	Make instance callable
`__iter__`, `__next__`	Iteration support
`__enter__`, `__exit__`	Context manager support
`__getattr__`, `__setattr__`, `__delattr__`	Attribute hooks
`__del__`	Destructor

🧠 Pro Tips

- Only use magic methods when you **need to customize default behaviors**.
- Always return `NotImplemented` in binary magic methods when types are incompatible.
- Pair magic methods with **docstrings** for clarity.

🧪 Challenge: Create Your Own List-Like Class

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

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

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

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

    def __str__(self):
        return str(self.items)

cl = CustomList()
cl.items = [10, 20, 30]
print(cl[1])      # 20
cl[1] = 99
print(cl)         # [10, 99, 30]
```
