Python


Python Dunder
Methods
__init__(
self.str= str
def __repr__(self)
return 'Object:
def __add__(self, o
return self.str
__name__ == '__main
str1 = String('

Dunder or magic methods in Python

Dunder or magic methods in Python are the methods having two prefix and suffix underscores in the method name. Dunder here means "Double Under (Underscores)". These are commonly used for operator overloading. Few examples for magic methods are: `__init__`, `__add__`, `__len__`, `__repr__` etc.

The `__init__` method for initialization is invoked without any call, when an instance of a class is created, like constructors in certain other programming languages such as C++, Java, C#, PHP etc. These methods are the reason we can add two strings with '+' operator without any explicit typecasting.

Here, the word dunder means double under (underscore). These special dunder methods are used in case of operator overloading (they provide extended meaning beyond the predefined meaning to an operator). Some of the examples of most common dunder methods in use are **__int__,__new__, __add__, __len__, and __str__ method**.

Therefore, **when you want to overload certain operators to make them work in a custom way with your own objects**, you need to implement the respective dunder methods. 09-Jul-2022

Here's a simple implementation :

```
# declare our own string class
class String:

    # magic method to initiate object
    def __init__(self, string):
        self.string = string

# Driver Code
if __name__ == '__main__':

    # object creation
    string1 = String('Hello')

    # print object location
    print(string1)
```

Output :

```
<__main__.String object at 0x7fe992215390>
```

The above snippet of code prints only the memory address of the string object. Let's add a `__repr__` method to represent our object.

```
# declare our own string class
class String:

    # magic method to initiate object
    def __init__(self, string):
        self.string = string

    # print our string object
    def __repr__(self):
        return 'Object: {}'.format(self.string)

# Driver Code
if __name__ == '__main__':

    # object creation
    string1 = String('Hello')

    # print object location
    print(string1)
```

Output :

```
Object: Hello
```

If we try to add a string to it :

```
# declare our own string class
class String:

    # magic method to initiate object
    def __init__(self, string):
        self.string = string

    # print our string object
    def __repr__(self):
        return 'Object: {}'.format(self.string)

# Driver Code
if __name__ == '__main__':

    # object creation
    string1 = String('Hello')

    # concatenate String object and a string
    print(string1 +' world')
```

Output :

```
TypeError: unsupported operand type(s) for +: 'String' and 'str'
```

Now add __add__ method to String class :

```
# declare our own string class
class String:

    # magic method to initiate object
    def __init__(self, string):
        self.string = string

    # print our string object
    def __repr__(self):
        return 'Object: {}'.format(self.string)

    def __add__(self, other):
        return self.string + other

# Driver Code
if __name__ == '__main__':

    # object creation
    string1 = String('Hello')

    # concatenate String object and a string
    print(string1 +' Geeks')
```

Output :

```
Hello Geeks
```

Dunder method	Usage / Needed for
`__init__`	Initialise object
`__new__`	Create object
`__del__`	Destroy object
`__repr__`	Compute “official” string representation / `repr(obj)`
`__str__`	Pretty print object / `str(obj)` / `print(obj)`
`__bytes__`	`bytes(obj)`
`__format__`	Custom string formatting
`__lt__`	`obj < ...`
`__le__`	`obj <= ...`
`__eq__`	`obj == ...`
`__ne__`	`obj != ...`
`__gt__`	`obj > ...`
`__ge__`	`obj >= ...`
`__hash__`	`hash(obj)` / object as dictionary key
`__bool__`	`bool(obj)` / define Truthy/Falsy value of object

`__getattr__`	Fallback for attribute access
`__getattribute__`	Implement attribute access: `obj.name`
`__setattr__`	Set attribute values: `obj.name = value`
`__delattr__`	Delete attribute: `del obj.name`
`__dir__`	`dir(obj)`
`__get__`	Attribute access in descriptor
`__set__`	Set attribute in descriptor
`__delete__`	Attribute deletion in descriptor
`__init_subclass__`	Initialise subclass
`__set_name__`	Owner class assignment callback
`__instancecheck__`	`isinstance(obj, ...)`
`__subclasscheck__`	`issubclass(obj, ...)`

Method	Operation	Method	Operation	Method	Operation
x.__divmod__(self, y)	divmod(x, y)	x.__rdivmod__(self, y)	divmod(y, x)		
x.__invert(self)	~x	x.__index__(self)	operator.index(x)	x.__ceil__(self)	ceil(x)
x.__pos__(self)	+x	x.__complex__(self)	complex(x)	x.__floor__(self)	floor(x)
x.__add__(self, y)	x + y	x.__radd__(self, y)	y + x	x.__iadd__(self, y)	x += y
x.__and__(self, y)	x & y	x.__rand__(self, y)	y & x	x.__iand__(self, y)	x &= y
x.__floordiv__(self, y)	x // y	x.__rfloordiv__(self, y)	y // x	x.__ifloordiv__(self, y)	x //= y
x.__lshift__(self, y)	x << y	x.__rlshift__(self, y)	y << x	x.__ilshift__(self, y)	x <<= y
x.__matmul__(self, y)	x @ y	x.__rmatmul__(self, y)	y @ x	x.__imatmul__(self, y)	x @= y
x.__mod__(self, y)	x % y	x.__rmod__(self, y)	y % x	x.__imod__(self, y)	x %= y
x.__mul__(self, y)	x * y	x.__rmul__(self, y)	y * x	x.__imul__(self, y)	x *= y
x.__or__(self, y)	x | y	x.__ror__(self, y)	y | x	x.__ior__(self, y)	x |= y
x.__pow__(self, y)	x ** y	x.__rpow__(self, y)	y ** x	x.__ipow__(self, y)	x **= y
x.__rshift__(self, y)	x >> y	x.__rrshift__(self, y)	y >> x	x.__irshift__(self, y)	x >>= y
x.__sub__(self, y)	x - y	x.__rsub__(self, y)	y - x	x.__isub__(self, y)	x -= y
x.__truediv__(self, y)	x / y	x.__rtruediv__(self, y)	y / x	x.__itruediv__(self, y)	x /= y
x.__xor__(self, y)	x ^ y	x.__rxor__(self, y)	y ^ x	x.__ixor__(self, y)	x ^= y
x.__abs__(self)	abs(x)	x.__float__(self)	float(x)	x.__round__(self[,n])	round(x)
x.__neg__(self)	-x	x.__int__(self)	int(x)	x.__trunc__(self)	trunc(x)