🪄 Magic Methods 🪄 and Operator Overloads

Announcements

- **LS14: Magic Methods and Operator Overloads** due today at 11:59pm
- **LS15: Class Survey** due Tuesday (April 7) at 11:59pm
- **EX07: River Simulation** released today, due Thursday (April 9) at 11:59pm

Review: Magic Methods

- Methods with built-in functionality!
- Not called *directly!*
- Names start and end with two underscores (__<method_name>__)

`__str__` method

```
def __str__(self) -> str:
    """Return a string representation for people."""
    return _____
```

`__repr__` method

```
def __repr__(self) -> str:
    """Return a string representation for Python evaluation."""
    return _____________________________
```

Operator Overloads

- You can write magic methods to give operators meaning!
- Think about operators you use on numbers that you'd like to use on other objects, e.g. `+`, `-`, `*`, `/`, `<`, `<=`, etc…
- This is called operator overloading

Arithmetic Operator Overloads

`+`	`__add__(self, other)`
`-`	`__sub__(self, other)`
`*`	`__mul__(self, other)`
`/`	`__truediv__(self, other)`
`**`	`__pow__(self, other)`
`%`	`__mod__(self, other)`

Comparison Operator Overloads

`<`	`__lt__(self, other)`
`>`	`__gt__(self, other)`
`<=`	`__le__(self, other)`
`>=`	`__ge__(self, other)`
`==`	`__eq__(self, other)`
`!=`	`__ne__(self, other)`

`__add__` Magic Method

- Allows you to define addition between an object of the class and another object. (Adds `self` and another object together.)
- Signature: `def __add__(self, <other_object>: <data_type>)`
- Call it by calling `<class_object> + <other_object>`

Practice: Consider this `BankAccount` class

```python
class BankAccount:
    """A simple bank account with a few attributes."""

    owner: str
    balance: float
    currency: str

    def __init__(self, owner: str, balance: float, currency: str):
        """Initialize a BankAccount instance."""
        self.owner = owner
        self.balance = balance
        self.currency = currency

    def __str__(self) -> str:
        """Return a human-readable string representation of the account."""
        return f"{self.owner}'s account: ${self.balance} {self.currency}"
```

Practice: Writing an `__add__` magic method

```python
class BankAccount:
    """A simple bank account with a few attributes."""

    owner: str
    balance: float
    currency: str

    def __init__(self, owner: str, balance: float, currency: str):
        """Initialize a BankAccount instance."""
        self.owner = owner
        self.balance = balance
        self.currency = currency

    def __str__(self) -> str:
        """Return a human-readable string representation of the account."""
        return f"{self.owner}'s account: ${self.balance} {self.currency}"
```

Our method should:

1. Take a single parameter `amount`: a `float` representing dollars to deposit
2. Return a **new** `BankAccount` object with the same `owner` and `currency`, but with `amount` added to the `balance` (do not modify `self`!)

Practice: Writing an `__add__` magic method

```
def __add__(self, amount: float) -> BankAccount:
    """Return a new BankAccount with `amount` deposited into it."""
    return BankAccount(self.owner, self.balance + amount, self.currency)
```

By defining `__add__` ourselves, we're telling Python exactly what the + operator should mean for a `BankAccount` object. (We're not changing how + works for any other data type.)

Example behavior:

```
account = BankAccount("Alice", 200.0)
richer  = account + 50.0
print(account)    # Alice's account: $200.0 USD
print(richer)     # Alice's account: $250.0 USD
```

Practice: Writing an `__mul__` magic method

```python
class BankAccount:
    """A simple bank account with a few attributes."""

    owner: str
    balance: float
    currency: str

    def __init__(self, owner: str, balance: float, currency: str):
        """Initialize a BankAccount instance."""
        self.owner = owner
        self.balance = balance
        self.currency = currency

    def __str__(self) -> str:
        """Return a human-readable string representation of the account."""
        return f"{self.owner}'s account: ${self.balance} {self.currency}"
```

Our method should:

1. Take a single parameter `factor`: a `float` to multiply the balance by (e.g. `1.05` for 5% interest)
2. Return a **new** `BankAccount` with the same `owner` and `currency`, but with `balance` multiplied by `factor` (do not modify `self`!)

Practice: Writing a `__mul__` magic method

```
def __mul__(self, factor: float) -> BankAccount:
    """Return a new BankAccount whose balance is scaled by `factor`."""
    return BankAccount(self.owner, self.balance * factor, self.currency)
```

By defining `__mul__` ourselves, we're telling Python exactly what the * operator should mean for a `BankAccount` object. (We're not changing how * works for any other data type.)

Example behavior:

```
Account = BankAccount("Alice", 200.0)
with_interest = account * 1.05
print(account)          # Alice's account: $200.00 USD
print(with_interest)    # Alice's account: $210.00 USD
```