Magic Methods + Operator Overloads

Review

What are unique properties of the __init__ method? (What sets it apart from other methods?)

Magic Methods

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

Question

When I call print(x), Python calls what function on x *before* printing?

__str__ Magic Method

- Gives a str representation to an object of a Class.
- Call it by calling str(<class_object>)

__str__ Magic Method

- Gives a str representation to an object of a Class.
- Call it by calling str(<class_object>)

Let's try it together!

Let's define a __str__ magic method that returns a string of information about a Profile object.

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)

For each magic method call, what is self and (if applicable) what is other?

str(a)	__str__(self)
a + b	__add__(self, other)
a – b	__sub__(self, other)
a * b	__mul__(self, other)
a < b	__lt__(self, other)
a == b	__eq__(self, other)

Class Writing

- Write a class called ShoppingGuide
- It should have three attributes, groceries: list[str], budget: float, and store: str
- Write a *constructor* that takes four parameters: self, groceries: list[str], budget: float, and store: str. It should update the attributes accordingly.
- Write a magic method __add__ that takes as parameters self, more_money: float.
    - It should return a new ShoppingGuide object with the same attribute values except it should add more_money to the budget

Instantiation

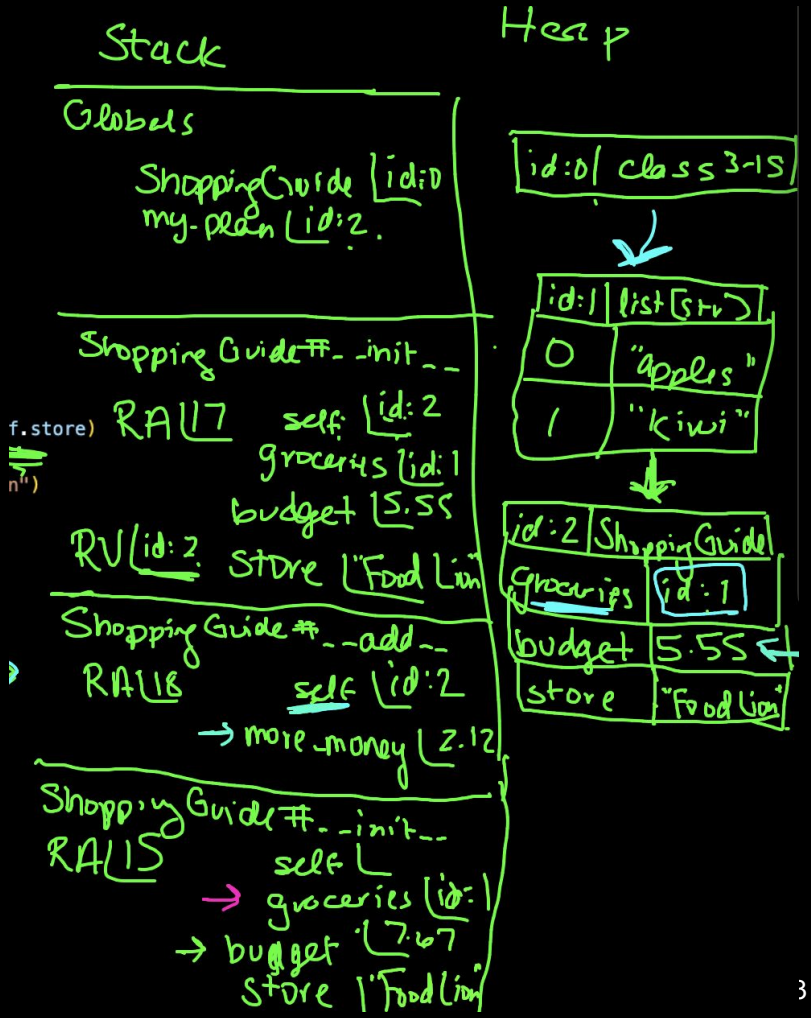
- Create a new variable my_plan that is reference to a Shopping Guide object with the groceries [“apples”, “kiwi”], the budget $5.55, and the store “Food Lion”.
- Now create another variable AJs_plan that is my_plan but with $2.12 added to the budget

Diagramming

```python
from __future__ import annotations

class ShoppingGuide:

    groceries: list[str]
    budget: float
    store: str

    def __init__(self, groceries: list[str], budget: float, store: str):
        self.groceries = groceries
        self.budget = budget
        self.store = store

    def __add__(self, more_money: float) -> ShoppingGuide:
        return ShoppingGuide(self.groceries, self.budget + more_money, self.store)

my_plan: ShoppingGuide = ShoppingGuide(["apples", "kiwi"], 5.55, "Food Lion")
AJs_plan: ShoppingGuide = my_plan + 2.12
```

Diagramming

```python
from __future__ import annotations

class ShoppingGuide:

    groceries: list[str]
    budget: float
    store: str

    def __init__(self, groceries: list[str], budget: float, store: str):
        self.groceries = groceries
        self.budget = budget
        self.store = store

    def __add__(self, more_money: float) -> ShoppingGuide:
        return ShoppingGuide(self.groceries, self.budget + more_money, self.store)

my_plan: ShoppingGuide = ShoppingGuide(["apples", "kiwi"], 5.55, "Food Lion")
AJs_plan: ShoppingGuide = my_plan + 2.12
```



Try it yourself!

- Write a __str__ magic method that gives me all the information of a ShoppingGuide object
- Change the __add__ magic method to add a list of more groceries instead of adding money to the budget. (Note that it still shouldn’t modify self!)