Inheritance

Announcements

Attribute Lookup Practice

Discussion Question: Class Attribute Assignment

Implement the **Place** class, which takes a **name.** Its **print_history**() method prints the **name** of the **Place** and then the names of all the **Place** instances that were created before it.

```
class Place:
    last = None
    def __init__(self, n):
        self.name = n
        self.then = ___________
        ___________ = self

    def print_history(self):
        print(self.name)
        if _________ is not None:
            ________________________
```

```
>>> places = [Place(x*2) for x in range(10)]
>>> places[2].print_history()
4
2
0
>>> places[6].print_history()
12
10
8
6
4
2
0
```

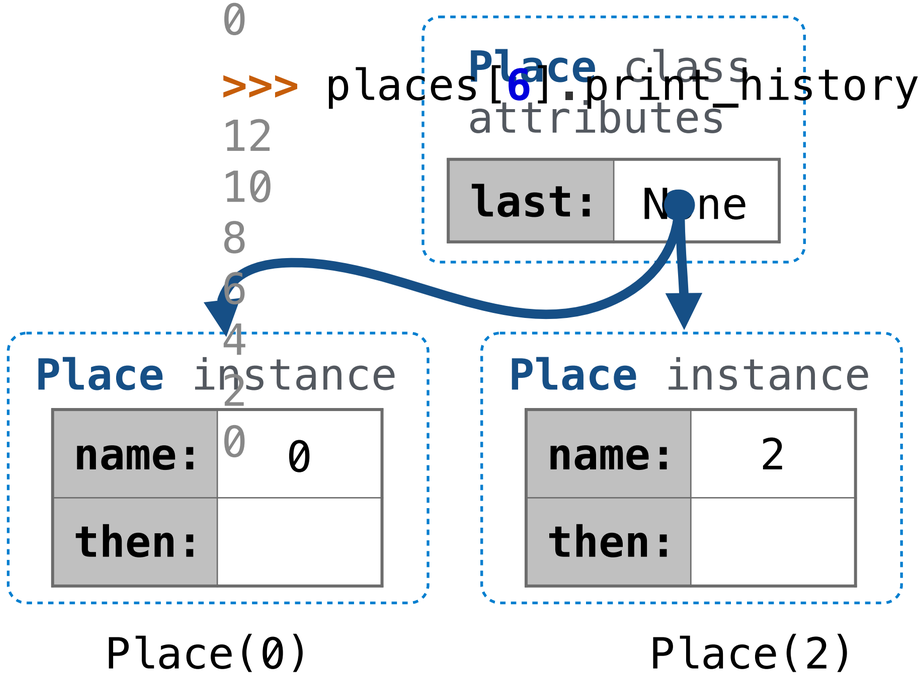
Discussion Question: Class Attribute Assignment

Implement the **Place** class, which takes a **name.** Its **print_history**() method prints the **name** of the **Place** and then the names of all the **Place** instances that were created before it.

```
class Place:
    last = None
    def __init__(self, n):
        self.name = n
        self.then = Place.last
        Place.last = self

    def print_history(self):
        print(self.name)
        if _________ is not None:
            ____________________________
```

```
>>> places = [Place(x*2) for x in range(10)]
>>> places[2].print_history()
4
2
0
>>> places[6].print_history()
12
10
8
6
4
2
0
```



Discussion Question: Class Attribute Assignment

Implement the **Place** class, which takes a **name.** Its **print_history**() method prints the **name** of the **Place** and then the names of all the **Place** instances that were created before it.

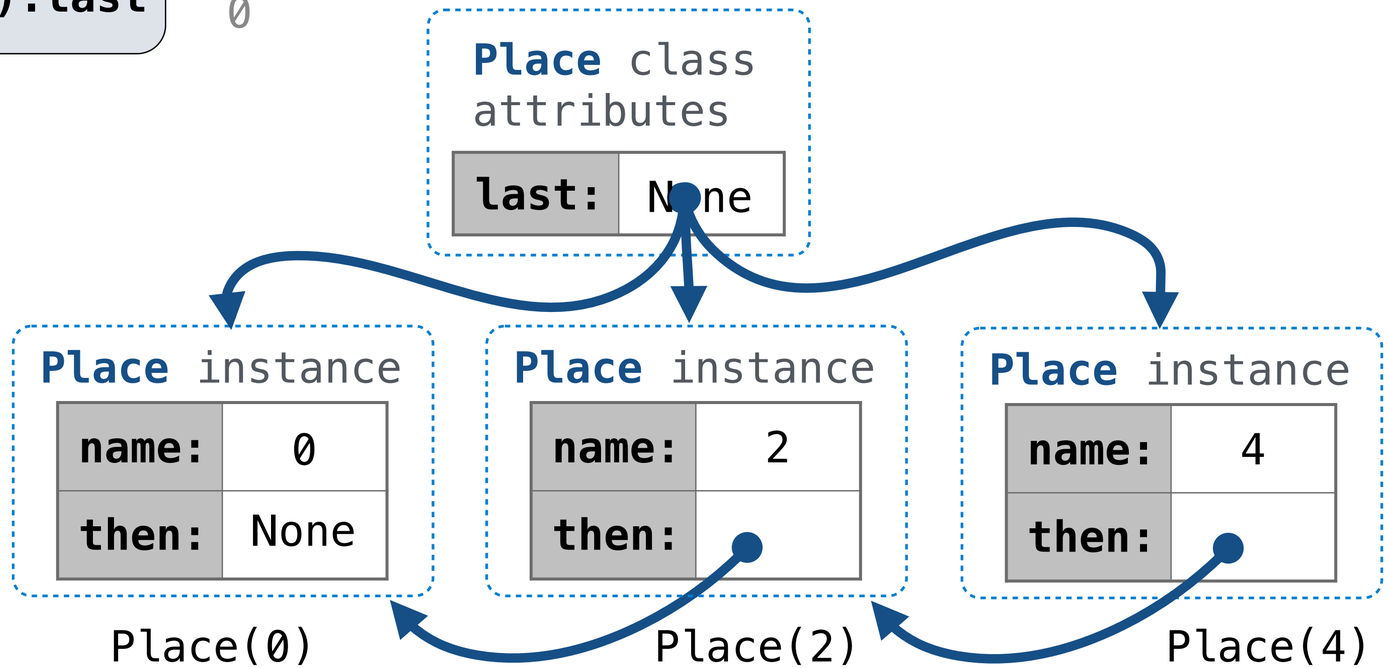
```
class Place:
    last = None
    def __init__(self, n):
        self.name = n
        self.then = Place.last
        Place.last = self

    def print_history(self):
        print(self.name)
        if _________ is not None:
            _________________________
```

OK to write **self.last** or **type(self).last**

Not ok to write **self.last**

```
>>> places = [Place(x*2) for x in range(10)]
>>> places[2].print_history()
4
2
0
```



Discussion Question: Class Attribute Assignment

Implement the **Place** class, which takes a **name.** Its **print_history**() method prints the **name** of the **Place** and then the names of all the **Place** instances that were created before it.

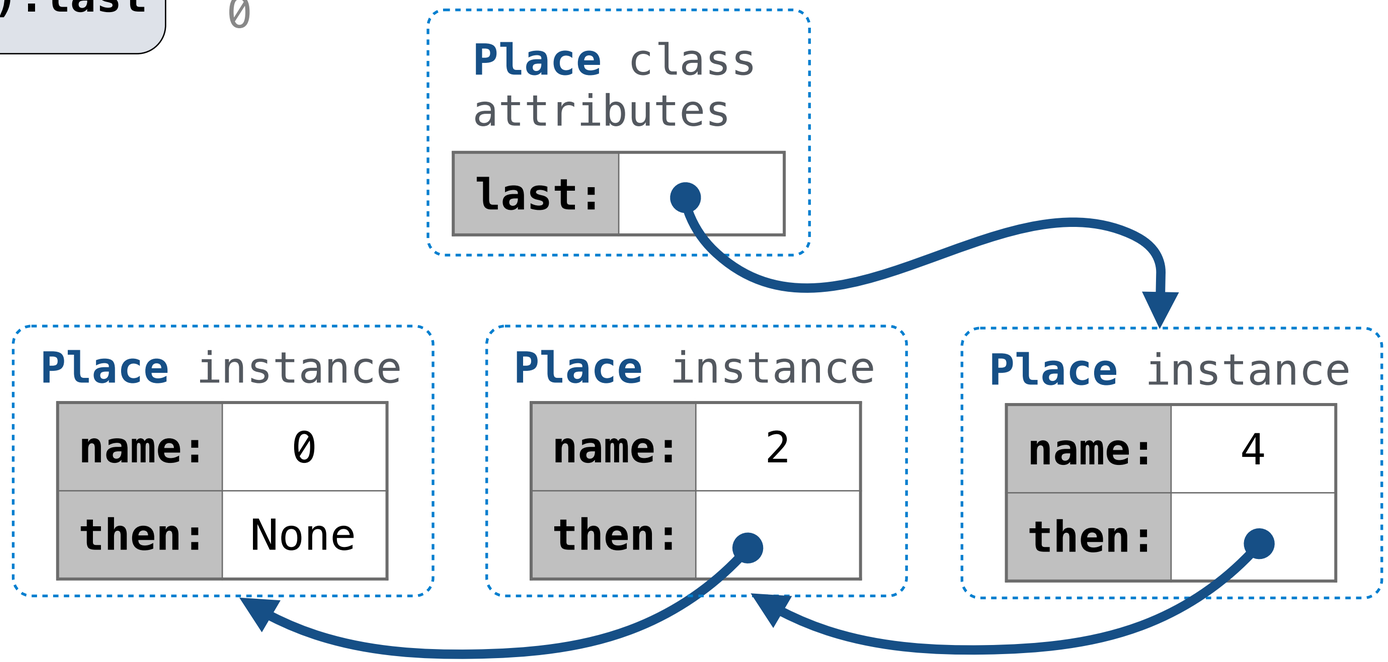
```
class Place:
    last = None
    def __init__(self, n):
        self.name = n
        self.then = Place.last
        Place.last = self

    def print_history(self):
        print(self.name)
        if self.then is not None:
            self.then.print_history()
```

OK to write **self.last** or **type(self).last**

Not ok to write **self.last**

```
>>> places = [Place(x*2) for x in range(10)]
>>> places[2].print_history()
4
2
0
```



Composition

Attribute Composition

Attribute values are themselves objects, which have attributes and methods

Any type of object can be an attribute value of any other type

When reading/writing a complicated expression, think about the type of each sub-expression:

- **self** always has the type of the class being defined
- For **self.x.y: self.x** is an object (an attribute of **self**) with an attribute **y.**
- For **self.f(x).y: self.f(x)** is the return value of method **f,** an object with an attribute **y.**
- For **self.f(x).g(y):** the return value of **self**'s method **f** is an object that has a method **g.**

Example: Message Delivery

```
@dataclass
class Email:
    msg: str
    sender: str
    recipient_name: str

class Server:
    def __init__(self):
        self.clients: dict[str, Client] = {}

    def send(self, email: Email) -> None:
        "Append the email to the inbox of the recipient's client."
        ____________________________________________________________

class Client:
    def __init__(self, server: Server, name: str):
        self.server = server
        self.name = name
        self.inbox: list[Email] = []
        server.____________________

    def compose(self, message: str, recipient: str) -> None:
        "Send an email with the given message to the recipient."
        self.server.send(_____________________________________)
```

A **Client** can **compose** an **Email** to **send** it to the **Server**, which appends it to the **inbox** of the **Client** named as the **recipient.**

Each **Server** has a **clients** dictionary from the **name** of each **Client** to that **Client**, and each **Client** adds itself to **clients.**

```
>>> s = Server()
>>> a = Client(s, 'Alice')
>>> b = Client(s, 'Bob')
>>> a.compose('Hello!', 'Bob')
>>> len(b.inbox)
1
>>> b.inbox[0].msg
'Hello!'
```

Example: Message Delivery

```
@dataclass
class Email:
    msg: str
    sender: str
    recipient_name: str

class Server:
    def __init__(self):
        self.clients: dict[str, Client] = {}
    def send(self, email: Email) -> None:
        "Append the email to the inbox of the recipient's client."
        self.clients[email.recipient_name].inbox.append(email)

class Client:
    def __init__(self, server: Server, name: str):
        self.server = server
        self.name = name
        self.inbox: list[Email] = []
        server.clients[name] = self

    def compose(self, message: str, recipient: str) -> None:
        "Send an email with the given message to the recipient."
        self.server.send(Email(message, self.name, recipient))
```

Server dict Client list[Email] Email

A **Client** can **compose** an **Email** to **send** it to the **Server**, which appends it to the **inbox** of the **Client** named as the **recipient.**

Each **Server** has a **clients** dictionary from the **name** of each **Client** to that **Client**, and each **Client** adds itself to **clients.**

```
>>> s = Server()
>>> a = Client(s, 'Alice')
>>> b = Client(s, 'Bob')
>>> a.compose('Hello!', 'Bob')
>>> b.inbox[0].msg
'Hello!'
```

Inheritance

Inheritance Example

A CheckingAccount is a specialized type of Account (Demo)

```
>>> ch = CheckingAccount('Tom')
>>> ch.interest     # Lower interest rate for checking accounts
0.01
>>> ch.deposit(20)  # Deposits are the same
20
>>> ch.withdraw(5)  # Withdrawals incur a $1 fee
14
```

Most behavior is shared with the base class Account

```
class CheckingAccount(Account):
    """A bank account that charges for withdrawals."""
    withdraw_fee = 1
    interest = 0.01
    def withdraw(self, amount):
        return Account.withdraw(self, amount + self.withdraw_fee)
```

or

```
        return super().withdraw(      amount + self.withdraw_fee)
```

Looking Up Attribute Names on Classes

Base class attributes *aren't* copied into subclasses!

To look up a name in a class:

1. If it names an attribute in the class, return the attribute value.

2. Otherwise, look up the name in the base class, if there is one.

```
>>> ch = CheckingAccount('Tom')  # Calls Account.__init__
>>> ch.interest     # Found in CheckingAccount
0.01
>>> ch.deposit(20)  # Found in Account
20
>>> ch.withdraw(5)  # Found in CheckingAccount
14
```