Assignment to Attributes

Assignment statements with a dot expression on their left-hand side affect attributes for the object of that dot expression.

- If the object is a class, then assignment sets a class attribute.
- If the object is an instance, then assignment sets an instance attribute.

Looking Up Attributes by Name

To evaluate a dot expression:

1. Evaluate the expression to the left of the dot, which yields the object of the dot expression.
2. The name is matched against the instance attributes of that object; if an attribute with that name exists, its value is returned.
3. If not, the name is looked up in the class, which yields a class attribute value.
4. That value is returned unless it is a function, in which case a bound method is evaluated instead.

Class Attributes

Class attributes are “shared” across all instances of a class because they are attributes of the class, not the instance.

```
class Account:
    interest = 0.02 # A class attribute

def __init__(self, account_holder):
    self.account_holder = account_holder

    # Additional methods would be defined here

>>> tom_account = Account('Tom')
>>> tom_account.deposit(100)
>>> tom_account.withdraw(50)
```

```
The interest attribute is part of the instance; it’s part of the class!
```

Attribute Assignment

Inheritance
Inheritance is a technique for relating classes together. A common use: Two similar classes differ in their degree of specialization.

The specialized class may have the same attributes as the general class, along with some special-case behavior.

Conceptually, the new subclass inherits attributes of its base class. The subclass may override certain inherited attributes. Using inheritance, we implement a subclass by specifying its differences from the base class.

Inheritance Example

A `CheckingAccount` is a specialized type of `Account`

```python
class CheckingAccount(Account):
    # A bank account that charges for withdrawals.
    withdraw_fee = 1

    def __init__(self, y):
        super().__init__(y)

    def withdraw(x, amount):
        return Account.withdraw(self, x)  # Calls Account.__init__

    # Deposit is the same.
    def deposit(x, amount):
        # Checks manually for special accounts.
        return Account.deposit(x, amount)
```

```
CheckingAccount('Tom').withdraw(20)  # Calls Account.__init__
```

Inheritance and Composition

Object-oriented programming shines when we adopt the metaphor

Inheritance is best for representing is-a relationships

- E.g., checking accounts have a specific type of account
- So, CheckingAccount inherits from Account

Composition is best for representing has-a relationships

- E.g., a bank has a collection of bank accounts it manages
- So, A bank has a list of accounts as an attribute

Looking Up Attribute Names on Classes

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.

```python
ch = CheckingAccount('Tom')  # Calls Account.__init__
```

Attributes Lookup Practice

- Attribute look-up on base class
- Preferred to `CheckingAccount.withdraw_fee`

```
>>> ch = CheckingAccount('Tom')  # Calls Account._init_
>>> ch.interest  # Found in CheckingAccount
0.01
>>> ch.deposit(20)  # Found in Account
20
```

Inheritance and Attribute Lookup

```python
class A:
    def f(self, x):
        return x

class B(A):
    def g(self, y):
        return y

# This evaluates to an integer.
a = B()
b = B()
a.a = 4
b.b = 5
```

Attributes Lookup Practice

- `a.a` is a preferred to `B.a`
- `b.b` is a preferred to `A.b`

```
class A:
    def g(self, y):
        return y

a = A()
b = A()
a.g()  # admixture of these
```

Multiple Inheritance

```python
class A:
    def f(self, x):
        return f

class B:
    def g(self, y):
        return g

a = A()
b = B()
```
Multiple Inheritance

A class may inherit from multiple base classes in Python.

```python
class AsSeenOnTVAccount(CheckingAccount, SavingsAccount):
    def __init__(self, account_holder):
        self.holder = account_holder
        self.balance = 1  # A free dollar!
```

```python
>>> such_a_deal = AsSeenOnTVAccount('John')
```

```python
>>> such_a_deal.balance
1
```

```python
>>> such_a_deal.deposit(20)
19
```

```python
>>> such_a_deal.withdraw(5)
13
```

Resolving Ambiguous Class Attribute Names

```python
>>> such_a_deal = AsSeenOnTVAccount('John')
```

```python
>>> such_a_deal.balance
1
```

```python
>>> such_a_deal.deposit(20)
19
```

```python
>>> such_a_deal.withdraw(5)
13
```

Complicated Inheritance

```python
def _init_(self, account_holder):
    self.holder = account_holder
    self.balance = 0  # A free dollar!
```

Biological Inheritance

Moral of the story: Inheritance can be complicated, so don’t overuse it!