Methods and Functions

Python distinguishes between:
- Functions, which we have been creating since the beginning of the course, and
- Bound methods, which couple together a function and the object on which that method will be invoked.

\[ \text{Object} \rightarrow \text{Function} \rightarrow \text{Bound Method} \]

```python
type(Account.deposit)
<class 'function'>
type(tom_account.deposit)
<class 'method'>
```

- **Accounts**: One object before the dot and other arguments within parentheses.

```
Account.deposit(item_account, 1001)
Function: all arguments within parentheses
```

- **Methods**: Functions are objects
- **Bound Methods**: Also objects: a function that has its first parameter "self" already bound to an instance
- **Dot Expressions**: Evaluate to bound methods for class attributes that are functions

```
<instance>.<method_name>
```

Terminology: Attributes, Functions, and Methods

All objects have attributes, which are name-value pairs.

- Classes are objects too, so they have attributes.
- Instance attributes: attributes of an instance
- Class attributes: attributes of the class of an instance

### Terminology:

- **Class attributes**: "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.balance = 0
        self.holder = account_holder

# Additional methods would be defined here
```

### Attribute Assignment

The interest attribute is "part of the instance," it's part of the class!

```
>>> tom_account = Account('Tom')
>>> jim_account = Account('Jim')
```

#### 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. `<name>` is matched against the instance attributes of that object; if an attribute with that name exists, its value is returned.
3. If not, `<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 returned instead.

```
<expression> . <name>
```
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 an instance, then assignment sets an instance attribute.
- If the object is a class, then assignment sets a class attribute.

```python
tom_account.interest = 0.08
```

But the name (“interest”) is not looked up.

Attribute assignment statement adds or modifies the attribute named “interest” of `tom_account`.

**Instance Attribute Assignment**:
```
Account.interest = 0.04
```

**Class Attribute Assignment**:
```
This expression evaluates to an object.
```

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.

```python

class <Name>(<Base Class>):
    """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)
```

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.

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.

```python
class <Name>(<Base Class>):
    """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)
```
Object-Oriented Design

Designing for Inheritance

Don't repeat yourself; use existing implementations

Attributes that have been overridden are still accessible via class objects
Look up attributes on instances whenever possible

```python
class CheckingAccount(Account):
    # A bank account that charges for withdrawals.
    interest = 0.01
    withdraw_fee = 1

    def withdraw(self, amount):
        return Account.withdraw(self, amount + self.withdraw_fee)
```

Inheritance and Composition

Object-oriented programming shines when we adopt the metaphor

Inheritance is best for representing is-a relationships
- E.g., a checking account is 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

Attributes Lookup Practice

```python
class A:
    a = -1
def f(self, x):
    return x-1

class B(A):
    b = 4
def __init__(self, y):
    if y:
        self.a = self.f(x)
    else:
        self.a = self.f(y)

class C(B):
    def f(self, x):
        return x

a = A()
b = B(1)
b.n = 5
```

>>> a.z == C.z
True

Which evaluates to an integer?

- `b.z`
- `b.z.z`
- `b.z.z.z.z`
- None of these

```python
>>> a.z == b.z
<__main__.A object at 0x7f8000000000>
```

Inheritance and Attribute Lookup

Multiple Inheritance

```python
class A:
    a = 1
def f(self, x):
    return x

class B(A):
    b = 2
def __init__(self, y):
    self.a = self.f(x)

class C(B):
    def f(self, x):
        return x

a = A()
b = B(1)
```

```console
>>> a.a
1
>>> b.a
2
```
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')
>>> such_a_deal.balance
1
>>> such_a_deal.deposit(20)
19
>>> such_a_deal.withdraw(5)
13
```

**Resolving Ambiguous Class Attribute Names**

- `Account`
- `CheckingAccount`
- `SavingsAccount`
- `AsSeenOnTVAccount`

```python
>>> such_a_deal = AsSeenOnTVAccount('John')
>>> such_a_deal.balance
1
>>> such_a_deal.deposit(20)
19
>>> such_a_deal.withdraw(5)
13
```

**Complicated Inheritance**

- **Grandma**
- **Grandpa**
- **Gramammy**
- **Grandaddy**
- **Aunt**
- **Double**
- **Quadruple**
- **Mom**
- **Dad**
- **You**
- **Half**
- **some_guy**
- **Double Half Uncle**
- **some_other_guy**
- **Double Cousin**

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