Terminology: Attributes, Functions, and Methods

All objects have attributes, which are name-value pairs
A class is a type (or category) of objects
Classes are objects too, so they have attributes
Instance attribute: attribute of an instance
Class attribute: attribute of the class of an instance

Terminology:
- Attributes, Functions, and Methods
- Bound methods are 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

Python object system:
- Functions are objects
- Bound methods are 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>
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

Class Attributes

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

```python
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
```

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
Attribute Assignment Statements

```
>>> jim_account = Account('Jim')
>>> tom_account = Account('Tom')
>>> tom_account.interest = 0.02
>>> jim_account.interest = 0.02
>>> Account.interest = 0.04
>>> tom_account.interest = 0.04
>>> jim_account.interest = 0.04
>>> jim_account.interest = 0.08
>>> tom_account.interest = 0.04
>>> Account.interest = 0.05
>>> tom_account.interest = 0.05
>>> jim_account.interest = 0.08
```

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.

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

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`.

```
>>> 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.

```python
>>> 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
```

Object-Oriented Design

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

(Demo)
Inheritance and Attribute Lookup

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

class B(A):
    n = 4
def __init__(self, y):
    if y:
        self.z = self.f(y)
    else:
        self.z = C(y+1)

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

a = A()
b = B(1)

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

>>> C(2).
n
4

Which evaluates to an integer?

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

Environment diagrams for objects aren't required, but can be very helpful!

Multiple Inheritance

```python
class SavingsAccount(Account):
    deposit_fee = 2
def deposit(self, amount):
        return Account.deposit(self, amount - self.deposit_fee)

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

A class may inherit from multiple base classes in Python.

**CleverBank** marketing executive has an idea:
- Low interest rate of 1%
- A $1 fee for withdrawals
- A $2 fee for deposits
- A free dollar when you open your account

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

A class may inherit from multiple base classes in Python.

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

Instance attribute
SavingsAccount method
CheckingAccount method

```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

Instance attribute
SavingsAccount method
CheckingAccount method

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