# Inheritance

# Announcements

# Attributes

# Terminology: Attributes, Functions, and Methods

All objects have attributes, which are name-value pairs

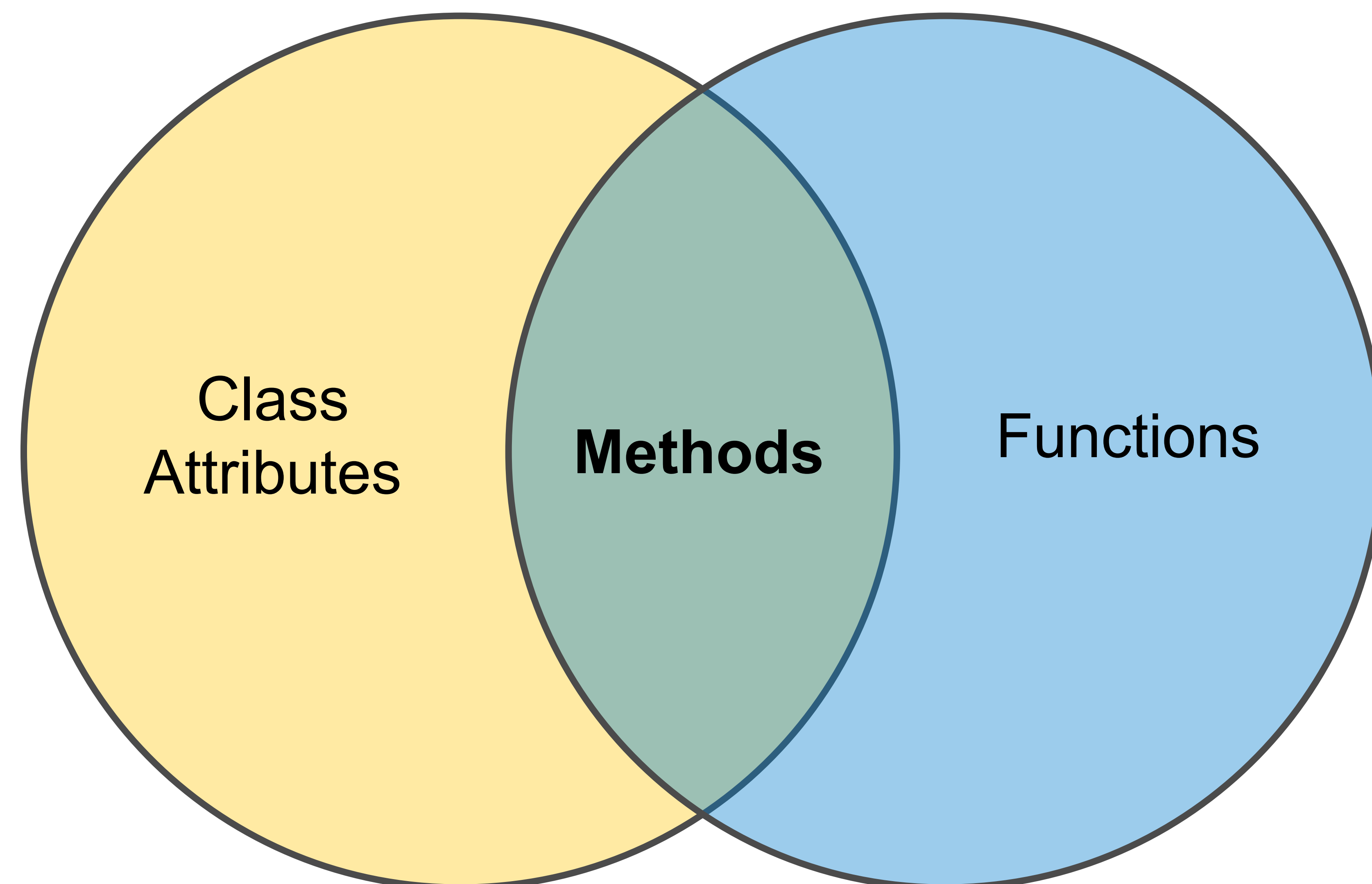
A class is a type (or category) of objects

Classes are objects too, so they have attributes

Instance attribute: attribute of an instance

Class attribute: attribute of the class of an instance

**Terminology:**




**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

<expression> . <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

```
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

>>> tim_account = Account('Tim')
>>> noor_account = Account('Noor')
>>> tim_account.interest
0.02
>>> noor_account.interest
0.02
```

The **interest** attribute is ***not*** part of the instance; it's part of the class!

# Attribute Assignment

# 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

```
class Account:
    interest = 0.02
    def __init__(self, holder):
        self.holder = holder
        self.balance = 0
    ...

tom_account = Account('Tom')
```

Instance Attribute Assignment :

tim_account.interest = 0.08

This expression evaluates to an object

But the name ("interest") is not looked up

Attribute assignment statement adds or modifies the attribute named "interest" of tom_account

Class Attribute Assignment

Account.interest = 0.04

# Attribute Assignment Statements

Account class attributes

interest: ~~0.02~~ ~~0.04~~ 0.05
(withdraw, deposit, __init__)

Instance attributes of noor_account

balance: 0
holder: 'Jim'
interest: 0.08

Instance attributes of tom_account

balance: 0
holder: 'Tim'

```
>>> noor_account = Account('Noor')
>>> tim_account = Account('Tim')
>>> tim_account.interest
0.02
>>> noor_account.interest
0.02
>>> Account.interest = 0.04
>>> tim_account.interest
0.04
>>> noor_account.interest
0.04
```

```
>>> noor_account.interest = 0.08
>>> noor_account.interest
0.08
>>> tim_account.interest
0.04
>>> Account.interest = 0.05
>>> tim_account.interest
0.05
>>> noor_account.interest
0.08
```

# Inheritance

# 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 the base class

# Inheritance Example

A CheckingAccount is a specialized type of Account

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

(Demo)

# Class Relationships

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

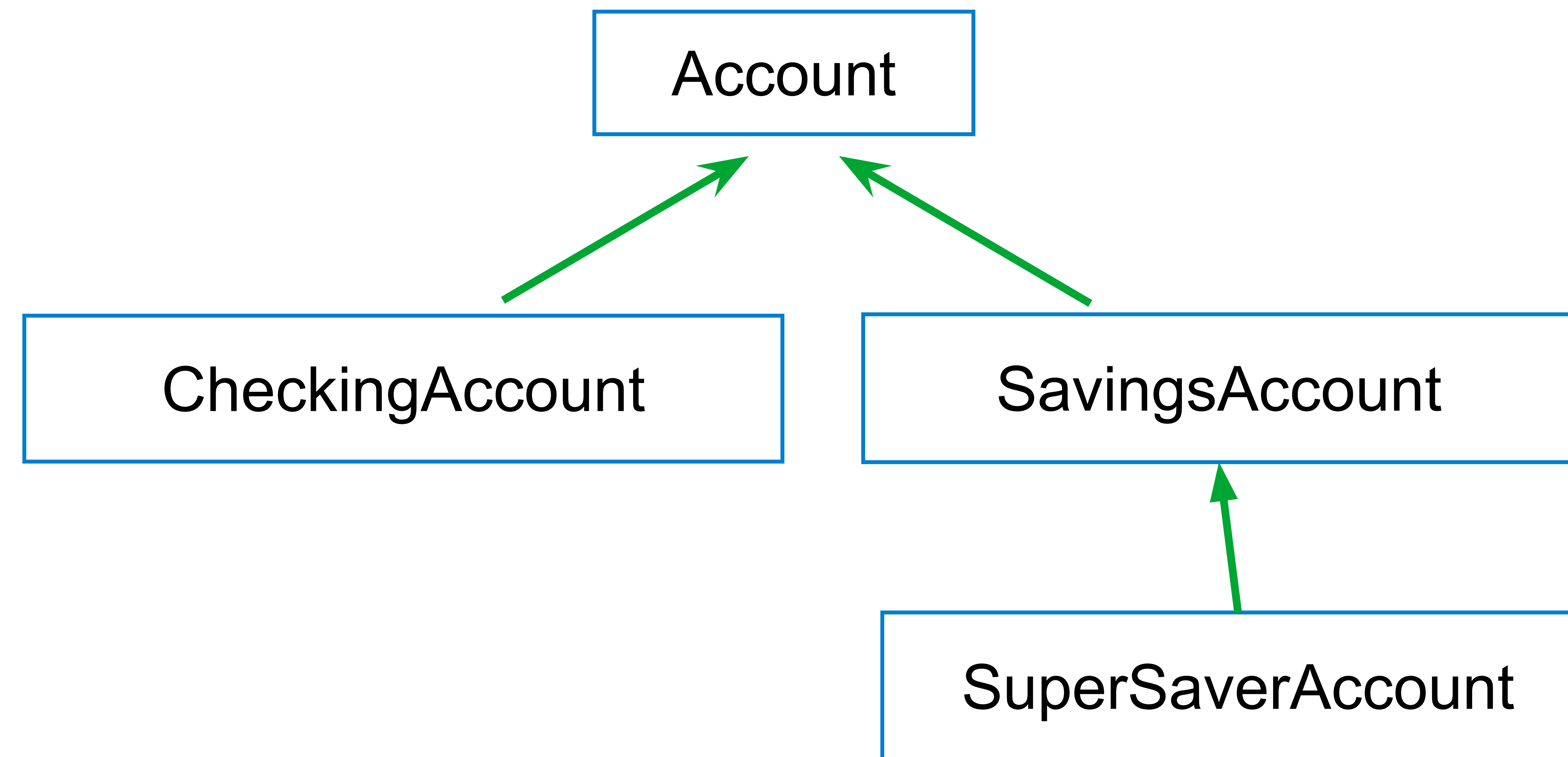
A class may inherit from an existing subclass.

SuperSaverAccount is a kind of SavingsAccount:

- Cannot withdraw until retired

```
class SuperSaverAccount(SavingsAccount):
    def withdraw(self, amount):
        if self.retired():
            return super().withdraw(amount)
    return self.balance

    # Additional methods like retired would be defined here
```

# Class Hierarchy




Account attribute

SavingsAccount method

SuperSaverAccount method

```
>>> super_saver = SuperSaverAccount('Tim')
>>> super_saver.interest
0.02
>>> super_saver.deposit(20)
18
>>> super_saver.withdraw(5) # Not retired
18
```

# 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

```
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)
```

Attribute look-up on base class

Preferred to CheckingAccount.withdraw_fee to allow for specialized accounts

# 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 Examples

(Demo)

# Review: Attributes Lookup, Methods, & Inheritance

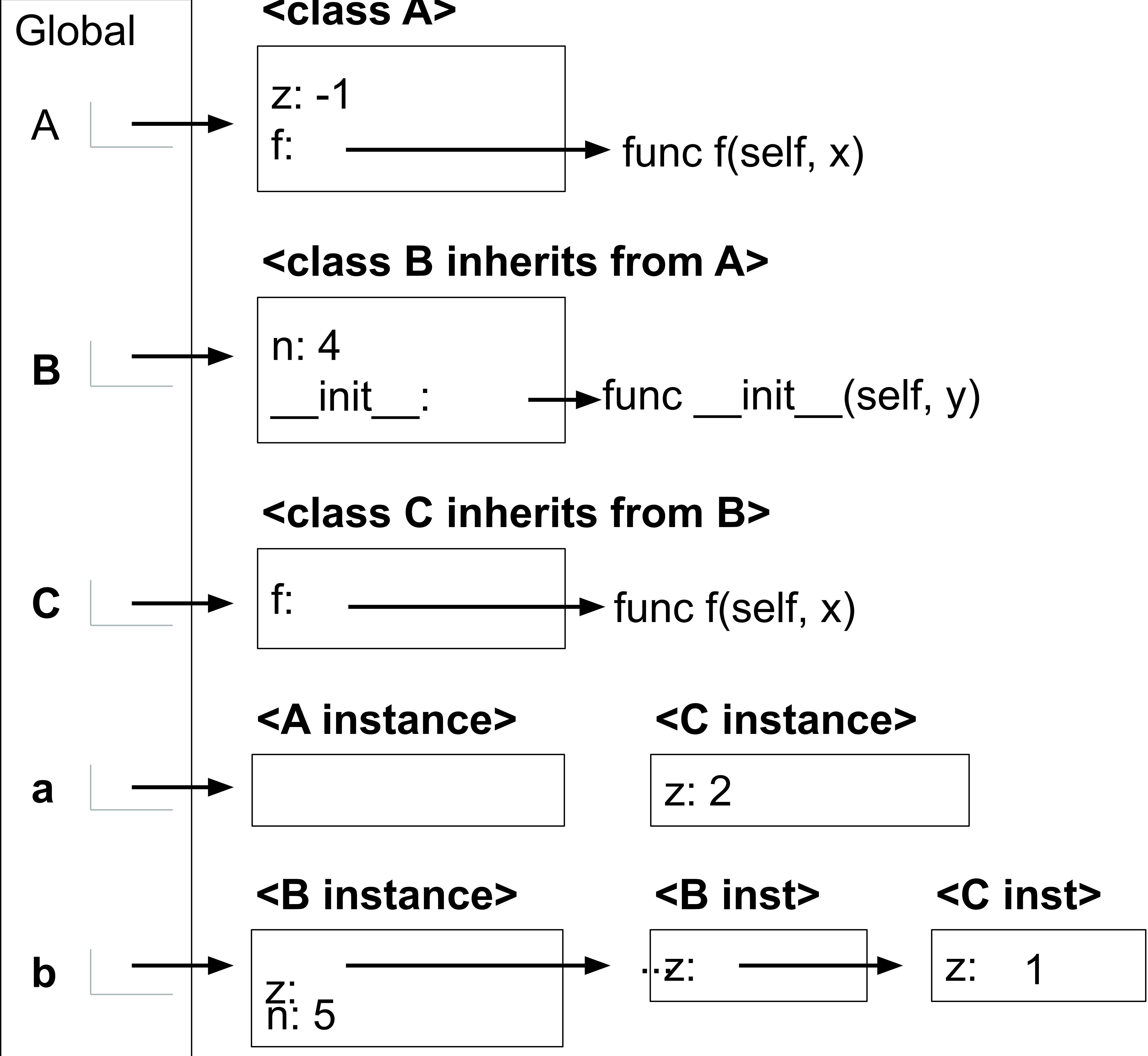
# Inheritance and Attribute Lookup

```
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)
b.n = 5
```

```
>>> C(2).n
4
>>> a.z == C.z
True
>>> a.z == b.z
False
```

**Which evaluates to an integer?**

- b.z
- b.z.z
- ▶ b.z.z.z
- b.z.z.z.z
- None of these




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

# Multiple Inheritance

# Multiple Inheritance

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

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

```
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!
```

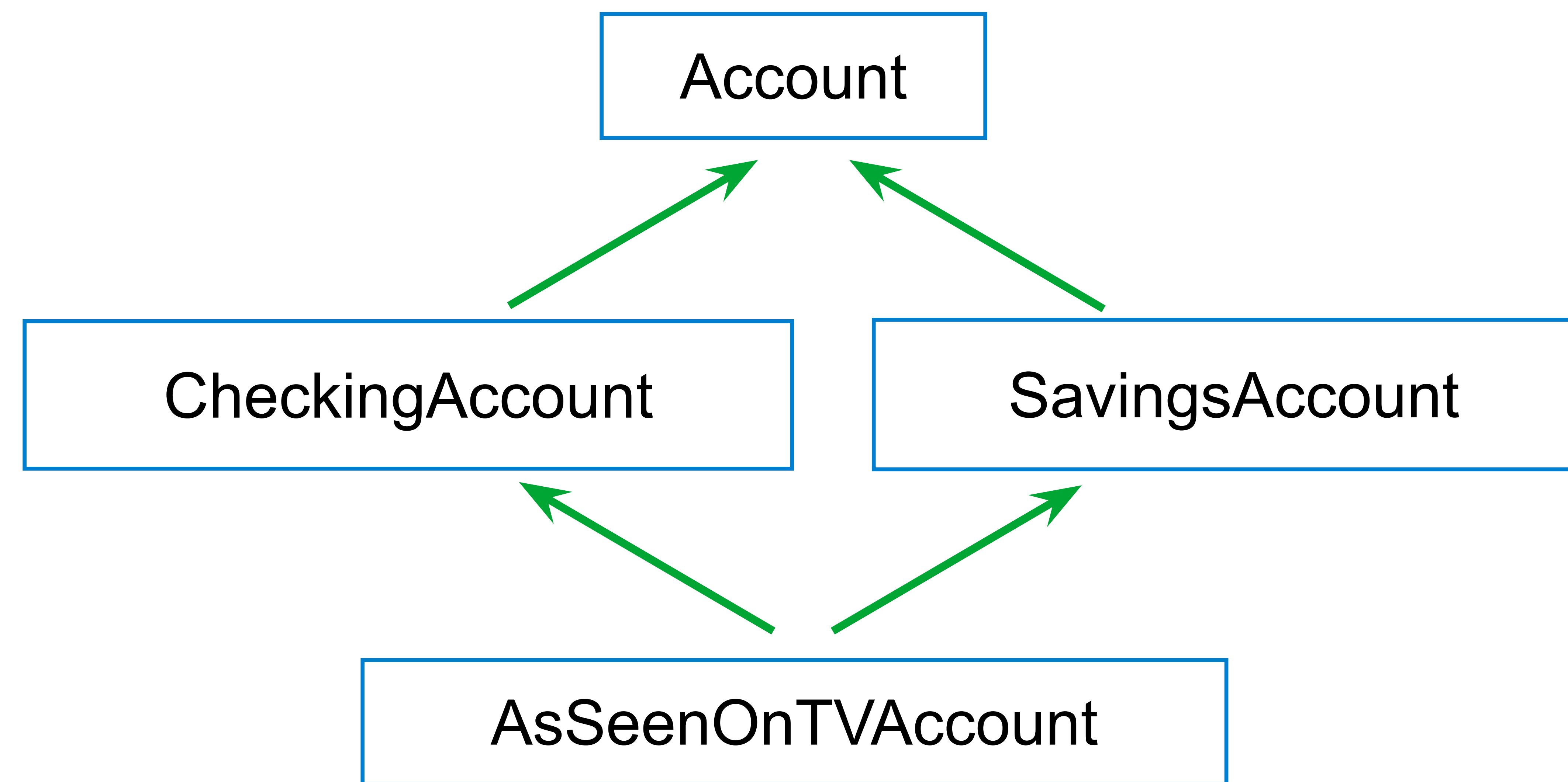
```
>>> such_a_deal = AsSeenOnTVAccount('John')
>>> such_a_deal.balance
1
>>> such_a_deal.deposit(20)
19
>>> such_a_deal.withdraw(5)
13
```

Instance attribute

SavingsAccount method

CheckingAccount method

# Resolving Ambiguous Class Attribute Names




Instance attribute

SavingsAccount method

CheckingAccount method

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

# Representation

# String Representations

# String Representations

An object value should behave like the kind of data it is meant to represent

For instance, by producing a string representation of itself

Strings are important: they represent language and programs

In Python, all objects produce two string representations:

- The **str** is legible to humans
- The **repr** is legible to the Python interpreter

The **str** and **repr** strings are often the same, but not always

# The repr String for an Object

The **repr** function returns a Python expression (a string) that evaluates to an equal object

```
repr(object) -> string

Return the canonical string representation of the object.
For most object types, eval(repr(object)) == object.
```

The result of calling **repr** on a value is what Python prints in an interactive session

```
>>> 12e12
12000000000000.0
>>> print(repr(12e12))
12000000000000.0
```

Some objects do not have a simple Python-readable string

```
>>> repr(min)
'<built-in function min>'
```

# The str String for an Object

Human interpretable strings are useful as well:

```
>>> from fractions import Fraction
>>> half = Fraction(1, 2)
>>> repr(half)
'Fraction(1, 2)'
>>> str(half)
'1/2'
```

The result of calling **str** on the value of an expression is what Python prints using the **print** function:

```
>>> print(half)
1/2
```

(Demo)

# F-Strings

# String Interpolation in Python

String interpolation involves evaluating a string literal that contains expressions.

Using string concatenation:

```
>>> from math import pi
>>> 'pi starts with ' + str(pi) + '...'
'pi starts with 3.141592653589793...'

>>> print('pi starts with ' + str(pi) + '...')
pi starts with 3.141592653589793...
```

Using string interpolation:

```
>>> f'pi starts with {pi}...'
'pi starts with 3.141592653589793...'

>>> print(f'pi starts with {pi}...')
pi starts with 3.141592653589793...
```

The result of evaluating an f-string literal contains the str string of the value of each sub-expression.

Sub-expressions are evaluated in the current environment.

(Demo)

# Using __str__ and __repr__