Fundamentals of Programming I

Commonly Used Methods

More Modeling

Making Meatballs Is Serious Business

Mind if I Lend a Hand?

The Interface of the **SavingsAccount** Class

```
SavingsAccount(name, pin, bal)  # Returns a new object

getBalance()              # Returns the current balance

deposit(amount)           # Makes a deposit

withdraw(amount)          # Makes a withdrawal

computeInterest()         # Computes the interest and
                          # deposits it
```

Defining the **SavingsAccount** Class

```
class SavingsAccount(object):
    """This class represents a savings account."""

    def __init__(self, name, pin, balance = 0.0):
        self.name = name
        self.pin = pin
        self.balance = balance

    # Other methods go here
```

Note that **name** is a method’s parameter, whereas **self.name** is an object’s instance variable

The Lifetime of a Variable

```
class SavingsAccount(object):
    """This class represents a savings account."""

    def __init__(self, name, pin, balance = 0.0):
        self.name = name
        self.pin = pin
        self.balance = balance

    # Other methods go here
```

Parameters exist only during the lifetime of a method call, whereas instance variables exist for the lifetime of an object

Parameters or Instance Variables?

- Use a parameter to send information through a method to an object

- Use an instance variable to retain information in an object

- An object’s *state* is defined by the current values of all of its instance variables

- References to instance variables must include the qualifier **`self`**

The Scope of a Variable

- The *scope* of a variable is the area of program text within which its value is visible
- The scope of a parameter is the text of its enclosing function or method
- The scope of an instance variable is the text of the enclosing class definition (perhaps many methods)

The Scope of a Variable

```python
class SavingsAccount(object):
    """This class represents a savings account."""

    def __init__(self, name, pin, balance = 0.0):
        self.name = name
        self.pin = pin
        self.balance = balance

    def deposit(self, amount):
        self.balance += amount

    def withdraw(self, amount):
        self.balance -= amount
```

`self.balance` always refers to the same storage area (for one object)

`amount` refers to a different storage area for each method call

The Interface of the **SavingsAccount** Class

```
SavingsAccount(name, pin, bal)  # Returns a new object

getBalance()              # Returns the current balance

deposit(amount)           # Makes a deposit

withdraw(amount)          # Makes a withdrawal

computeInterest()         # Computes the interest and
                          # deposits it
```

The Interface of the **SavingsAccount** Class

```
SavingsAccount(name, pin, bal)  # Returns a new object

getBalance()              # Returns the current balance

deposit(amount)           # Makes a deposit

withdraw(amount)          # Makes a withdrawal

computeInterest()         # Computes the interest and
                          # deposits it

str(account)              # String representation of account

a1 == a2                  # Test for equality
```

Accessing Data in an Object

```
>>> account = SavingsAccount('Ken', '3322', 1000.00)

>>> print('Name:    ' + account.getName() + '\n' + \
          'PIN:     ' + account.getPin() + '\n' + \
          'Balance: ' + str(account.getBalance()))
Name:    Ken
PIN:     3322
Balance: 1000.00
```

An object’s data can be viewed or accessed by using its *accessor methods*

String Representation

```
>>> account = SavingsAccount('Ken', '3322', 1000.00)

>>> print(str(account))  # Same as account.__str__()
Name:    Ken
PIN:     3322
Balance: 1000.00
```

Each class can include a string conversion method named **`__str__`**

This method is automatically called when the **`str`** function is called with the object as a parameter

String Representation

```
>>> account = SavingsAccount('Ken', '3322', 1000.00)

>>> str(account)
'Name:    Ken\nPIN:     3322\nBalance: 1000.00'

>>> print(account)               # Better still
Name:    Ken
PIN:     3322
Balance: 1000.00
```

Each class can include a string conversion method named **`__str__`**

`print` runs **`str`** if it’s given an object to print - way cool!

The __str__ Method

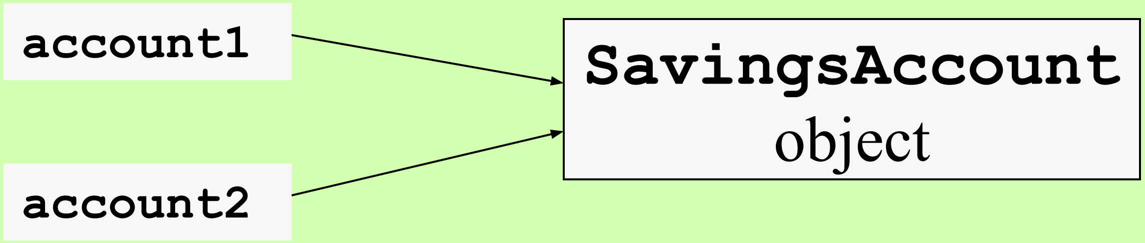
```
class SavingsAccount(object):
    """This class represents a savings account."""

    def __init__(self, name, pin, balance = 0.0):
        self.name = name
        self.pin = pin
        self.balance = balance

    def __str__(self):
        return 'Name:    ' + self.name + '\n' + \
               'PIN:     ' + self.pin + '\n' + \
               'Balance: ' + str(self.balance)
```

As a rule of thumb, you should include an **`__str__`** method in each new class that you define

Equality with ==

```
>>> account1 = SavingsAccount("ken", "1000", 4000.00)
>>> account2 = account1
>>> account1 == account2
True
```

The two variables refer to the same, identical object



account1

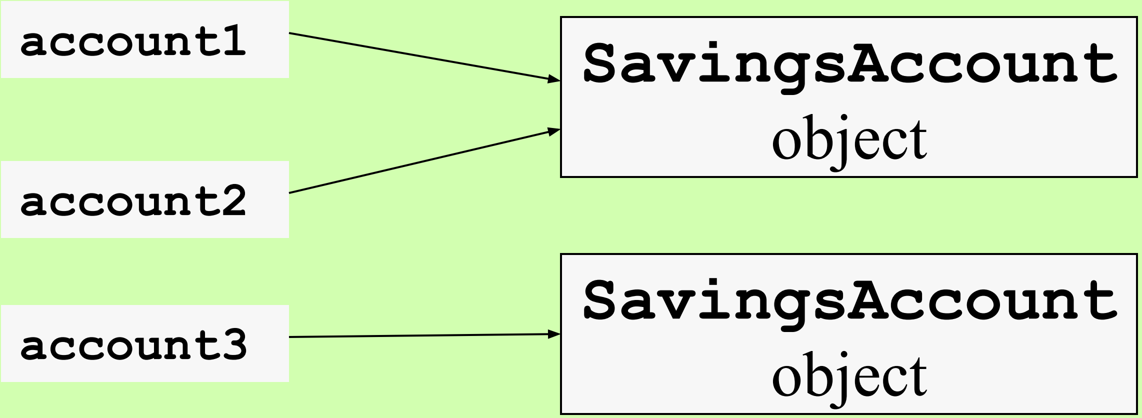
account2

SavingsAccount
object

Equality with ==

```
>>> account1 = SavingsAccount("ken", "1000", 4000.00)
>>> account2 = account1
>>> account1 == account2
True
>>> account3 = SavingsAccount("ken", "1000", 4000.00)
>>> account1 == account3
False
```

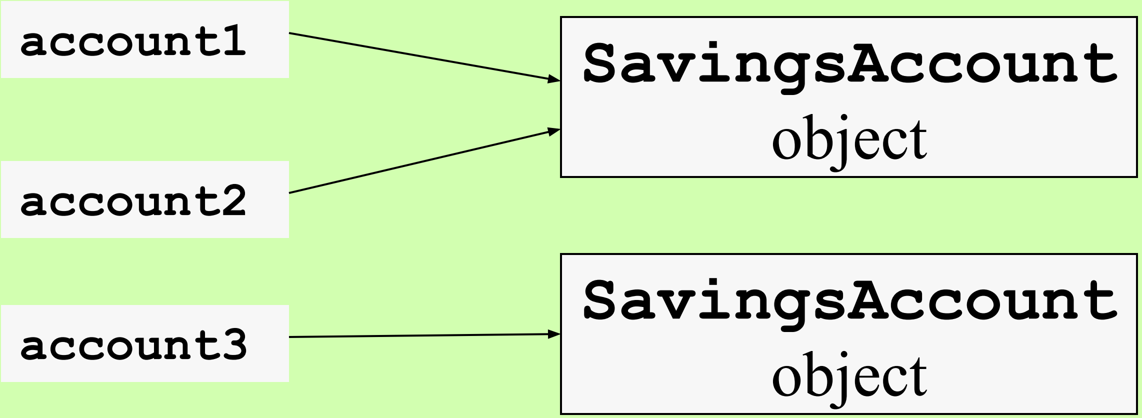
The two account objects have the same contents but aren’t equal



Equality with **`is`**

```
>>> account1 = SavingsAccount("ken", "1000", 4000.00)
>>> account2 = account1
>>> account1 is account2
True
>>> account3 = SavingsAccount("ken", "1000", 4000.00)
>>> account1 is account3
False
```

By default, **`==`** uses **`is`**, which tests for object identity



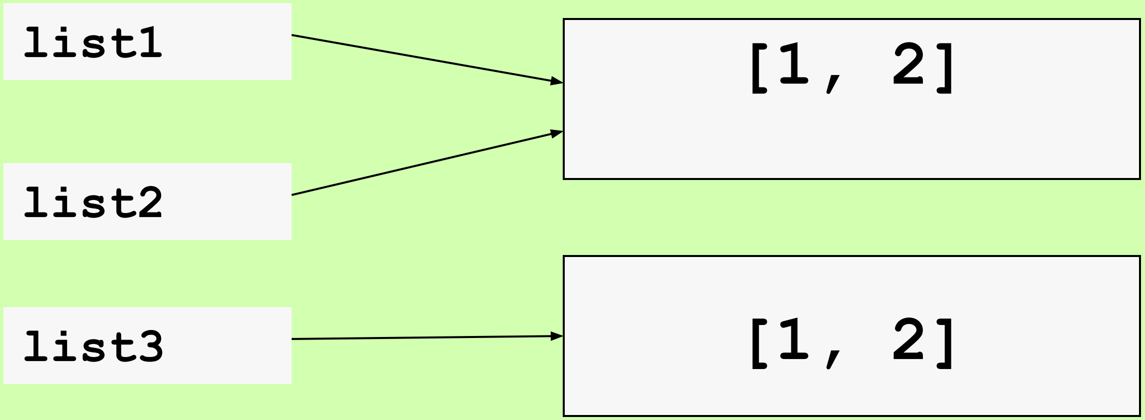
What Is Equality?

- *Object identity*: two variables refer to the exact same object
- *Structural equivalence*: two variables refer to distinct objects that have the same contents
- Object identity is pretty strict, maybe too strict
- **==** actually tests for structural equivalence with Python’s data structures

== Should Do Structural Equivalence

```
>>> list1 = [1, 2]
>>> list2 = list1
>>> list3 = [1, 2]
>>> list1 is list2
True
>>> list1 is list3
False
>>> list1 == list3
True
```

Two lists are equal if they are identical or have the same elements



list1

list2

list3

[1, 2]

[1, 2]

Define the Method `__eq__`

```python
class SavingsAccount(object):
    """This class represents a savings account."""

    def __init__(self, name, pin, balance = 0.0):
        self.name = name
        self.pin = pin
        self.balance = balance

    def __eq__(self, other):
        if self is other: return True
        if type(other) != SavingsAccount: return False
        return self.name == other.name and \
               self.pin == other.pin
```

Test for identity, then type, then equality of selected attributes

The operator `==` actually calls the method `__eq__`