OBJECT-ORIENTATION & CLASS INHERITANCE

LECTURE 08-1

JIM FIX, REED COLLEGE CSCI 121

COURSE INFO

▸ The **1st Midterm Exam** is Wednesday
▸ **Today:** we continue looking at object-orientation in Python
  • a few more examples
  • special methods
  • class inheritance

▸ **Reading:** Python object-orientation
  ➡ TP**2e** Ch 15-18
    ✦ at https://greenteapress.com/thinkpython2/thinkpython2.pdf
  ➡ CP Ch 2.5-2.8

SYNTAX: CLASS DEFINITION

Below is a template for most class definitions:

```
class class-name:
    def __init__(self, parameter-list):
        statements that set each of self's attributes

        ...
    def method-name (self, parameter-list):
        statements that access self's attributes

        ...

    ...
```

SYNTAX: CLASS DEFINITION

Below is a template for most class definitions:

```
class class-name:
      def __init__(self,parameter-list):
            statements that set each of self's attributes

            ...
      def method-name (self,parameter-list):
            ...self.attribute... # attribute access
            ...

      ...
```

SYNTAX: CLASS DEFINITION

Below is a template for most class definitions:

```
class class-name:
    def __init__(self, parameter-list):
        statements that set each of self's attributes

        ...
    def method-name(self, parameter-list):

        ...
        # method invocation
        self.method(parameters)

        ...

    ...
```

SYNTAX: CLASS DEFINITION

Below is a template for most class definitions:

```
class class-name:
        def __init__(self,parameter-list):
                ...
        def method-name(self,parameter-list):
                ...
        ...
```

Here is client code for creating a new object instance:

```
thing = class-name(parameters)
thing.method-name(parameters)
```

EXAMPLE: RATIONAL NUMBER CLASS

Here is our rational number data structure as an object class

```python
class Rational:
    def __init__(self,n,d):
        if d < 0:
            n *= -1
            d *= -1
        g = GCD(n,d)
        self.numerator = n // g
        self.denominator = d // g

    def getNumerator(self):
        return self.numerator

    def getDenominator(self):
        return self.denominator
```

EXAMPLE: RATIONAL NUMBER ADDITION METHOD

We can define multiplication of rational numbers as we did before:

```
class Rational:
    def __init__(self,n,d): …
    def getNumerator(self): …
    def getDenominator(self): …

    def times(self,other):
        sn = self.getNumerator()
        sd = self.getDenominator()
        on = other.getNumerator()
        od = other.getDenominator()
        return Rational(sn*on, sd*od)
```

EXAMPLE: RATIONAL NUMBER ADDITION METHOD

We can define addition of rational numbers as we did before:

```python
class Rational:
    def __init__(self,n,d): …
    def getNumerator(self): …
    def getDenominator(self): …
    def times(self,other): …
    def plus(self,other):
        sn = self.getNumerator()
        sd = self.getDenominator()
        on = other.getNumerator()
        od = other.getDenominator()
        return Rational(sn*od + on*sd, sd*od)
```

OUR RATIONAL NUMBER OBJECT IN ACTION

▸ With these defined, here is an interaction:

```
>>> a = Rational(1, 3)
>>> a.asString()
'1 / 3'
>>> b = Rational(1, 2)
>>> ba = b.times(a)
>>> ba.asString()
'1 / 6'
>>> c = a.plus(ba)
>>> c.asString()
'1 / 2'
```

OUR RATIONAL NUMBER OBJECT IN ACTION

▸ Wouldn't this be great to see instead?

```
>>> a = Rational(1, 3)
>>> a
1 / 3
>>> b = Rational(1, 2)
>>> b * a
1 / 6
>>> a + b * a
1 / 2
```

EXAMPLE: DEFINING THE TIMES OPERATION

Python has "special methods" that provide hooks to using operator syntax:

```python
class Rational:
    def __init__(self,n,d): …
    …

    # defines r1 * r2
    def __mul__(self,other):
        sn = self.getNumerator()
        sd = self.getDenominator()
        on = other.getNumerator()
        od = other.getDenominator()
        return Rational(sn*on, sd*od)
```

EXAMPLE: DEFINING THE PLUS OPERATION

```python
class Rational:
    def __init__(self,n,d): …
    def getNumerator(self): …
    def getDenominator(self): …
    def __mul__(self,other): …

    # defines r1 + r2
    def __add__(self,other):
        sn = self.getNumerator()
        sd = self.getDenominator()
        on = other.getNumerator()
        od = other.getDenominator()
        return Rational(sn*od + on*sd, sd*od)
```

SPECIAL METHODS

Python has "special methods" for lots of built-in syntax.

▸ They are surrounded by a double underscore (_)

▸ Documented at this technical page:

  ➡ https://docs.python.org/3/reference/datamodel.html#special-method-names

▸ Nice overview here:

  ➡ https://www.pythonlikeyoumeanit.com/Module4_OOP/Special_Methods.html

Example:

```
def __mul__(self,other):
    ...
```

▸ Defines `x * y` to mean `x.__mul__(y)`

SPECIAL METHODS

Python has "special methods" for lots of built-in syntax.

▸ They are surrounded by a double underscore (_)

▸ Documented at this technical page:

➡ https://docs.python.org/3/reference/datamodel.html#special-method-names

▸ Nice overview here:

➡ https://www.pythonlikeyoumeanit.com/Module4_OOP/Special_Methods.html

Example:
```
      def __eq__(self,other):
          ...
```
▸ Defines `x == y` to mean `x.__eq__(y)`

SPECIAL METHODS

Python has "special methods" for lots of built-in syntax.
- They are surrounded by a double underscore (_)
- Documented at this technical page:
  - ➡ https://docs.python.org/3/reference/datamodel.html#special-method-names
- Nice overview here:
  - ➡ https://www.pythonlikeyoumeanit.com/Module4_OOP/Special_Methods.html

Example:
```
def __getitem__(self,index):
    …
```
- Defines `x[i]` to mean `x.__getitem__(i)`

SPECIAL METHODS

Python has "special methods" for lots of built-in syntax.

▸ They are surrounded by a double underscore (_)

▸ Documented at this technical page:

➡ https://docs.python.org/3/reference/datamodel.html#special-method-names

▸ Nice overview here:

➡ https://www.pythonlikeyoumeanit.com/Module4_OOP/Special_Methods.html

Example:

```
def __str__(self):
    …
```

▸ Defines `str(x)` to mean `x.__str__()`

▸ Also used for `print(x)`. It means `print(x.__str__())`

SPECIAL METHODS

Python has "special methods" for lots of built-in syntax.
- They are surrounded by a double underscore (_)
- Documented at this technical page:
  - ➡ https://docs.python.org/3/reference/datamodel.html#special-method-names
- Nice overview here:
  - ➡ https://www.pythonlikeyoumeanit.com/Module4_OOP/Special_Methods.html

Example:
```
    def __repr__(self):
        ...
```
- Defines the string "representation" of an object.

SPECIAL METHODS

Python has "special methods" for lots of built-in syntax.

▸ They are surrounded by a double underscore (_)

▸ Documented at this technical page:

    ➡ https://docs.python.org/3/reference/datamodel.html#special-method-names

▸ Nice overview here:

    ➡ https://www.pythonlikeyoumeanit.com/Module4_OOP/Special_Methods.html

Example:

```
    def __repr__(self):
        ...
```

▸ Used by the interpreter to display the object's value, like so:
```
>>> Rational(27, 33)
9 / 11
```

RECALL: ACCOUNT CLASS

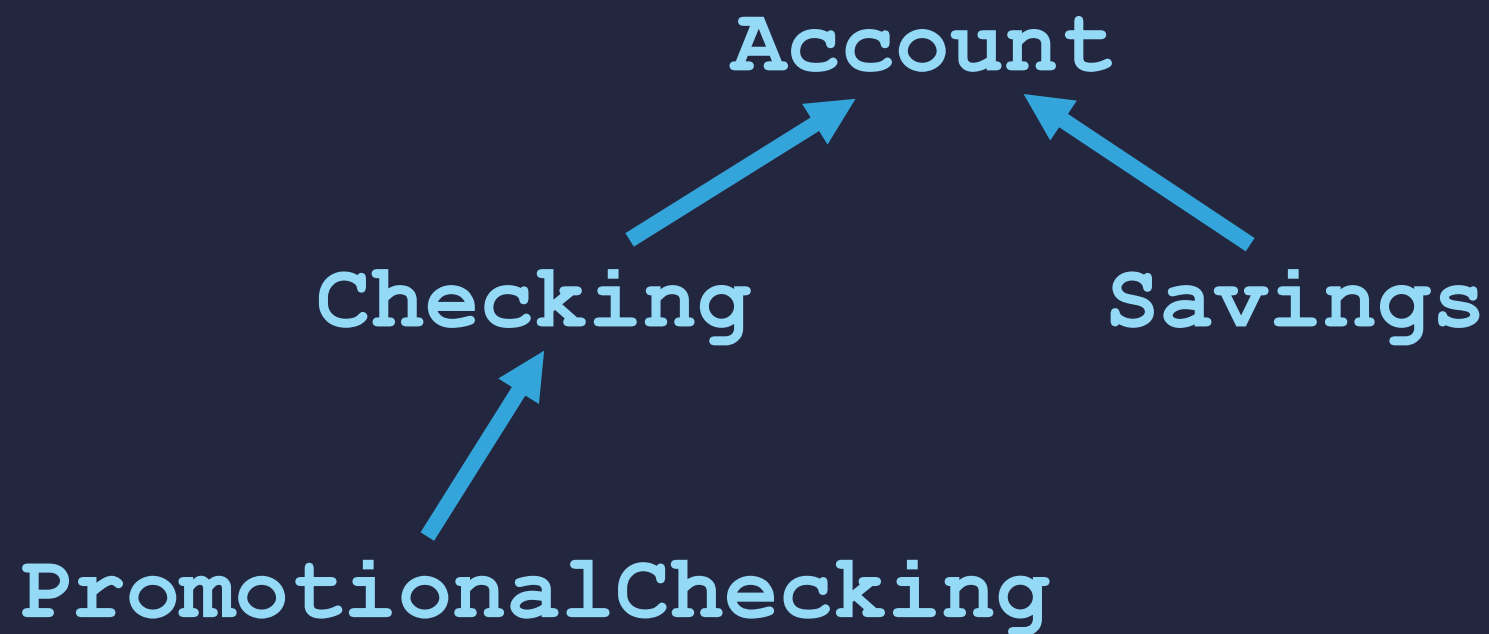▸ Here is the class definition of a new **Account** type:

```
class Account:
interest_rate = .02
def __init__(self, amount):
        self.balance = amount
def deposit(self, amount):
        self.balance += amount
def payInterest(self):
        self.balance *= 1.0 + self.interest_rate
def getBalance(self):
        return self.balance
```

▸ Here is **Account** in use:

```
>>> a = Account(150)
>>> a.deposit(50)
>>> a.payInterest()
>>> a.getBalance()
204.0
```

AN ACCOUNT CLASS HIERARCHY

▸ We can build *hierarchies* of different accounts:

Account

Checking   **Savings**

PromotionalChecking

▸ We make *subclasses* that *inherit* the attributes of their "*superclasses*"
- A **Savings** account has all the info and operations of an **Account**.
- But it has features and behavior more specific to checking accounts
  - ✦ This is called subclass *specialization*.
  - ✦ We *extend* the superclass with additional attributes.
- It also *overrides* some of the behavior it inherits from **Account**.

INHERITANCE EXAMPLE: A SAVINGS ACCOUNT

```python
class Account:
    interest_rate = .02
    def __init__(self, amount):
        self.balance = amount
    def deposit(self, amount):
        self.balance += amount
    def withdraw(self, amount):
        self.balance -= amount
    def payInterest(self):
        self.balance *= 1.0 + self.interest_rate

class Savings(Account):
    interest_rate = 0.04
    withdraw_fee = 1.0
    def withdraw(self, amount):
        Account.withdraw(self, amount + self.withdraw_fee)
```

INHERITANCE EXAMPLE: A SAVINGS ACCOUNT

```python
class Account:
    interest_rate = .02
    def __init__(self, amount):
        self.balance = amount
    def deposit(self, amount):
        self.balance += amount
    def withdraw(self, amount):
        self.balance -= amount
    def payInterest(self):
        self.balance *= 1.0 + self.interest_rate

class Savings(Account):  # inherit the methods and class variables of Account
    interest_rate = 0.04
    withdraw_fee = 1.0
    def withdraw(self, amount):
        Account.withdraw(self, amount + self.withdraw_fee)
```

INHERITANCE EXAMPLE: A SAVINGS ACCOUNT

```python
class Account:
    interest_rate = .02
    def __init__(self, amount):
        self.balance = amount
    def deposit(self, amount):
        self.balance += amount
    def withdraw(self, amount):
        self.balance -= amount
    def payInterest(self):
        self.balance *= 1.0 + self.interest_rate

class Savings(Account):
    interest_rate = 0.04  # overrides the class variable inherited from Account
    withdraw_fee = 1.0
    def withdraw(self, amount):
        Account.withdraw(self, amount + self.withdraw_fee)
```

INHERITANCE EXAMPLE: A SAVINGS ACCOUNT

```python
class Account:
    interest_rate = .02
    def __init__(self, amount):
        self.balance = amount
    def deposit(self, amount):
        self.balance += amount
    def withdraw(self, amount):
        self.balance -= amount
    def payInterest(self):
        self.balance *= 1.0 + self.interest_rate

class Savings(Account):
    interest_rate = 0.04
    withdraw_fee = 1.0 # extends with a specializing class variable
    def withdraw(self, amount):
        Account.withdraw(self, amount + self.withdraw_fee)
```

INHERITANCE EXAMPLE: A SAVINGS ACCOUNT

```python
class Account:
    interest_rate = .02
    def __init__(self, amount):
        self.balance = amount
    def deposit(self, amount):
        self.balance += amount
    def withdraw(self, amount):
        self.balance -= amount
    def payInterest(self):
        self.balance *= 1.0 + self.interest_rate

class Savings(Account):
    interest_rate = 0.04
    withdraw_fee = 1.0
    def withdraw(self, amount): # overrides a method inherited from Account
        Account.withdraw(self, amount + self.withdraw_fee)
```

INHERITANCE EXAMPLE: A SAVINGS ACCOUNT

```python
class Account:
    interest_rate = .02
    def __init__(self, amount):
        self.balance = amount
    def deposit(self, amount):
        self.balance += amount
    def withdraw(self, amount):
        self.balance -= amount
    def payInterest(self):
        self.balance *= 1.0 + self.interest_rate

class Savings(Account):
    interest_rate = 0.04
    withdraw_fee = 1.0
    def withdraw(self, amount):    # overrides a method inherited from Account
        Account.withdraw(self, amount + self.withdraw_fee)
        # explicitly invokes the method of its superclass
```

INHERITANCE EXAMPLE: A SAVINGS ACCOUNT

```python
class Account:
    interest_rate = .02
    def __init__(self, amount):
        self.balance = amount
    def deposit(self, amount):
        self.balance += amount
    def withdraw(self, amount):
        self.balance -= amount
    def payInterest(self):
        self.balance *= 1.0 + self.interest_rate

class Savings(Account):
    interest_rate = 0.04
    withdraw_fee = 1.0
    def withdraw(self, amount): # overrides a method inherited from Account
        Account.withdraw(self, amount + self.withdraw_fee)
        # explicitly invokes the method of its superclass
```

ACCOUNT VERSUS SAVINGS

▸ Here is **Account** in use:

```
>>> a = Account(100)
>>> a.balance
100.0
>>> a.payInterest()
>>> a.balance
102.0
>>> a.withdraw(20)
>>> a.balance
82.0
```

▸ Here is **Savings** in use:

```
>>> a = Savings(100)
>>> a.balance
100.0
>>> a.payInterest()
>>> a.balance
104.0
>>> a.withdraw(20)
>>> a.balance
83.0
```

INHERITANCE EXAMPLE: A CHECKING ACCOUNT

```python
class Account:
    interest_rate = .02
    def __init__(self, amount):
        self.balance = amount
    def deposit(self, amount):
        self.balance += amount
    def withdraw(self, amount):
        self.balance -= amount
    def payInterest(self):
        self.balance *= 1.0 + self.interest_rate


class Checking(Account):

    min_balance = 1000.0

    def payInterest(self):
        if self.balance >= self.min_balance:
            Account.payInterest(self)
```

CHECKING ACCOUNT INTERACTION

▸ Here is **Checking** in use:

```
>>> a = Checking(1000.0)
>>> a.balance
1000.0
>>> a.payInterest()
>>> a.balance
1040.0
>>> a.withdraw(50.0)
>>> a.balance
990.0
>>> a.payInterest()
>>> a.balance
990.0
```

INHERITANCE EXAMPLE: A PROMOTIONAL CHECKING ACCOUNT

```python
class Checking(Account):

    min_balance = 1000.0

    def payInterest(self):
        if self.balance >= self.min_balance:
            Account.payInterest(self)

class PromotionalChecking(Checking):

    reward = 50

    def __init__(self,amount):
        Checking.__init__(self,amount+self.reward)
        # The code above explicitly uses the initializer code from Checking
```

INHERITANCE EXAMPLE: A PROMOTIONAL CHECKING ACCOUNT

```python
class Checking(Account):

    min_balance = 1000.0

    def payInterest(self):
        if self.balance >= self.min_balance:
            Account.payInterest(self)

class PromotionalChecking(Checking):

    reward = 50

    def __init__(self,amount):
        super().__init__(amount+self.reward)
        # The code above explicitly uses the initializer code from Checking
```

INHERITANCE EXAMPLE: A PROMOTIONAL CHECKING ACCOUNT

```python
class Checking(Account):

    min_balance = 1000.0

    def payInterest(self):
        if self.balance >= self.min_balance:
            Account.payInterest(self)


class PromotionalChecking(Checking):

    reward = 50

    def __init__(self,amount):
        super().__init__(amount+self.reward)
        # The code above uses the initializer code from Checking that was inherited from Account
        # Using super() references self as though it is an instance of its superclass
```

OBJECT TAKEAWAYS

▸ New object types are defined with class.

▸ Within the class you define these things:

- **`__init__`**

- other methods

- (maybe) class attributes

▸ Method parameters are `self` followed by the others.

▸ Object dot notation:

- Methods are called using `receiver.method(...)`

- Instance variables are accessed by `receiver.variable`

- We use self. notation inside a method to access these things too.

▸ New instances are built with `class-name(...)`

INHERITANCE TAKEAWAYS

‣ A class inherits from its superclass with
  ⇨ `class class-name(super-class-name):`

‣ You can call the superclass initializer with the syntax:
  ⇨ `super-class-name.__init__(self,...)`

‣ You can call the superclass methods with the syntax:
  ⇨ `super-class-name.method(self,...)`

‣ Subclasses inherit the methods of their superclass.

‣ They can be *specialized* in two ways:

  • You can add additional attributes and methods.

  • You can override super-class methods.