DATA ABSTRACTIONS & OBJECT-ORIENTATION

LECTURE 07-2

JIM FIX, REED COLLEGE CSCI 121

AFTER BREAK

- **In-Class Midterm Exam:** Wednesday, March 22nd
  - *closed note, closed computer, hand-written*
  - about 6 or 7 problems similar to quiz and homework problems
  - **Topics covered:**
    - scripting, including `input` and `print`
    - `int` and `str` operations
    - function and procedure `def`; `return`; the `None` value
    - conditional `if`-`else` statements; `while` loops; `bool`
    - lists and dictionaries
  - There is a practice exam on the course website under the Handouts tab
  - I will post practice exam solutions on Monday, March 20th.

AFTER BREAK

- **Project 2 due:** Monday, March 30^{th} by 1pm
  - ***stats and chats***

TODAY

- **Today:**
  - inventing your own data structures and data types
  - object-oriented programming in Python
- **Reading:** on Python object-orientation
  - TP Ch 12, 14-16
  - CP Ch 2.5-2.8

FUNCTIONAL/PROCEDURAL ABSTRACTION

Idea: invent new operations and actions that constitute your program

- We use the `def` statement to define ***functions*** and ***procedures***
  - We give them meaningful and memorable names.
  - We take care to make them broadly useful.

- Good definitions enhance code ***modularity***
  - They can be made part of a ***library*** used by several programs.
  - Makes code collaboration easier and larger programs easier to write.

FUNCTIONAL/PROCEDURAL ABSTRACTION

- Functions/procedures create a useful ***barrier of abstraction***.
  - Make code easier to read
  - You need not know all the details.
  - Only need to know the function's interface and behavior.

```
def removeDuplicates(someList):
    """This modifies a list so each item occurs just once."""
    ...messy code details here and below...
```

DATA ABSTRACTION

Idea: invent a new data object that your program needs.

- Determine its features and components.
  - These are its ***attributes***.

- Consider the *operations* you'd like it to support.
  - e.g. access, queries, look-ups, checks, changes, actions, activities, ...
  - These are its ***methods***.

DATA ABSTRACTION

Idea: invent a new data object that your program needs.

- Determine its features and components.
  - These are its ***attributes***.
- Consider the *operations* you'd like it to support.
  - e.g. access, queries, look-ups, checks, changes, actions, activities, ...
  - These are its ***methods***.
- Sometimes the object is a *collection*, organized in a useful way.
  - In that case it's a ***data structure***.
- Python provides a few: "tuples" (e.g. pairs), strings, lists, dictionaries.
- Others: vectors, stacks, queues, linked lists, trees, graphs, ...

DATA ABSTRACTION: ADVANTAGES

Idea: invent a new data object that your program needs.

- Can be special purpose, geared for a specific application or algorithm.
  - Tuples, lists, and dictionaries can sometimes be too generic, featureless.
  - Can write code that reads how you think about your program's activity.
  - This is the data analog to functional abstraction.
- Some data abstractions have universal value, can be reused.
  - A good design saves programming effort in the future
- Abstraction forces a modular design.
  - It makes code easier to understand; easier to get right.
  - May even be useful elsewhere.

EXAMPLE: RATIONAL NUMBER OBJECT

- We can represent a rational number in Python with a list
  - It stores two items: its integer numerator and denominator.

- Here are some basic operations on our rational number object:
  - Make a new rational number (an object *constructor*):

```
def createRational(n, d):
    return [n, d]
```

  - Get the numerator (object's *accessor* or “getter”):

```
def numerator(r):
    return r[0]
```

  - Get the denominator (another “getter”):

```
def denominator(r):
    return r[1]
```

EXAMPLE: RATIONAL NUMBER OBJECT

- We can build operations that work with rational number objects:

$$\frac{3}{4} * \frac{2}{3} = \text{???}$$

  - We can invent rational number multiplication:

```
def rationalProduct(r, s):
    newNumer = numerator(r) * numerator(s)
    newDenom = denominator(r) * denominator(s)
    return createRational(newNumer,newDenom)
```

EXAMPLE: RATIONAL NUMBER OBJECT

- We can build operations that work with rational number objects:

$$\frac{3}{4} * \frac{2}{3} = \frac{3 * 2}{4 * 3} =$$

- We can invent rational number multiplication:

```
def rationalProduct(r, s):
    newNumer = numerator(r) * numerator(s)
    newDenom = denominator(r) * denominator(s)
    return createRational(newNumer,newDenom)
```

EXAMPLE: RATIONAL NUMBER OBJECT

- We can build operations that work with rational number objects:

$$\frac{3}{4} * \frac{2}{3} = \frac{3 * 2}{4 * 3} = \frac{6}{12}$$

- We can invent rational number multiplication:

```
def rationalProduct(r, s):
    newNumer = numerator(r) * numerator(s)
    newDenom = denominator(r) * denominator(s)
    return createRational(newNumer,newDenom)
```

EXAMPLE: RATIONAL NUMBER OBJECT

- We can build operations that work with rational number objects:

$$\frac{3}{4} + \frac{2}{3} = ???$$

- We can invent rational number addition:

```
def rationalSum(r, s):
    nr,dr = numerator(r),denominator(r)
    ns,ds = numerator(s),denominator(s)
    newNumer = nr*ds + ns*dr
    newDenom = ds*dr
    return createRational(newNumer,newDenom)
```

EXAMPLE: RATIONAL NUMBER OBJECT

- We can build operations that work with rational number objects:

$$\frac{3}{4} + \frac{2}{3} = \frac{3 * 3}{4 * 3} + \frac{2 * 4}{3 * 4}$$

- We can invent rational number addition:

```
def rationalSum(r, s):
    nr,dr = numerator(r),denominator(r)
    ns,ds = numerator(s),denominator(s)
    newNumer = nr*ds + ns*dr
    newDenom = ds*dr
    return createRational(newNumer,newDenom)
```

EXAMPLE: RATIONAL NUMBER OBJECT

- We can build operations that work with rational number objects:

$$\frac{3}{4} + \frac{2}{3} = \frac{9}{12} + \frac{8}{12} = \frac{17}{12}$$

- We can invent rational number addition:

```
def rationalSum(r, s):
    nr,dr = numerator(r),denominator(r)
    ns,ds = numerator(s),denominator(s)
    newNumer = nr*ds + ns*dr
    newDenom = ds*dr
    return createRational(newNumer,newDenom)
```

EXAMPLE: RATIONAL NUMBER OBJECT

- We can build operations that work with rational number objects:

$$\frac{a}{b} == \frac{c}{d} \quad \text{whenever} \quad a_* d == c_* b$$

- We can check whether two rational numbers are the same:

```
def areSameRationals(r, s):
    nr,dr = numerator(r),denominator(r)
    ns,ds = numerator(s),denominator(s)
    return (nr*ds == ns*dr)
```

EXAMPLE: RATIONAL NUMBER OBJECT

- We can build operations that work with rational number objects:
  - We can invent ways of displaying and reporting rational numbers

```
def stringOfRational(r):
    ntext = str(numerator(r))
    dtext = str(denominator(r))
    return ntext + "/" + dtext

def outputRational(r):
    print(stringOfRational(r))
```

  - Other operations: subtraction, division, conversion to `float`, ...

OUR RATIONAL NUMBER OBJECT IN ACTION

- With these defined, here is an interaction:

```
>>> a = createRational(1, 3)
>>> b = createRational(1, 2)
>>> c = rationalSum(a, rationalProduct(b, a))
>>> outputRational(c)
9 / 18
```

- Here, we are relying on functional abstraction to provide data abstraction.
  - The function calls hide the underlying representation.

- This allows us to change that underlying implementation easily:
  - We can enhance or rewrite the underlying code...
  - ...with no change to the "client" code that relies on it.

- Provides an ***abstraction barrier*** that makes code maintainable.
  - The details are hidden from the code that uses the object.

EXAMPLE: AN ENHANCED RATIONAL NUMBER OBJECT

- We change our constructor from this...

```
def createRational(n, d):
    return [n, d]
```

- ...to this, which simplifies the numerator and denominator with the *GCD*:

```
def createRational(n, d):
    g = GCD(n,d)          # Find greatest common divisor
    return [n//g, d//g]
```

- Our script doesn't need to change, but the object's behavior is improved:

```
>>> a = createRational(1, 3)
>>> b = createRational(1, 2)
>>> c = rationalSum(a, rationalProduct(b, a))
>>> outputRational(c)
1 / 2
```

EXAMPLE: RATIONAL OBJECT USING A DICTIONARY INSTEAD

- Note that we could use a dictionary instead:

```
def createRational(n, d):
    g = GCD(n,d)
    return {"numerator":n//g, "denominator":d//g}

def numerator(r):
    return r["numerator"]

def denominator(r):
    return r["denominator"]
```

- Client code need not change since it uses the getters and constructor:

```
def rationalSum(r, s):
    nr,dr = numerator(r),denominator(r)
    ns,ds = numerator(s),denominator(s)
    newNumer = nr*ds + ns*dr
    newDenom = ds*dr
    return createRational(newNumer,newDenom)
```

EXAMPLE: A GIFT CARD OBJECT

- Here is a gift card object's use:

```
>>> gc = createGiftCard(100)
>>> spend(gc,20)
80
>>> spend(gc,45)
35
>>> spend(gc,50)
'Insufficient funds'
>>> spend(gc,20)
15
```

EXAMPLE: GIFT CARD OBJECT USING A DICTIONARY

- We could use a dictionary to represent a gift card:

```
def createGiftCard(amount):
    return {"balance":amount}
```

EXAMPLE: GIFT CARD OBJECT USING A DICTIONARY

- We could use a dictionary to represent a gift card:

```
def createGiftCard(amount):
    return {"balance":amount}

def spend(giftCard,amount):
    balance = giftCard["balance"]
    if amount > balance:
        return “Insufficient funds”
    balance -= amount
    # update the object's info
    giftCard["balance"] = balance
    return balance
```

EXAMPLE: GIFT CARD OBJECT USING A DICTIONARY

- We could use a dictionary to represent a gift card:

```
def createGiftCard(amount):
    return {"balance":amount}

def spend(giftCard,amount):
    balance = giftCard["balance"]
    if amount > balance:
        return "Insufficient funds"
    balance -= amount
    # update the object's info
    giftCard["balance"] = balance
    return balance

def addFunds(giftCard,amount):
    giftCard["balance"] += amount
    return giftCard["balance"]
```

GIFT CARD SUMMARY

- We made a gift card object that responds to two kinds of request:
  - We could spend money from the card.
  - We could add funds to the card.

- We built these as two different functions.

OBJECT TERMINOLOGY

- `spend` and `addFunds` are *messages* to which gift card objects respond.
- Their code are the gift card's ***methods*** for handling each request.
- The suite of messages that an object supports is its ***interface***.

OBJECT ORIENTATION

- Many languages support coding up data abstractions in this style.
  - They allow you to invent your own type of object.
  - They let you define its attributes, the information each object stores.
  - They allow you to define a set of operations on that type.

- Your code is organized as a ***class definition*** for that object type.

OBJECT ORIENTATION

- These are called ***class-based object-oriented*** languages.
  - **Python** is an example, as is **C++** and **Java**.

- Object-oriented languages have special syntax for:
  - constructors
  - attribute access
  - method definition

EXAMPLE: GIFT CARD CLASS

- Here is the class definition of a new **`GiftCard`** type:

```
class GiftCard:

    def __init__(self, amount): # used by the constructor
        self.balance = amount

    def addFunds(self, amount): # a method definition
        self.balance = self.balance + amount
        return self.balance

    def spend(self, amount):    # another method definition
        if amount > self.balance:
            return "Insufficient funds"
        self.balance = self.balance - amount
        return self.balance
```

EXAMPLE: GIFT CARD CLASS

- Here is the class definition of a new `GiftCard` type:

```
class GiftCard:

    def __init__(self, amount): # used by the constructor
        self.balance = amount

    def addFunds(self, amount): # a method definition
        self.balance = self.balance + amount
        return self.balance

    def spend(self, amount):    # another method definition
        if amount > self.balance:
            return “Insufficient funds”
        self.balance = self.balance – amount
        return self.balance

    def getBalance(self):      # a balance “getter”
        return self.balance
```

EXAMPLE: USING A GIFT CARD OBJECT

- Here is a gift card object's use, assuming there is a `"GiftCard.py"` file:

```
>>> from GiftCard import GiftCard
>>> gc = GiftCard(100) # use the constructor; it calls __init__
>>> gc.spend(20)
80
>>> gc.spend(45)
35
>>> gc.spend(50)
'Insufficient funds'
>>> gc.getBalance()
35
>>> gc.addFunds(20)
55
>>> gc.spend(50)
5
>>> gc.balance    # Python lets a client access attributes EEK!
5
```

EXAMPLE: ACCOUNT CLASS

- Here is the class definition of a new `Account` type:

```
class Account:

    rate = .02

    def __init__(self, amount):
        self.balance = amount

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

    def payInterest(self):
        self.balance *= 1.0 + rate
```

EXAMPLE: ACCOUNT (

```
class Account:

    rate = .02

    def __init__(self, amount):
        self.balance = amount

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

    def payInterest(self):
        self.balance *= 1.0 + rate
```

- Here is **`Account`** in use:

```
>>> a = Account(100)
```

EXAMPLE: ACCOUNT (

```
class Account:

    rate = .02

    def __init__(self, amount):
        self.balance = amount

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

    def payInterest(self):
        self.balance *= 1.0 + rate
```

- Here is **`Account`** in use:

```
>>> a = Account(100)
```

COMMENTARY

- Class name is used like a function. We are calling the *constructor*.
  → this creates a new object, an instance of class **`Account`**
- **`__init__`** code runs with this new object passed as **`self`**.
- The argument is passed as the other parameter to **`__init__`**.

EXAMPLE: ACCOUNT (

```
class Account:

    rate = .02

    def __init__(self, amount):
        self.balance = amount

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

    def payInterest(self):
        self.balance *= 1.0 + rate
```

- Here is **`Account`** in use:

```
>>> a = Account(100)
```

COMMENTARY

- Class name is used like a function. We are calling the *constructor*.
  - this creates a new object, an instance of class **`Account`**
- **`__init__`** code runs with this new object passed as **`self`**.
- The argument is passed as the other parameter to **`__init__`**.

EXAMPLE: ACCOUNT C

```
class Account:

    rate = .02

    def __init__(self, amount):
        self.balance = amount

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

    def payInterest(self):
        self.balance *= 1.0 + rate
```

- Here is **`Account`** in use:

```
>>> a = Account(100)
```

COMMENTARY

- Class name is used like a function. We are calling the *constructor*.
  ➡ this creates a new object, an instance of class **`Account`**
- **`__init__`** code runs with this new object passed as **`self`**.
- The argument is passed as the other parameter to **`__init__`**.

EXAMPLE: ACCOUNT (

```
class Account:

    rate = .02

    def __init__(self, amount):
        self.balance = amount

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

    def payInterest(self):
        self.balance *= 1.0 + rate
```

- Here is `Account` in use:

```
>>> a = Account(100)
```

COMMENTARY

- Class name is used like a function. We are calling the *constructor*.
  ➡ this creates a new object, an instance of class `Account`
- `__init__` code runs with this new object passed as `self`.
- The argument is passed as the other parameter to `__init__`.

EXAMPLE: ACCOUNT C

```
class Account:

    rate = .02

    def __init__(self, amount):
        self.balance = amount

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

    def payInterest(self):
        self.balance *= 1.0 + rate
```

- Here is **Account** in use:

```
>>> a = Account(100)
```

COMMENTARY

- Class name is used like a function. We are calling the *constructor*.
  - this creates a new object, an instance of class **Account**
- **__init__** code runs with this new object passed as **self**.
- The argument is passed as the other parameter to **__init__**.

EXAMPLE: ACCOUNT (

- Here is **`Account`** in use:

```
>>> a = Account(100)
>>> a.balance
100
```

```
class Account:

    rate = .02

    def __init__(self, amount):
        self.balance = amount

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

    def payInterest(self):
        self.balance *= 1.0 + rate
```

EXAMPLE: ACCOUNT C

```
class Account:

    rate = .02

    def __init__(self, amount):
        self.balance = amount

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

    def payInterest(self):
        self.balance *= 1.0 + rate
```

- Here is `Account` in use:

```
>>> a = Account(100)
>>> a.balance
100
```

COMMENTARY

- This expression performs an access of an ***instance variable***.
  - Syntax: ***object . attribute-name***
- Gets the value of an attribute with that name from the object.

EXAMPLE: ACCOUNT C

```
class Account:

    rate = .02

    def __init__(self, amount):
        self.balance = amount

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

    def payInterest(self):
        self.balance *= 1.0 + rate
```

- Here is **Account** in use:

```
>>> a = Account(100)
>>> a.balance
100
```

COMMENTARY

- This expression performs an access of an ***instance variable***.
  - Syntax: ***object . attribute-name***
  - Gets the value of an attribute with that name from the object.

EXAMPLE: ACCOUNT (

```
class Account:

    rate = .02

    def __init__(self, amount):
        self.balance = amount

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

    def payInterest(self):
        self.balance *= 1.0 + rate
```

- Here is `Account` in use:

```
>>> a = Account(100)
>>> a.balance
100
```

COMMENTARY

- This expression performs an access of an ***instance variable***.
  - Syntax: ***object . attribute-name***
    - Gets the value of an attribute with that name from the object.

EXAMPLE: ACCOUNT (

```
class Account:

    rate = .02

    def __init__(self, amount):
        self.balance = amount

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

    def payInterest(self):
        self.balance *= 1.0 + rate
```

- Here is `Account` in use:

```
>>> a = Account(100)
>>> a.balance
100
```

COMMENTARY

- This expression performs an access of an ***instance variable***.
  - Syntax: ***object . attribute-name***
- Gets the value of an attribute with that name from the object.

EXAMPLE: ACCOUNT C

```
class Account:

    rate = .02

    def __init__(self, amount):
        self.balance = amount

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

    def payInterest(self):
        self.balance *= 1.0 + rate
```

- Here is **`Account`** in use:

```
>>> a = Account(100)
>>> a.balance
100
>>> a.rate
0.02
```

EXAMPLE: ACCOUNT C

```
class Account:

    rate = .02

    def __init__(self, amount):
        self.balance = amount

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

    def payInterest(self):
        self.balance *= 1.0 + rate
```

- Here is **Account** in use:

```
>>> a = Account(100)
>>> a.balance
100
>>> a.rate
0.02
```

COMMENTARY

- The same notation is used to look up ***class variables***.
- If an object is missing an attribute, the class is checked instead.
- You can also access it directly inside the class.

EXAMPLE: ACCOUNT (

```
class Account:

    rate = .02

    def __init__(self, amount):
        self.balance = amount

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

    def payInterest(self):
        self.balance *= 1.0 + rate
```

- Here is **`Account`** in use:

```
>>> a = Account(100)
>>> a.balance
100
>>> a.rate
0.02
```

COMMENTARY

- The same notation is used to look up ***class variables***.
- If an object is missing an attribute, the class is checked instead.
- You can also access it directly inside the class.

EXAMPLE: ACCOUNT (

```
class Account:

    rate = .02

    def __init__(self, amount):
        self.balance = amount

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

    def payInterest(self):
        self.balance *= 1.0 + rate
```

- Here is **Account** in use:

```
>>> a = Account(100)
>>> a.balance
100
>>> Account.rate
0.02
```

COMMENTARY

- The same notation is used to look up ***class variables***.
- If an object is missing an attribute, the class is checked instead.
- You can also access a class variable by "dotting" with the class.

EXAMPLE: ACCOUNT (

- Here is **Account** in use:

```
>>> a = Account(100)
>>> a.balance
100
>>> a.rate
0.02
>>> a.deposit(50)
```

```
class Account:

    rate = .02

    def __init__(self, amount):
        self.balance = amount

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

    def payInterest(self):
        self.balance *= 1.0 + rate
```

EXAMPLE: ACCOUNT (

```
class Account:

    rate = .02

    def __init__(self, amount):
        self.balance = amount

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

    def payInterest(self):
        self.balance *= 1.0 + rate
```

- Here is `Account` in use:

```
>>> a = Account(100)
>>> a.balance
100
>>> a.rate
0.02
>>> a.deposit(50)
```

COMMENTARY

- This expression requests execution of a ***method***.
  - Similar syntax: ***object . method-name (...arguments...)***
- This behaves a lot like a function call.

EXAMPLE: ACCOUNT (

```
class Account:

    rate = .02

    def __init__(self, amount):
        self.balance = amount

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

    def payInterest(self):
        self.balance *= 1.0 + rate
```

- Here is **`Account`** in use:

```
>>> a = Account(100)
>>> a.balance
100
>>> a.rate
0.02
>>> a.deposit(50)
```

COMMENTARY

- This expression requests execution of a ***method***.
    - Similar syntax: ***object . method-name (...arguments...)***
- This behaves a lot like a function call.

EXAMPLE: ACCOUNT (

```
class Account:

    rate = .02

    def __init__(self, amount):
        self.balance = amount

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

    def payInterest(self):
        self.balance *= 1.0 + rate
```

- Here is `Account` in use:

```
>>> a = Account(100)
>>> a.balance
100
>>> a.rate
0.02
>>> a.deposit(50)
```

COMMENTARY

- This expression requests execution of a ***method***.
  - Similar syntax: ***object . method-name (...arguments...)***
- This behaves a lot like a function call.

EXAMPLE: ACCOUNT (

```
class Account:

    rate = .02

    def __init__(self, amount):
        self.balance = amount

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

    def payInterest(self):
        self.balance *= 1.0 + rate
```

- Here is **Account** in use:

```
>>> a = Account(100)
>>> a.balance
100
>>> a.rate
0.02
>>> a.deposit(50)
```

COMMENTARY

- This expression requests execution of a ***method***.
  - Similar syntax: ***object . method-name (...arguments...)***
- This behaves a lot like a function call.
  - The argument value is passed as the parameter **amount**.

EXAMPLE: ACCOUNT (

```
class Account:

    rate = .02

    def __init__(self, amount):
        self.balance = amount

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

    def payInterest(self):
        self.balance *= 1.0 + rate
```

- Here is **Account** in use:

```
>>> a = Account(100)
>>> a.balance
100
>>> a.rate
0.02
>>> a.deposit(50)
```

COMMENTARY

- This expression requests execution of a ***method***.
  - Similar syntax: ***object . method-name (...arguments...)***
- This behaves a lot like a function call.
  - The ***message receiver object*** is passed as **self**.

EXAMPLE: ACCOUNT (

```
class Account:

    rate = .02

    def __init__(self, amount):
        self.balance = amount

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

    def payInterest(self):
        self.balance *= 1.0 + rate
```

- Here is `Account` in use:

```
>>> a = Account(100)
>>> a.balance
100
>>> a.rate
0.02
>>> a.deposit(50)
>>> a.payInterest()
```

COMMENTARY

- This expression requests execution of a ***method***.
  - Similar syntax: ***object . method-name (...arguments...)***
- Methods with no arguments just have a receiver parameter `self`.

EXAMPLE: ACCOUNT (

```
class Account:

    rate = .02

    def __init__(self, amount):
        self.balance = amount

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

    def payInterest(self):
        self.balance *= 1.0 + rate
```

- Here is **Account** in use:

```
>>> a = Account(100)
>>> a.balance
100
>>> a.rate
0.02
>>> a.deposit(50)
>>> Account.rate
0.02
```

COMMENTARY

- In a way, a class is like an object. It can have attributes.
- There is only one "class object", so only one `Account.rate`
- There is a different `balance` for every `Account` **instance**.

EXAMPLE: ACCOUNT (

```
class Account:

    rate = .02

    def __init__(self, amount):
        self.balance = amount

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

    def payInterest(self):
        self.balance *= 1.0 + rate
```

- Here is `Account` in use:

```
>>> a = Account(100)
>>> a.balance
100
>>> a.rate
0.02
>>> a.deposit(50)
>>> Account.rate
0.02
>>> Account.deposit(a,10)
```

COMMENTARY

- You can also call an instance's method using its class name:
    - Syntax: *class-name* . *instance-method-name* (*receiver* , *arguments*)

EXAMPLE: ACCOUNT (

```
class Account:

    rate = .02

    def __init__(self, amount):
        self.balance = amount

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

    def payInterest(self):
        self.balance *= 1.0 + rate
```

- Here is `Account` in use:

```
>>> a = Account(100)
>>> a.balance
100
>>> a.rate
0.02
>>> a.deposit(50)
>>> Account.rate
0.02
>>> Account.deposit(a,10)
```

COMMENTARY

- You can also call an instance's method using its class name:
  - Syntax: *class-name* . *instance-method-name* (*receiver* , *arguments*)
- It is as if `deposit` is a function attached to the `Account` class.

EXAMPLE: ACCOUNT (

```
class Account:

    rate = .02

    def __init__(self, amount):
        self.balance = amount

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

    def payInterest(self):
        self.balance *= 1.0 + rate
```

- Here is **Account** in use:

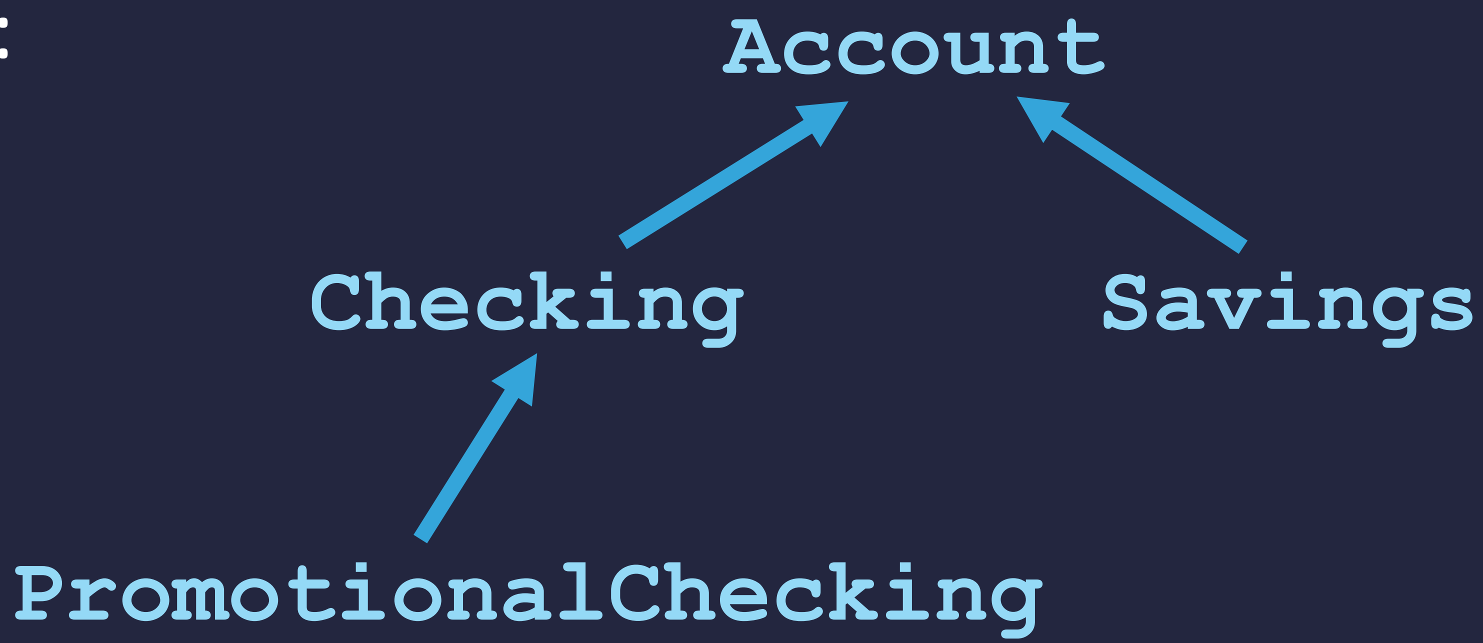
```
>>> a = Account(100)
>>> a.balance
100
>>> a.rate
0.02
>>> a.deposit(50)
>>> Account.rate
0.02
>>> Account.deposit(a,10)
```

COMMENTARY

- You can also call an instance's method using its class name:
  - Syntax: *class-name* . *instance-method-name* (*receiver* , *arguments*)
- You pass the receiver as the first argument to that "function."

NEXT TIME

- We will build ***hierarchies*** of different classes that relate to each other:



- We make ***subclasses*** that ***inherit*** the attributes of their "***superclasses***"
  - A `Checking` account has all the info and operations of an `Account`.
  - But it might also have "specialized" features and behavior.
    - I.e. it might have additional attributes.
  - It might ***override*** the behavior it inherits.