DATA ABSTRACTIONS & OBJECT-ORIENTATION

LECTURE 06-2

JIM FIX, REED COLLEGE CSCI 121

QUIZ ON MONDAY

- **In-Class Quiz:** Monday, March 9th
  - *closed note, closed computer, hand-written*
  - **Topic covered:**
    - ➡ lists

MONDAY BEFORE BREAK

- **In-Class Midterm Exam:** Monday, March 16th
  - *closed note, closed computer, hand-written*
  - about 5 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`
    - ➡ (basic) list and dictionary use
  - I will post a practice exam next week.
  - I will post a practice exam solution on Friday, March 13th.

JUST BEFORE BREAK

- **Project 2 due:** Friday, March 20th
  - ***ciphers***

TODAY

- **Today:**
  - inventing your own data structures and data types
  - object-oriented programming in Python
- **Reading:** on Python object-orientation
  - ➡ PP Ch 3
  - ➡ TP Ch 12, 14-16
  - ➡ CP Ch 2.5-2.8

- We'll end lecture today early for a survey.

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 remove_all(x, someList):
"""Modifies list, removing elements equal to the value."""
```

...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: strings, lists, dictionaries, "tuples" (e.g. pairs).
- 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.
  - 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:

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

- We can invent rational number multiplication:

```
def rationalProduct(r, s):
    newNumerator = numerator(r) * numerator(s)
    newDenominator = denominator(r) * denominator(s)
    return createRational(newNumerator, newDenominator)
```

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):
    newNumerator = numerator(r) * numerator(s)
    newDenominator = denominator(r) * denominator(s)
    return createRational(newNumerator, newDenominator)
```

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):
    newNumerator = numerator(r) * numerator(s)
    newDenominator = denominator(r) * denominator(s)
    return createRational(newNumerator, newDenominator)
```

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 addition:

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

EXAMPLE: RATIONAL NUMBER OBJECT

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

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

- We can invent rational number addition:

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

EXAMPLE: RATIONAL NUMBER OBJECT

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

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

- We can invent rational number addition:

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

EXAMPLE: RATIONAL NUMBER OBJECT

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

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

- We can invent rational number addition:

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

EXAMPLE: RATIONAL NUMBER OBJECT

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

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

- We can invent rational number addition:

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

EXAMPLE: RATIONAL NUMBER OBJECT

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

$$\frac{a}{b} == \frac{c}{d} \quad \textit{whenever} \quad a*d == c*b$$

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

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

- No changes below and elsewhere because we used the getters and the constructor!

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

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: RATIONAL NUMBER CLASS

Here is our rational number data structure as an object class

```
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:

```
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:

```
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

```
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:

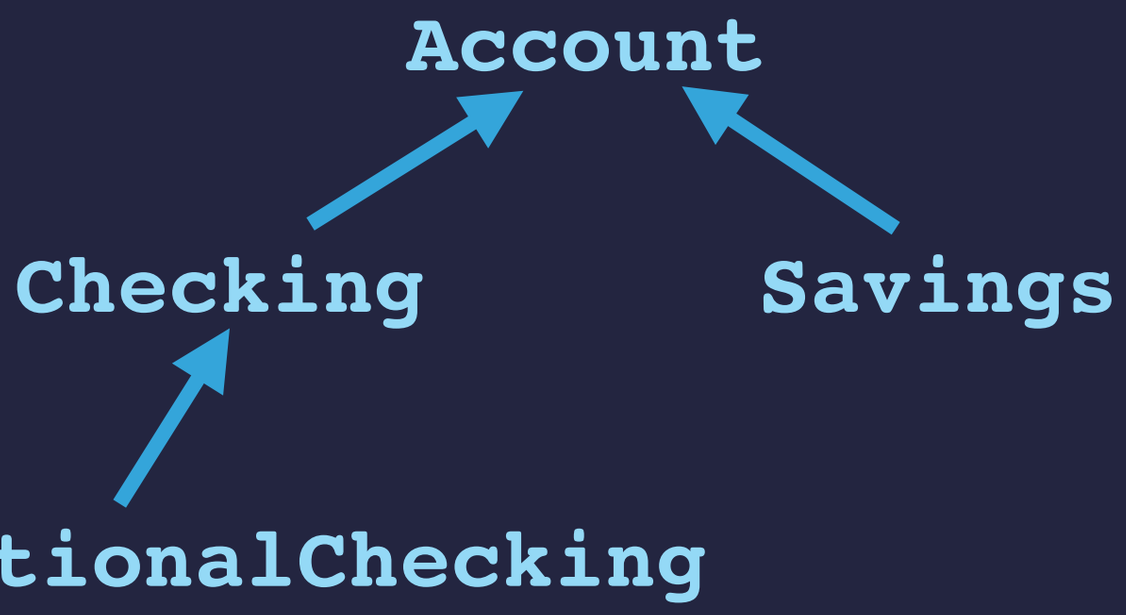
```
>>> Rational(27, 33)
9 / 11
```

OBJECT TAKEAWAYS

- New object types are defined with `class`.
- Within the class you define these things:
  - `__init__`
  - other methods
- Method parameters are `self` followed by the others.
- Object dot notation:
  - Methods are called using `receiver.method(...)`
  - Object attributes are accessed by `receiver.variable`
  - We use `self.` notation inside a method to access these things too.
- New instances are built with `class-name(...)`

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.
    - It might have additional attributes.
    - It might ***override*** the behavior it inherits.

SURVEY TIME!