Recreation

Given that
\[
\log(1 + x) = x - \frac{1}{2}x^2 + \frac{1}{3}x^3 - \ldots
\]

why is it not the case that

\[
\log 2 = 1 - \frac{1}{2} + \frac{1}{3} - \frac{1}{4} + \frac{1}{5} - \frac{1}{6} + \frac{1}{7} - \frac{1}{8} + \frac{1}{9} - \ldots
\]

\[
= (1 + \frac{1}{3} + \frac{1}{5} + \frac{1}{7} + \frac{1}{9} + \ldots) - (\frac{1}{2} + \frac{1}{4} + \frac{1}{6} + \frac{1}{8} + \ldots)
\]

\[
= (1 + \frac{1}{3} + \frac{1}{5} + \frac{1}{7} + \frac{1}{9} + \ldots) + (\frac{1}{2} + \frac{1}{4} + \frac{1}{6} + \frac{1}{8} + \ldots)
\]

\[
- 2(\frac{1}{2} + \frac{1}{4} + \frac{1}{6} + \frac{1}{8} + \ldots)
\]

\[
= (1 + \frac{1}{2} + \frac{1}{3} + \frac{1}{4} + \ldots) - (1 + \frac{1}{2} + \frac{1}{3} + \frac{1}{4} + \ldots)
\]

\[
= 0?
\]
Basic Idea.

- **Function-based programs** are organized primarily around the functions (methods, etc.) that do things. Data structures (objects) are considered separate.

- **Object-based programs** are organized around the *types of objects* that are used to represent data; methods are grouped by type of object.

- Simple banking-system example:

  - **Function-based**
    - account
    - deposit
    - withdraw
    - account
    - account
    - withdraw
    - withdraw
    - account
    - deposit
    - account

  - **Object-based**
    - Account
      - balance: 1420
      - deposit
      - withdraw
      - Exported methods
      - Exported field
Philosophy

- Idea (from 1970s and before): An abstract data type is
  - a set of possible values (a domain), plus
  - a set of operations on those values (or their containers).

- In IntList, for example, the domain was a set of pairs: \((\text{head}, \text{tail})\), where head is an int and tail is a pointer to an IntList.

- The IntList operations consisted only of assigning to and accessing the two fields (head and tail).

- In general, we prefer a purely procedural interface, where the functions (methods) do everything—no outside access to the internal representation (i.e., instance variables).

- That way, implementor of a class and its methods has complete control over behavior of instances.

- In Java, the preferred way to write the “operations of a type” is as instance methods.
You Saw It All (Maybe) in CS61A: The Account Class

class Account:
    balance = 0
    def __init__(self, balance0):
        self.balance = balance0

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

    def withdraw(self, amount):
        if self.balance < amount:
            raise ValueError("Insufficient funds")
        else:
            self.balance -= amount
            return self.balance

myAccount = Account(1000)
print(myAccount.balance)
myAccount.deposit(100)
myAccount.withdraw(500)
You Also Would Have Seen It All in CS61AS

```
(define-class (account balance0)
  (instance-vars (balance 0))
  (initialize
    (set! balance balance0)))

(method (deposit amount)
  (set! balance (+ balance amount))
  balance)

(method (withdraw amount)
  (if (< balance amount)
      (error "Insufficient funds")
      (begin
        (set! balance (- balance amount))
        balance)))

(define my-account
  (instantiate account 1000))
(ask my-account 'balance)
(ask my-account 'deposit 100)
(ask my-account 'withdraw 500)
```

```
public class Account {
  public int balance;
  public Account(int balance0) {
    balance = balance0;
  }
  public int deposit(int amount) {
    balance += amount; return balance;
  }
  public int withdraw(int amount) {
    if (balance < amount)
      throw new IllegalStateException("Insufficient funds");
    else balance -= amount;
    return balance;
  }
}

Account myAccount = new Account(1000);
myAccount.balance
myAccount.deposit(100);
myAccount.withdraw(500);
```
The Pieces

- **Class declaration** defines a *new type of object*, i.e., new type of structured container.

- **Instance variables** such as `balance` are the simple containers within these objects (*fields* or *components*).

- **Instance methods**, such as `deposit` and `withdraw` are like ordinary (static) methods that take an invisible extra parameter (called `this`).

- The `new` operator creates (*instantiates*) new objects, and initializes them using constructors.

- **Constructors** such as the method-like declaration of `Account` are special methods that are used only to initialize new instances. They take their arguments from the `new` expression.

- **Method selection** picks methods to call. For example,

  ```java
  myAccount.deposit(100)
  ```

  tells us to call the method named `deposit` that is defined for the object pointed to by `myAccount`. 
Getter Methods

• Slight problem with Java version of Account: anyone can assign to the balance field

• This reduces the control that the implementor of Account has over possible values of the balance.

• Solution: allow public access only through methods:

```java
public class Account {
    private int _balance;
    ...
    public int balance() { return _balance; }
    ...
}
```

• Now Account._balance = 1000000 is an error outside Account.

• (I use the convention of putting ‘_’ at the start of private instance variables to distinguish them from local variables and non-private variables. Could actually use balance for both the method and the variable, but please don’t.)
Class Variables and Methods

- Suppose we want to keep track of the bank’s total funds.
- This number is not associated with any particular Account, but is common to all—it is class-wide. In Java, “class-wide” \(\equiv\) static.

```java
public class Account {
    ...
    private static int _funds = 0;
    public int deposit(int amount) {
        _balance += amount;
        _funds += amount; // or this._funds or Account._funds
        return _balance;
    }
    public static int funds() {
        return _funds; // or Account._funds
    }
    ...
    // Also change withdraw.
}
```

- From outside, can refer to either `Account.funds()` or to `myAccount.funds()` (same thing).
Instance Methods

- Instance method such as

```java
int deposit(int amount) {
    _balance += amount;
    _funds += amount;
    return balance;
}
```

behaves sort of like a static method with hidden argument:

```java
static int deposit(final Account this, int amount) {
    this._balance += amount;
    _funds += amount;
    return this._balance;
}
```

- NOTE: Just explanatory: Not real Java (not allowed to declare 'this'). (final is real Java; means “can’t change once initialized.”)
Calling Instance Methods

/** (Fictional) equivalent of deposit instance method. */
static int deposit(final Account this, int amount) {
    this._balance += amount;
    _funds += amount;
    return this._balance;
}

• Likewise, the instance-method call `myAccount.deposit(100)` is like a call on this fictional static method:

        Account.deposit(myAccount, 100);

• Compare this with Python, where the extra parameter is not fictional (and one can choose the name—usually `self`.)

• Inside a real instance method, as a convenient abbreviation, one can leave off the leading 'this.' on field access or method call if not ambiguous. (Unlike Python)
'Instance' and 'Static' Don't Mix

• Since real static methods don’t have the invisible this parameter, makes no sense to refer directly to instance variables in them:

```java
public static int badBalance(Account A) {
    int x = A._balance;  // This is OK
    // (A tells us whose balance)

    return _balance;     // WRONG! NONSENSE!
}
```

• Reference to _balance here equivalent to this._balance,

• But this is meaningless (whose balance?)

• However, it makes perfect sense to access a static (class-wide) field or method in an instance method or constructor, as happened with _funds in the deposit method.

• There’s only one of each static field, so don’t need to have a ‘this’ to get it. Can just name the class (or use no qualification inside the class, as we’ve been doing).
Constructors

• To completely control objects of some class, you must be able to set their initial contents.

• A constructor is a kind of special instance method that is called by the new operator right after it creates a new object, as if

\[
\begin{align*}
L &= \text{new IntList}(1, \text{null}) \\
&\quad \implies \begin{cases}
\text{tmp = pointer to 0} \\
\text{tmp.IntList}(1, \text{null}); \\
L = \text{tmp};
\end{cases}
\end{align*}
\]
Multiple Constructors and Default Constructors

- All classes have constructors. In the absence of any explicit constructor, get the **default constructor**, as if you had written:

  ```java
  public class Foo {
      public Foo() {
      }
  }
  ```

- Multiple **overloaded** constructors are possible, and they can use each other (although the syntax is odd):

  ```java
  public class IntList {
      public IntList(int head, IntList tail) {
          this.head = head; this.tail = tail;
      }

      public IntList(int head) {
          this(head, null); // Calls first constructor.
      }

      ...  
  }
  ```
Constructors and Instance Variables

- Instance variables initializations are moved inside constructors that don’t start with `this(...)`.

```java
class Foo {
    int x = 5;

    Foo(int y) {
        DoStuff(y);
    }

    Foo() {
        this(42);
    }
}
```

```java
class Foo {
    int x;

    Foo(int y) {
        x = 5;
        DoStuff(y);
    }

    Foo() {
        this(42); // Assigns to x
    }
}
```
## Summary: Java vs. Python

<table>
<thead>
<tr>
<th>Java</th>
<th>Python</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>class Foo: ...</td>
</tr>
<tr>
<td></td>
<td>x = ...</td>
</tr>
<tr>
<td></td>
<td>def <strong>init</strong>(self, ...)::</td>
</tr>
<tr>
<td></td>
<td>...</td>
</tr>
<tr>
<td></td>
<td>def f(self, ...)::</td>
</tr>
<tr>
<td></td>
<td>...</td>
</tr>
<tr>
<td></td>
<td>y = 21</td>
</tr>
<tr>
<td></td>
<td># Referred to as Foo.y</td>
</tr>
<tr>
<td></td>
<td>@staticmethod</td>
</tr>
<tr>
<td></td>
<td>def g(...):</td>
</tr>
<tr>
<td></td>
<td>...</td>
</tr>
<tr>
<td></td>
<td>aFoo.f(...)</td>
</tr>
<tr>
<td></td>
<td>aFoo.x</td>
</tr>
<tr>
<td>new Foo(...)</td>
<td>aFoo.f(...)</td>
</tr>
<tr>
<td>this</td>
<td>aFoo.x</td>
</tr>
<tr>
<td></td>
<td>Foo(...)</td>
</tr>
<tr>
<td></td>
<td>self # (typically)</td>
</tr>
</tbody>
</table>