Object-Oriented Programming

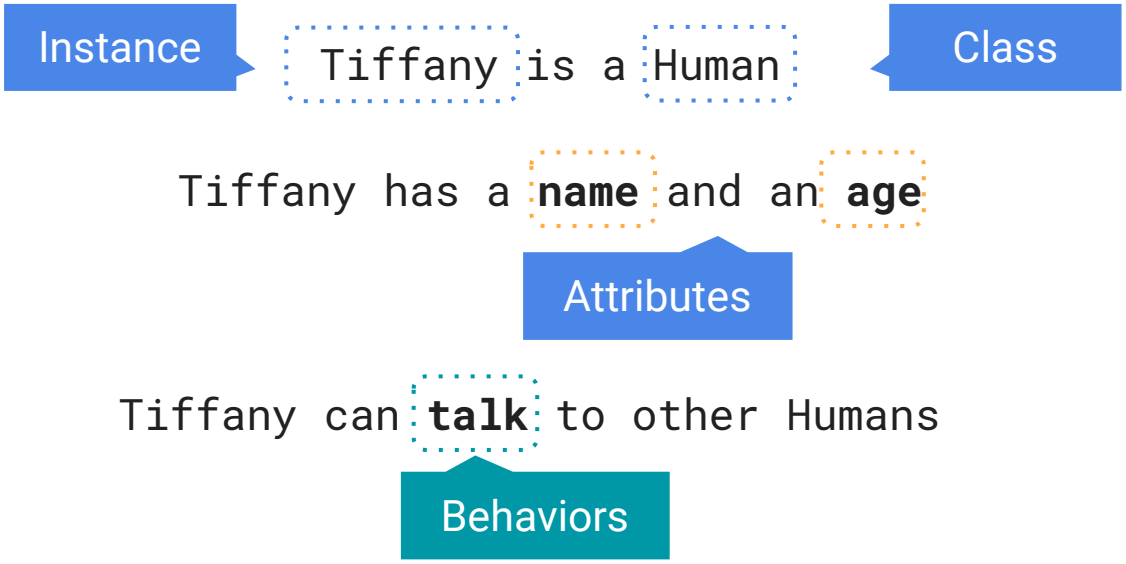
Why OOP?

Object-Oriented Programming

- A (hopefully) more intuitive way of representing data
- A common method for organizing programs
- Formally split "global state" and "local state" for every object

Classes

- Every object is an **instance** of a **class**
- A class is a **type** or category of objects
- A class serves as a blueprint for its instances



Example: Bank Accounts

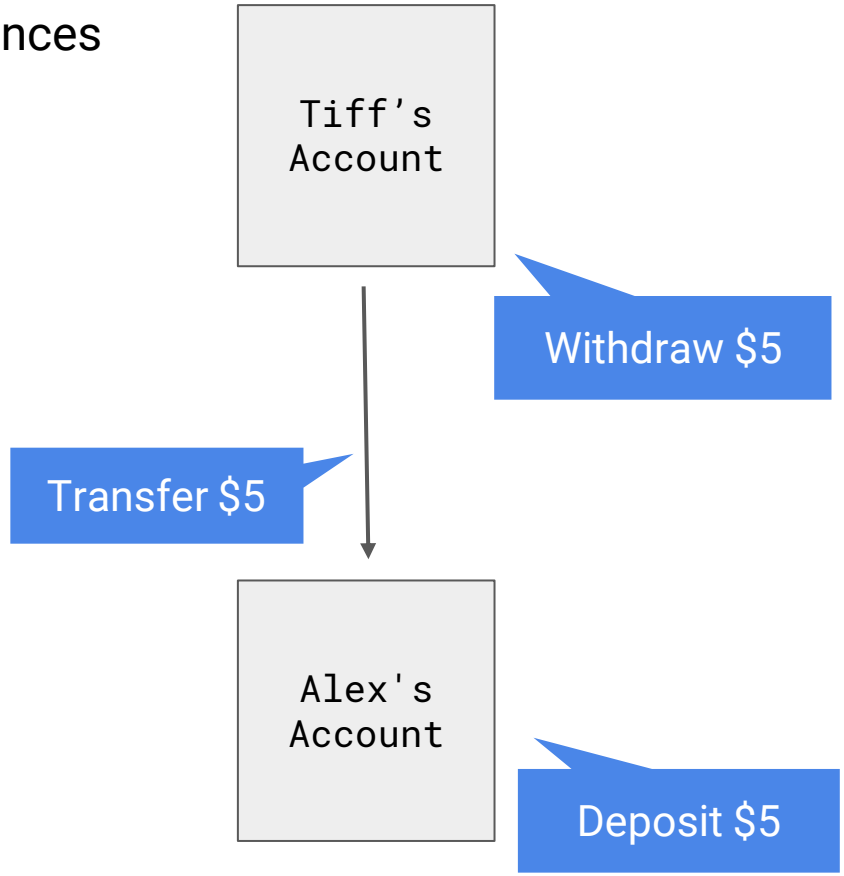
A class serves as a blueprint for its instances

Idea: All bank accounts have a balance and an account holder;
the Account class should define those values for each newly created **instance**.

Idea: All bank accounts should have "withdraw" and "deposit" **behaviors** that all work in the **same way**.

Better Idea: All bank accounts **share** a "withdraw" and "deposit" method.

All accounts might also share other characteristics like maximum withdrawal, or loan limits.



The Class Statement

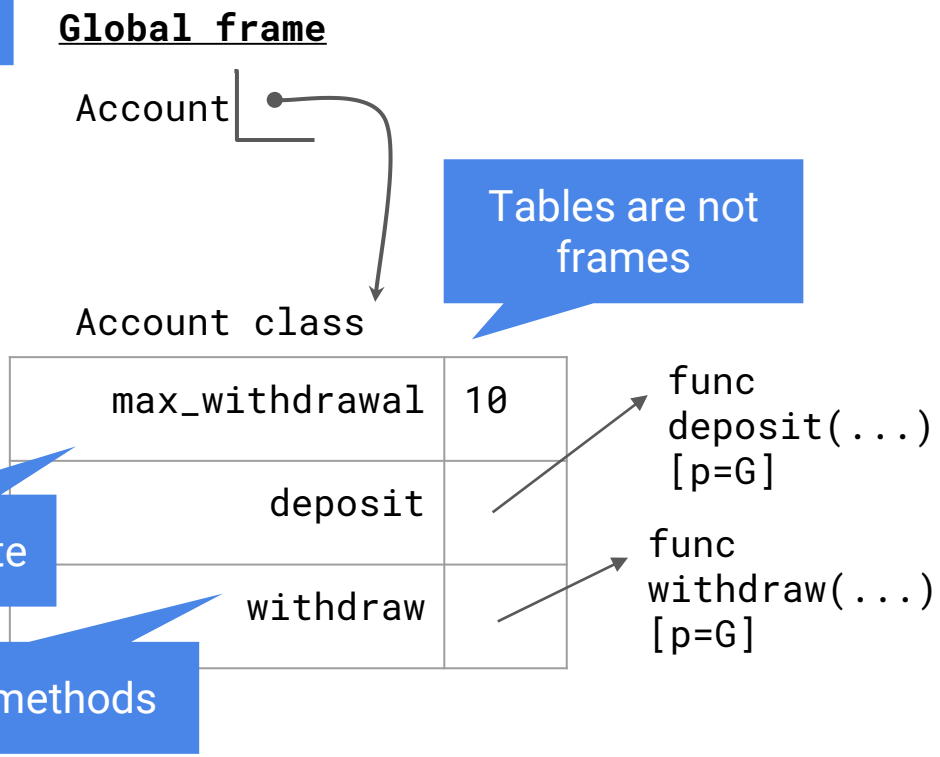
Class Statements defines the blueprint

Assignment & def statements in the Class Statement create class attributes.

```
class <name>:
    <suite>
```

Lots of … don't worry!

```
class Account:
    max_withdrawal = 10
    def deposit(...):
        ...
    def withdraw(...):
        ...
    ...
```



The Class Statement

Before we start implementing our methods, we need to talk about how to create Accounts.

Idea: All bank accounts have a balance and an account holder. These are not shared across Accounts.

When a class is called:

1. A new instance of that class is created.
2. The __init__ method of a class is called with the new object as its first argument (named self), along with additional arguments provided in the call expression.

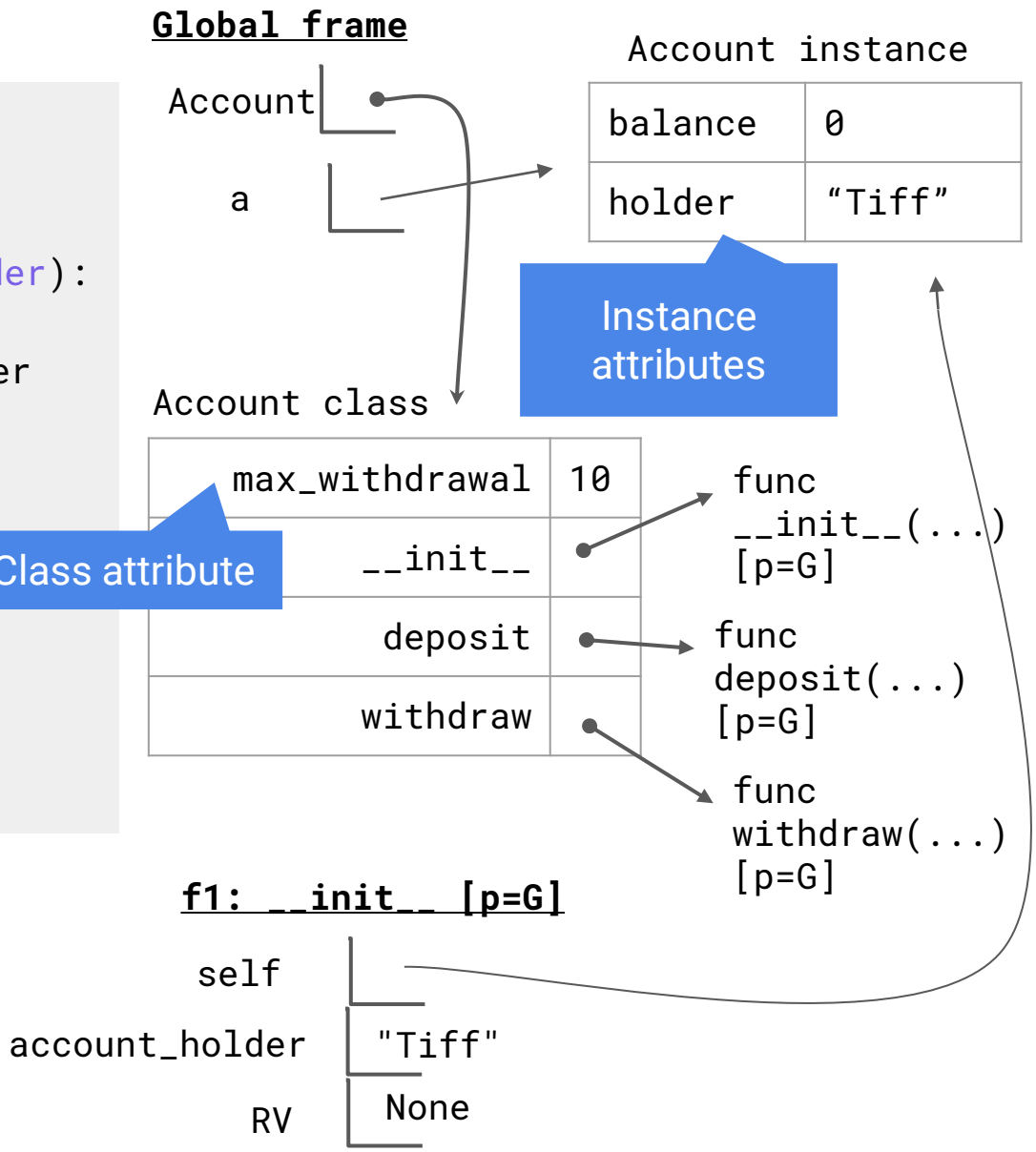
__init__ is called the constructor

```
class Account:
    ...
    def __init__(self, account_holder):
        self.balance = 0
        self.holder = account_holder
    ...
```

The Class Statement

```
class Account:
    max_withdrawal = 10

    def __init__(self, account_holder):
        self.balance = 0
        self.holder = account_holder

    def deposit(self, amount):
        ...

    def withdraw(self, amount):
        ...

>>> a = Account('Tiff')
```

Python Tutor link



Terminology: Attributes, Functions, and Methods

All objects have **attributes**, which are name-value pairs

Classes are objects too, so they have attributes

Instance attribute: attribute of an instance

Class attribute: attribute of the class of an instance

Object Identity

Python Tutor link

- Every object that is an instance of a user-defined class has a unique identity
- Identity operators “is” and “is not” test if two expressions evaluate to the same object (the arrows point to the same place)
- Binding an object to a new name using assignment does not create a new object

Dot Expressions

Python Tutor link

You can access class or instance attributes with dot notation.

```
<expression>.<name>
```

The <expression> can be any valid Python expression that evaluates to a **class** or **instance**. The <name> must be an **attribute** or a **method**.

```
tiff_account.max_withdrawal
```

To evaluate a dot expression:

1. Evaluate the <expression> to the left of the dot, which yields the object of the dot expression
2. <name> is matched against the instance attributes of that object; if an attribute with that name exists, its value is returned
3. If not, <name> is looked up in the class, which yields a class attribute or a method
4. The corresponding attribute is returned or corresponding method is called

Methods and Functions

Methods are functions defined in the suite of a class statement.

However methods that are accessed through an instance will be bound methods. **Bound methods** couple together a function and the object on which that method will be invoked. This means that when we invoke bound methods, the instance is automatically passed in as the first argument.

```
>>> a = Account("Tiffany")
>>> Account.deposit
<function>
>>> a.deposit
<bound method>
```

Invoking Methods

We can call class methods in two ways: as a bound method and as a function.

Invoking class methods as a bound method:

- Bound methods are accessed through the instance and implicitly pass the instance object in as the first argument of the method.
- <instance>.<method_name>(<arguments>)
- a.deposit(5)

Invoking class methods as functions:

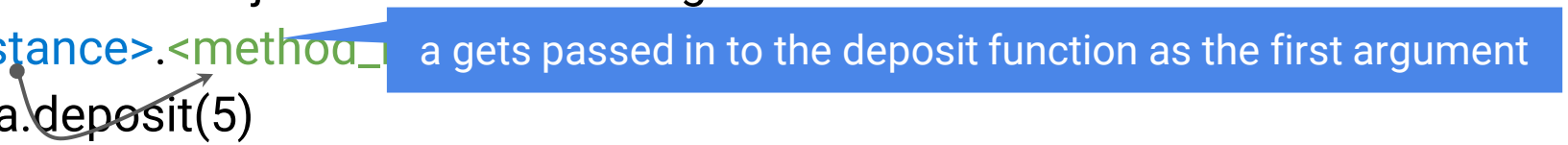
- We can use the class name to directly call a method. These follow our typical function call rules and nothing is implicitly passed in.
- <class_name>.<method_name>(<instance>, <arguments>)
- Account.deposit(a, 5)

Invoking Methods

We can call class methods in two ways: as a bound method and as a function.

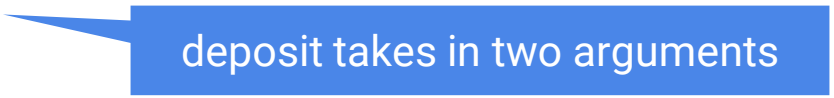
Invoking class methods as a bound method:

- Bound methods are accessed through the instance and implicitly pass the instance object in as the first argument of the method.
- <instance>.<method_
- a.deposit(5)



Invoking class methods as functions:

- We can use the class name to directly call a method. These follow our typical function call rules and nothing is implicitly passed in.
- <class_name>.<method_name>(<instance>, <arguments>)
- Account.deposit(a, 5)



Implementing the Account Class

Python Tutor link

```
class Account:
        max_withdrawal = 10
        def __init__(self, account_holder):
                """Creates an instance of the Account class"""
                self.balance = 0
                self.holder = account_holder

        def deposit(self, amount):
        """Deposits amount to the account."""
        self.balance = self.balance + amount
                return self.balance

        def withdraw(self, amount):
        """Subtracts amount from the account."""
                if amount > self.max_withdrawal or amount >
self.balance:
                        return "Can't withdraw this amount"
                self.balance = self.balance - amount
                return self.balance
```

Accessing Attributes

There are built-in functions that can help us access attributes.

Using getattr, we can look up an attribute using a string instead.

- getattr(<expression>, <attribute_name (string)>)
- getattr(a, 'balance') is the same as a.balance
- getattr(Account, 'balance') is the same as Account.balance

Using hasattr, we can check if an attribute exists.

- hasattr(<expression>, <attribute_name (string)>)
- hasattr(a, 'balance') returns True
- hasattr(Account, 'balance') returns False

Assigning Attributes

Python Tutor link

We saw this before, but let's formalize the rules for assigning/re-assigning instance and class attributes.

<expression>.<name> = <value>

Change attributes for the **object of that dot expression.**

If the expression evaluates to an instance: then assignment sets an instance attribute, even if it exists in the class.

If the expression evaluates to a class: then assignment sets a class attribute

Summary

- Object-oriented programming is another way (paradigm) to organize and reason about programs
- The Python class statement allows us to create user-defined data types that can be used just like built-in data types
  - Class attributes are variables shared across instances
  - Instance attributes are unique to each instance
  - Two ways to invoke methods, implicitly and explicitly.
  - Tomorrow, we'll discover how to define the relationships between different classes
- In lab, you'll use OOP to make a trading card game!