Object-Oriented Programming

A method for organizing programs
- Data abstraction
- Bundling together information and related behavior

A metaphor for computation using distributed state
- Each object has its own local state
- Each object also knows how to manage its own local state, based on method calls
- Method calls are messages passed between objects
- Several objects may all be instances of a common type
- Different types may relate to each other

Specialized syntax & vocabulary to support this metaphor

John’s Account

Steven’s Account

Apply for a loan!

Classes

A class serves as a template for its instances

Idea: All bank accounts have a balance and an account holder; the Account class should add those attributes to 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 method and a deposit method

---

The Class Statement

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

A class statement creates a new class and binds that class to `<name>` in the first frame of the current environment.

Assignment & def statements in `<suite>` create attributes of the class (not names in frames)

---

Object Construction

Idea: All bank accounts have a balance and an account holder; the Account class should add those attributes to each of its instances

```python
>>> a = Account('Jim')
>>> a.balance
0
>>> a.holder
'Jim'
```

When a class is called:
1. A new instance of that class is created:
   ```python
   >>> a = Account('Jim')
   >>> a.balance
   0
   >>> a.holder
   'Jim'
   ```
2. The `__init__` method of the class is called with the new object as its first argument (named `self`), along with any additional arguments provided in the call expression
   ```python
def __init__(self, account_holder):
   self.balance = 0
   self.holder = account_holder
```

---

Announcements
Object Identity

Every object that is an instance of a user-defined class has a unique identity:

```python
>>> a = Account('Jim')
>>> b = Account('Jack')
>>> a.balance
d
>>> b.holder
'Jack'
```

Identity operators "is" and "is not" test if two expressions evaluate to the same object:

```python
>>> a is a
True
>>> a is not b
True
```

Binding an object to a new name using assignment does not create a new object:

```python
>>> c = a
>>> c is a
True
```

Methods

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

```python
class Account:
    def __init__(self, account_holder):
        self.balance = 0
        self.holder = account_holder
    def deposit(self, amount):
        self.balance = self.balance + amount
        return self.balance
    def withdraw(self, amount):
        if amount > self.balance:
            return 'Insufficient funds'
        self.balance = self.balance - amount
        return self.balance
```

These `def` statements create function objects as always, but their names are bound as attributes of the class:

```python
self should always be bound to an instance of the Account class
```

Dot Expressions

Objects receive messages via dot notation:

Dot notation accesses attributes of the instance or its class:

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

The `<expression>` can be any valid Python expression.

The `<name>` must be a simple name.

Evaluates to the value of the attribute looked up by `<name>` in the object that is the value of the `<expression>`:

```python
tom_account.deposit(10)
```

Methods and Functions

Python distinguishes between:

- Functions, which we have been creating since the beginning of the course, and
- Bound methods, which couple together a function and the object on which that method will be invoked:

```python
Object + Function = Bound Method
```

```python
>>> type(Account.deposit)
<class 'function'>
```

```python
>>> type(tom_account.deposit)
<class 'method'>
```

```python
>>> Account.deposit(tom_account, 100)  # Function: all arguments within parentheses
101
```

```python
>>> tom_account.deposit(100)  # Method: one object before the dot and other arguments within parentheses
201
```

Accessing Attributes

Using `getattr`, we can look up an attribute using a string:

```python
>>> getattr(tom_account, 'balance')
10
```

```python
>>> hasattr(tom_account, 'deposit')
True
```

`getattr` and dot expressions look up a name in the same way:

- One of its instance attributes, or
- One of the attributes of its class.
Looking Up Attributes by Name

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 value
4. That value is returned unless it is a function, in which case a bound method is returned instead

Class Attributes

Class attributes are "shared" across all instances of a class because they are attributes of the class, not the instance

```python
class Account:
    interest = 0.02  # A class attribute
    def __init__(self, account_holder):
        self.balance = 0
        self.holder = account_holder
    # Additional methods would be defined here

>>> tom_account = Account('Tom')
>>> jim_account = Account('Jim')
>>> tom_account.interest
0.02
>>> jim_account.interest
0.02
```

The interest attribute is not part of the instance; it's part of the class!