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

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

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

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

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

```python
>>> a.deposit(15)
5
>>> a.balance
15
```

Better idea: All bank accounts share a withdraw method and a deposit method

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

    def withdraw(self, amount):
        if self.balance >= amount:
            self.balance -= amount
            return True
        else:
            return False

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

a = Account('John')
>>> a.withdraw(10)
'Insufficient funds'
```

Class Statements

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)

```python
class Clown:
    nose = 'big and red'
    def dance(self):
        return 'No thanks'

>>> Clown.nose
'big and red'
>>> Clown.dance()
'No thanks'
```

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('John')
>>> a.holder
'John'
>>> a.balance
0
```

When a class is called:

1. A new instance of that class is created:
   ```python
   class Account:
       def __init__(self, account_holder):
           self.balance = 0
           self.holder = account_holder
   account = Account('John')
       >>> account.holder
   'John'
   >>> account.balance
   0
   ```

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

a = Account('John')
>>> a.holder
'John'
>>> a.balance
0
```
Object Identity

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

```python
g>>> a = Account('John')
```

```python
g>>> b = Account('Jack')
```

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

```python
g>>> a is a
True
```

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

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

```python
g>>> c = a
```

```python
g>>> c is a
True
```

Every call to Account creates a new Account instance. There is only one Account class.

```python
g>>> a.balance
0
```

```python
g>>> b.holder
'Jack'
```

Methods

Methods are functions defined in the suite of a class statement

```python
gclass Account:

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

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

def withdraw(self, amount):
    if amount > self.balance:
        return 'Insufficient funds'
    self.balance -= amount
    return self.balance
```

Methods are always bound to an instance of the Account class

```python
g>>> Account.deposit(tom_account, 100)
101
```

```python
g>>> Account.deposit(1000)
2001
```

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
g>>> type(Account.deposit)
geclass 'function'
```

```python
g>>> type(tom_account.deposit)
geclass 'method'
```

```python
g>>> Account.deposit(tom_account, 1001) -- Function: all arguments within parentheses
1001
```

```python
g>>> tom_account.deposit(1004) -- Method: one object before the dot and other arguments within parentheses
2005
```
Looking Up Attributes by Name

<expression> . <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.

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')

The interest attribute is _not_ part of the instance; it's part of the class!

0.02
0.02