Lecture 16: Object-Oriented Programming II

Marvin Zhang
07/19/2016

Announcements

Survey Responses (Thanks!)
Highlights from the survey:
• Many students reevaluated their starting ability
• Lab checkoffs: most think they’re worthwhile
  Others think it’s stressful or it’s too easy
  They should be easy and not stressful
  It’s not unreasonable to ask you to come to lab once a week
• Homework 3 and Quiz 4 were so hard!
  Homework assignments are graded on effort
  We will do coding quizzes a little differently

More Survey Responses
• Remove the auto-grader delay on projects!
  Nope, it’s for your own good
• Have two midterms instead of quizzes!
  Nope, it’s for your own good
• Brian and I will slow down the demos in lecture
  When we can
• Brian’s office hours are great
• Some administrative things are out of our control
• 1/6 students came to the potluck, 5/6 want another one

Roadmap

Introduction
Functions
Data
Mutability
Objects
Interpretation
Paradigms
Applications

Inheritance

• Powerful idea in Object-Oriented Programming
• Way of relating similar classes together
• Common use: a specialized class inherits from a more general class

\[
\text{class } <\text{new class}>(<\text{base class}>): \\
... 
\]

• The new class shares attributes with the base class, and overrides certain attributes
• Implementing the new class is now as simple as specifying how it’s different from the base class
Inheritance Example (demo)

```python
class Pokemon:
    """A Pokemon."""
...

classElectricType(Pokemon):
    """An electric-type Pokemon."""
...

• Pokémon have:
  • a name
  • a trainer
  • a level
  • an amount of HP (life)
  • a basic attack: tackle

• Pokémon can:
  • say their name
  • attack other Pokémon
```

Designing for Inheritance

- Don’t repeat yourself! Use existing implementations
- Reuse overridden attributes by accessing them through the base class
- Look up attributes on instances if possible

```python
class ElectricType(Pokemon):
    basic_attack = 'thunder shock'
    prob = 0.1

def attack(self, other):
    if random() < self.prob and type(other) != ElectricType:
        other.paralyzed = True
        print(other.name, 'is paralyzed!')
```

Multiple Inheritance

- In Python, a class can inherit from multiple base classes
- This exists in many but not all object-oriented languages
- This is a tricky and often dangerous subject, so proceed carefully!

```python
class FlyingType(Pokemon):
    basic_attack = 'peck'
    damage = 35

def fly(self, location):
    print(self.trainer, 'flew to', location)
```

Multiple Inheritance Example

- Zapdos is a legendary bird Pokémon
  - Zapdos’ attack, thunder, does a lot of damage
  - Zapdos can paralyze when attacking
  - Zapdos can fly
  - Zapdos can’t say its own name

```python
class Zapdos(ElectricType, FlyingType):
    basic_attack = 'thunder'
    damage = 120

def speak(self):
    print('EEEEEEEE')
```

Multiple Inheritance Example (demo)

```
Pokemon

ElectricType

FlyingType

Zapdos
```

More on Design

- This example has been shortened for lecture purposes, and could have better design if done properly
- We should create a class for every species of Pokémon
  - Consequently, we should not create instances of the Pokemon, ElectricType, or FlyingType classes
- We should create classes for different types of attacks, with damage and special effect attributes
  - The relationship between classes that reference each other (e.g., Pokemon and Tackle) is called composition
- Good design is a bigger topic in future classes
Complicated Inheritance

To show how complicated inheritance can be, let's look at an analogy through biological inheritance.

Moral of the story:
Inheritance, especially multiple inheritance, is complicated and weird. Use it carefully!

Exceptions

Raising and handling exceptions

Exceptions

• In Python, exceptions alter the control flow of programs for exceptional circumstances, e.g., errors
• Exceptions cause the program to halt immediately and print a stack trace if not handled
• There are many different types of exceptions

Raising Exceptions

• We can cause an exception in our program by using the `raise` statement:

```python
raise <expression>
```

• `<expression>` must evaluate to either an exception class or instance
• Otherwise, an error occurs...

• An exception class is any class that inherits from the built-in `BaseException` class
• Almost all built-in exceptions inherit from the `Exception` class, which inherits from `BaseException`

Handling Exceptions

• The `try` statement allows us to handle exceptions and continue running our program

```python
try:
    <try suite>
except <exception type> as <name>:
    <except suite>
```

Execution Rule for `try` Statements:

1. Execute the `<try suite>`.
2. If an exception of `<exception type>` is raised, switch to executing the `<except suite>` with `<name>` bound to the exception that was raised.

User-defined Exceptions

• It's possible to create our own exception types by defining a new class that inherits from `Exception` or a subclass of `Exception`
• These user-defined exceptions can then be used in `raise` statements, just like any other exception
• There aren't many reasons to create new exceptions, since Python already has so many

```python
class MySpecialException(Exception):
    def __init__(self, msg):
        # special magic
        raise MySpecialException('so special')
```
Computer science often involves communication between different components.

- Communication between the program and the user, between two different programs, between two objects in the same program, etc.
- This can get very complicated, since these components often have different behaviors and specifications.

Interfaces specify rules for communication between these components, and this is a form of abstraction!

- E.g., to use an object, we don’t need to know how it is implemented if we know the interface for the object.
- There are several common interfaces that are widely used in Python, called protocols.

In Python, object interfaces are usually implemented through magic methods.

- Special methods surrounded by double underscores (e.g., `__init__`) that add “magic” to your classes.

We will look at two examples of these interfaces:

- The arithmetic interface
- The (mutable) container protocol

For more information, see: http://www.rafekettler.com/magicmethods.html

Python has many built-in container types: lists, tuples, ranges, dictionaries, etc.

Python also has a protocol for defining custom container classes.

- Defining custom containers is as easy as implementing the `__len__`, `__getitem__`, and `__contains__` magic methods.
- `__len__` is called by `len`, `__getitem__` is used in indexing, and `__contains__` is used in membership.
- To create a mutable container, we can also implement the `__setitem__` and `__delitem__` methods.

Inheritance allows us to implement relationships between classes and simplify our programs.

Interfaces allow for standardized interaction between different components by defining rules for communication.

- Implementing interfaces in Python can allow our custom classes to behave like built-in classes.

Both are tools for abstraction, and learning them well is one of the keys to becoming a great object-oriented programmer.