More Object Oriented Programming (OOP)

- Help documentation
- Copying object state
- Reading/writing objects to a file
- A closer look at attributes
- Implementing magic methods
- Inheritance

Help documentation

You can provide "help" documentation at the start of the class and the start of each method

- Define a help string """like this"""
- For an example, see accounting2.py

You can then get help for the class or methods via the help() function in the Python shell

Copying object state

When you assign one object reference to another:

- It just copies the object reference
- So both references refer to the same actual object

If you want to create a copy of an object:

- Call the copy() function, defined in the copy module

Example: See demoCopying.py

Reading/writing objects to a file

A common requirement is to read/write objects to a file

There are various ways to do this in Python:

- As JSON
- As XML
- As CSV

You can also write your own custom code See accounting3.py, clientcodeReadWriteObjects.py

A closer look at attributes

- Determining an object's attributes
- Adding and removing object attributes
- Built-in class attributes

Determining an object's attributes

Python provides several global functions that allow you to manage attributes on an object

```
from accounting import BankAccount

acc1 = BankAccount("Fred")

setattr(acc1, "bonus", 2000)

if hasattr(acc1, "bonus"):
    print("acc1.bonus is %d" % acc1.bonus)

delattr(acc1, "bonus")
```

Adding and removing object attributes

You can also add and remove attributes on an object directly, as follows:

```
from accounting import BankAccount

acc1 = BankAccount("Fred")

# Add an attribute to an object.
acc1.flag = "Whao watch this guy"
print("acc1.flag is %s" % acc1.flag)

# Remove an attribute from an object.
del acc1.flag
```

Built-in class attributes

Every class provides metadata via the following built-in attributes

- You can also get metadata about an object too

```
from accounting import BankAccount

print("BankAccount.__doc__:",    BankAccount.__doc__)
print("BankAccount.__name__:",   BankAccount.__name__)
print("BankAccount.__module__:", BankAccount.__module__)
print("BankAccount.__bases__:",  BankAccount.__bases__)
print("BankAccount.__dict__:",   BankAccount.__dict__)

acc1 = BankAccount("Ola")
print("acc1.__dict__:", acc1.__dict__)
```

Implementing magic methods

- Overview
- Implementing constructors and destructors
- Implementing stringify methods
- Implementing operator methods

Overview

There are various "special" methods you can implement in your Python classes

- These methods allow your class objects to take advantage of standard Python idioms

It’s good practice to implement these methods where relevant

- Python programmers will recognise these methods immediately
- Makes your classes easier to maintain

Implementing constructors and destructors

Constructor

- `__init__(self, otherArgs)`

Destructor

- `__del__(self)`

Example

```
class Person:

    def __init__(self, name, age):
        self.name = name
        self.age = age
        print("In __init__() for %s and %d" % (self.name, self.age))

    def __del__(self):
        print("In __del__() for %s and %d" % (self.name, self.age))

p1 = Person("Bill", 23)
p2 = Person("Ben", 25)
```

Implementing stringify methods

Return a machine-readable representation of an object

- `__repr__(self)`

Return a human-readable representation of an object

- `__str__(self)`

Example

```
class Person:

    def __repr__(self):
        return "{0} instance, name: {1}, age: {2}".format( \
                                                self.__class__.__name__,\
                                                self.name, self.age)
    def __str__(self):
        return "{0} is {1}.".format(self.name, self.age)
    …

print(repr(p1))
print(str(p2))
```

Implementing operator methods

There are a large number of method that represent standard operators, including:

- `__eq__(self, other)`
- `__ne__(self, other)` Etc...

Example

```
class Person:

    def __eq__(self, other):
        return self.age == other.age

    def __ne__(self, other):
        return self.age ≠ other.age

    …

…

print("p1 == p2 gives %s" % (p1 == p2))
print("p1 ≠ p2 gives %s" % (p1 ≠ p2))
```

Inheritance

- Overview of inheritance
- Superclasses and subclasses
- Sample hierarchy
- Defining a subclass
- Adding new members
- Defining constructors
- Overriding methods
- Multiple inheritance

Overview of inheritance

Inheritance is a very important part of object-oriented development

- Allows you to define a new class based on an existing class
- You just specify how the new class differs from the existing class

Terminology:

- For the "existing class": Base class, superclass, parent class
- For the "new class": Derived class, subclass, child class

Potential benefits of inheritance:

- Improved OO model
- Faster development
- Smaller code base

Superclasses and subclasses

The subclass inherits everything from the superclass (except constructors)

- You can define additional variables and methods
- You can override existing methods from the superclass
- You typically have to define constructors too
- Note: You can’t cherry pick or "blank off" superclass members

Sample hierarchy

We'll see how to implement the following simple hierarchy:

BankAccount <- SavingsAccount

Note:

- BankAccount defines common state and behaviour that is relevant for all kinds of account
- SavingsAccount "is a kind of" BankAccount that earns interest

We might define additional subclasses in the future…

- E.g. CurrentAccount, a kind of BankAccount that has cheques

Defining a subclass

To define a subclass, use the following syntax

- Note that a Python class can inherit from multiple superclasses
- We'll discuss multiple inheritance later in this chapter

```
class Subclass(Superclass1, Superclass2, …) :

    # Additional attributes and methods …

    # Constructor(s) …

    # Overrides for superclass methods, if necessary …
```

Example:

```
class SavingsAccount(BankAccount):
    …
    …
    …
```

Adding new members

The subclass inherits everything from the superclass

- (Except for constructors)
- The subclass can define additional members if it needs to …

Example:

```
class SavingsAccount(BankAccount):

    __DEFAULT_INTEREST_RATE = 1.5


    def earnInterest(self):
        self.balance *= (1 + self.interestRate)
        return self.balance

    …
```

Defining constructors

A subclass doesn't inherit the constructor from superclass

- So, define a constructor in the subclass, to initialize subclass state

The subclass constructor should invoke the superclass constructor, to initialize superclass data

- Call `super().__init__(params)`

Example:

```
class SavingsAccount(BankAccount):

    def __init__(self, accountHolder="Anonymous", interestRate=None):

        super().__init__(accountHolder)

        if interestRate is None:
            self.interestRate = SavingsAccount.__DEFAULT_INTEREST_RATE
        else:
            self.interestRate = interestRate
    …
```

Overriding methods

The subclass can override superclass instance methods

- To provide a different (or supplementary) implementation
- No obligation ☺

An override can call the original superclass method, to leverage existing functionality

- Call super().methodName(params)

Example:

```
class SavingsAccount(BankAccount):

    def withdraw(self, amount):
        if amount > self.balance:
            print("You can't go overdrawn in a savings account!")
        else:
            super().withdraw(amount)
        return self.balance
    …
```

Multiple inheritance (1/2)

Python supports multiple inheritance

```
class Logger:
    def log(self, msg):
        print(msg)

class Beeper:
    def beep(self, duration):
        winsound.Beep(2500, duration)
```

```
class Alerter(Logger, Beeper):
    def doShortAlert(self, msg):
        super().log(msg)
        super().beep(250)

    def doMediumAlert(self, msg):
        super().log(msg)
        super().beep(1000)

    def doLongAlert(self, msg):
        super().log(msg)
        super().beep(2500)
```

Multiple inheritance (2/2)

Client code can access public members in the subclass or in any superclass

```
alerter = Alerter()

alerter.log("Wakey wakey!")
for i in range(30):
    alerter.beep(50)

msg = input("Enter an alert message: ")
alerter.doShortAlert(msg)

msg = input("Enter another alert message: ")
alerter.doMediumAlert(msg)

msg = input("And another: ")
alerter.doLongAlert(msg)
```

Any questions?