SciComp with Py

OOP: Part 2

Vladimir Kulyukin
Department of Computer Science
Utah State University

Outline

- Inheritance
- Persistent Objects
- Magic Methods & Protocols
- Exceptions

Quote

I don't predict the demise of object-oriented programming, by the way. Though I don't think it has much to offer good programmers, except in certain specialized domains, it is irresistible to large organizations. Object-oriented programming offers a sustainable way to write spaghetti code. It lets you accrete programs as a series of patches. Large organizations always tend to develop software this way, and I expect this to be as true in a hundred years as it is today.

Paul Graham, The Hundred Year Language

Inheritance

Inheritance

- Inheritance is an OOP principle that supports code reuse and abstraction
- If a class **C** defines a set of attributes (data and methods), the programmer can derive a subclass of **C** without having to reimplement **C**'s attributes
- In OOP, subclasses are said to ***inherit*** attributes of superclasses

Old-Style vs. New-Style Classes in Py2

- In Py2, there are two types of classes: old-style and new-style
- If you use multiple inheritance (more on it later in the lecture), you should always go with new-style classes, because it is the only way to call specific superclass constructors
- The way to ensure the use of new-style classes is to explicitly inherit from object, e.g. class **C(object)**
- In Py3, all classes are new-style by default

Attribute Method hello() is Inherited

```
class A:
    def hello(self):
        print('Hi, I am A!')


class B(A):
    pass
```

```
>>> a = A()
>>> b = B()
>>> a.hello()
Hi, I am A!
>>> b.hello()
Hi, I am A!
```

hello() is inherited by B

source in py2/inheritance_01.py and py3/inheritance_01.py

Overriding hello() in class B

```
class A:
    def hello(self):
        print('Hi, I am A!')


class B(A):
    def hello(self):
        print('Hi, I am B!')
```

```
>>> a = A()
>>> b = B()
>>> a.hello()
Hi, I am A!
>>> b.hello()
Hi, I am B!
```

hello() is overridden in B

source in py2/inheritance_02.py and py3/inheritance_02.py

Overriding Constructor __init__()

__init__() is overridden in SongBird

```
class Bird:
    def __init__(self):
        self.hungry = True
    def eat(self):
        if self.hungry:
            print('Aaaah...')
            self.hungry = False
        else:
            print('No, thanks!')

class SongBird(Bird):
    def __init__(self):
        self.sound = 'Squawk!'
    def sing(self):
        print(self.sound)
```

Common Problem: Attribute is Not Constructed

Py2 source in py2/birds.py, py2/birds2.py, and py2/birds3.py

__init__() is overridden in SongBird
eat() is inherited but the attribute hungry is not constructed in SonBird

```
>>> b = Bird()
>>> b.eat()
Aaaah...
>>> b.eat()
No, thanks!
>>> sb = SongBird()
>>> sb.sing()
Squawk!
>>> sb.eat()
AttributeError: SongBird instance has no attribute 'hungry'
```

Fixing the Problem with Explicit Call to Superclass __init__()

Explicit call to Bird.__init__()

Py

```
class Bird:
    def __init__(self):
        self.hungry = True
    def eat(self):
        if self.hungry:
            print('Aaaah...')
            self.hungry = False
        else:
            print('No, thanks!')
class SongBird(Bird):
    def __init__(self):
        Bird.__init__(self)
        self.sound = 'Squawk!'
    def sing(self):
        print(self.sound)
```

Output

```
>>> sb = SongBird()
>>> sb.sing()
Squawk!
>>> sb.eat()
Aaaah...
```

Fixing the Problem with SUPER in Py2

Using new-style classes

Call to superclass constructor

Py

```
__metaclass__ = type
class Bird:
    def __init__(self):
        self.hungry = True
    def eat(self):
        if self.hungry:
            print('Aaaah...')
            self.hungry = False
        else:
            print('No, thanks!')
class SongBird(Bird):
    def __init__(self):
        super(SongBird, self).__init__()
        self.sound = 'Squawk!'
    def sing(self):
        print(self.sound)
```

Output

```
>>> sb = SongBird()
>>> sb.sing()
Squawk!
>>> sb.eat()
Aaaah...
```

Fixing the Problem with SUPER in Py3

Py3 source in py3/birds.py, py3/birds2.py, an py3/birds3.py

Call to superclass constructor

Py

```
class Bird:
    def __init__(self):
        self.hungry = True
    def eat(self):
        if self.hungry:
            print('Aaaah...')
            self.hungry = False
        else:
            print('No, thanks!')
class SongBird(Bird):
    def __init__(self):
        super(SongBird, self).__init__()
        self.sound = 'Squawk!'
    def sing(self):
        print(self.sound)
```

Output

```
>>> sb = SongBird()
>>> sb.sing()
Squawk!
>>> sb.eat()
Aaaah...
```

Multiple Inheritance

```
class A:
    def __init__(self):
        self.x = “A's x”
        print('A()')
    def f(self):
        print(“A's f”)

class B:
    def __init__(self):
        self.x = “B's x”
        print('B()')
    def f(self):
        print(“B's f”)
```

List of superclasses: A and B are CAB's superclasses

```
class CAB(A, B):
    def __init__(self):
        print('CAB()')
```

B and A are CBA's superclasses

```
class CBA(B, A):
    def __init__(self):
        print('CBA()')
```

Multiple Inheritance

```
>>> cab = CAB()
CAB()
>>> cba = CBA()
CBA()
```

```
>>> isinstance(cab, CAB)
True
>>> isinstance(cab, A)
True
>>> isinstance(cab, B)
True
>>> isinstance(cba, CBA)
True
>>> isinstance(cba, A)
True
>>> isinstance(cba, B)
True
```

Py2 source in py2/multinh.py and py2/multinh2.py
Py3 source in py3/multinh.py and py3/multinh2.py

Question

```
>>> cab = CAB()
CAB()
>>> cba = CBA()
CBA()
>>> cab.f()
???
>>> cba.f()
???
```

Answer

This is because CAB inherites first from A and then from B

This is because CBA inherites first from B and then from A

```
>>> cab = CAB()
CAB()
>>> cba = CBA()
CBA()
>>> cab.f()
A's f
>>> cba.f()
B's f
```

Question

```
>>> cab.x
???
>>> cba.x
???
```

Answer

This is because CAB does not call the constructor's of its super class

```
>>> cab.x
AttributeError: CAB instance has no attribute 'x'
>>> cba.x
AttributeError: CBA instance has no attribute 'x'
```

This is because CBA does not call the constructor's of its super class

Fixing AttributeError with New-Style Classes

A is a new-style class because it inherits from object

```
class A(object):
    def __init__(self):
        self.x = 'A\'s x'
        print('A()')
    def f(self):
        print('A\'s f')

class B(object):
    def __init__(self):
        self.x = 'B\'s x'
        print('B()')
    def f(self):
        print('B\'s f')
```

```
class CAB(A, B):
    def __init__(self):
        super(CAB, self).__init__()
        print('CAB()')

class CBA(B, A):
    def __init__(self):
        super(CBA, self).__init__()
        print('CBA()')
```

Multiple Inheritance

```
>>> cab = CAB()
A()
CAB()
>>> cba = CBA()
B()
CBA()
>>> cab.x
"A's x"
>>> cba.x
"B's x"
```

Persistent Objects

Persistent Objects

- It is easy to save numbers and strings for later use in a file.
- Things get complicated when you need to save lists, tuples, and dictionaries
- Things get even more complicated when you need to save arbitrary objects for later use

Pickle

- Persistent objects can be saved to file and later read back from that file
- The module pickle provides tools to convert objects to strings and save them to files.
- Converting objects to strings is called serialization
- Converting strings back to objects is called deserialization

Example

pickle_trial.py

```
import pickle
import sys

with open(sys.argv[1], 'wb') as outfile:
    lst = ['a', 'b', [1, 2, 3], 'c', 'd']
    pickle.dump(lst, outfile)
    print('Dumped ' + str(lst))

with open(sys.argv[1], 'rb') as infile:
    lst = None
    lst = pickle.load(infile)
    print('Loaded ' + str(lst))
```

output

```
$ python pickle_trial.py dump.pck
Dumped ['a', 'b', [1, 2, 3], 'c', 'd']
Loaded ['a', 'b', [1, 2, 3], 'c', 'd']
```

Py2 source in py2/pickle_trial.py
Py3 source in py3/pickle_trial.py

Example with User Objects

pickle_objects.py

```
import pickle
import sys
from Date import Date
date_list = [Date(m=1, d=1, y=2017), Date(m=2, d=1, y=2017),
                Date(m=2, d=10, y=2017)]
with open(sys.argv[1], 'wb') as outfile:
    pickle.dump(date_list, outfile)
    sys.stdout.write('Dumped dates\n')
with open(sys.argv[1], 'rb') as infile:
    loaded_dates = None
    loaded_dates = pickle.load(infile)
    sys.stdout.write('Loaded dates:\n')
    for date in loaded_dates:
        sys.stdout.write(date.toMDYString() + '\n')
```

output

```
$ python pickle_objects.py dates.pck

Dumped dates

Loaded dates:

1/1/2017

2/1/2017

2/10/2017
```

Py2 source in py2/pickle_objects.py
Py3 source in py3/pickle_objects.py

cPickle

- Python has the cPickle module
- cPickle provides the same support for object serialization (pickling) and object deserialization (unpickling)
- Python documentation states that “cPickle can be up to 1000 times faster than pickle because the former is implemented in C”

Example

cpickle_trial.py

```
import cPickle as pickle
import sys

with open(sys.argv[1], 'wb') as outfile:
    lst = ['a', 'b', [1, 2, 3], 'c', 'd']
    pickle.dump(lst, outfile)
    print('Dumped ' + str(lst))

with open(sys.argv[1], 'rb') as infile:
    lst = None
    lst = pickle.load(infile)
    print('Loaded ' + str(lst))
```

output

```
$ python pickle_trial.py dump.pck

Dumped ['a', 'b', [1, 2, 3], 'c', 'd']

Loaded ['a', 'b', [1, 2, 3], 'c', 'd']
```

Py2 source in py2/cpickle_trial.py
Py3 source in py3/cpickle_trial.py

Magic Methods & Protocols

Magic Methods

- Magic methods in Python start and end with double underscores (dunders) __
- Magic methods are called by Python under specific circumstances:
  - **__init__()** - called when objects are constructed
  - **__eq__()** - called when objects are compared

Magic Method __del__()

When a class defines **__del__()**, it is called when instances of that class are destroyed by the garbage collector

Py

```
class A3:
    def __init__(self, x=10):
       self.my_list = [i for i in xrange(x)]
    def __del__(self):
        print 'Destroying an A3 object'
        del self.my_list
```

Output

```
>>> a = A3(x=200)

>>> del a

Destroying an A3 object
```

Magic Methods & Protocols

Protocol

- A protocol is a set of rules for governing a programmatic behavior
- A protocol states which methods must be implemented and what those methods should do
- In Python, a protocol does not require objects to belong to a certain class; in this sense, protocols are similar to Java/C++ interfaces

Sequence and Mapping Protocol

- **__len__(self)**: called when len is called
- **__getitem__(self, key)**: called when **[ ]** is called
- **__setitem__(self, key, value)**: called when **[ ]** is called with assignment
- **__delitem__(self, key)**: called del is called
- If you want your class to behave like sequences or dictionaries, you must implement these magic methods

Exceptions

What's an Exception?

- An exception is an class that represents an erroneous condition
- When a running program encounters an error, it raises an exception
- If an exception is handled properly, the program continues
- If an exception is not handled, the program terminates

Raising Exceptions

```
>>> raise Exception
Exception
>>> raise Exception('this is my exception')
Exception: this is my exception
>>> e = Exception('this is my exception')
>>> raise e
Exception: this is my exception
```

Catching Exceptions

If anything goes wrong inside try try block, control goes into except block; every try must have a matching except

```
>>> for i in [0, 1, 2, 3, 4, 0, 45]:
        try:
            print(100.0/i)
        except Exception:
            print('Something\'s wrong')
Something's wrong
100.0
50.0
33.333333333325.0
Something's wrong
2.2222222
```

Built-in Exceptions

- Exception – base class
- AttributeError – failure of attribute reference or assignment
- IOError – open a nonexistent file
- KeyError – nonexistence index on a sequence
- NameError – a variable is not defined
- SyntaxError – code is syntactically wrong
- TypeError – function is applied to a argument of a wrong type
- ValueError – correct type but inappropriate value
- ZeroDivisionError – division by zero

Customizing Exceptions

```
## This is a custom exception class.
class MyException(Exception):
    def __init__(self, error):
        self.__errorMessage = error

## raise an instance of MyException
raise MyException('This is the message')
```

Handling User Input

```
# Exceptions come in handy when handing user input:
try:
    x = input('Enter the first number: ')
    y = input('Enter the second number: ')
    print x/y
except ZeroDivisionError:
    print 'The second number cannot be zero!'
```

Handling User Input

```
# else clause runs only if no exception is raised.
while True:
    try:
        x = input('Enter the first number: ')
        y = input('Enter the second number: ')
        value = x/y
        print x, '/', y, '=', value
    except Exception, e:
        print 'Invalid input:', e
        print 'Please try again'
    else:
        break
```

Catching Multiple Exceptions

```
# You can catch more than one exception in
# the same try block
try:
    x = input('Enter the first number: ')
    y = input('Enter the second number: ')
    print x/y
except ZeroDivisionError:
    print 'The second number cannot be zero!'
except TypeError:
    print 'Both inputs must be numbers.'
```

Binding Exceptions to Variables

```
#You can bind exception objects to variables for more advanced
#debugging.
    try:
        x = input('Enter the first number: ')
        y = input('Enter the second number: ')
        value = x/y
        print x, '/', y, '=', value
    except Exception, e:
        print 'Invalid input:', e
        print 'Please try again'
```

Finally Clause

```
# finally clause always runs.
try:
    x = input('Enter the first number: ')
    y = input('Enter the second number: ')
    print x, '/', y, '=', x/y
except Exception, e:
    print 'Invalid input:', e
else:
    print 'All went well'
finally:
    print 'Cleaning up'
    del x, y
```

Exception Propagation

- Unhandled (unexcepted) exceptions propagate (bubble) upward until they are handled
- An exception raised in a function propagates to the place where the function is called
- If a raised exception reaches the global scope, the running program halts