Inf-1400 - Object Oriented Programming

Chapter 2 - 3

John Markus Bjørndalen

Department of Computer Science
University of Tromsø
Norway

2017-01-27

Topics

Chapter 2:

- Python modules
- Python docstrings

Chapter 3:

- basic inheritance
- extending built-ins
- overriding and super
- multiple inheritance
- diamond problem
- method resolution order
- polymorphism

Python Modules

Simplest version: a Python file is a module that can be imported and used by other Python files.

Creating a module:

```
# simplemod.py
somevar = 42
def somefunc(v):
    print(v)
```

Using a module:

```
# Using simplemod.py as a module
import simplemod
print('Value of simplemod var is', simplemod.somevar)
simplemod.somefunc(9000)
```

Python Modules

Main uses:

- organize code
- provide libraries

Python Modules

Entire packages/libraries can be imported as modules. Requires `__init__.py` in the package directory.

```
somelib/
   __init__.py      <-- contains variable 'somevar'
   blarb.py
   sublib/
     __init__.py   <-- contains variable 'foo'
     other.py
```

Can be used:

```python
import somelib
import somelib.sublib

print(somelib.somevar)
print(somelib.sublib.foo)
```

Python Modules

Python has several libraries included:

`https://docs.python.org/3/py-modindex.html`

Also third party packages (such as PyGame). One resource is

`https://pypi.python.org/pypi`

Python Docstrings

Document your code. Code is “write once read many” (not quite but almost)

Most people reading the code will not know (or have forgotten) what, why and how. That includes yourself in the future.

What does this code do?
`http://www.ioccc.org/2013/birken/birken.c`

Solution:
`http://www.ioccc.org/2013/birken/hint.html`

From **`http://www.ioccc.org/`**

Python Docstrings

This type of documentation is useful

```python
def foo(bar):
    # This function does nothing
    pass
```

However, you need to read the source to view this documentation.

Python Docstrings

Java: javadoc based on specially formatted comments.
`http://www.oracle.com/technetwork/articles/java/index-137868.html`

`http://docs.visualillusionsent.net/CanaryLib/1.2.0-RC1/index.html`

Python Docstrings

Python: make the comments available as a property of functions, objects and modules: a docstring.

```
#!/usr/bin/env python
"""This is a file/module docstring. It has to be a string and be
the first object in the file."""

def foo(bar):
    """This is a function docstring. It must be the first object
    in the function."""
    pass
```

Documentation available at run-time and at the interactive prompt (ex: the **`help`**`()` function).

Python Docstrings

Taking one step further: showing example code and output in the docstring helps the programmer understand how to use the library.

It can also be used for automated testing.

`https://docs.python.org/3/library/doctest.html`

Basic Inheritance

Sub-class/Child class: derives from (or extends) parent.

```
class Parent():
    def somemethod(self):
        print("oink")


class Child(Parent):
    def othermethod(self):
        """"extends parent with a new method"""
```

Classes provide type identity which gives a contract about interface.
Children will have the same interface, but can choose to extend it.

Extending built-ins

Built in types can also be extended[1].

```
class ContactList(list):
    def search(self, name):
        '''Return all contacts that contain the search value
        in their name.'''
        matching_contacts = []
        for contact in self:
            if name in contact.name:
                matching_contacts.append(contact)
        return matching_contacts

class Contact:
    all_contacts = ContactList()
    def __init__(self, name, email):
        self.name = name
        self.email = email
        self.all_contacts.append(self)
```

[1] Code from the OOP book. See handout code for extended examples.

Overriding and super

Overriding the behaviour/method of a parent. The interface is the same, but what the method does is changed.

```
class Contact:
    all_contacts = ContactList()    # class level  / shared
    def __init__(self, name, email):
        self.name = name
        self.email = email
        self.all_contacts.append(self)
    def foo(self):
        print('Foo')

class Friend(Contact):
    def __init__(self, name, email, phone):
        self.name = name
        self.email = email
        self.phone = phone
    def foo(self):
        print('Bazooka')
```

Multiple Inheritance

Multiple Inheritance: a class specifies several classes that it inherits from.

```
class A():
    def athing(self):
         pass
class B():
    def bthing(self):
        pass
class M(A,B):
    def mthing(self):
        pass

m = M()
```

Now `M` inherits from both `A` and `B`. We can call

```
m.athing()
m.bthing()
m.mthing()
```

Multiple Inheritance

Mix-ins: classes that are not intended to be used on their own. Extends or modifies other classes.

Example use in standard library:
`https://docs.python.org/3/library/socketserver.html`

Mix-ins specifies first to override method(s) in the normal base/parent class.

Multiple Inheritance

Multiple inheritance can be complicated.

Some languages (like Java) do not allow multiple inheritance to force programmers to avoid potential complexity issues.

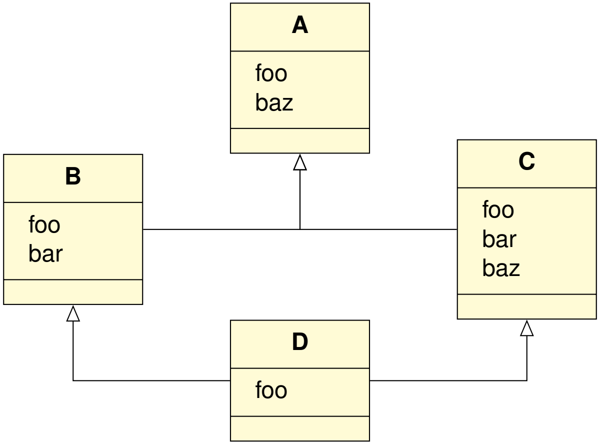
Java provides *Interfaces* to let a class inherit the interface of multiple types. A Java class can inherit from one class and several interfaces. The child class must implement each of the methods in the interfaces as there is no code to inherit.

Diamond problem

```
class A:
    def foo(self):
        print("This is A")
    def baz(self):
        print("baz A")

class B(A):
    def foo(self):
        print("This is B")
        super().foo()
    def bar(self):
        print("bar B")

class C(A):
    def foo(self):
        print("This is C")
        super().foo()
    def bar(self):
        print("bar C")
    def baz(self):
        print("baz C")
```

```
class D(B,C):
    def foo(self):
        print("This is D")
        super().foo()

b = B()
c = C()
d = D()

# which order are the methods called?
print("Calling b.foo():")
b.foo()
print("Calling c.foo():")
c.foo()
print("Calling d.foo():")
d.foo()
print("Calling d.bar():")
d.bar()
print("Calling d.baz():")
d.baz()
```

Diamond problem



Method resolution order

The defined search order through the objects and classes.

Think of this as a searching problem: Python searches objects and classes to find the first matching attribute or method.

Can be modified by modifying the `__mro__` method (beyond the scope of this lecture).

`super``()` finds the next class in the search list, not the parent of the current object/class. For a D object, the order will be the d object, then the classes in the following order: D, B, C, A

Polymorphism

One basic definition: when the type of the object determines the behaviour when calling a method or interface.

```
class AudioFile:
    def __init__(self, filename):
        if not filename.endswith(self.ext):
            raise Exception("Invalid file format")
        self.filename = filename

class MP3File(AudioFile):
    ext = "mp3"
    def play(self):
        print("playing {} as mp3".format(self.filename))

class WavFile(AudioFile):
    ext = "wav"
    def play(self):
        print("playing {} as wav".format(self.filename))

someaudiofile.play()
```

We have seen this with the initial PyGame examples.

Polymorphism

Python duck typing goes beyond this by only caring about the interface and not the type.

```python
print(3 + 5)
print(3.1415926 + 5.8)
print('Hello ' + 'there')

class Tomato:
    def __init__(self, desc="Tomato"):
        self.desc = desc
    def __repr__(self):
        return self.desc
    def __add__(self, other):
        print("Smashing {} and {}".format(self.desc, other))
        return Tomato("Ketchup")

t1 = Tomato()
t2 = Tomato()
print(t1 + t2)  # What does this print?
```