Object-oriented programming

Computer & Information Sciences

W. H. Bell

Overview

- Object-oriented programming.
- Classes.
- Inheritance.
- Encapsulation.
- Applications.
- Summary.

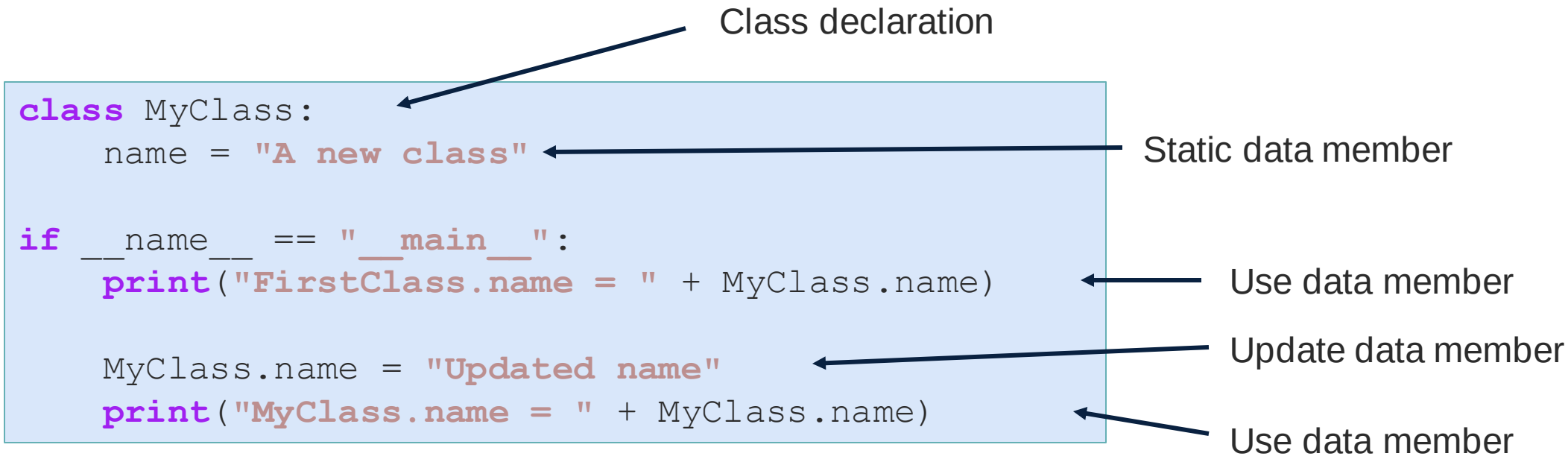
Object-oriented programming

- Many languages support object-oriented (OO) programming.
- C++, C#, Java, Python, Ruby and others.
- The implementation of OO programming differs between languages.
- Need to learn how to implement OO programming in each language.
- Compilers may optimise more or less of the OO syntax.
- E.g. best implementation in Python and Java is not the same.

Classes

- An object is an instance of a class.
- A class definition may include data members and functions.
- Data members and functions can be static or associated with an instance of an object.
- Static members can be called using the class name or object.
- Non-static members can only be called from the object.

Python Class: static data member



Output

```
MyClass.name = A new class
MyClass.name = Updated name
```

Python Class: non-static data member

```
class MyClass:
    def __init__(self):
        self.name = "A new class"

if __name__ == "__main__":
    m1 = MyClass()
    m2 = MyClass()
    m1.name = "Updated name"
    print("m1.name = " + m1.name)
    print("m2.name = " + m2.name)
```

Constructor

Data member

Instantiating objects

Update data member

Print values

Output

```
m1.name = Updated name
m2.name = A new class
```

Python Class: static function

Class declaration

```
class MyClass:
    def staticFunction():
        return True

if __name__ == "__main__":
    returnValue = MyClass.staticFunction()
    print("staticFunction() returns " + str(returnValue))
```

Static member function

Use member function

Output

```
staticFunction() returns True
```

Python Class: non-static function

```
class MyClass:
    def __init__(self):
        self.name = "MyClass"

    def fullName(self):
        return self.name + " is an example"

if __name__ == "__main__":
    m = MyClass()
    m.name = "New name"
    print("fullName = " + m.fullName())
```

Constructor

Member function

Instantiate object

Use member function

Output

```
fullName = New name is an example
```

Inheritance

- Classes can inherit from another class.
- Data members and functions become part of derived class.
- A derived class can directly use member functions and data if they are public or protected.
- Private member functions and data cannot be directly accessed by derived classes.

Inheritance

```
class Coordinates:
    def __init__(self):
        self.latitude = 0.
        self.longitude = 0.

class Position(Coordinates):
    def __init__(self):
        self.elevation = 0.

if __name__ == "__main__":
    m = Coordinates()
    m.latitude = 13.0
    m.longitude = -10.0
    p = Position()
    p.latitude = 55.860916
    p.longitude = -4.251433
    p.elevation = 16
```

Public, Protected and Private

- Functions and data members within a class can be public, protected and private.
- **Public** – function or data member is accessible from outside the class.
- **Protected** - function or data member is accessible from a derived class, but not from outside the derived or base class.
- **Private** – function or data member is not accessible from outside the class.

Public, Protected and Private

```
class MyClass:
    def __init__(self):
        self.name = "MyClass"
        self._protectedName = "Only derived know"
        self.__privateName = "Only this class knows"

    def publicFunction(self):
        return "This a public function"

    def _protectedFunction(self):
        return "This is a protected function"

    def __privateFunction(self):
        return "This is a private function"
```

"_" (single underscore) => protected
"__" (double underscore) => private

Encapsulation

- Data and member functions together within one class.
- Restricted direct access to class components.
- Used to hide the values or state of data.
- Require accessor functions to get or set private or protected values.

Accessors

Avoid using Accessors with Python. Use public data members instead.

```
class MyClass:
    def __init__(self):
        self.__name = "MyClass"

    def setName(self, name):
        self.__name = name

    def getName(self):
        return self.__name

if __name__ == "__main__":
    m = MyClass()
    m.setName("New name")
    print(m.getName())
```

Accessor functions

Accessors can be used with C++, C# and Java with reduced overhead.

Accessors

- Accessors are member functions that are used to set or get data member values.
- Python programmers tend to avoid using accessors.
- Use public data members instead and directly access them.
- There is a processing cost overhead for accessors.
- The processing cost overhead is reduced slightly in compiled languages.

Polymorphism

- Ability for function to have different forms.
- Can call a function in the same way for different classes.
- Normally implemented using inheritance.
- May inherit from an interface or base class, where the function is defined as virtual.
- May have to explicitly require polyporphism, depending on language.

Polymorphism in Python

- Python automatically provides polymorphism.
- Polymorphism is available using inheritance and using function definitions.
- Therefore, inheritance may not be necessary.
- No interfaces, but abstract base classes are possible.
- Beyond the scope of this course.

Polymorphism using functions

```
class Algorithm:
    def result(self):
        return "Result from Algorithm"

class Calculator:
    def result(self):
        return "Result from Calculator"

if __name__ == "__main__":
    algorithms = [ Algorithm(), Calculator() ]
    for algorithm in algorithms:
        print(algorithm.result())
```

Output

```
Result from Algorithm
Result from Calculator
```

Object-oriented programming dangers

- When a program is first written, only a subset of requirements may be known.
- Difficult to encapsulate all data and functionality.
- Incorrect encapsulation may result in large changes to address other requirements.
- Incorrect use of inheritance may increase effort needed to re-write software.
- State split between objects.
- Obscure data flow or copy data around problem needlessly.

Normal class applications: Data like

- Read from or written to central data store.
- Generic container, may include member functions to return transient data.
- Example, map coordinates, where transient data could be an angle.

Normal class applications: Algorithms

- Contain functions to perform operations on input data.
- May contain configuration settings.
- May use polymorphism to allow algorithms to be called in a similar way.
- May not need a class for some functions.
- In some languages where classes are required (Java), use static functions.
- In Python, functions in a module might be a better choice than a class.

Summary

- Introduced object-oriented programming.
- Discussed applications to Python.
- Discussed dangers and applications to other languages.
- Further reading and practice needed to understand ideas.