Object-Oriented Programming

Peter Hill

Outline

- What is Object-Oriented Programming?
- Why use it?
- General concepts of OOP
- How to use OOP in Python
- How to use OOP in Fortran
- How to use OOP in C++

Programming Paradigms

Procedural/imperative programming

- Series of statements
  - “Do this then do that”
- Call functions (procedures) sequentially that may modify data
- Languages: C, C++, Fortran, Python, Matlab

```
B_field = 0.0
update_B(B_field, x0, y0, current0)
update_B(B_field, x1, y1, current1)
```

Programming Paradigms

Declarative programming

- Series of declarations
  - “I want this thing to be done”
- Mostly for databases and optimisation problems
- Languages: SQL, Prolog, Make (?)

```
SELECT SUM(B_field) FROM coils;
```

Programming Paradigms

Functional programming

- Series of expressions or chained functions
  - “This is how you do that”
- Pass in data, get different data out: no mutable state!
- Languages: Haskell, Python, C++

```
coils = [(x0, y0, current0), (x1, y1, current1)]
B_field = sum(map(calculate_B, coils))
```

Programming Paradigms

Object oriented programming

- Series of verbs acting on nouns
  - "Do this to that thing"
- Objects wrap up both data and functions that operate it
- Languages: C++, Python, Fortran, Java

```
coils = Coils([(x0, y0, current0), (x1, y1, current1)])
B_field = coils.calculate_B()
```

Programming Paradigms

- These are all *choices*
  - All Turing-complete languages can do *everything* any other language can... it just might be easier in one language than another (e.g. string manipulation in Fortran is horrible)
- What's the easiest/best way to map your problem onto a program?
- What does your data look like, and what are you doing with it?
- Pick the right tool for the right job
  - OOP probably not well suited to pure data analysis
  - Declarative programming not well suited to simulations

Why use it?

Modular

- A `Tokamak` is made of `Coils` and `Walls`
- `Coils` and `Walls` can be developed separately from each other

Code Reuse

- Reuse the `Tokamak`, `Coils` and `Walls` objects in a different code

May map conceptually better

- We're used to dealing with concrete objects in the real world
- Can be easier to think about objects interacting with each other than passing numbers around

Why not use OOP?

- Problem might not map onto objects
  - Pure data analysis:
    - Take data from experiment
    - Normalise
    - Apply correction
    - Calculate derived quantity
    - Plot graph
- Structure of arrays vs array of structures

General concepts

Abstraction

- Wrap up several concepts into a higher-level abstraction
- An example particle code:

```
ke = calculate_kinetic_energy(mass1, charge1, position1,
                              velocity1, E_field)
force = coulomb_force(charge1, charge2, position1, position2)
update_position(position1, mass1, charge1, velocity1, force)
```

- We keep passing around the same bundle of information!
- Abstract a `Particle`, wrapping up mass, charge, position, etc., and how to calculate energy, force, etc.

```
ke = particle1.kinetic_energy(E_field)
particle1.set_coulomb_force(particle2)
particle1.push()
```

- Reduces cognitive load, freeing up mental energy to think about more important things

General concepts

Encapsulation

- An object may need information that the user doesn't need to care about, or shouldn't be able to change
- A function that returns the kinetic energy of a `Particle`, but don't let the user set the energy directly
- That information can be hidden away as an implementation detail
- `particle.push()` may have some internal work array for doing calculations, but we don't care about that
- If we change how `particle.push()` works internally, the user doesn't even need to know

General concepts

Inheritance

- Objects can be a specialisation of another type of object
- Classic example:

```
class Animal:
    def talk(self):
        pass

class Cat(Animal):
    def talk(self):
        return "Meow!"

class Dog(Animal):
    def talk(self):
        return "Woof!"
```

General concepts

Polymorphism

- Polymorphism ("many shapes") allows us to act on different types of objects with the same function
- Classic example:

```
def make_a_noise(animal):
    print(animal.talk())

ziggy = Cat()
ben = Dog()

make_a_noise(ziggy) # Meow!
make_a_noise(ben) # Woof!
```

Ducking-typing vs polymorphism

A brief diversion about typing

- Static typing: checked at compile-time (C, Fortran)

  ```
  void make_a_noise(Animal animal) {
      std::cout << animal.talk();
  }
  ```

  This won't work if `animal` is not a subtype of `Animal`
- Dynamic typing: checked at runtime (Python)

  ```
  def make_a_noise(animal):
      print(animal.talk())
  ```

  This will work as long as `animal` has a `talk()` method

Some terms

- **Class**: The *type* that defines the data and functions
- **Object**: An *instance* of a class (i.e. a variable whose type is `class`)
- **Attribute/member/component/field**: A variable belonging to a class
- **Method**: A function belonging to a class

Using OOP in Python

Constructor and `self`

- Often need to *initialise* an object when we *instantiate* (create) it
- The method that does this is called the *constructor*
- In Python, this is done with `__init__` method
  - Double underscores in Python indicate "magic"
- First argument of any method is `self`: the instance of the class being used

```python
class Animal:
    def __init__(self, noise):
        self.noise = noise

    def talk(self):
        return self.noise
```

Using OOP in Python

More about `self`

- Normally passed invisibly:

```python
ziggy = Animal("Meow")
ziggy.talk()

# exactly the same as:
Animal.talk(ziggy)
```

- Name `self` is just convention – in other languages, it may be a keyword (e.g. `this` in C++)

Using OOP in Python

Operators

```
class RationalNumber:
    def __init__(self, numerator, denominator):
        self.numerator = numerator
        self.denominator = denominator

    def __str__(self):
        return "{}/{}".format(self.numerator,
                              self.denominator)

    def __add__(self, other):
        numerator = self.numerator * other.denominator \
                  + other.numerator * self.denominator
        denominator = self.denominator * other.denominator
        return RationalNumber(numerator, denominator)
```

Using OOP in Python

Using the RationalNumber class

```
>>> half = RationalNumber(1, 2)
>>> third = RationalNumber(1, 3)
>>> print("{} + {} = {}".format(half, third, half+third))
1/2 + 1/3 = 5/6
```

Other operators

- Numeric operations:

`__sub__`, `__mul__`, `__div__`

- Comparison:

`__eq__`, `__lt__`, `__gt__`

- Fancier features:

`__enter__`, `__exit__`, `__getitem__`, `__iter__`

Using OOP in Fortran

Basic Animals “derived type”

```
module animal_mod
  implicit none
  type :: AnimalType
      character(len=:), allocatable, private :: noise
  contains
      procedure :: talk
  end type AnimalType
 contains
  function talk(this)
      class(AnimalType), intent(in) :: this
      character(len=:), allocatable :: talk
      talk = this%noise
  end function
end module
```

Using OOP in Fortran

Using the type

- Fortran defines a default "structure constructor" that initialises all the members in order

```fortran
program animals
  use animal_mod
  implicit none
  type(AnimalType) :: ziggy

  ziggy = AnimalType("Meow")
  print*, ziggy%talk() ! Meow
end program animals
```

Using OOP in Fortran

Defining our own constructor

- Overload the type name

```
interface AnimalType
    module procedure new_animal_type
end interface
...
function new_animal_type(noise) result(this)
    type(AnimalType), intent(out) :: this
    character(len=*), intent(in) :: noise
    this%noise = '"' // noise // '!"'
end function
...
print*, ziggy%talk() ! "Meow!"
```

Using OOP in Fortran

Operators

```
module rational_mod

  type RationalNumber
    integer :: numerator, denominator
  contains
    private
    procedure :: rational_add
    generic, public :: operator(+) => rational_add
  end type RationalNumber

 contains
 ...
```

Using OOP in Fortran

Operators...

```
  ...
  function rational_add(this, other)
    class(RationalNumber), intent(in) :: this, other
    type(RationalNumber) :: rational_add
    integer :: numerator, denominator

    numerator = this%numerator * other%denominator &
         + other%numerator * this%denominator
    denominator = this%denominator * other%denominator

    rational_add = RationalNumber(numerator, denominator)
  end function rational_add
end module rational_mod
```

Using OOP in Fortran

Operators. . .

```
program rational_numbers
  use rational_mod
  implicit none
  type(RationalNumber) :: half, third, sum
  half = RationalNumber(1, 2)
  third = RationalNumber(1, 3)
  sum = half + third

  print('(I0,A,I0)'), sum%numerator, "/", sum%denominator
end program rational_numbers
```

Using OOP in Fortran

Pretty-printing

```
SUBROUTINE my_write_formatted (var,unit,iotype,vlist,iostat,iomsg)
dtv-type-spec,INTENT(IN) :: var
INTEGER,INTENT(IN) :: unit
CHARACTER(*),INTENT(IN) :: iotype
INTEGER,INTENT(IN) :: vlist(:)
INTEGER,INTENT(OUT) :: iostat
CHARACTER(*),INTENT(INOUT) :: iomsg
END
```

Using OOP in C++

RationalNumbers again

```
class RationalNumber:
public:
    int numerator, denominator;

    RationalNumber(int numerator, int denominator) :
        numerator(numerator), denominator(denominator) {}

    RationalNumber operator+(const RationalNumber& other) {
        ...
        return RationalNumber(numerator, denominator);
    }
};
```

Using OOP in C++

RationalNumbers again

```
#include <iostream>
#include "RationalNumbers.hxx"

int main() {
    RationalNumber half{1, 2}, third{1, 3}, sum;
    sum = half + third;
    std::cout << sum.numerator << "/" << sum.denominator << "\n";
}
```

Conclusions

- Object-oriented programming is a way to wrap up data and functions that operate on that data
- Can be a good mental fit for lots of problems in physics
- OOP encourages modular code that can be reused
- Four “pillars”:
  - Abstraction
  - Encapsulation
  - Inheritance
  - Polymorphism