Chapter 7

Polymorphism, Encapsulation and Inheritance

The three OOP factors

- Remember, there were 3 factors that distinguished an OOP:
  - encapsulation
  - inheritance
  - polymorphism

We are still at encapsulation

What is encapsulation?

- hid details of the implementation so that the program was easier to read and write.
- modularity, make an object so that it can be reused in other contexts.
- providing an interface (the methods) that are the approved way to deal with the class.

One more aspect

A new aspect we should have is *consistency*

Remember Rule 9: Do the right thing

- A new class should be consistent with the rules of the language.
- It should respond to standard messages, it should behave properly with typical functions (assuming the type allows that kind of call).

An example

Consider a Rational number class. It should respond to:

- construction
- printing
- arithmetic ops (+, -, *, /)
- comparison ops (<, >, <=, >=)

example program

```
# get our rational number class named frac_class
>>> from frac_class import *
>>> r1 = Rational(1,2)      # create the fraction 1/2
>>> r2 = Rational(3,2)      # create the fraction 3/2
>>> r3 = Rational(3)        # default denominator is 1, so really creating 3/1
>>> r_sum = r1 + r2         # use "+" in a familiar way
>>> print(r_sum)            # use "print" in a familiar way
4/2
>>> r_sum                   # display value in session in a familiar way
4/2
>>> if r1 == r1:            # use equality check "==" in a familiar way
...     print('equal')
... else:
...     print('not equal')
...
equal
>>> print(r3 - r2)          # combine arithmetic and printing in a familiar way
3/2
```

just like any other number

- by building the class properly, we can make a new instance of Rational look like any other number syntactically.
- the instance responds to all the normal function calls because it is properly encapsulated, it is much easier to use

But how can that work?

Two parts:

- Python can distinguish which operator to use based on types.
- Python provides more standard methods that represent the action of standard functions in the language
  - by defining them in our class, Python will call them in the "right way"

Introspection

- Python does not have a type associated with any variable, since each variable is allowed to reference any object.
- however, we can query any variable as to what type it presently references.
- this is often called ***introspection***. That is, while the program is running we can determine the type a variable references

Python introspection ops

- `type(variable)` returns its type as an object.
- `isinstance(variable,type)` returns a boolean indicating if the variable is of that type.

```
1  def special_sum(a,b):
2      ''' sum two ints or convert params to ints
3      and add. return 0 if conversion fails '''
4      if type(a)==int and type(b)==int:
5          result = a + b
6      else:
7          try:
8              result = int(a) + int(b)
9          except ValueError:
10             result = 0
11     return result
```

Program Development with Classes: Predator-Prey Problem

Predator–prey problems

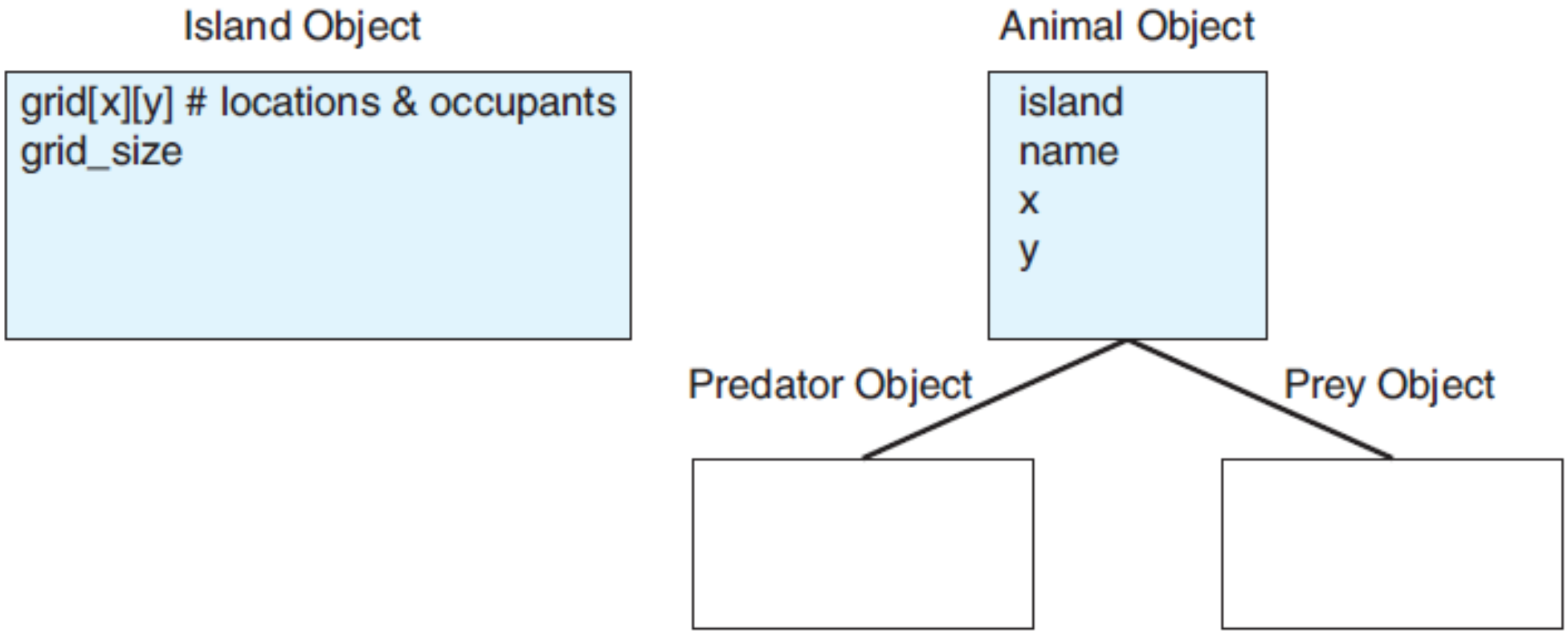
To examine changes in population sizes between two interacting animal groups.

- **Prey**: The population that serves as a food source.
- **Predators**: The population that relies on preying on the other group for survival.
- **OOP Simulation**: Models the dynamic interaction between predator and prey populations.
- **Outcome**: Illustrates a survival struggle, often resembling a "habitat war," as each population competes to sustain or grow.

Rules

- **Clock Ticks**: The habitat updates in units of time called clock ticks, during which each animal gets a chance to act.
- **Movement**: Animals can move to an adjacent empty space once per tick.
- **Reproduction**: Predators and prey reproduce after a fixed breed time. If an animal survives for the breed time, it reproduces in an adjacent empty space and resets its breed timer.
- **Predator Survival**: Predators must find prey within a fixed starve time. If they fail, they die.
- **Predation**: When a predator eats prey, it moves into the prey's space, resets its starve timer, and counts eating as its move for that tick.
- **Clock Update**: At each tick’s end, breed and starve times for all animals are decremented.

Objects:
island, animals, predators, and prey



Inheritance

Class-Instance relations

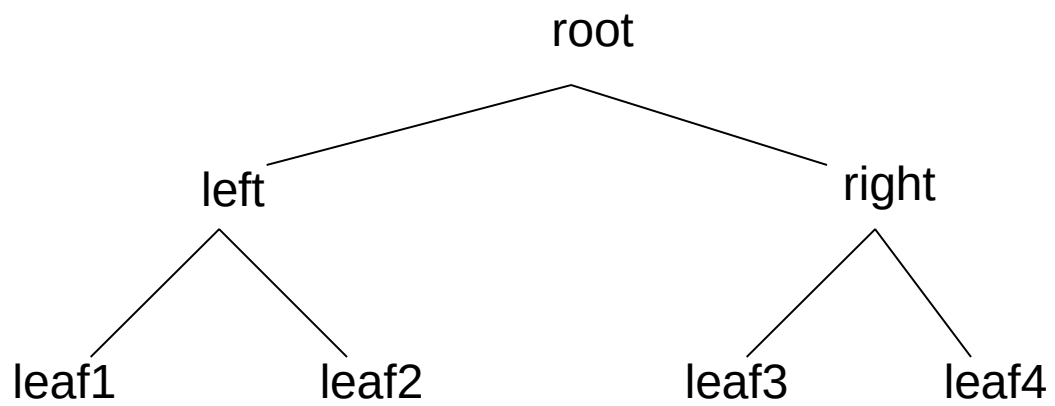
- Remember the relationship between a class and its instances
  - a class can have many instances, each made initially from the constructor of the class
  - the methods an instance can call are initially shared by all instances of a class

Class-Class relations

- Classes can also have a separate relationship with other classes
- the relationships forms a hierarchy
  - ***hierarchy***: A body of persons or things ranked in grades, orders or classes, one above another

computer science 'trees'

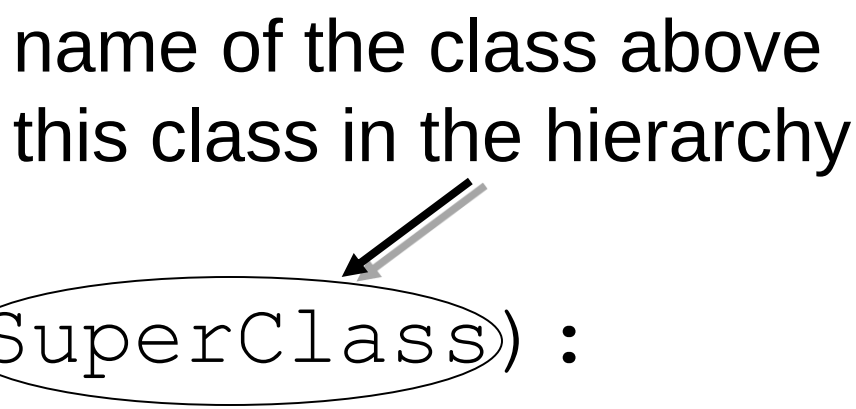
- the hierarchy forms what is called a tree in computer science. Odd 'tree' though



Classes related by a hierarchy

- when we create a class, which is itself another object, we can state how it is related to other classes
- the relationship we can indicate is the class that is 'above' it in the hierarchy

class statement



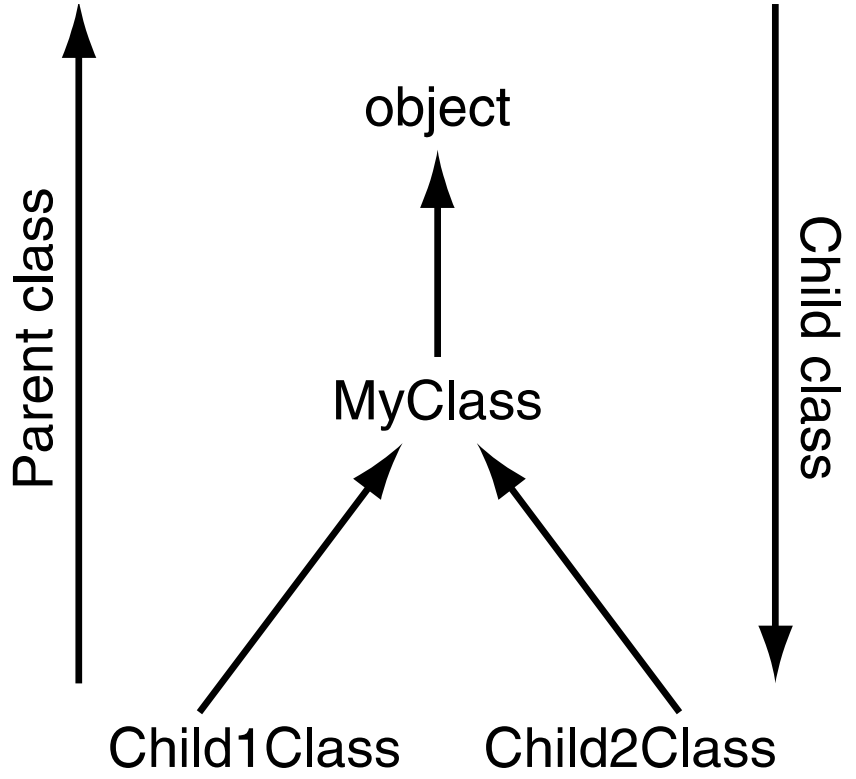

```
class MyClass (SuperClass):
    pass
```

- The top class in Python is called `object`.
- it is predefined by Python, always exists
- use `object` when you have no superclass

```
class MyClass (object):
    pass


class Child1Class (MyClass):
    pass


class Child2Class (MyClass):
    pass
```



```
1  class MyClass (object):
2      ''' parent is object '''
3      pass
4
5  class MyChildClass (MyClass):
6      ''' parent is MyClass '''
7      pass
8
9  my_child_instance = MyChildClass()
10 my_class_instance = MyClass()
11
12 print(MyChildClass.__bases__)          # the parent class
13 print(MyClass.__bases__)               # ditto
14 print(object.__bases__)                # ditto
15
16 print(my_child_instance.__class__)     # class from which the instance came
17 print(type(my_child_instance))         # same question, asked via function
```

is-a, super and sub class

- the class hierarchy imposes an ***is-a*** relationship between classes
  - MyChildClass ***is-a*** (or is a kind of) MyClass
  - MyClass ***is-a*** (or is a kind of) object
  - object has as a subclass MyClass
  - MyChildClass has as a superclass MyClass

Scope for objects, the full story

1. Look in the object for the attribute
2. If not in the object, look to the object's class for the attribute
3. If not in the object's class, look up the hierarchy of that class for the attribute
4. If you hit object, then the attribute does not exist

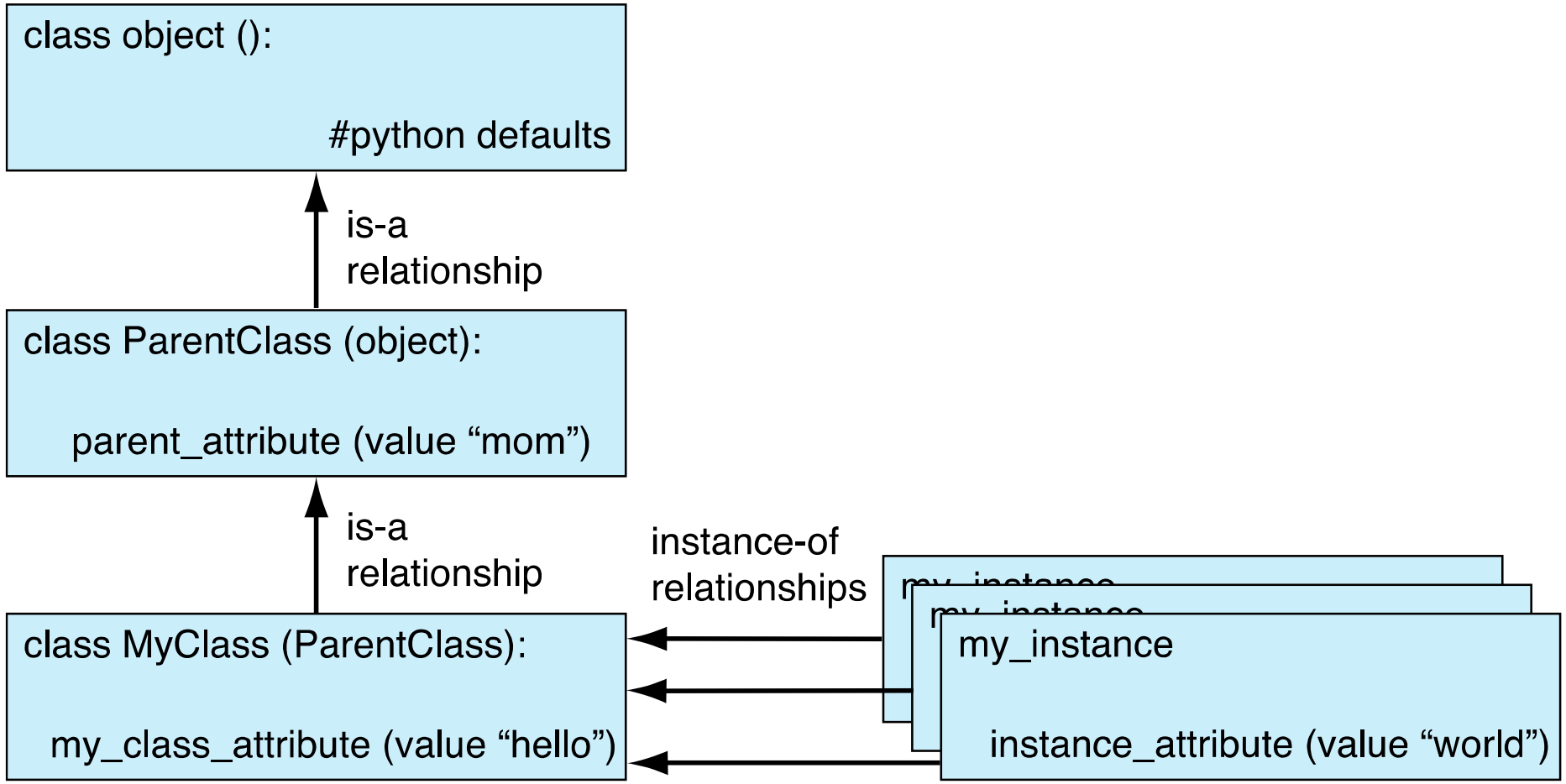


FIGURE 12.2 The players in the “find the attribute” game.

Inheritance is powerful but also can be complicated

- many powerful aspects of OOP are revealed through uses of inheritance
- However, some of that is a bit detailed and hard to work with. Definitely worth checking out but a bit beyond us and our first class

The Standard Model

builtins are objects too

- One nice way, easy way, to use inheritance is to note that all the builtin types are objects also
- Thus you can inherit the properties of built-in types then modify how they get used in your subclass
- you can also use any of the types you pull in as modules

specializing a method

- One technical detail. Normal method calls are called ***bound methods***. Bound methods have an instance in front of the method call and automatically pass self

```
my_inst = MyClass()
my_inst.method(arg1,arg2)
```

- `my_inst` is an instance, so the method is bound

unbound methods

it is also possible to call a method without Python binding `self`. In that case, the user has to do it.

- unbound methods are called as part of the class but `self` passed by the user

```
my_inst = MyClass()
MyClass.method(my_inst, arg2, arg3)
```

`self` is passed **explicitly** (`my_inst` here)!

Why???

- Consider an example. We want to specialize a new class as a subclass of list.

```
class MyClass(list):
```

- easy enough, but we want to make sure that we get our new class instances initialized the way they are supposed to, by calling __init__ of the super class

Why call the super class init?

If we don't explicitly say so, our class may inherit stuff from the super class, but we must make sure we call it in the proper context. For example, our __init__ would be:

```
def __init__(self):
    list.__init__(self)
# do anything else special to MyClass
```

explicit calls to the super

- we explicitly call the super class constructor using an unbound method (why not a bound method????)
- then, after it completes we can do anything special for our new class
- We **specialize** the new class but inherit most of the work from the super. Very clever!

Gives us a way to organize code

- ***specialization***. A subclass can inherit code from its superclass, but modify anything that is particular to that subclass
- ***over-ride***. change a behavior to be specific to a subclass
- ***reuse-code***. Use code from other classes (without rewriting) to get behavior in our class.

Reminder, rules so far

1. Think before you program!
2. A program is a human-readable essay on problem solving that also happens to execute on a computer.
3. The best way to imporve your programming and problem solving skills is to practice!
4. A foolish consistency is the hobgoblin of little minds
5. Test your code, often and thoroughly
6. If it was hard to write, it is probably hard to read. Add a comment.
7. All input is evil, unless proven otherwise.
8. A function should do one thing.
9. Make sure your class does the right thing.