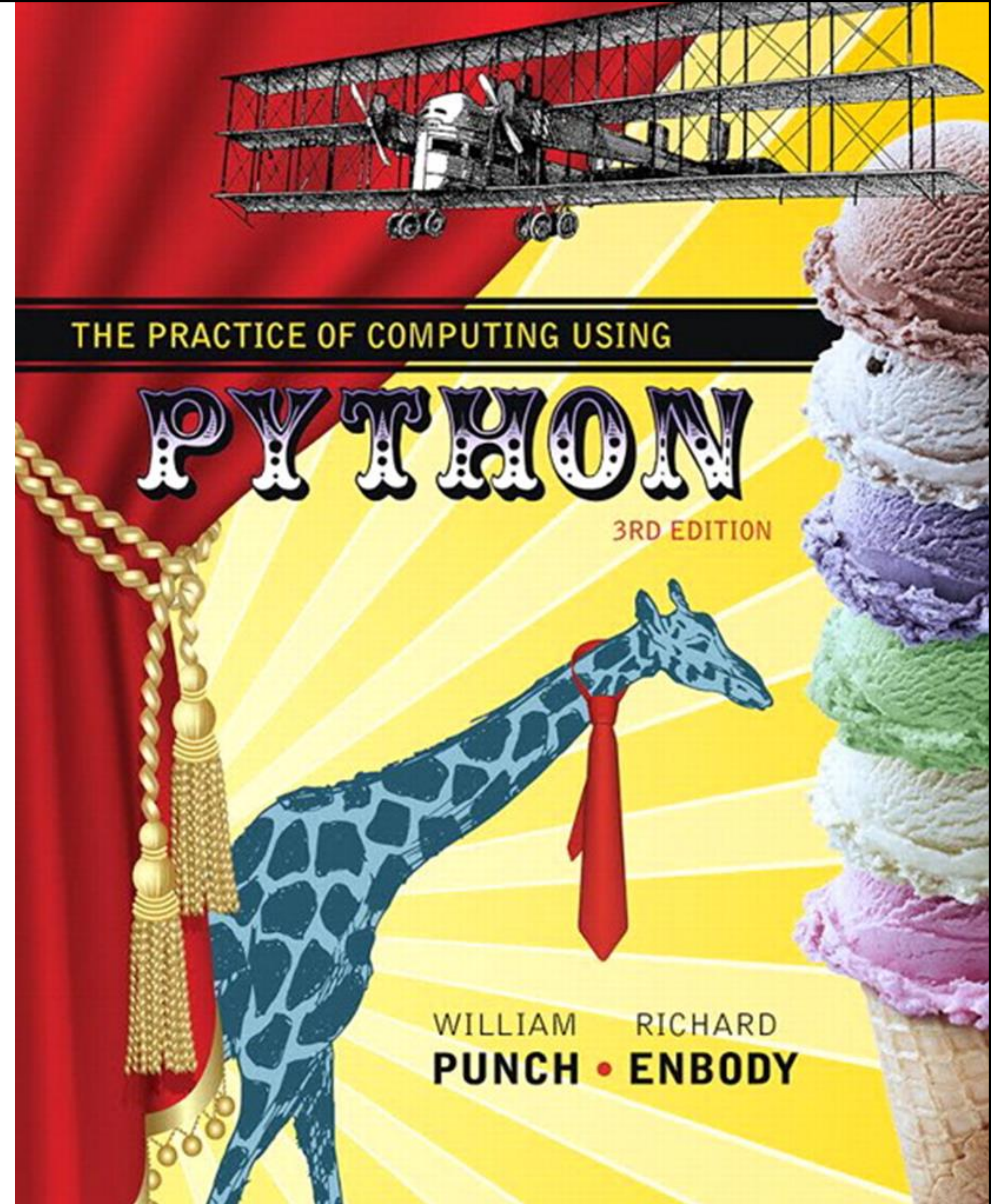
chapter 12

More on Classes



Three OOP Features

- three features distinguish an object-oriented programming language from other types of languages
  - encapsulation
  - inheritance
  - polymorphism

"The Practice of Computing Using Python, 3rd Edition", Punch & Enbody, Copyright © 2017 Pearson Education, Inc.

Encapsulation

- hides details of the implementation so that the program is easier to read and write
- aids in modularity, so that an object can be reused in other contexts
- provides an interface (through methods) that are the approved way to deal with the class

"The Practice of Computing Using Python, 3rd Edition", Punch & Enbody, Copyright © 2017 Pearson Education, Inc.

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)

"The Practice of Computing Using Python, 3rd Edition", Punch & Enbody, Copyright © 2017 Pearson Education, Inc.

Example

consider a Rational number class

it should respond to

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

"The Practice of Computing Using Python, 3rd Edition", Punch & Enbody, Copyright © 2017 Pearson Education, Inc.

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
```

"The Practice of Computing Using Python, 3rd Edition", Punch & Enbody, Copyright © 2017 Pearson Education, Inc.

Rational Should Work 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

"The Practice of Computing Using Python, 3rd Edition", Punch & Enbody, Copyright © 2017 Pearson Education, Inc.

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"

"The Practice of Computing Using Python, 3rd Edition", Punch & Enbody, Copyright © 2017 Pearson Education, Inc.

More on Type

as mentioned previously, a class is essentially a new type

- when we make an instance of a class, we create an object of a particular type
  - 1.36 is a float
- **`my_instance = MyClass()`**
  - **`my_instance`** is of type **`MyClass`**

Punch & Enbody, Copyright © 2017 Pearson Education, Inc.

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

"The Practice of Computing Using Python, 3rd Edition", Punch & Enbody, Copyright © 2017 Pearson Education, Inc.

Python Introspection Operations

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

"The Practice of Computing Using Python, 3rd Edition", Punch & Enbody, Copyright © 2017 Pearson Education, Inc.

Code Listing 12.1

```python
def special_sum(a,b):
    ''' sum two ints or convert params to ints
    and add. return 0 if conversion fails '''
    if type(a)==int and type(b)==int:
        result = a + b
    else:
        try:
            result = int(a) + int(b)
        except ValueError:
            result = 0
    return result
```


Punch & Enbody, Copyright © 2017 Pearson Education, Inc.

Operator Overloading

"The Practice of Computing Using Python, 3rd Edition",
Punch & Enbody, Copyright © 2017 Pearson Education, Inc.

So What Does `var1+var2` Mean?

it depends on is the type

- the + operation has two operands – what are their types?
- Python uses introspection to find the type and then selects the correct operator

We've Seen This Before

what does **`var1+var2`** do?

- with two strings, we get concatenation
- with two integers, we get addition
- with an integer and a string we get:

```
Traceback (most recent call last):
 File "<pyshell#9>", line 1, in <module>
   1+'a'
TypeError: unsupported operand type(s) for
   +: 'int' and 'str'
```

"The Practice of Computing Using Python, 3rd Edition", Punch & Enbody, Copyright © 2017 Pearson Education, Inc.

Operator Overloading

- the plus operator is ***overloaded***
- that is, the operator can do or mean different things (have multiple/overloaded meanings) depending on the types involved
- if Python does not recognize the operation and that combination of types, you get an error

"The Practice of Computing Using Python, 3rd Edition", Punch & Enbody, Copyright © 2017 Pearson Education, Inc.

Python Overloaded Ops

- Python provides a set of operators that can be overloaded
  - not all of the operators, but many
- like all the special class operations, they use the two underlines before and after
- they come in three general classes
  - numeric type operations (`+`,`-`,`<`,`>`, etc.)
  - container operations (`[]`, iterate, `len`, etc.)
  - general operations (printing, construction)

"The Practice of Computing Using Python, 3rd Edition", Punch & Enbody, Copyright © 2017 Pearson Education, Inc.

Math-like Operators		
Expression	**Method name**	**Description**
x + y	`__add__()`	Addition
x − y	`__sub__()`	Subtraction
x * y	`__mul__()`	Multiplication
x / y	`__div__()`	Division
x == y	`__eq__()`	Equality
x > y	`__gt__()`	Greater than
x >= y	`__ge__()`	Greater than or equal
x < y	`__lt__()`	Less than
x <= y	`__le__()`	Less than or equal
x != y	`__ne__()`	Not equal
Sequence Operators		
len(x)	`__len__()`	Length of the sequence
x in y	`__contains__()`	Does the sequence y contain x?
x[key]	`__getitem__()`	Access element *key* of sequence x
x[key]=y	`__setitem__()`	Set element *key* of sequence x to value y
General Class Operations		
x=myClass()	`__init__()`	Constructor
print (x), str(x)	`__str__()`	Convert to a readable string
	`__repr__()`	Print a Representation of x
	`__del__()`	Finalizer, called when x is garbage collected

TABLE 12.1 Python Special Method Names

"The Practice of Computing Using Python, 3rd Edition", Punch & Enbody, Copyright © 2017 Pearson Education, Inc.

"The Practice of Computing Using Python, 3rd Edition", Punch & Enbody, Copyright © 2017 Pearson Education, Inc.

```python
1 class MyClass(object):
2         def __init__(self, param1=0):
3             ''' constructor, sets attribute value to
4             param1, default is 0'''
5             print('in constructor')
6             self.value = param1
7
8         def __str__(self):
9             ''' Convert val attribute to string.'''
10            print('in str')
11            return 'Val is: {}'.format(str(self.value))
12
13        def __add__(self,param2):
14            ''' Perform addition with param2, a MyClass instance.
15            Return a new MyClass instance with sum as value attribute'''
16            print('in add')
17            result = self.value + param2.value
18            return MyClass(result)
```


"The Practice of Computing Using Python, 3rd Edition", Punch & Enbody, Copyright © 2017 Pearson Education, Inc.

How Does **`v1+v2`** Map to **`__add__`**?

`v1 + v2`

is turned into, by Python,

`v1.__add__(v2)`

- these are ***exactly equivalent expressions***
  - the first variable calls the **`__add__`** method with the second variable passed as an argument
- **`v1`** is `bound` to **`self`**, **`v2`** is bound to **`param2`**

"The Practice of Computing Using Python, 3rd Edition", Punch & Enbody, Copyright © 2017 Pearson Education, Inc.

Calling `__str__`

- when does the `__str__` method get called?
  - whenever a string representation of the instance is required
    - directly, by calling `str(my_instance)`
    - indirectly, by calling `print(my_instance)`

Simple Rational Number Class

- a Rational is represented by two integers, the numerator and the denominator
- we can apply many of the numeric operators to Rational

"The Practice of Computing Using Python, 3rd Edition", Punch & Enbody, Copyright © 2017 Pearson Education, Inc.

Code Listing 12.3

"The Practice of Computing Using Python, 3rd Edition",
Punch & Enbody, Copyright © 2017 Pearson Education, Inc.

```
1  class Rational(object):
2      """ Rational with numerator and denominator. Denominator
3      parameter defaults to 1"""
4      def __init__(self,numer,denom=1):
5          print('in constructor')
6          self.numer = numer
7          self.denom = denom
8
9      def __str__(self):
10         """ String representation for printing"""
11         print('in str')
12         return str(self.numer)+'/'+str(self.denom)
13
14     def __repr__(self):
15         """ Used in interpreter. Call __str__ for now """
16         print('in repr')
17         return self.__str__()
```


"The Practice of Computing Using Python, 3rd Edition", Punch & Enbody, Copyright © 2017 Pearson Education, Inc.

`__str__` vs. `__repr__`

- `__repr__` is what the interpreter will call when you type an instance
  - potentially, the representation of the instance, something you can recreate an instance from
- `__str__` is a conversion of the instance to a string
  - often we define `__str__`, and have `__repr__` call `__str__`
  - note the call: `self.__str__()`

"The Practice of Computing Using Python, 3rd Edition", Punch & Enbody, Copyright © 2017 Pearson Education, Inc.

The __init__ Method

- each instance gets an attribute **`numer`** and **`denom`** to represent the numerator and denominator of that instance's values

Implementing Addition

remember how to add fractions

- if the denominator is the same, add the numerators
- if not, find a new common denominator that each denominator divides without remainder
- modify the numerators and add

"The Practice of Computing Using Python, 3rd Edition", Punch & Enbody, Copyright © 2017 Pearson Education, Inc.

The LCM and GCD

- the least common multiple (LCM) finds the smallest number that each denominator divides without remainder
- the greatest common divisor (GCD) finds the largest number two numbers can divide into without remainder

"The Practice of Computing Using Python, 3rd Edition", Punch & Enbody, Copyright © 2017 Pearson Education, Inc.

LCM in terms of GCD

$$LCM(a,b) = \frac{a * b}{GCD(a,b)}$$

OK, how do we find the GCD?

"The Practice of Computing Using Python, 3rd Edition", Punch & Enbody, Copyright © 2017 Pearson Education, Inc.

GCD and Euclid

- one of the earliest algorithms recorded was the GCD by Euclid in his book *Elements* around 300 B.C.
  - he originally defined it in terms of geometry, but the result is the same

"The Practice of Computing Using Python, 3rd Edition", Punch & Enbody, Copyright © 2017 Pearson Education, Inc.

"The Practice of Computing Using Python, 3rd Edition", Punch & Enbody, Copyright © 2017 Pearson Education, Inc.

GCD Algorithm

GCD(a,b)

1. if one of the numbers is 0, return the other and halt
2. otherwise, find the integer remainder of the larger number divided by the smaller number
3. reapply GCD(a,b) with a as the smaller element and b the remainder from step 2

"The Practice of Computing Using Python, 3rd Edition", Punch & Enbody, Copyright © 2017 Pearson Education, Inc.

```python
def gcd(bigger, smaller):
    """Calculate the greatest common divisor of two positive integers."""
    if not bigger > smaller:         # swap if necessary so bigger > smaller
        bigger, smaller = smaller, bigger
    while smaller != 0:                              # 1. if smaller == 0, halt
        remainder = bigger % smaller                 # 2. find remainder
        print('calculation, big:{}, small:{}, rem:{}'.\
              format(bigger, smaller, remainder)) # debugging
        bigger, smaller = smaller, remainder  # 3. reapply
    return bigger
```

```python

def lcm (a,b):
    """Calculate the lowest common multiple of two positive integers."""
    return (a*b)//gcd(a,b)  # Equation 12.1, // ensures an int is returned
```


"The Practice of Computing Using Python, 3rd Edition", Punch & Enbody, Copyright © 2017 Pearson Education, Inc.

The `__add__` Method (Code Listing 12.6)

```
38     def __add__(self, param_Rational):
39         """ Add two Rationals"""
40         print('in add')
41         # find a common denominator (lcm)
42         the_lcm = lcm(self.denom, param_Rational.denom)
43         # multiply each by the lcm, then add
44         numerator_sum = (the_lcm * self.numer/self.denom) + \
45                         (the_lcm * param_Rational.numer/param_Rational.denom)
46         return Rational(int(numerator_sum), the_lcm)
47
48     def __sub__(self, param_Rational):
49         """ Subtract two Rationals"""
50         print('in sub')
51         # subtraction is the same but with '-' instead of '+'
52         the_lcm = lcm(self.denom, param_Rational.denom)
53         numerator_diff = (the_lcm * self.numer/self.denom) - \
54                          (the_lcm * param_Rational.numer/param_Rational.denom)
55         return Rational(int(numerator_diff), the_lcm)
```


"The Practice of Computing Using Python, 3rd Edition", Punch & Enbody, Copyright © 2017 Pearson Education, Inc.

Equality

- the equality method is `__eq__`
- it is invoked with the == operator

  `1/2 == 1/2` is equivalent to

  `1/2.__eq__(1/2)`
- it should be able to deal with non-reduced fractions

  `1/2 == 1/2` is True

  `2/4 == 3/6` is True

"The Practice of Computing Using Python, 3rd Edition", Punch & Enbody, Copyright © 2017 Pearson Education, Inc.

"The Practice of Computing Using Python, 3rd Edition", Punch & Enbody, Copyright © 2017 Pearson Education, Inc.

```python
def reduce_rational(self):
    """ Return the reduced fractional value as a Rational"""
    print('in reduce')
    # find the gcd and then divide numerator and denominator by gcd
    the_gcd = gcd(self.numer,self.denom)
    return Rational(self.numer//the_gcd, self.denom//the_gcd)

def __eq__(self,param_Rational):
    """ Compare two Rationals for equality, return Boolean"""
    print('in eq')
    # reduce both; then check that numerators and denominators are equal
    reduced_self = self.reduce_rational()
    reduced_param = param_Rational.reduce_rational()
    return reduced_self.numer == reduced_param.numer and\
        reduced_self.denom == reduced_param.denom
```


"The Practice of Computing Using Python, 3rd Edition", Punch & Enbody, Copyright © 2017 Pearson Education, Inc.

Fitting In

- what is amazing about the traces of these methods is how many of them are called in service of the overall goal
- all we did was provide the basic pieces and Python orchestrates how they all fit together
- Rule 9 rules!

"The Practice of Computing Using Python, 3rd Edition", Punch & Enbody, Copyright © 2017 Pearson Education, Inc.

What Doesn't Work

"The Practice of Computing Using Python, 3rd Edition",
Punch & Enbody, Copyright © 2017 Pearson Education, Inc.

So `r1+r2` Works, But What About

- we said the add we defined would work for two rationals, but what about…

```
r1 + 1    # Rational + integer
1 + r1    # commutativity
```

- neither works right now
- how do we fix it?

"The Practice of Computing Using Python, 3rd Edition", Punch & Enbody, Copyright © 2017 Pearson Education, Inc.

`r1 + 1`

- what's the problem?
  - add expects a Rational number as the second argument
  - Python used to have a coercion operator, but that is deprecated
    - coerce: force conversion to another type
    - deprecated: 'disapproval', an approach that is no longer supported
- our constructor supports conversion of an int to a Rational
  - how/where to do this?

"The Practice of Computing Using Python, 3rd Edition", Punch & Enbody, Copyright © 2017 Pearson Education, Inc.

Introspection in **`__add__`**

- the **`__add__`** operator is going to have to check the types of the parameter and then decide what should be done
  - if the type is an integer, convert it to Rational
  - if the type is Rational, do what we did before
  - if the type is anything else, it will not be allowed

"The Practice of Computing Using Python, 3rd Edition", Punch & Enbody, Copyright © 2017 Pearson Education, Inc.

"The Practice of Computing Using Python, 3rd Edition",
Punch & Enbody, Copyright © 2017 Pearson Education, Inc.

```
1     def __add__(self, param):
2         """ Add two Rationals. Allows int as a parameter"""
3         print('in add')
4         if type(param) == int:  # convert ints to Rationals
5             param = Rational(param)
6         if type(param) == Rational:
7             # find a common denominator (lcm)
8             the_lcm = lcm(self.denom, param.denom)
9             # multiply each by the lcm, then add
10            numerator_sum = (the_lcm * self.numer/self.denom) + \
11                (the_lcm * param.numer/param.denom)
12            return Rational(int(numerator_sum),the_lcm)
13        else:
14            print('wrong type')  # problem: some type we cannot handle
15            raise(TypeError)
```


Punch & Enbody, Copyright © 2017 Pearson Education, Inc.

What about `1 + r1`?

- what's the problem?
  - the mapping is wrong
  - `1 + r1` maps to `1.__add__(r1)`
  - no such method for integers (and besides, it would be a real pain to have to add a new method to every type we want to include)
  - user should expect that this should work since addition is commutative!

Punch & Enbody, Copyright © 2017 Pearson Education, Inc.

`__radd__` Method

- Python allows the definition of an `__radd__` method
  - called when the `__add__` method fails because of a type mismatch
  - reverses the two arguments in the call

"The Practice of Computing Using Python, 3rd Edition", Punch & Enbody, Copyright © 2017 Pearson Education, Inc.

`__radd__` vs. `__add__`

- 1 + r1

  try `1.__add__(r1)`

  if failure, look for an `__radd__`

  if it exists, remap

- 1 + r1

  `r1.__radd__(1)`

`__radd__`

- essentially, all we need **`__radd__`** to do is remap the parameters
- after that, it is just add all over again, so we call **`__add__`** directly
- using this approach, we need only update **`__add__`** if any changes are required

```
def __radd__(self,f):
    return self.__add__(f)
```

"The Practice of Computing Using Python, 3rd Edition", Punch & Enbody, Copyright © 2017 Pearson Education, Inc.

Inheritance

"The Practice of Computing Using Python, 3rd Edition", Punch & Enbody, Copyright © 2017 Pearson Education, Inc.

Class-Instance Relationships

- 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

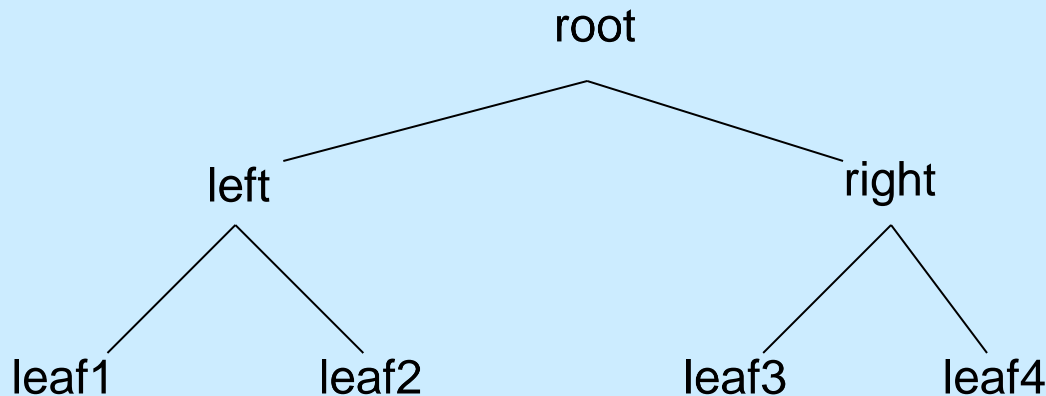
Class-Class Relationships

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

Punch & Enbody, Copyright © 2017 Pearson Education, Inc.

Computer Science 'Trees'

- the hierarchy forms what is called a tree in computer science (odd 'tree' though)



"The Practice of Computing Using Python, 3rd Edition", Punch & Enbody, Copyright © 2017 Pearson Education, Inc.

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

"The Practice of Computing Using Python, 3rd Edition", Punch & Enbody, Copyright © 2017 Pearson Education, Inc.

Class Statement

name of the class above this class in the hierarchy

```
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

"The Practice of Computing Using Python, 3rd Edition", Punch & Enbody, Copyright © 2017 Pearson Education, Inc.

```
class MyClass (object):
    pass


class Child1Class (MyClass):
    pass


class Child2Class (MyClass):
    pass
```

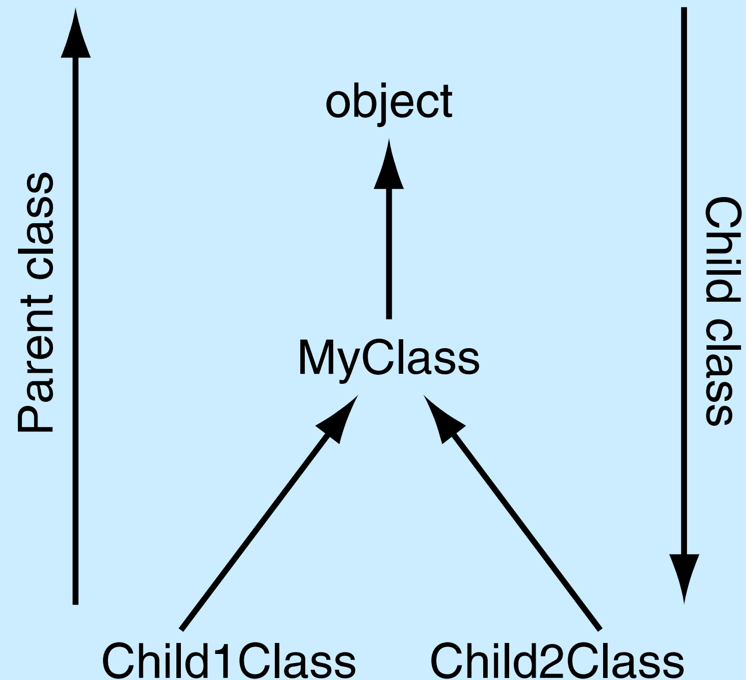


FIGURE 12.1 A simple class hierarchy.

"The Practice of Computing Using Python, 3rd Edition",
Punch & Enbody, Copyright © 2017 Pearson Education, Inc.

"The Practice of Computing Using Python, 3rd Edition",
Punch & Enbody, Copyright © 2017 Pearson Education, Inc.

```python
class MyClass (object):
    ''' parent is object'''
    pass

class MyChildClass (MyClass):
    ''' parent is MyClass '''
    pass

my_child_instance = MyChildClass()
my_class_instance = MyClass()

print(MyChildClass.__bases__)          # the parent class
print(MyClass.__bases__)               # ditto
print(object.__bases__)                # ditto

print(my_child_instance.__class__)     # class from which the instance came
print(type(my_child_instance))         # same question, asked via function
```


"The Practice of Computing Using Python, 3rd Edition", Punch & Enbody, Copyright © 2017 Pearson Education, Inc.

Class Relationships

- 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`**

"The Practice of Computing Using Python, 3rd Edition", Punch & Enbody, Copyright © 2017 Pearson Education, Inc.

Why is This Important?

- the hope of such an arrangement is the saving or re-use of code
- superclass code contains general code that is applicable to many subclasses
- subclass uses superclass code (via sharing), but specializes code for itself when necessary

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

"The Practice of Computing Using Python, 3rd Edition", Punch & Enbody, Copyright © 2017 Pearson Education, Inc.

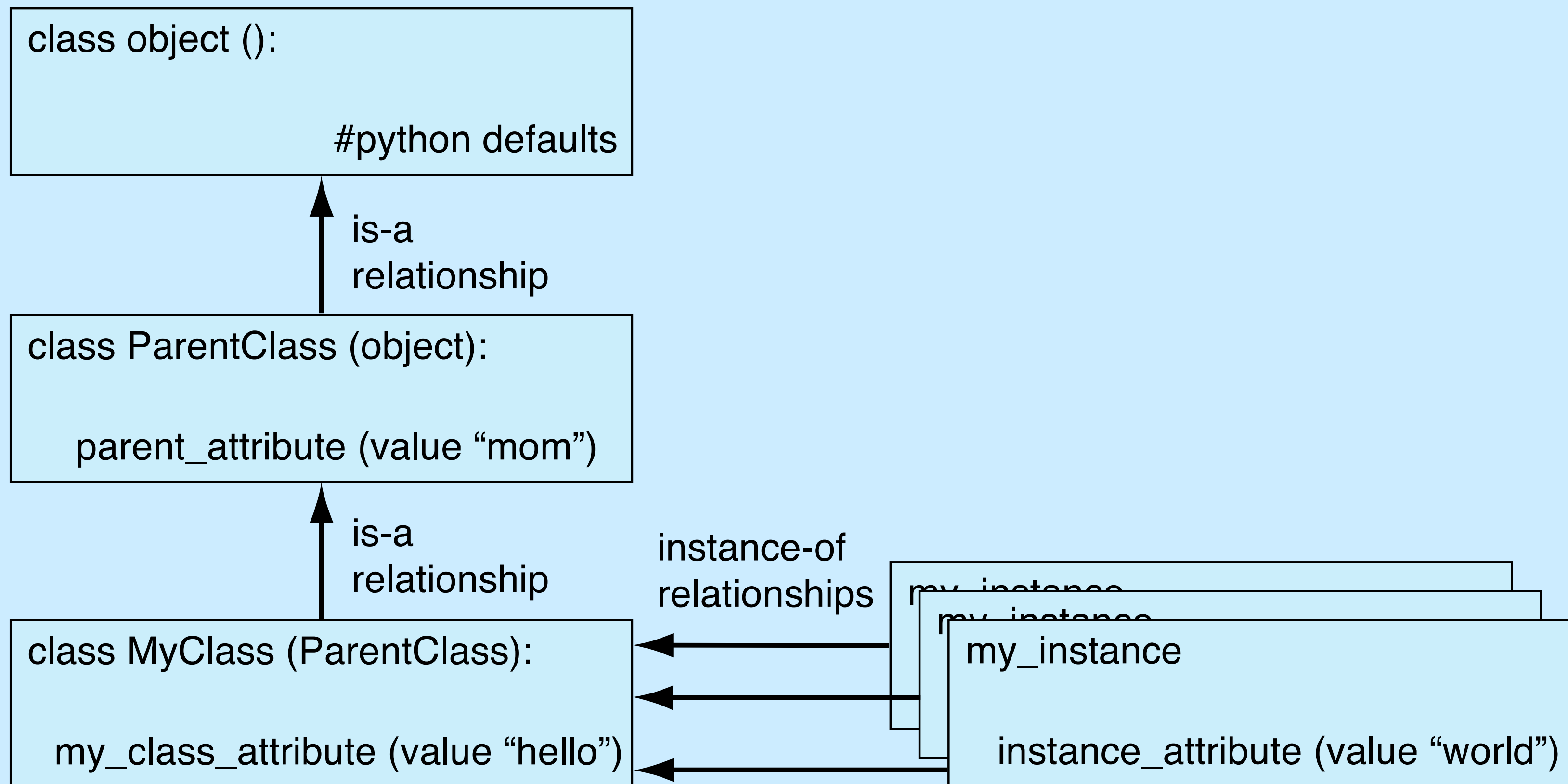


FIGURE 12.2 The players in the “find the attribute” game.

"The Practice of Computing Using Python, 3rd Edition", Punch & Enbody, Copyright © 2017 Pearson Education, Inc.

Inheritance is Powerful, but 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 Practice of Computing Using Python, 3rd Edition", Punch & Enbody, Copyright © 2017 Pearson Education, Inc.

The Standard Model

Punch & Enbody, Copyright © 2017 Pearson Education, Inc.

Built-ins are Objects, Too

- one nice, easy way, to use inheritance is to note that all the built-in 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

"The Practice of Computing Using Python, 3rd Edition", Punch & Enbody, Copyright © 2017 Pearson Education, Inc.

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

"The Practice of Computing Using Python, 3rd Edition", Punch & Enbody, Copyright © 2017 Pearson Education, Inc.

Unbound Methods

- it is also possible to call a method without Python binding **`self`**
  - in that case, the user must 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)!

"The Practice of Computing Using Python, 3rd Edition", Punch & Enbody, Copyright © 2017 Pearson Education, Inc.

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 superclass

"The Practice of Computing Using Python, 3rd Edition", Punch & Enbody, Copyright © 2017 Pearson Education, Inc.

Why Call the Superclass `__init__`?

- if we don't explicitly say so, our class may inherit stuff from the superclass, 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
```

"The Practice of Computing Using Python, 3rd Edition", Punch & Enbody, Copyright © 2017 Pearson Education, Inc.

Explicit Calls to the Superclass

- we explicitly call the superclass 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!)

"The Practice of Computing Using Python, 3rd Edition", Punch & Enbody, Copyright © 2017 Pearson Education, Inc.

Used to Organize Code

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

"The Practice of Computing Using Python, 3rd Edition", Punch & Enbody, Copyright © 2017 Pearson Education, Inc.

Reminder, Rules So Far

1. Think before you program!
2. A program is a human-readable essay on problem solving that executes on a computer.
3. The best way to improve 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.

"The Practice of Computing Using Python, 3rd Edition", Punch & Enbody, Copyright © 2017 Pearson Education, Inc.