PC204 Lecture 8

Conrad Huang
conrad@cgl.ucsf.edu
Genentech Hall, N453A
x6-0415

Topics

- Homework review
- Review of OOP
- Inheritance
- Polymorphism

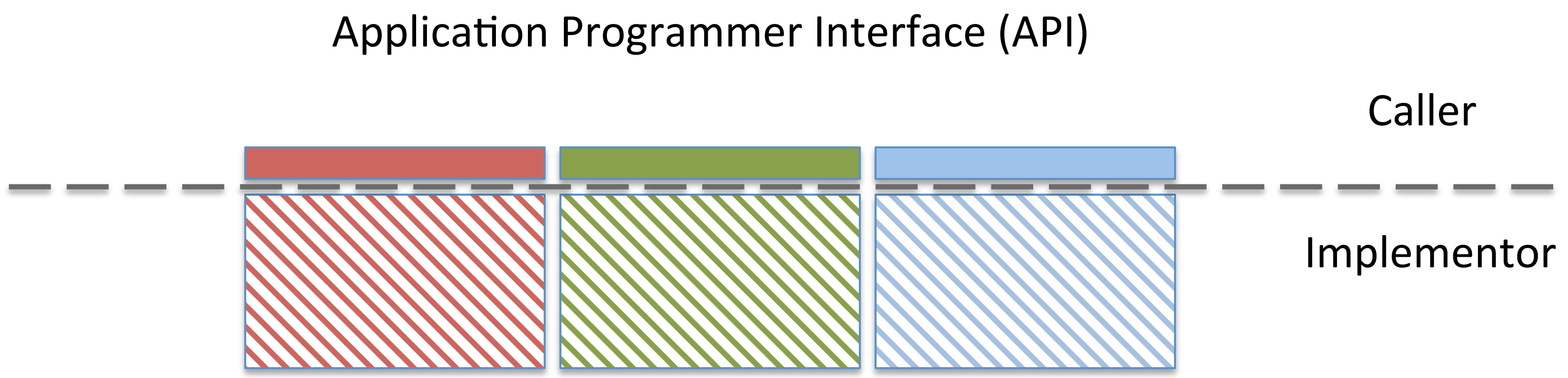
Homework Review

- 7.1 – Rectangle methods
- 7.2 – Rectangle __add__ method

Review of OOP

- Object-oriented programming is about grouping data and functions together into units (objects) that can be manipulated using an external ***interface*** and whose self-consistency is maintained by the internal ***implementation***
- The ultimate goal is to minimize complexity

Interface *vs* Implementation



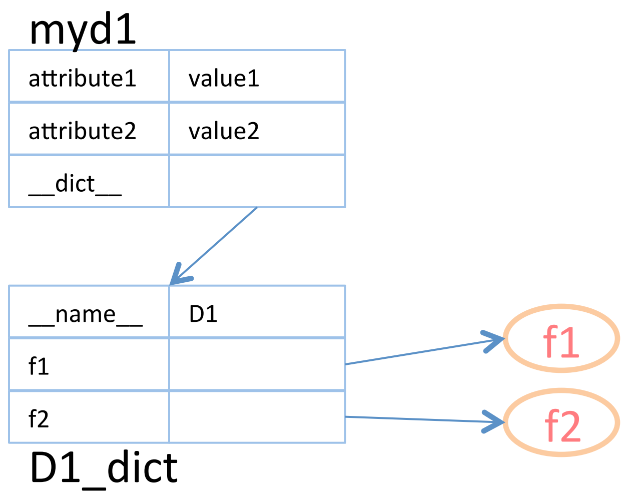
OOP with Dictionaries

- Suppose we have data type **D1** and data dictionary **d1** and **myd1**

```
def f1(d1):
        # do something with d1
def f2(d1):
        # do something with d1
D1_dict = {
        "__name__": "D1",
        "f1": f1,
        "f2": f2,
}
def D1():
        d1 = dict()
        d1["__dict__"] = D1_dict
```

```
# Create a D1 data dictionary
myd1 = D1()

# Apply the f1 function to myd1
myd1["__dict__"]["f1"](myd1)
```



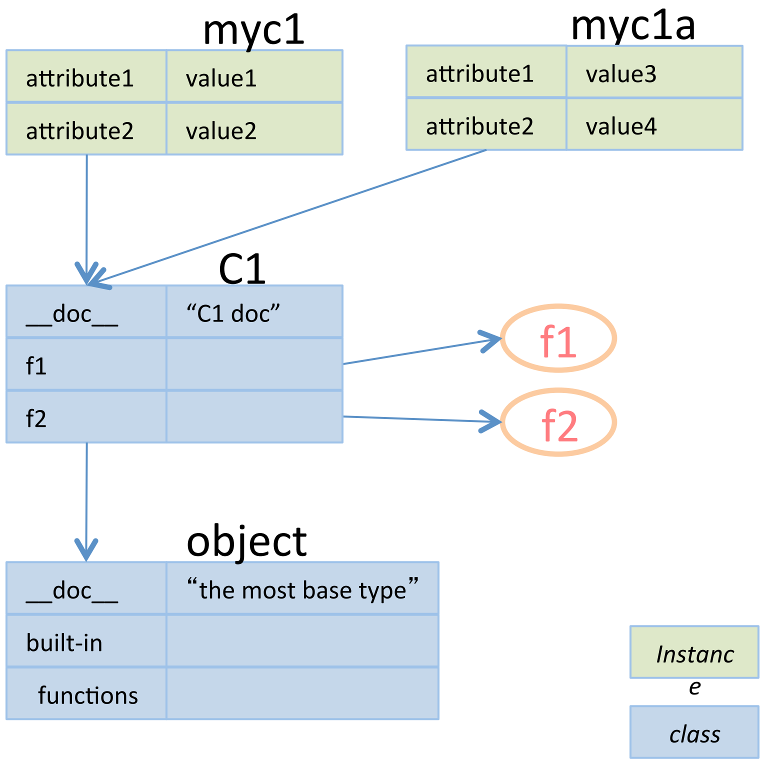
OOP with Classes

- Suppose we have class **C1** and instances **myc1** and **myc1a**

```
class C1(object):
        "C1 doc"
        def f1(self):
                # do something with self
        def f2(self):
                # do something with self

# create C1 instances
myc1 = C1()
myc1a = C1()

# call f2 method on one instance
myc1.f2()
```



OOP with Classes (cont.)

- The **object** class is created automatically by Python
- Executing the “class” statement creates the **C1** class
  - Note **C1** is actually a variable: a reference to a **class** object; this is analogous to the “import” statement where the result is a variable referring to a **module** object
  - Note also that the class object contains data, eg **__doc__**, as well as method references, eg **f1** and **f2**

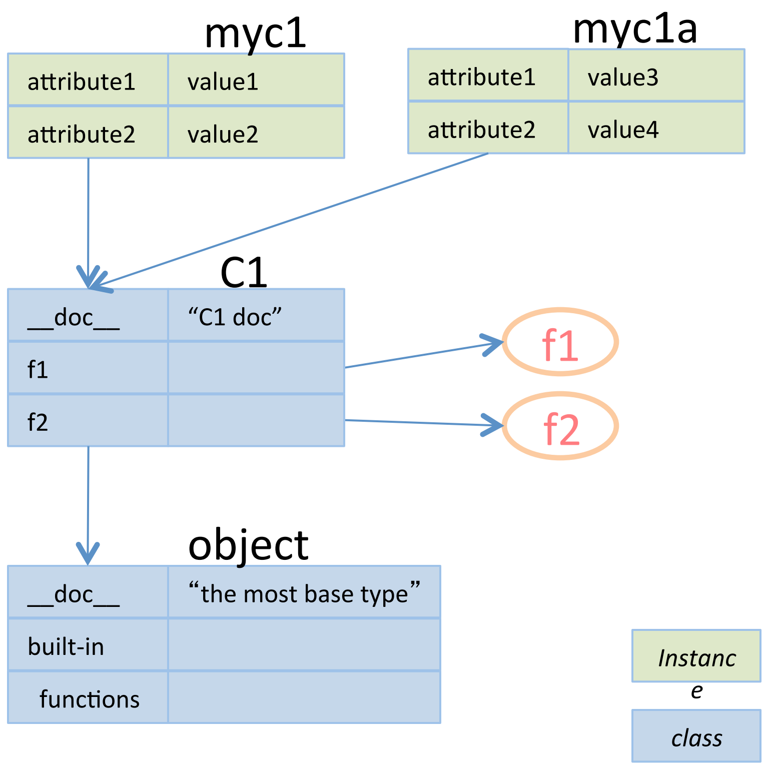
```
class C1(object):
        “C1 doc”
        def f1(self):
                # do something with self
        def f2(self):
                # do something with self

# create a C1 instance
myc1 = C1()
myc1a = C1()

# call f2 method
myc1.f2()
```

OOP with Classes (cont.)

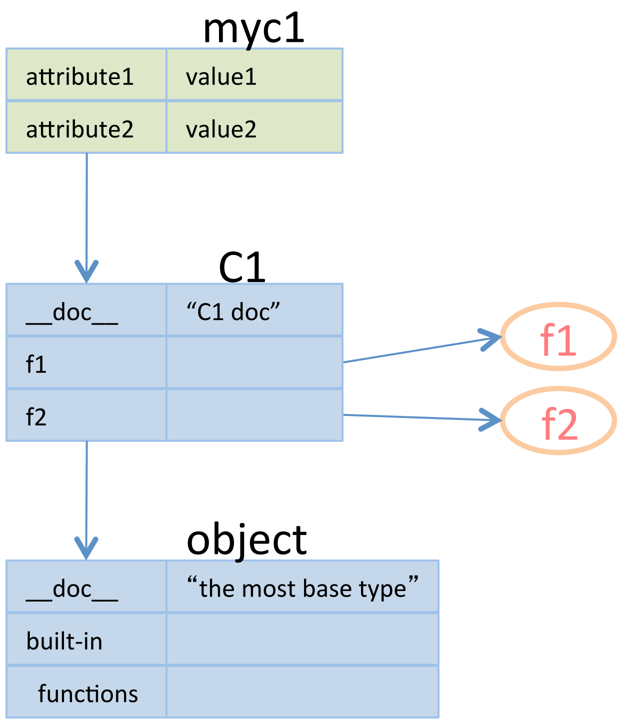
- Creating an instance creates a new attribute namespace
- Each instance has its own attribute namespace, but they all share the same *class* namespace(s)
- Both instance and class attributes may be accessed using the *instance.attribute* syntax

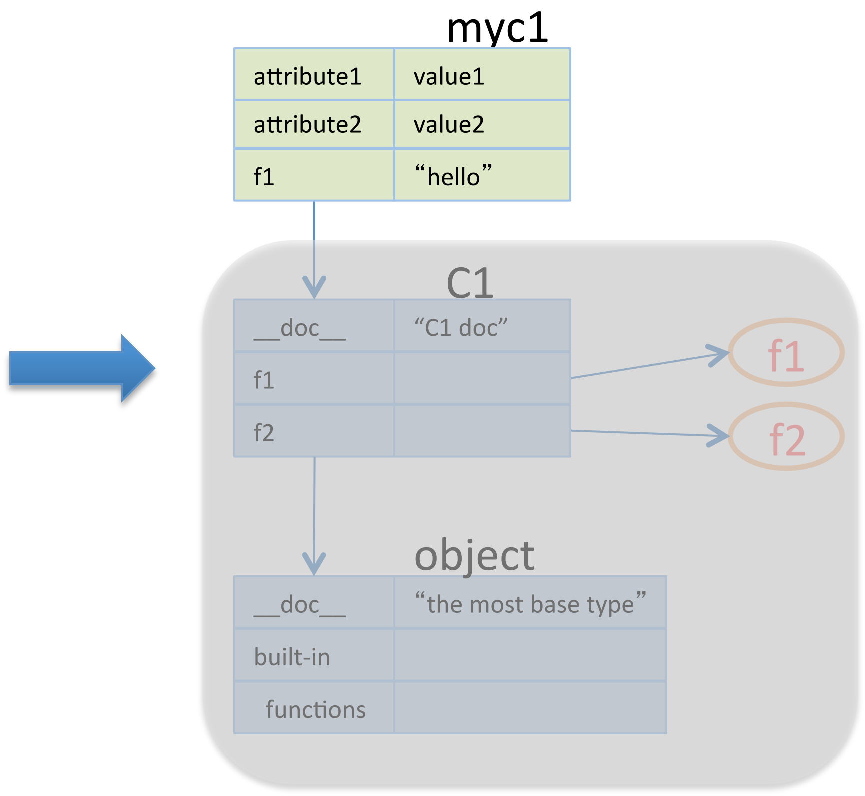


Accessing Attributes

- Setting an instance attribute

```
myc1.f1 = “hello”
```

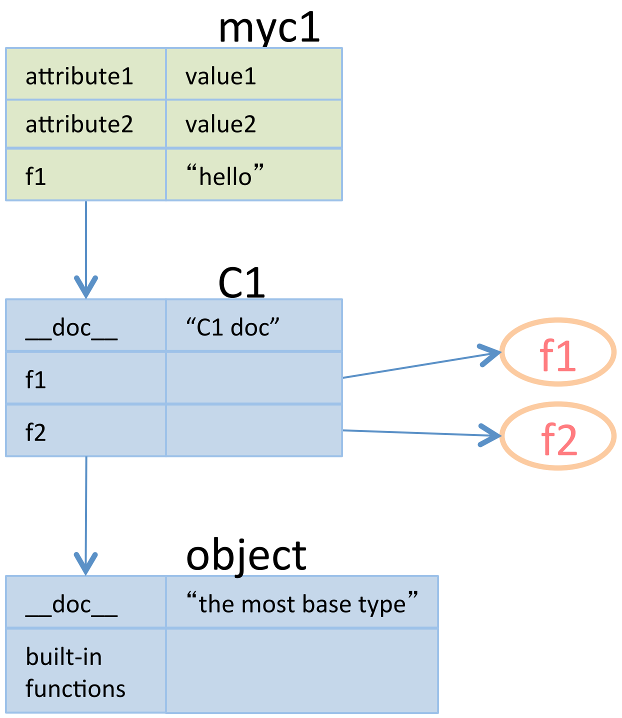




Accessing Attributes (cont.)

- Looking up instance attributes

```
>>> print myc1.f1
hello
>>> print myc1.f2
<bound method C1.f2 of <__main__.C1
object at 0x1401d6b50>>
>>> print myc1.__doc__
C1 doc
>>> myc1.f1()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'str' object is not callable
```



Accessing Attributes (cont.)

- Setting and looking up class attributes
  - Class attributes may be looked up via the instances, but they cannot be modified using the *instance.attribute* syntax
  - To access and manipulate class attributes, use the class variable

```
>>> C1.count = 12
>>> print C1.count
12
>>> C1.f1
<unbound method C1.f1>
>>> C1.f1(myc1)
>>> print C1.__doc__
C1 doc
>>> C1.__doc__ = "new documentation"
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: attribute '__doc__' of
'type' objects is not writable
>>> help(C1)
…
```

Attribute Pitfall

- Attribute lookup and assignment are not symmetrical

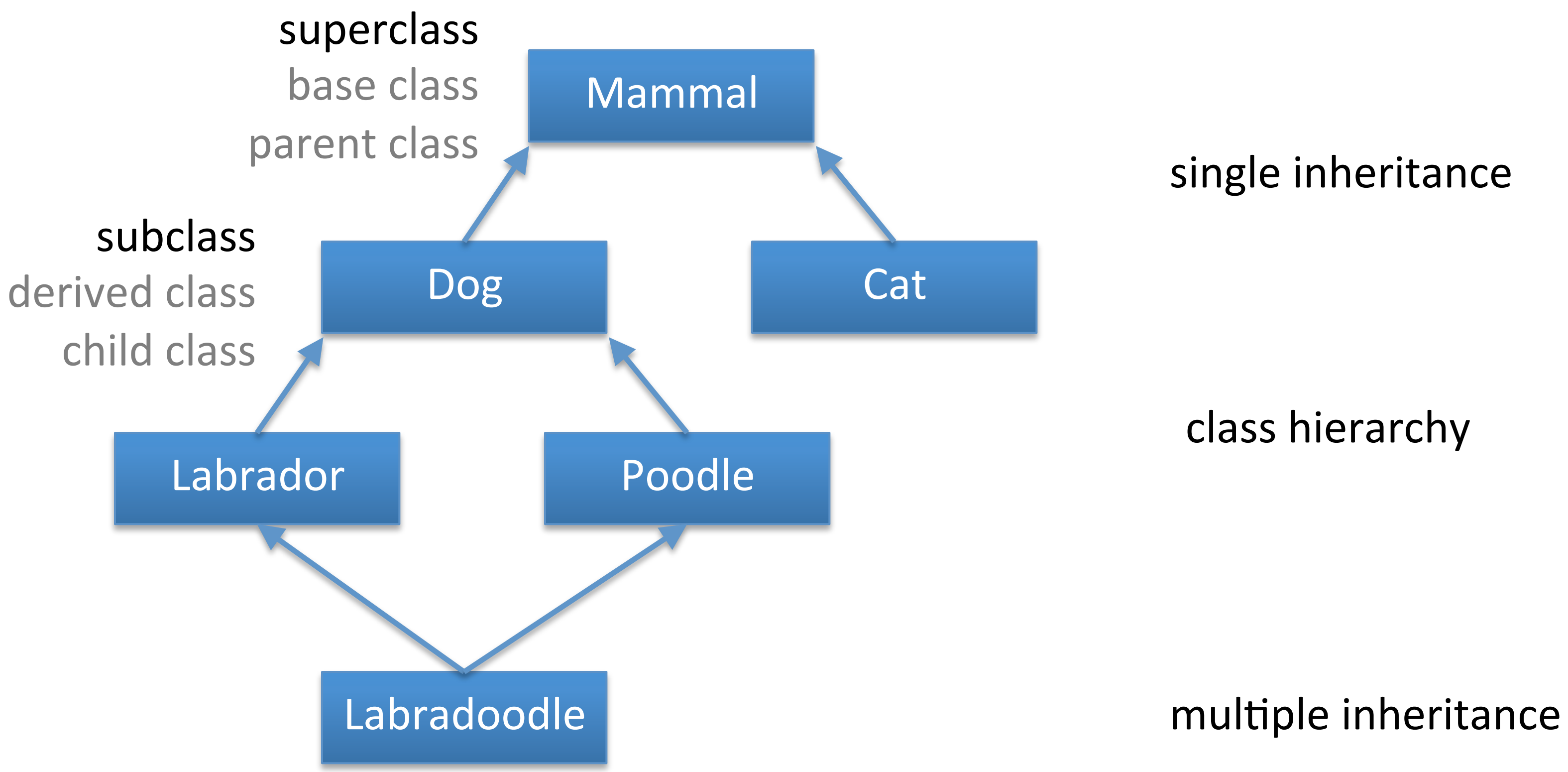
```
>>> class C1:
...     count = 12
...
>>> myc1 = C1()
>>> print myc1.count
12
>>> myc1.count = 20
>>> print myc1.count
20
>>> print C1.count
12
```

OOP Inheritance

- "Inheritance is the ability to define a new class that is a modified version of an existing class." – Allen Downey, *Think Python*
- "A relationship among classes, wherein one class shares the structure or behavior defined in one (single inheritance) or more (multiple inheritance) other classes. Inheritance defines a "kind of" hierarchy among classes in which a subclass inherits from one or more superclasses; a subclass typically augments or redefines the existing structure and behavior of superclasses." – Grady Booch, *Object-Oriented Design*

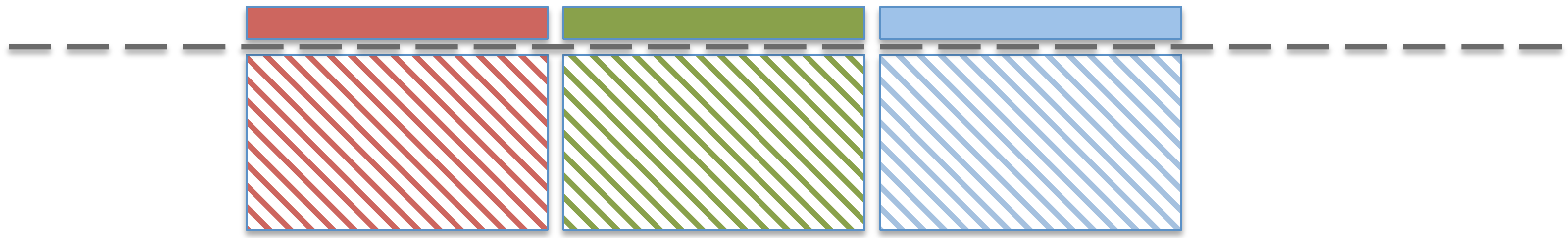
OOP Inheritance (cont.)

- Conceptual example:

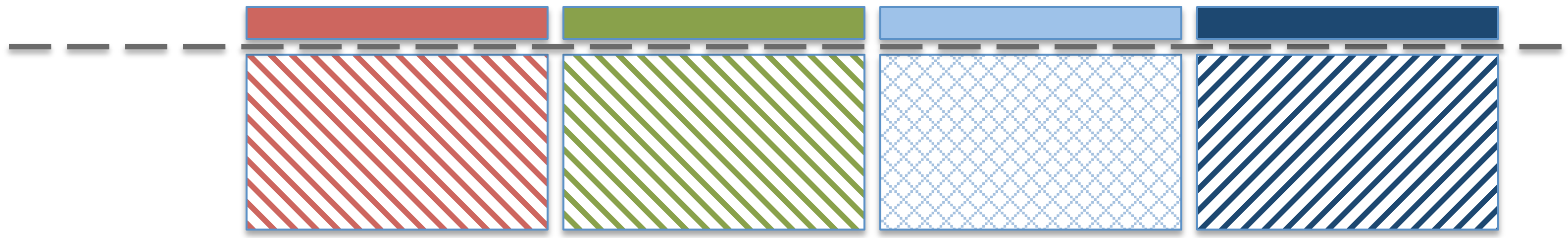


Base Class *vs* Derived Class

Base class



Derived class



Inheritance Syntax

- The syntax for inheritance was already introduced during class declaration
  - **C1** is the name of the subclass
  - **object** is the name of the superclass
  - for multiple inheritance, superclasses are declared as a comma-separated list of class names

```
class C1(object):
        “C1 doc”
        def f1(self):
                # do something with self
        def f2(self):
                # do something with self

# create a C1 instance
myc1 = C1()

# call f2 method
myc1.f2()
```

Inheritance Syntax (cont.)

- Superclasses may be either Python- or user-defined classes
  - For example, suppose we want to use the Python list class to implement a stack (last-in, first-out) data structure
  - Python list class has a method, **pop**, for removing and returning the last element of the list
  - We need to add a **push** method to put a new element at the end of the list so that it gets popped off first

```
class Stack(list):
    “LIFO data structure"
    def push(self, element):
        self.append(element)
    # Might also have used:
    #push = list.append

st = Stack()
print "Push 12, then 1"
st.push(12)
st.push(1)
print "Stack content", st
print "Popping last element", st.pop()
print "Stack content now", st
```

Inheritance Syntax (cont.)

- A subclass inherits all the methods of its superclass
- A subclass can **override** (replace or augment) methods of the superclass
  - Just define a method of the same name
  - Although not enforced by Python, keeping the same arguments (as well as pre- and post-conditions) for the method is highly recommended
  - When augmenting a method, call the superclass method to get its functionality
- A subclass can serve as the superclass for other classes

Overriding a Method

- __init__ is frequently overridden because many subclasses need to both (a) let their superclass initialize their data, and (b) initialize their own data, usually in that order

```
class Stack(list):
    push = list.append

class Calculator(Stack):
    def __init__(self):
        Stack.__init__(self)
        self.accumulator = 0
    def __str__(self):
        return str(self.accumulator)
    def push(self, value):
        Stack.push(self, value)
        self.accumulator = value

c = Calculator()
c.push(10)
print c
```

Multiple Inheritance

- Python supports multiple inheritance
- In the **class** statement, replace the single superclass name with a comma-separated list of superclass names
- When looking up an attribute, Python will look for it in "method resolution order" (MRO) which is approximately left-to-right, depth-first
- There are (sometimes) subtleties that make multiple inheritance tricky to use, eg superclasses that derive from a common super-superclass
- Most of the time, single inheritance is good enough

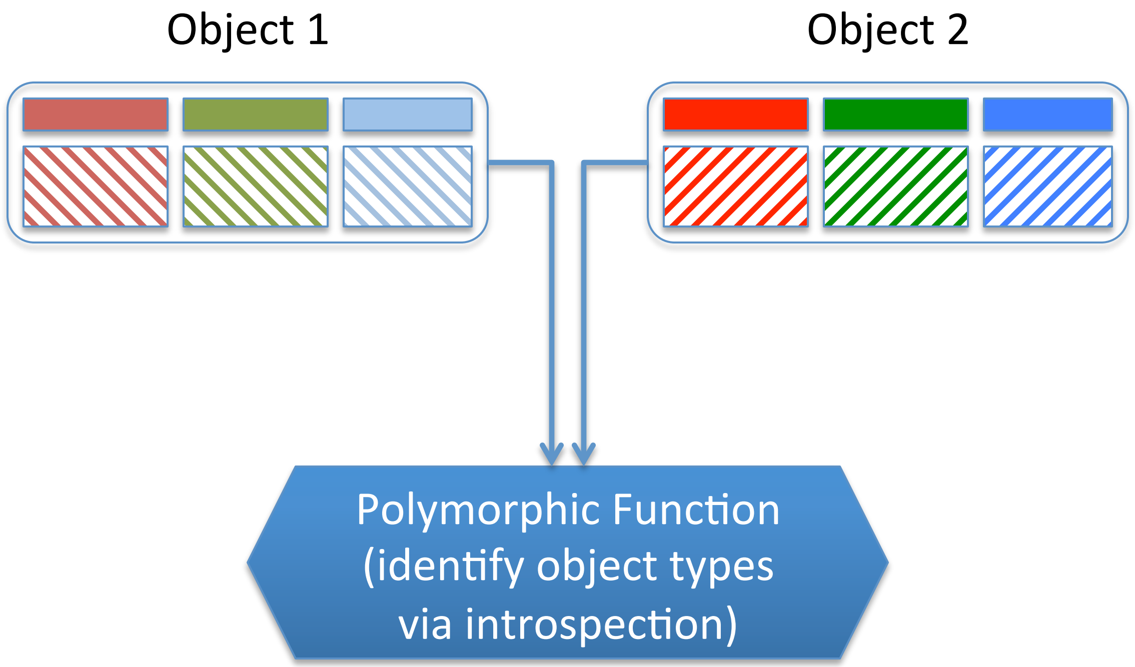
Class Diagrams

- Class diagrams are visual representations of the relationships among classes
  - They are similar in spirit to entity-relationship diagrams, unified modeling language, *etc* in that they help implementers in understanding and documenting application/library architecture
  - They are more useful when there are more classes and attributes
  - They are also very useful (along with documentation) when the code is unfamiliar

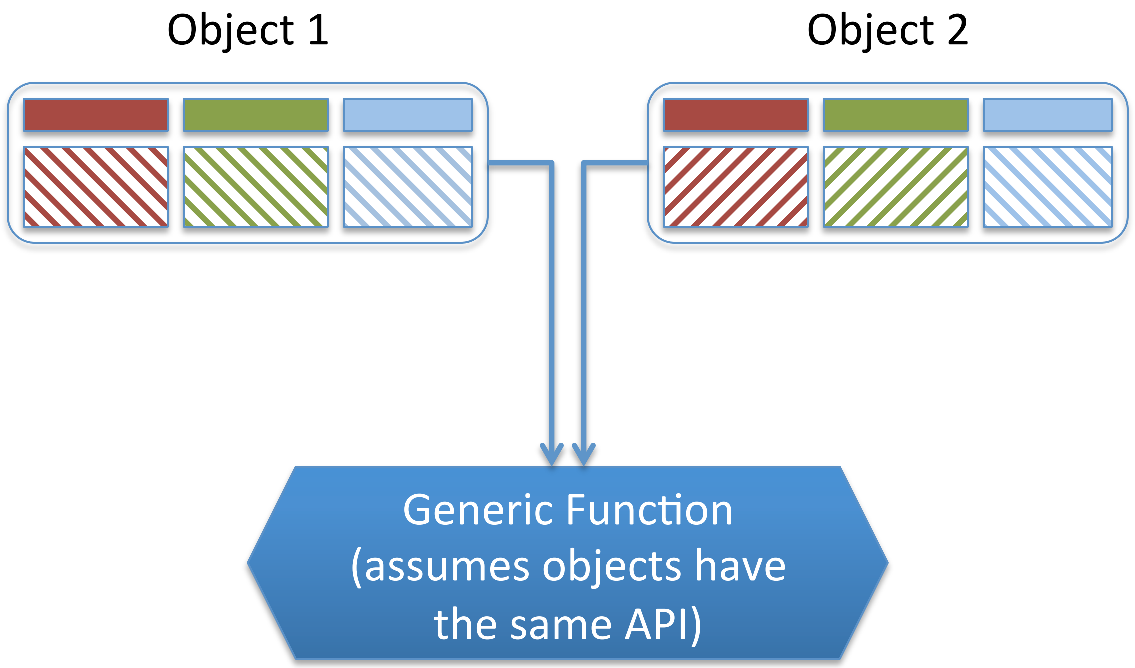
Polymorphism

- "Functions that can work with several types are called **polymorphic**." – Downey, *Think Python*
- "The primary usage of **polymorphism** in industry (object-oriented programming theory) is the ability of objects belonging to different types to respond to method, field, or property calls of the same name, each one according to an appropriate type-specific behavior. The programmer (and the program) does not have to know the exact type of the object in advance, and so the exact behavior is determined at run time (this is called **late binding** or **dynamic binding**)." - *Wikipedia*

Polymorphic Function



Polymorphic Classes



Polymorphism (cont.)

- The critical feature of polymorphism is a shared **interface**
  - Using the Downey definition, we present a common interface where the same function may be used regardless of the argument type
  - Using the Wikipedia definition, we require that polymorphic objects share a common interface that may be used to manipulate the objects regardless of type (class)

Polymorphism (cont.)

- Why is polymorphism useful?
  - By reusing the same interface for multiple purposes, polymorphism reduces the number of "things" we have to remember
  - It becomes possible to write a "generic" function that perform a particular task, eg sorting, for many different classes (instead of one function for each class)

Polymorphism (cont.)

- To define a polymorphic function that accepts multiple types of data requires the function either:
  - be able to distinguish among the different types that it should handle, or
  - be able to use other polymorphic functions, methods or syntax to manipulate any of the given types

Type-based Dispatch

- Python provides several ways of identifying data types:
  - **isinstance** function
  - **hasattr** function
  - **__class__** attribute

```
def what_is_this(data):
        if isinstance(data, basestring):
                # Both str and unicode derive
                # from basestring
                return "instance of string"
        elif hasattr(data, "__class__"):
                return ("instance of %s" %
                   data.__class__.__name__)
        raise TypeError(”unknown type: %s" %
                                str(data))

class NC(object): pass
class OC: pass

print what_is_this("Hello")
print what_is_this(12)
print what_is_this([1, 2])
print what_is_this({12:14})
print what_is_this(NC())
print what_is_this(OC())
```

Polymorphic Syntax

- Python uses the same syntax for a number of data types, so we can implement polymorphic functions for these data types if we use the right syntax

```
def histogram(s):
        d = dict()
         for c in s:
                 d[c] = d.get(c, 0) + 1
         return d

print histogram("aabc")
print histogram([1, 2, 2, 5])
print histogram(("abc", "abc", "xyz"))
```

Polymorphic Classes

- Classes that share a common interface
  - A function implemented using only the common interface will work with objects from any of the classes
- Although Python does not require it, a simple way to achieve this is to have the classes derive from a common superclass
  - To maintain polymorphism, methods overridden in the subclasses ***must*** keep the same arguments as the method in the superclass

Polymorphic Classes (cont.)

```
class InfiniteSeries(object):
        def next(self):
                raise NotImplementedError(”next")
class Fibonacci(InfiniteSeries):
        def __init__(self):
                self.n1, self.n2 = 1, 1
        def next(self):
                n = self.n1
                self.n1, self.n2 = self.n2, self.n1 + self.n2
                return n
class Geometric(InfiniteSeries):
        def __init__(self, divisor=2.0):
                self.n = 1.0 / divisor
                self.nt = self.n / divisor
                self.divisor = divisor
        def next(self):
                n = self.n
                self.n += self.nt
                self.nt /= self.divisor
                return n
def print_series(s, n=10):
        for i in range(n):
                print "%.4g" % s.next(),
        print
```

- The superclass defining the interface often has no implementation and is called an **abstract base class**
- Subclasses of the abstract base class override interface methods to provide class-specific behavior
- A generic function can manipulate all subclasses of the abstract base class

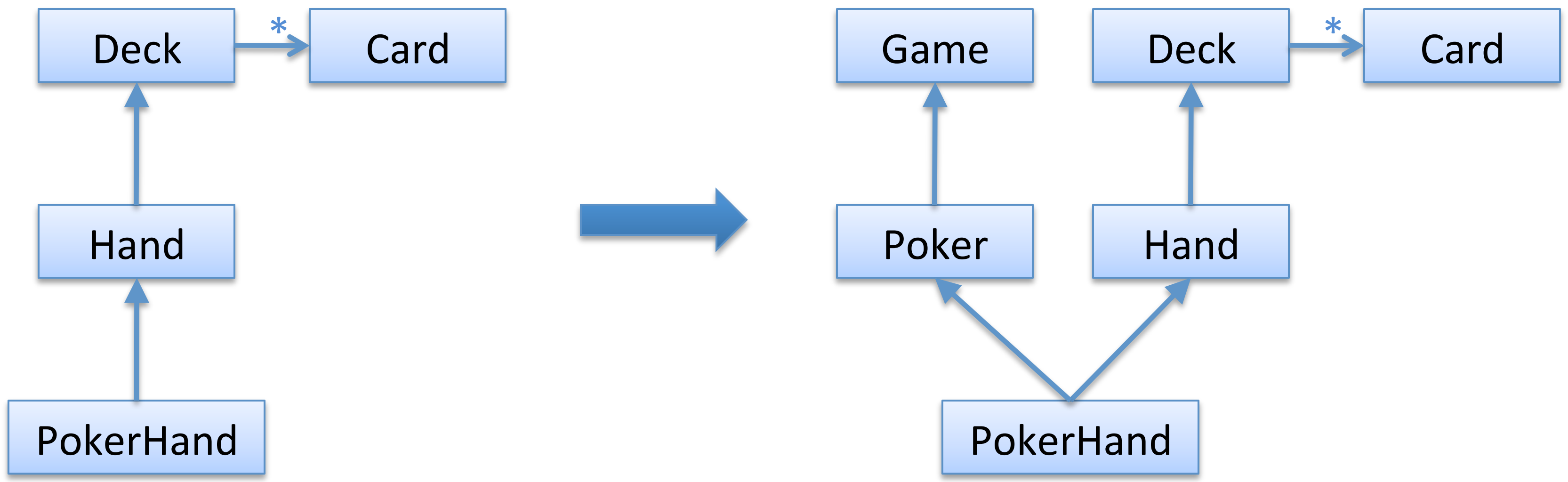
```
print_series(Fibonacci())
print_series(Geometric(3.0))
print_series(InfiniteSeries())
```

Polymorphic Classes (cont.)

- In our example, all three subclasses overrode the **next** method of the base class, so they each have different behavior
- If a subclass does ***not*** override a base class method, then it **inherits** the base class behavior
  - If the base class behavior is acceptable, the writer of the subclass does not need to do anything
  - There is only one copy of the code so, when a bug is found it the inherited method, only the base class needs to be fixed
- ***instance.method()*** is preferable over ***class.method(instance)***
  - Although the code still works, the explicit naming of a class in the statement suggests that the method is defined in the class when it might actually be inherited from a base class

Cards, *Decks* and *Hands*

- Class diagram of example in Chapter 18 and Exercise 18.6



Is More Complex Better?

- Advantages
  - Each class corresponds to a real concept
  - It should be possible to write a polymorphic function to play cards using only Game and Hand interfaces
  - It should be easier to implement other card games
- Disadvantages
  - More classes means more things to remember
  - Need multiple inheritance (although in this case it should not be an issue because the class hierarchy is simple)

Debugging

- Python is capable of **introspection**, the ability to examine an object at run-time without knowing its class and attributes *a priori*
- Given an object, you can
  - get the names and values of its attributes (including inherited ones)
  - get its class
  - check if it is an instance of a class or a subclass of a class
- Using these tools, you can collect a lot of debugging information using polymorphic functions

Debugging with Introspection

```
def tell_me_about(data):
	print str(data)
	print " Id:", id(data)
	if isinstance(data, basestring):
		# Both str and unicode
		# derive from basestring
		print " Type: instance of string"
	elif hasattr(data, "__class__"):
		print (" Type: instance of %s" %
				data.__class__.__name__)
	else:
		print " Type: unknown type"
	if hasattr(data, "__getitem__"):
		like = []
		if hasattr(data, "extend"):
				like.append("list-like")
		if hasattr(data, "keys"):
				like.append("dict-like")
		if like:
				print " %s" % ", ".join(like)
```

```
tell_me_about({12:14})
class NC(object): pass
nc = NC()
nc_copy = nc
tell_me_about(nc)
tell_me_about(nc_copy)
tell_me_about(NC())
```

```
{12: 14}
 Id: 5370941216
 Type: instance of dict
 dict-like
<__main__.NC object at 0x1401d6410>
 Id: 5370635280
 Type: instance of NC
<__main__.NC object at 0x1401d6410>
 Id: 5370635280
 Type: instance of NC
<__main__.NC object at 0x1401d6490>
 Id: 5370635408
 Type: instance of NC
```

More Introspection

```
def list_attributes(obj):
    for attr_name in dir(obj):
        print " %s:" % attr_name,
        value = getattr(obj, attr_name)
        if callable(value):
            print "function/method"
        else:
            print value
list_attributes(list)
```

Homework

- Assignment 8.1
- Assignment 8.2
  - A one-paragraph description is sufficient