Inheritance and Object-oriented Design

This reading is relatively new, and your feedback will help us improve it! If you notice mistakes (big or small), if you have questions, if anything is unclear, if there are things not covered here that you'd like to see covered, or if you have any other suggestions; please get in touch during office hours or open lab hours, or via e-mail at `6.101-help@mit.edu`.

Table of Contents

1) Introduction

In the last reading, we introduced the `class` keyword, which allows us to make new *types* of Python objects. While using classes is not strictly necessary (in the sense that any program that we can write using classes could also be written without them), they turn out to be a powerful organizational tool, and, as we saw last week, they also give us some neat ways to integrate our custom types into Python so that they can make use of built-in operators and functions.

In this reading, we'll build on what we learned last week and introduce several new features that we can make use of when defining classes of our own, and we hope to do this in a way that demonstrates the power of classes as an organizational tool.

We're going to start this reading with somewhat-small, somewhat-contrived examples that we'll use to review some of the features introduced in the last reading and also to introduce the idea of inheritance. Once we've walked through several examples like that, we'll pull back and take a look at using these ideas to organize a bigger, more-authentic program.

But we'll start with a small example, which should just be review from last time:

Show/Hide Line Numbers

```
1 x = "dog"
2
3 class A:
4     pass
5
6 a = A()
7 print(a.x)
```

Which of the following is printed when we run this program? Using the rules established in the last reading, draw an environment diagram to explain this result.

- ○ `dog`
- ○ `cat`
- ○ `ferret`
- ○ `tomato`
- ○ `fish`
- ○ `lemon`
- ○ None of the above (we get a different value instead)
- ○ None of the above (a `NameError` is raised)
- ○ None of the above (an `AttributeError` is raised)
- ○ None of the above (a `TypeError` is raised)
- ○ None of the above (some other exception is raised)

Now, let's consider a slight modification of the program above (where the body of the `A` class is `x = 'cat'` rather than `pass`):

Show/Hide Line Numbers

```
1 x = "dog"
2
3 class A:
4     x = "cat"
5
6 a = A()
7 print(a.x)
```

Think about the differences here. How will the environment diagram be different from what we saw above, and how will that affect the result?

Which of the following is printed when we run this program? Draw an environment diagram that explains this result.

- ○ `dog`
- ○ `cat`
- ○ `ferret`
- ○ `tomato`
- ○ `fish`
- ○ `lemon`
- ○ None of the above (we get a different value instead)
- ○ None of the above (a `NameError` is raised)
- ○ None of the above (an `AttributeError` is raised)
- ○ None of the above (a `TypeError` is raised)
- ○ None of the above (some other exception is raised)

2) Inheritance

Now that we've gone through a couple of examples to refamilizarize ourselves with some of last week's rules, let's introduce some new machinery. Last week, when we introduced attributes, we described the rules by which variables and attributes are looked up:

From last time...

To look up a variable:

1. look in the current frame first
2. if not found, look in the *parent* frame
3. if not found, look in that frame's parent frame
4. `...` (keep following that process, looking in parent frames)
5. if not found, look in the global frame
6. if not found, look in the builtins (where things like `print`, `len`, etc. are bound)
7. if not found, raise a `NameError`

To look up an attribute inside of an object:

1. look in the object itself
2. if not found, look in that object's class
3. if not found, look in that class's superclass
4. if not found, look in *that* class's superclass
5. `...` (keep following that process, looking in superclasses)
6. if not found and no more superclasses, raise an `AttributeError`

Our rules for attribute lookup involved not only looking in an instance and its class but possibly looking in other classes as well (specifically, the *superclass*, if any, of the class whose instance we're working with). But so far we've not yet shown any such relationships in our environment diagrams, nor talked about the practical effects of that relationship on our code, nor how we

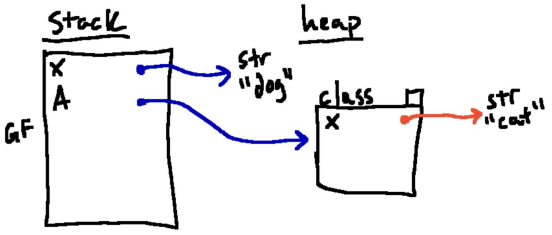
can leverage this when designing programs. But don't fret, because that's exactly the focus of this reading! Let's go ahead and jump right in to an example, which uses some syntax that may or may not be familiar from 6.100A:

Show/Hide Line Numbers

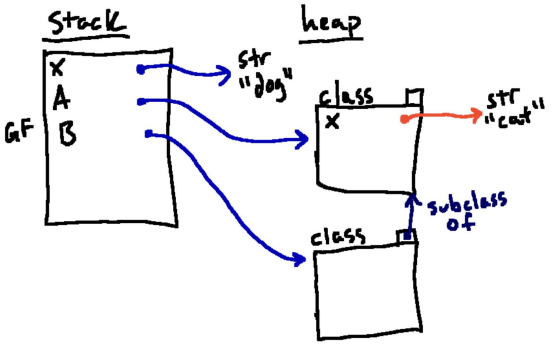
```
1  x = "dog"
2
3  class A:
4      x = "cat"
5
6  class B(A):
7      pass
8
9  b = B()
10 print(b.x)
```

Note that here we've left the definition of `A` unchanged. But when we defined `B`, we indicated (by way of the `A` in parentheses next to it) that our new class should be, as we say, a *subclass* of the class called `A`. Let's see how this relationship plays out in our environment diagrams by way of example.

Up through line 5, our diagram is exactly the same as it has been for the examples we saw earlier:

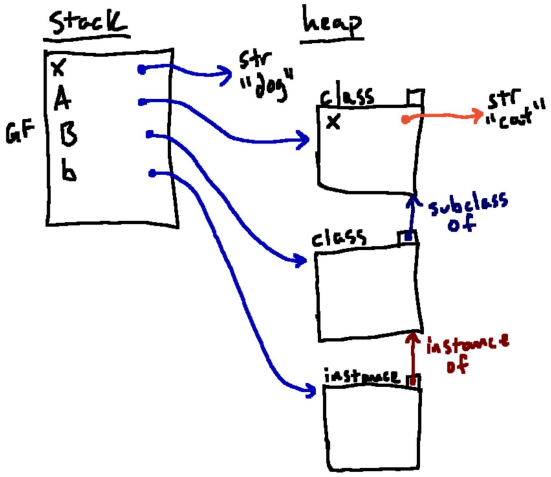


But things change when we define `B` starting on line 6. When Python sees this class definition, it will make a new class (like before), but it will also store away, as part of that class object, a reference to the class that it is a subclass of. In our diagrams, we'll draw that relationship in the following way:



What this means is that, if we're ever looking for an attribute or method inside of this class (either because we looked directly in the class or because we started by looking in an instance of the class and failed to find it there), if that attribute doesn't exist in the class, we will continue looking in its *superclass* (the class it's a subclass of). In that way, every attribute we define in `A` that isn't also defined in `B` will be accessible from `B`. We say that `B` "inherits" attributes and methods from `A`, and this relationship is generally referred to as *inheritance*.

So as we continue through our example, the next step is line 9, where we make an instance of the class called `B` and associate it with the name `b`:



Then we look up `b.x`. In so doing, Python starts by evaluating the name `b`, finding our new instance. So we look there for an attribute called `x`. Not finding one, we follow the arrow and look in `B`. We still don't find it there, so we look in `A`, where we find `"cat"`, so that's what `b.x` evaluates to (and, thus, what is printed to the screen).

3) More Examples: Environment Diagrams

That's it for the rules we're going to introduce for working with classes! But now that we've seen those rules play out in a few small examples, let's get some additional practice by carrying them out on a few more programs.

For each of the pieces of code below, try drawing an environment diagram and using it to explain what happens when the code is run. For each, the differences from the previous piece of code are highlighted in yellow.

Here's our first example, a slight change from the code above:

Show/Hide Line Numbers

```
x = "dog"

class A:
    x = "cat"

class B(A):
    x = "ferret"

b = B()
print(b.x)
```

Use an environment diagram to model the execution of this program. Which of the following is printed when we run this program?

- ○ `dog`
- ○ `cat`
- ○ `ferret`
- ○ `tomato`
- ○ `fish`
- ○ `lemon`
- ○ None of the above (we get a different value instead)
- ○ None of the above (a `NameError` is raised)
- ○ None of the above (an `AttributeError` is raised)
- ○ None of the above (a `TypeError` is raised)
- ○ None of the above (some other exception is raised)

3.1) Adding an `__init__` Method

Now let's add an initializer to the `B` class:

Show/Hide Line Numbers

```
1  x = "dog"
2
3  class A:
4      x = "cat"
5
6  class B(A):
7      x = "ferret"
8      def __init__(self):
9          self.x = "tomato"
10
11 b = B()
12 print(b.x)
```

Remember that when we make a new instance of a class, Python will look up the name `__init__` in that new instance according to our normal attribute-lookup rules; and if it finds a method called `__init__`, it will implicitly call that method with our new instance passed in as the first argument.

Try stepping through an environment diagram for this program on your own, and use it to answer the following question:

Use an environment diagram to model the execution of this program. Which of the following is printed when we run this program?

- ○ `dog`
- ○ `cat`
- ○ `ferret`
- ○ `tomato`
- ○ `fish`
- ○ `lemon`
- ○ None of the above (we get a different value instead)
- ○ None of the above (a `NameError` is raised)
- ○ None of the above (an `AttributeError` is raised)
- ○ None of the above (a `TypeError` is raised)
- ○ None of the above (some other exception is raised)

Now let's try a few different variations on this theme, making some small changes to the `__init__` method. Next, let's look at the following, which only contains a small syntactical difference compared to our last example:

Show/Hide Line Numbers

```
1  x = "dog"
2
3  class A:
4      x = "cat"
5
6  class B(A):
7      x = "ferret"
8      def __init__(self):
9          x = "tomato"
10
11 b = B()
12 print(b.x)
```

Use an environment diagram to model the execution of this program. Which of the following is printed when we run this program?

- ○ `dog`
- ○ `cat`
- ○ `ferret`
- ○ `tomato`
- ○ `fish`
- ○ `lemon`
- ○ None of the above (we get a different value instead)
- ○ None of the above (a `NameError` is raised)
- ○ None of the above (an `AttributeError` is raised)
- ○ None of the above (a `TypeError` is raised)
- ○ None of the above (some other exception is raised)

Let's try one more example with `A` and `B` before we add in another wrinkle. Consider the following change to `B.__init__`:

Show/Hide Line Numbers

```
1  x = "dog"
2
3  class A:
4      x = "cat"
5
6  class B(A):
7      x = "ferret"
8      def __init__(self):
9          self.x = x
10
11 b = B()
12 print(b.x)
```

Carefully use an environment diagram to model the execution of this code. What is printed to the screen when we run the code?

- ○ `dog`
- ○ `cat`
- ○ `ferret`
- ○ `tomato`
- ○ `fish`
- ○ `lemon`
- ○ None of the above (we get a different value instead)
- ○ None of the above (a `NameError` is raised)
- ○ None of the above (an `AttributeError` is raised)
- ○ None of the above (a `TypeError` is raised)
- ○ None of the above (some other exception is raised)

3.2) Adding Another Class

Now for our last couple of examples, we'll introduce a third class into our program:

Show/Hide Line Numbers

```
1  x = "dog"
2
3  class A:
4      x = "cat"
5
6  class B(A):
7      x = "ferret"
8      def __init__(self):
9          self.x = x
10
11 class C(B):
12     x = "fish"
13
14 c = C()
15 print(c.x)
```

Use an environment diagram to model the execution of this code (how will this new class `C` manifest in the diagram?) and use it to answer: What is printed to the screen when we run the code?

- ○ `dog`
- ○ `cat`
- ○ `ferret`
- ○ `tomato`
- ○ `fish`
- ○ `lemon`
- ○ None of the above (we get a different value instead)
- ○ None of the above (a `NameError` is raised)
- ○ None of the above (an `AttributeError` is raised)
- ○ None of the above (a `TypeError` is raised)
- ○ None of the above (some other exception is raised)

And as one last example before moving on, let's consider what happens when we give `C` its own `__init__` method (in this case, an empty one).

Show/Hide Line Numbers

```
x = "dog"

class A:
    x = "cat"

class B(A):
    x = "ferret"
    def __init__(self):
        self.x = x

class C(B):
    x = "fish"
    def __init__(self):
        pass

c = C()
print(c.x)
```

Use an environment diagram to model the execution of this code and use it to answer: What is printed to the screen when we run the code?

- ○ `dog`
- ○ `cat`
- ○ `ferret`
- ○ `tomato`
- ○ `fish`
- ○ `lemon`
- ○ None of the above (we get a different value instead)
- ○ None of the above (a `NameError` is raised)
- ○ None of the above (an `AttributeError` is raised)
- ○ None of the above (a `TypeError` is raised)
- ○ None of the above (some other exception is raised)

4) Example: Drawing with Shapes

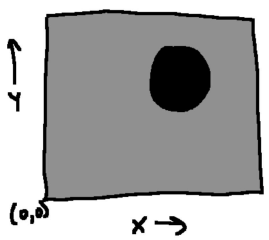
In the past several sections, we've been looking at small examples to get a feel for what is going on "behind the scenes" when we write code that uses inheritance. That knowledge is critical, particularly when unexpected things happen in code that makes use of these features. But so far, we have not described *why* or *when* we might want to use these features in our own programs. In this section, we'll talk through the construction of a more authentic program, with the goal of further exploring the question of how we can use these ideas to organize our code so as to manage complexity in a large program.

The context we'll use to explore this idea is a drawing library, which we'll use to augment the kinds of capabilities we developed in labs 1 and 2 to enable drawing complex shapes. We'll go about this by implementing representations for various shapes in Python, and we'll build in a functionality for drawing those shapes onto images so that we can display them to the screen or save them as image files, etc.

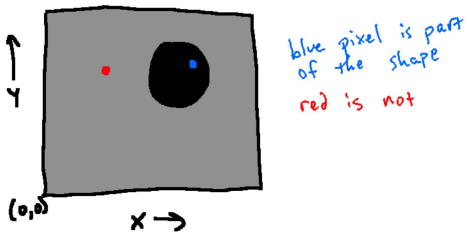
We've made a code skeleton available here: `shapes.py`, which you can use to follow along on your own machine if you want to. There are some helper functions up near the top of that code for working with our earlier image representation, and we'll use those as we work through building up our representation of shapes. Our main focus is not going to be on that code but rather on our representation for shapes and on the organization of the code used to realize that representation.

The way we'll define a shape in this context is as an arbitrary collection of pixels, and the main question we're going to need to be able to ask of any shape is: here is a pixel location (x, y); is that a part of this shape or not?

For example, we have an image where $(0, 0)$ is in the bottom left, x increases to the right and y increases upward. Within that image, we want to represent a shape, say a circle (like the one drawn in black below):



The way we'll define this circle is by asking which pixels are part of the shape and which aren't:



We're going to represent shapes using classes in our code, and the way that we'll implement the question above is as a method `__contains__`. `__contains__` is one of the special "dunder" methods discussed in the last reading, and it's used to implement the `in` keyword for custom types. So by making sure that every shape has a `__contains__` method, we can then say something like `(x, y) in s` where `s` is an instance of one of our shape classes, and Python will automatically interpret that as `s.__contains__((x, y))`.

We also want to be able to ask any shape what pixel value is at its center, so we'll set things up so that, for any shape `s`, `s.center` is always an `(x, y)` tuple representing the (x, y) location of that shape's center point.

And, finally, we want to be able to draw these shapes onto images, so we will define an additional method `draw` such that for any shape `s`, `s.draw(im, color)` will draw `s` onto the given image `im` (in our lab 2 format) in the specified `color` (as an `(r, g, b)` tuple).

Our starter code sets up this kind of structure:

```
class Shape:
    """
    Represents a 2D shape that can be drawn.

    All subclasses MUST implement the following:

    __contains__(self, p) returns True if point p is inside the shape
    represented by self

        Note that "(x, y) in s" for some instance of Shape
        will be translated automatically to "s.__contains__((x, y))"

    s.center should give the (x,y) center point of the shape

    draw(self, image, color) should mutate the given image to draw the shape
    represented by self on the given image in the given color
    """

    def __contains__(self, p):
        pass

    def draw(self, image):
        pass
```

4.1) Circles and Rectangles

Let's start by implementing two particular kinds of shapes: circles and rectangles. We'll do this by implementing two subclasses of `Shape`:

```
class Circle(Shape):
    pass
```

```
class Rectangle(Shape):
    pass
```

Check Yourself:

Let's think through how we can implement `Circle`. In particular, think about:

- what information do we need to store to represent a circle?
- (we'll come back to this later, but...) how can we use that information to implement `__contains__` and `center` and `draw`?

Show/Hide

Check Yourself:

Similarly, let's think about rectangles from the same kind of lens. How can we represent rectangles?

Show/Hide

4.2) `__contains__` and an Organizational Principle

Now that we have representations built up for rectangles and circles, we can go ahead and start thinking about implementing `__contains__`. One place where we might start is by thinking that, since both `Circle` and `Rectangle` are subclasses of `Shape`, they both inherit the `__contains__` method from `Shape`. So we can fill in the definition of `__contains__` in the `Shape` class while leaving `Circle` and `Rectangle` unchanged, something like:

```
class Shape:
    def __contains__(self, p):
        if isinstance(self, Circle):
            # do some circle-y computations
            # based on self.radius and self.center
        elif isinstance(self, Rectangle):
            # do some rectangle-y computations
            # based on self.lower_left, self.width, and self.height

    def draw(self, image):
        pass


class Circle(Shape):
    def __init__(self, center, radius):
        self.center = center
        self.radius = radius


class Rectangle(Shape):
    def __init__(self, lower_left, width, height):
```

```
        self.lower_left = lower_left
        self.width = width
        self.height = height
```

It's worth mentioning that this structure *can work*, i.e., it is technically OK to implement things this way (if we have an instance of `Circle` or `Rectangle` and we look for a `__contains__` method within it, then Python will eventually find the one implementation of `__contains__` in the `Shape` class).

But we're going to recommend **avoiding** this kind of code (specifically, code that does explicit type checking like this), for a few reasons.

The main reason has to do with thinking ahead and imagining that we may want to expand our library beyond just circles and rectangles in the future. Currently, if we think about *adding a new kind of shape*, then we need to modify code in multiple places (we need to make a new class and make sure it's set up with the right attributes, but then we also need to jump up to the `Shape` class to add a new conditional to the `__contains__` method, and we need to make sure that those two places agree!). Not only that, but if we continue adding shapes, this `__contains__` method is going to get really big and really complicated, which would make it a good place for bugs to hide.

It would be nice if adding a new shape were easier than that, i.e., if everything that it meant to be a *particular kind of shape* existed in one place (in the subclass). So organizationally, what we're going to try to accomplish here is:

- only things that are *general to all shapes* are defined in the `Shape` class, and
- everything that is *specific to a certain kind of shape* is defined in a subclass for that kind of shape.

Hopefully we'll see over the remainder of this reading that this approach can help us keep things relatively small, relatively simple, and relatively easy to debug.

Despite this goal, though, we *do* need a way to differentiate these behaviors. However, we're going to take advantage of the rules we've been learning over this reading and the last one to avoid writing our own explicit type checks, effectively making Python do those checks for us using the built-in machinery we've already seen. And an effective way to do this is to define `__contains__` within each subclass:

```
class Shape:
    def __contains__(self, p):
        raise NotImplementedError("Subclass of Shape didn't define __contains__")

    def draw(self, image):
        pass


class Circle(Shape):
    def __init__(self, center, radius):
        self.center = center
        self.radius = radius

    def __contains__(self, p):
        # do some circle-y computations
        # based on self.radius and self.center


class Rectangle(Shape):
    def __init__(self, lower_left, width, height):
        self.lower_left = lower_left
        self.width = width
        self.height = height

    def __contains__(self, p):
```

```
        # do some rectangle-y computations
        # based on self.lower_left, self.width, and self.height
```

This will still work. If we have an instance in-hand, remember that that instance knows what class it's an instance of. So if we have an instance of `Circle`, then looking up `__contains__` inside of it will find `Circle`'s `__contains__` method, and similarly for other kinds of shapes. But here, the win is an organizational one: everything that it means to be a `Circle`, for example, is all in one place! With this approach, Python does the type checking for us implicitly instead of explicitly, in a way that keeps our code cleaner and more modular (which will make understanding, testing, debugging, and expanding the program easier!).

One could also argue that there's an efficiency win here as well. In our original formulation (with explicit type checking in `Shape.__contains__`), figuring out which block of code to execute required checking against multiple types one after the other. But in this formulation (with `__contains__` defined in each subclass), we don't need to do that; each instance knows right away which `__contains__` method is the right one to use. That's not a big win but is perhaps worth mentioning (though again, the real advantage here is in terms of organization and code clarity).

Notice that we've also left something in `Shape.__contains__`. The idea here is that every time we define a new shape, it's going to take care of its own `__contains__`; so we should never call `Shape.__contains__` directly. So we're doing a little bit of "defensive programming" here by having that method raise an exception indicating that we're expecting each subclass to implement `__contains__` for itself (so that if we forget it when implementing a new kind of shape, Python will let us know that we've forgotten!).

4.2.1) Implementing `__contains__`

Now let's go ahead and implement `__contains__` for these two shapes. Try this on your own and write out some test cases for yourself, before looking at our solutions below:

Show/Hide

4.3) `draw` and Another Organizational Principle

Now that we have `__contains__` defined, we can think about implementing `draw`, which should make testing a little bit easier (since it will allow us to start making images!). Remember that `draw` should take an image and a color (both in our lab 2 representation), and it should mutate the given image by drawing our shape in the appropriate color.

To start, let's try organizing this the same way as we did for `__contains__`, by implementing `draw` in each subclass.

4.3.1) `Circle.draw`

Let's start by thinking about how to implement this for the `Circle` class. Try to write `draw` for the `Circle` class before looking at the discussion below:

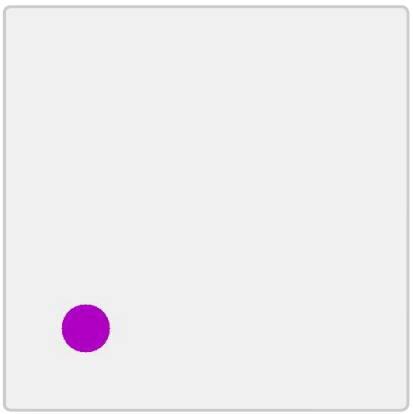
Show/Hide

Now with that implemented, we can try drawing something! Let's try drawing a little circle, setting up some code to make it easy to draw more shapes later on:

```
out_image = new_image(500, 500)

shapes = [
```

```
    (Circle((100, 100), 30), COLORS['purple']),
]

for shape, color in shapes:
    shape.draw(out_image, color)

save_color_image(out_image, "test.png")
```

And we see the following result:



Hooray! It looks like our little library is working so far (and hopefully yours is, too, on your own machine!) 🎈🥳🎊

But drawing circles was only part of the goal, so let's continue on and add `draw` functionality to our rectangles as well!

4.3.2) `Rectangle.draw`

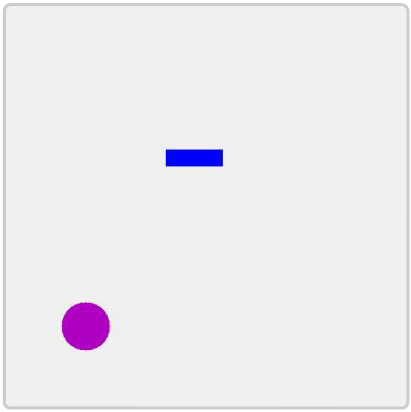
Now let's try to implement `Rectangle.draw`. Try to implement it on your own before looking at the implementation below:

Show/Hide

Having implemented this, we can try out our code by trying to draw a rectangle as well:

```
out_image = new_image(500, 500)

shapes = [
    (Circle((100, 100), 30), COLORS['purple']),
    (Rectangle((200, 300), 70, 20), COLORS['blue']),
]

for shape, color in shapes:
    shape.draw(out_image, color)

save_color_image(out_image, "test.png")
```

Running this code results in the following image:



Double hooray! 🎈🥳🎪🎈🥳🎪

4.3.3) Refactoring

Unfortunately, however, if we take a second look at the code that we've just written, our mood may start to dampen somewhat. The code is working, it's true (and that's great!), but the way that we've just implemented `Rectangle.draw` (and the resulting code) is the kind of thing that may make us feel uneasy at this point in the term. To drive the point home, let's look at the two functions next to each other:

```
class Circle:
    # ...
    def draw(self, image, color):
        for x in range(image['width']):
            for y in range(image['height']):
                if (x, y) in self:  # if self.__contains__((x, y)):
                    set_pixel(image, x, y, color)

class Rectangle:
    # ...
    def draw(self, image, color):
        for x in range(image['width']):
            for y in range(image['height']):
                if (x, y) in self:  # if self.__contains__((x, y)):
                    set_pixel(image, x, y, color)
```

These functions are *identical*! In general, duplicating complex code like this is just inviting bugs into your code; it's giving them lots of place to hide and making them harder to find and squash when something goes wrong.

But even more than that, `draw` as implemented now is completely general. It will work not just for squares or for rectangles but for *any* shape that knows how to do a containment check (via a `__contains__` method).

What we're going to do to remedy this is to move this method (which is completely general) out of the subclasses (which are intended to contain only those things that are specific to a given type of shape) and lift it into the `Shape` class instead:

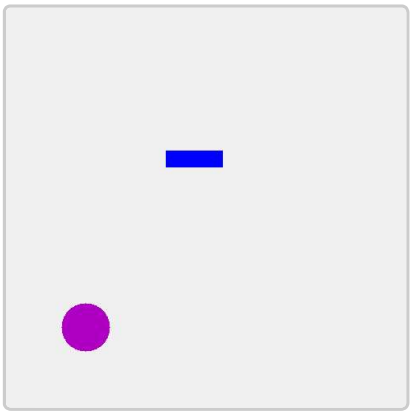
Show/Hide

Once we've done this, any time we look for a `draw` method inside of any subclass of `Shape`, we won't find draw there, but we will chain up to look in `Shape`, where we will find this one method.

Something may seem a little bit weird about this, which is that we've defined `draw` inside of the `Shape` class, but *inside* of `draw`, it's calling out to a `__contains__` method. So will we get the `NotImplementedError` we raised from `Shape.__contains__` earlier?

It turns out that this will work just fine! In general, the way we're going to get to the point of calling `draw` is by way of an instance of `Circle` or `Square` or some other shape. So inside of `draw`, the name `self` is going to refer to an instance of one of those subclasses. Thus, when I look up `self.__contains__`, Python will find *the right `__contains__` method* for whatever kind of shape we were calling `draw` from. This may feel a little bit strange at first, but this idea (moving common pieces of code into a method in the superclass that then calls out to specific methods defined in subclasses) is a very common and very powerful way of organizing things to avoid duplicate code.

Now that we've made that change, it's a good idea to test things again to make sure that, after our refactoring, we're still getting the image we expect to get. And, indeed, we do:



So, double hooray for real this time, because not only do we have working code, but there's something quite nice about this code from a stylistic/organizational perspective as well. 🎈🥳🎊🎈🥳🎊

4.4) `center` and the `@property` Decorator

We can take a moment to celebrate what we've done so far, but let's not rest on our laurels for too long. There's still another part of the specification that we've not addressed yet: the `center` attribute.

Conveniently, our `Circle` class already has this done since we're storing away the center as part of the class definition! But we need to make sure to implement it for `Rectangle` as well.

One way that we could do this is familiar: we could simply add another attribute at initialization time, like so:

```
class Rectangle(Shape):
    def __init__(self, lower_left, width, height):
        self.lower_left = lower_left
        self.width = width
        self.height = height
        self.center = (lower_left[0] + width//2, lower_left[1] + height//2)
```

And perhaps that's the right thing to do (it all depends on your choice of implementation!). But let's imagine for a moment that I wanted to allow my shapes to be *mutable*, i.e., I wanted to be able to change `r.lower_left` for some `Rectangle` instance. In that case, we kind of have a problem here, in that moving the lower-left corner should also affect the center, but the code above won't do that!

In cases like this where we have related values stored in our instance, one way to avoid the issue would be to define `center` as a method instead of as an attribute:

```
class Rectangle(Shape):
    def __init__(self, lower_left, width, height):
        self.lower_left = lower_left
```

```
            self.width = width
            self.height = height

        def center(self):
            return (self.lower_left[0] + self.width//2, self.lower_left[1] + self.height//2)
```

This kind of a structure would fix the problem from above, in that if we changed `r.lower_left`, then `r.center()` would adjust appropriately! But it has one downside, which is it doesn't match our specification anymore. Our spec said that `r.center` should be a tuple, but with this code we would need to write `r.center()` instead.

But Python gives us a way around this, by way of the `@property` decorator. If we write `@property` just above that method's definition:

```
class Rectangle(Shape):
    def __init__(self, lower_left, width, height):
        self.lower_left = lower_left
        self.width = width
        self.height = height

    @property
    def center(self):
        return (self.lower_left[0] + self.width//2, self.lower_left[1] + self.height//2)
```

then when we say `r.center` (without the round brackets), Python will automatically call that method and give us the result (thus giving us a way to meet the given specification even with our dynamically computed center value!).

It turns out that this syntax with the `@` sign is more general than this (and we'll talk more about that in the next reading), but for now it's fine to treat `@property` as a special piece of syntax.

It's perhaps also worth mentioning that there is an equivalent operation for setting a value in such a way that it has a dynamic effect. So if we wanted to be able to say `r.center = (2, 3)`, for example, but `r.center` was dynamically computed as a `@property`, we could write something like the following, which would adjust the `lower_left` attribute so that `center` would have the given value:

```
class Rectangle(Shape):
    def __init__(self, lower_left, width, height):
        self.lower_left = lower_left
        self.width = width
        self.height = height

    @property
    def center(self):
        return (self.lower_left[0] + self.width//2, self.lower_left[1] + self.height//2)

    @center.setter
    def center(self, value):
        self.lower_left = (value[0] - self.width//2, value[1] - self.height//2)
```

4.5) Adding Another Shape

Now let's build on things a little bit by adding another new shape: a square! We'll implement this as a new class `Square`, and all instances of `Square` need to support all of the same operations as our other shapes.

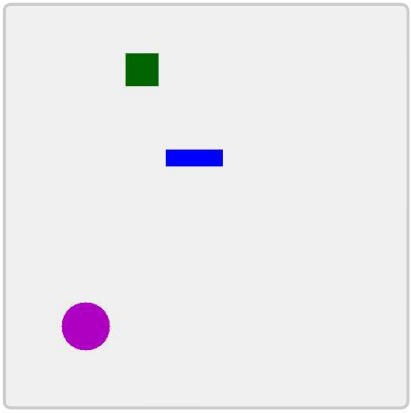
Take a moment to think about `Square`, and to try to implement it for yourself. What should `Square`'s superclass be? What arguments should its initializer take as input? What methods do we need to rewrite?

Show/Hide

With that code written, we can test again:

```
out_image = new_image(500, 500)

shapes = [
    (Circle((100, 100), 30), COLORS['purple']),
    (Rectangle((200, 300), 70, 20), COLORS['blue']),
    (Square((150, 400), 40), COLORS['green']),
]


for shape, color in shapes:
    shape.draw(out_image, color)

save_color_image(out_image, "test.png")
```

and we see the following image:

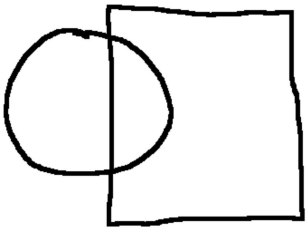


4.6) Shape Combinations

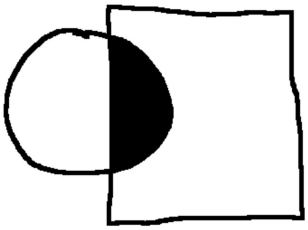
We're nearing the end of our adventure with drawing shapes, but let's explore just a few more examples first.

So far, we've defined a few basic shapes. But one of the things we talked about as being powerful, generally, when talking about complex systems is an ability to *combine primitive pieces together to make more complex pieces*. So let's spend a little bit of time thinking about some *combinations* of shapes we could implement.

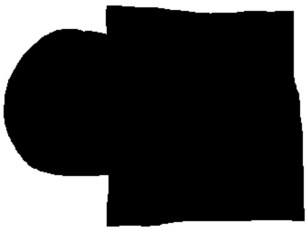
Let's imagine that I have two shapes, a circle and a square, that overlap, like so:



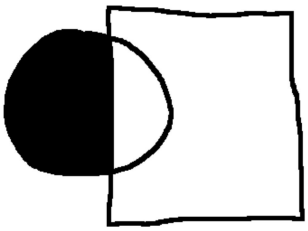
Given these two shapes, we might think about ways that we could combine them together to make new shapes. For example, I could think about the *intersection* of the circle and square as being a new shape containing only pixels that exist in both shapes:



The *union* of the circle and square is a new shape containing all the pixels that are in either (or both) of the original shapes:



And the *difference* between the shapes (e.g., the circle minus the square) might be a new shape with all pixels that are in the circle but not the square:



So let's think about implementing new classes to represent these kinds of combinations.

4.6.1) Union

Let's start by implementing a new kind of shape that represents the *union* of two shapes. Think for a little while about how we might implement this: What inputs should it take at initialization time? What attributes should it store? How and where should we implement `__contains__`?

Give it a try on your own before looking below:

Show/Hide

4.6.2) Intersection and Difference

Let's go ahead and implement a class to represent an intersection as well. Once again, think about this (and try to implement it) before looking below:

Show/Hide

And, while we're at it, try writing code for a difference as well. One solution is below:

Show/Hide

4.6.3) Refactoring

The above code works! It allows us to use code like the following:

```
out_image = new_image(500, 500)
c1 = Circle((250, 250), 100)
c1 = Difference(c1, Circle((220, 280), 20))
c1 = Difference(c1, Circle((280, 280), 20))
c1 = Difference(c1, Difference(Circle((250, 250), 80), Rectangle((0, 250), 500, 500)))
c1.draw(out_image, COLORS['grey'])
save_color_image(out_image, "test.png")
```

to make the following image:

But you may have noticed something about the implementations above. Once again, there's a lot of repeated code between them! There are ways to improve `__contains__` as well, but for now, let's focus on `__init__`, which is *identical* between all three of these classes.

Before, when we noticed this kind of similarity, we moved the offending function up into the `Shape` class so that everyone could inherit from it. Does that make sense here?

Unfortunately, no! Remember that our `Shape` class is the place where we put information and behaviors that are shared between *all* shapes. While these three combinations share something between the three of them (the fact that they're made up of two shapes), that property is not shared by all shapes! So what are we to do?

Well, noticing this similarity, we can create a new class for ourselves representing a combination! Since a combination is a type of shape, it can inherit from our `Shape` class; and since `Intersection`, `Union`, and `Difference` are all combinations, they can all inherit from this new class. This gives us a place to put behaviors that are common between all of these combinations but not common between all shapes.

Try thinking about how to structure this (and maybe implementing it for yourself) before looking at our code below:

Show/Hide

Again, this approach saves us a lot of repeated code! The general approach we're advocating here involves looking for common behaviors between classes and moving those general behaviors into superclasses (in whole or in part), while letting the subclasses worry only about behaviors and information that are specific to them.

4.7) Optional Extra Challenges

Hopefully this has been a useful example, demonstrating not only a little bit about how classes work but also how we can use them (and, in particular, use inheritance) to help us avoid repetitious or overly complex code. We'll leave that last example as our ending point for the day, but there are lots of things that we could still do to improve on this code! If you have the time and interest, it might be fun to tackle any of the following additions/improvements (or others of your own devising):

- Even though we were able to move some redundant code from `Union`, `Intersection`, and `Difference` into the `Combination` class, there is still a lot of similarity in their `__contains__` methods. Try finding a way to move the common behavior from their `__contains__` methods into the `Combination` class without resorting to explicit checking of types anywhere.
- Add support for built-in operations to create combinations. For example, `s1 | s2` could result in `Union(s1, s2)`, and we could do similar things for the other combinations. How can we accomplish this, and in which classes should the associated methods be implemented?
- Make more kinds of primitive shapes (how could you implement a triangle? an octagon? etc?).
- Make a new kind of shape representing an *outline* of a given shape, so that `Outline(s, w)`, for example, would be a new shape that contains pixels that are within `w` pixels of the *edge* of an abitrary shape `s`. This can be done purely in terms of `s.__contains__`, without needing to store any additional information in `s`.
- Implement one or more "transformations" of shapes, for example:
  - `Scaled(s, n)` could be a version of `s` scaled up in size by a factor of `n`.
  - `Rotated(s, d)` could be a version of `s` rotated by `d` degrees about its center.
  - `Translated(s, dx, dy)` could be a version of `s` moved through space by `dx` pixels horizontally and `dy` pixels vertically.
- Use the shapes library you've written to draw cool pictures!
- Our `draw` code is inefficient because it iterates over every pixel of an image to find which pixels we should draw for a specific `Shape`. Typically, the shapes are much smaller than the entire image. Implement the notion of a *bounding box*, namely the smallest `Rectangle` that includes all the pixels of a `Shape`. With this, re-write `draw` to make use of the bounding box so that only the pixels within it need to be scanned to see which are actually in the shape and thus which need to be drawn onto the image.

If you take on any of these tasks, we'd be interested to hear about them (and/or to help you get them to work if you're having trouble!).

5) Summary

Per usual, we've covered a lot of ground in this reading. Our main goal for today was to introduce the notion of *inheritance*, by which classes can share methods.

We started by approaching this from a very low level, looking at small, contrived examples to try to develop an intuition for how this new machinery works.

Then we pulled back and looked at inheritance as an *organizational tool* in the context of a real program. In particular, we looked at leveraging inheritance to reduce the amount of redundant code in our programs to make them clearer and more extensible.

As you're thinking about applying these ideas to your own programs, many of these ideas are the kind of thing for which some amount of prior planning can be done (and it's always a good idea!). We can think ahead-of-time about what classes we're going to implement, the relationships between them, the operations they support, etc. But some of these things may also be

things that you notice *as you're writing code*. If that happens, don't be afraid to go back and refactor your code; a little bit of time refactoring now can often save a lot of time debugging later on! Over time, and with more practice, you'll be better able to recognize ahead-of-time what structures are going to work well and what ones aren't.