Custom Types and the Environment Model

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

We're not going to leave recursion behind, per se, but our focus will shift a little bit as we move forward. For the

next couple of weeks, we're going to be talking about *classes*, which provide a mechanism for making our own custom types; and about the mechanisms Python gives us for integrating those custom types tightly into the Python language.

We'll start by talking about how classes fit into the environment model that we've been developing over the term, to understand something about how classes are implemented and about how they work "behind the scenes" so that we can bring that knowledge to bear and use these features in our own programs. Then we'll end our discussion by walking through a concrete example of creating a custom type from start to finish.

2) The Power of Abstraction Revisited

In an earlier reading, we introduced the notion of *abstraction* as a means of controlling complexity in our programs. It may go without saying, but thinking about complicated systems is complicated, and thinking about simpler systems is often simpler. So as we move to analyzing or designing bigger and more complex systems, it is important to have strategies for breaking those complex systems down into simpler pieces that are easier to understand. In the reading about functions, we introduced a framework for thinking about complex systems that involved thinking about:

- **Primitives**: What are the smallest and simplest building blocks of which the system is comprised?
- Means of **Combination**: How can we combine these building blocks together to build more complicated compound structures?
- Means of **Abstraction**: What mechanisms do we have for treating those complicated compound structures as building blocks themselves?

We start by thinking about *primitives*: what are the things that are built into the system that we have to work with? Then, we can think about how we can take those pieces and combine them to make more complicated things. Then, finally, the real power comes from *abstraction*: the idea that we can take one of these (arbitrarily complex) pieces, draw a box around it, give it a name, and then treat it as though it were a primitive (combining it with other pieces, abstracting those away, etc.).

When we first introduced this idea, it was in the context of functions, and we looked at an example in terms of *operations* that we could ask Python to perform.

In that context, we could think about *primitive* operations, including arithmetic (`+`, `*`, etc.), comparisons (`==`, `!=`, etc.), Boolean operators (`and`, `or`, etc.), built-in functions (`abs`, `len`, etc.), and more. Then we can *combine* those things together, using things like conditionals (`if`, `elif`, `else`), looping structures (`for` and `while`), and function composition (`f(g(x))`). Then, after we've combined some pieces together to represent some new operations, we can *abstract* away the details of those new operations and treat them as though they had been built-in to Python from the start by defining functions corresponding to those operations, using the `def` or `lambda` keywords.

Hopefully by now you've already seen (and *felt*, deep in your soul) the usefulness of this idea when implementing your own code, for example by making nice helper functions for yourself to help organize and manage your code in the labs.

But this kind of analysis, it turns out, is not limited to operations; Python also gives us mechanisms to create our own *data types*; and we can apply a similar kind of analysis here.

We can start by thinking about *primitive* types, for example things like `int`s, `float`s, and `str`s. We also have ways to combine those things together into more complex structures like `list`s, `set`s, or `dict`s. We've seen an example of this idea already in lab 0, where we worked with structures like the following:

```
{
    'rate': 8000,
    'left': [1, 2, 3, 4, 5],
    'right': [6, 7, 8, 9, 10],
}
```

Here we've made a complex structure by combining together a few different types of objects. But the real magic happened when we started to think of this not just as a *dictionary containing lists of numbers* but as a *sound*, and when we defined functions that operated on it under that assumption. This idea, where we shifted our focus from the details of the Python objects to the things they represent, is another example of abstraction.

In this reading, we'll extend this idea further by exploring the `class` keyword, which provides us a nice way to create our own data types and integrate them tightly into Python so that they behave as though they had been built into Python from the start. So a custom type like a *sound* could be represented as a class, instead of as a dictionary with associated helper functions.

3) Custom Types and the Environment Model

The main goal of this section is to introduce classes in terms of how they behave in our environment model.

Throughout this section, we're going to work on building a class as a way to see how these things manifest in our environment diagrams (which will help us explain behaviors we see when using these tools in our own code).

We're going to come at this in stages, and the code will continue to evolve as we go. At first, our code will be non-idiomatic Python; that is, code that is correct but not written in the way you'd typically see Python programs written. By the end of this reading, we will have worked our way to more conventional code.

We take this approach so that we can start, essentially, from first principles in terms of understanding how classes and instances fit into our environment model. By starting with the very basics, we'll be able to lean heavily on the conventions we've already seen in our environment diagrams over the course of the semester. So we'll start small and gradually introduce new complexity; and in so doing, we hope to present a smooth path toward understanding how even idiomatic code is behaving in terms of our environment model.

3.1) Example: 2-d Vectors

Our running example throughout this section will be a custom type that represents 2-d vectors. The idea is that we will define a *class* that represents the general notion of a 2-d vector, including both the data we need to represent a vector and the operations that we can perform on that vector. Once we've made that class, we will be able to create *instances* of that class to represent specific 2-d vectors.

And again, we'll start small and simple (although nonidiomatic) and build up to something more conventional. So

our starting point is really just about the simplest class we can construct:

```
class Vector2D:
    pass
```

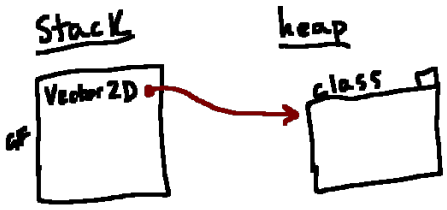
In general, we define a class using the `class` keyword, which is followed by a name and an indented body. Here, we're keeping the body as simple as possible: it's effectively empty (the `pass` means "do nothing"). So let's start our journey by seeing how Python responds to this piece of code. The `class` keyword in Python does two things:

- It creates a new object on the heap to represent this new class, and
- it associates that class object with a name in the frame in which we were running when we encountered the `class` keyword.

When running the code above, we start off running in the global frame as usual. In this example, the first statement that Python encounters is the class definition (indicated by the `class` keyword), and so it makes a new class object. We'll generally represent a class in our diagrams using a box similar to those we used for functions, though we'll label this box with the word "class" to help us keep track of the fact that it represents a class.

After making the fresh class object, the next step that Python takes is to run the code in the body of the class (in a particular way, which we'll expand on shortly). Here our class body is just `pass`, so there is nothing to do.

Finally, Python associates the new class object with the name `Vector2D` in the global frame. All told, the resulting diagram looks like the following:



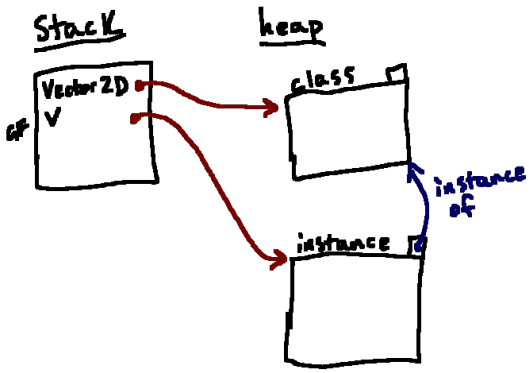
It's worth noting that, even though we'll draw class objects in a similar way to how we draw function objects, classes don't have enclosing frames. We still draw classes with a little tab on their top-right corner, but it will have a different meaning, which we'll discuss in more detail later.

Now that we've got our class represented, we can make an instance of that class like so:

```
v = Vector2D()
```

Python sees this as a regular old assignment statement, so we'll start by evaluating the expression on the right-hand side of the equals sign. This looks like a *call*, which we've seen before. So Python first evaluates the thing to the left of the round brackets to figure out which object we're going to be calling. In this case, it's what we get by evaluating the name `Vector2D`, i.e., the class object we just created in the previous step. In Python, when we call a class object using this syntax, Python proceeds by making a new object representing an instance of that class.

We'll represent instances in our diagrams with a similar kind of box, but labeled as "instance" instead of "function" or "class." Each instance will also store a reference to the class that it's an instance of. So after creating that new instance and associating it with the name `v`, our diagram will look like the following:

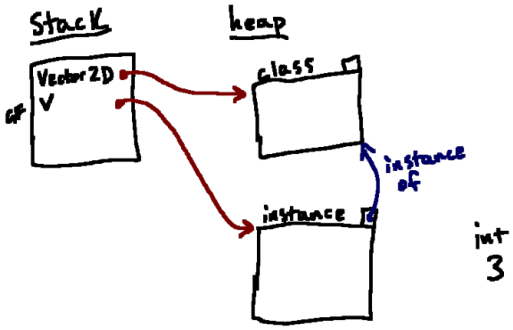


So far, our class and instance are somewhat boring, but this is indeed where we intended to start, so that we could see how classes and instances get made, as well as how we'll represent those things in our diagrams. With those objects made, we can introduce a new piece of syntax, so-called "dot notation."

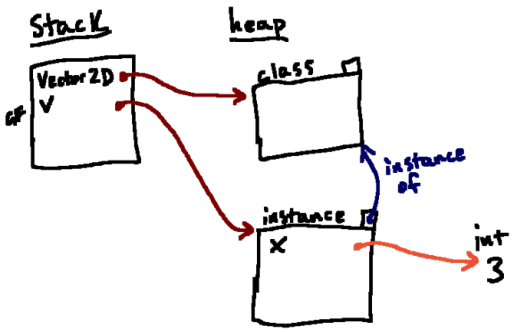
Let's try the following now:

```
v.x = 3
v.y = 4
```

In some sense, this syntax is new. But in some sense, it's familiar. These are both assignment statements, and they'll behave like the assignment statements we've seen before. What's different is simply: what are we assigning to? Let's start with the first of these: `v.x = 3`. Python starts, as always, by evaluating the right-hand side of the equals sign, resulting in a new object representing `3`:

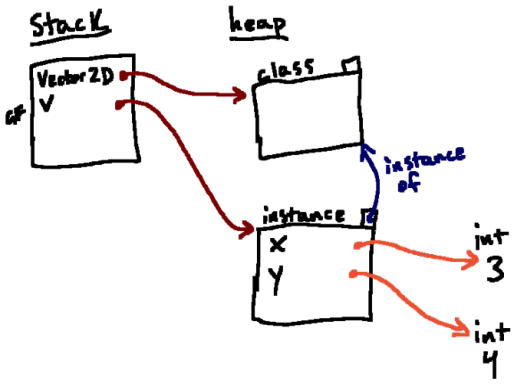


But then what are we assigning that to? When Python sees `v.x`, this means: first evaluate `v`, and then, *within whatever object we get,* associate this object with the name `x`. So the end result here looks like the following:



Note that the place where we associated the name `x` with the value `3` was *inside of the instance*. To introduce a bit of terminology, we call this kind of association (inside of an object rather than a frame) an *attribute* of that object.

For completeness, let's also carry out the next line, which says `v.y = 4`. Here we follow a similar process, resulting in a diagram like the following:

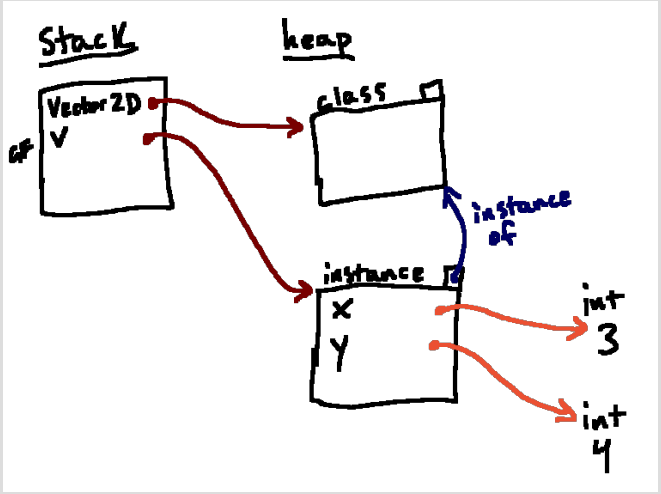


Starting from the diagram above, which of the following are valid ways to refer to the integer `3` on the heap? Check all that apply.

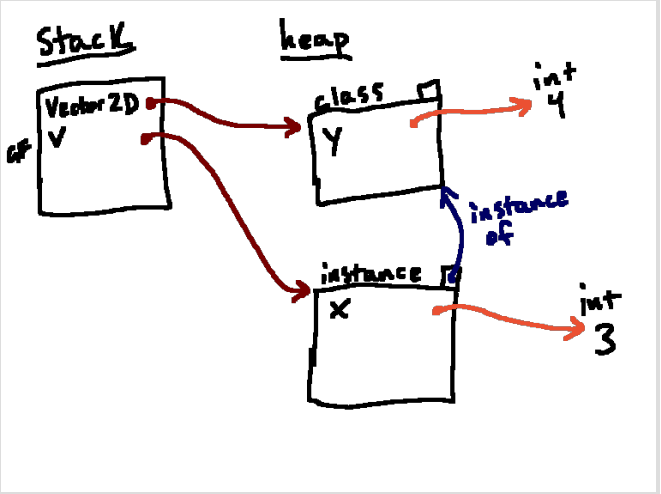
- [ ] `v`
- [ ] `v.v`
- [ ] `v.x`
- [ ] `x`
- [ ] `x.v`
- [ ] `x.x`
- [ ] `v.v.x`
- [ ] `x.x.v`

If the last line had been `y = 4` instead of `v.y = 4`, which of the following diagrams would have resulted?

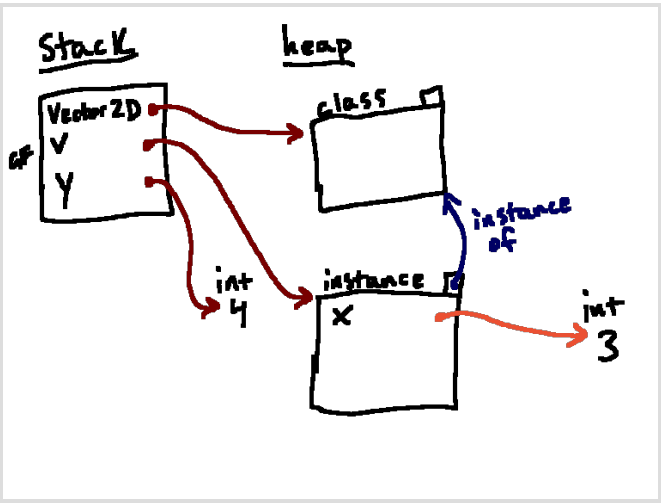
A



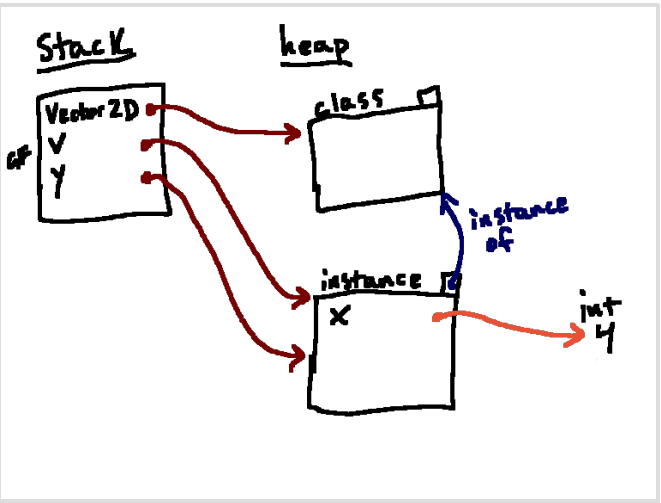
B



C



D



Matching diagram:

--

3.1.1) Magnitude Function

Now that we have something in code to represent a vector, let's make a function for working with these objects. Since we have stored attributes `x` and `y` inside of our instance, we can make a function that anticipates this. Consider the following code (which includes everything from above for clarity):

```
class Vector2D:
    pass

v = Vector2D()
v.x = 3
v.y = 4
```

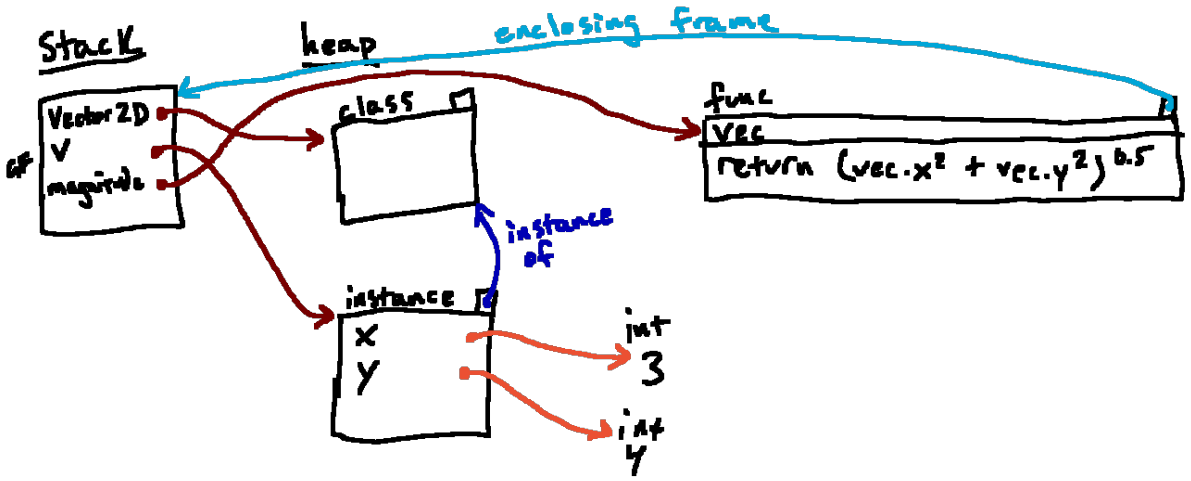
```
def magnitude(vec):
    return (vec.x**2 + vec.y**2) ** 0.5
```

Check Yourself:

What does the environment diagram look like after defining `magnitude`?

Show/Hide

Note that `magnitude` is defined in the global frame (it's not indented inside of the class). So this definition works the same way as other function definitions we've seen in the past, resulting in the following diagram:

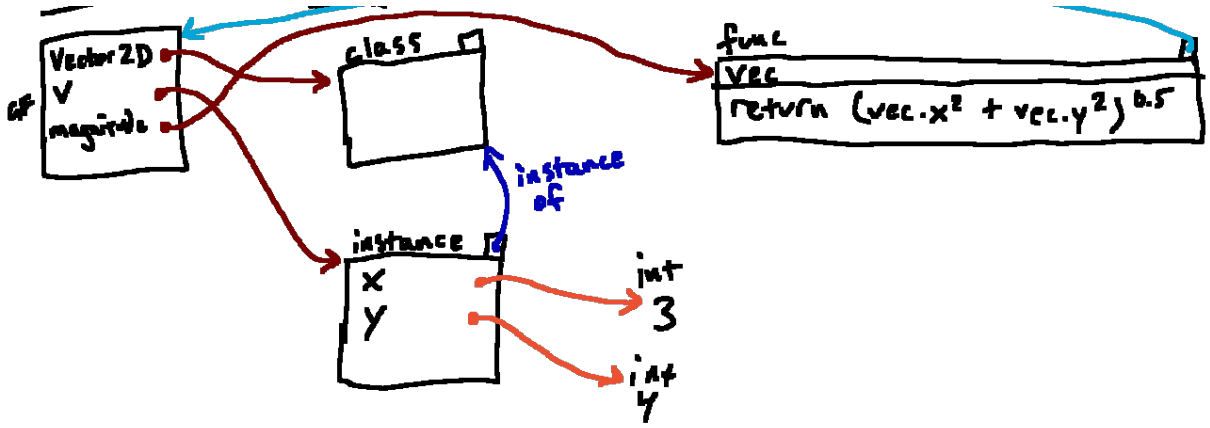


Given the code we've written above, which of the following are valid ways to compute the magnitude of the vector represented by `v`? Check all that apply.

- [ ] `magnitude()`
- [x] `magnitude(v)`
- [ ] `v.magnitude()`
- [ ] `v.magnitude(v)`
- [ ] `Vector2D.magnitude()`
- [ ] `Vector2D.magnitude(v)`

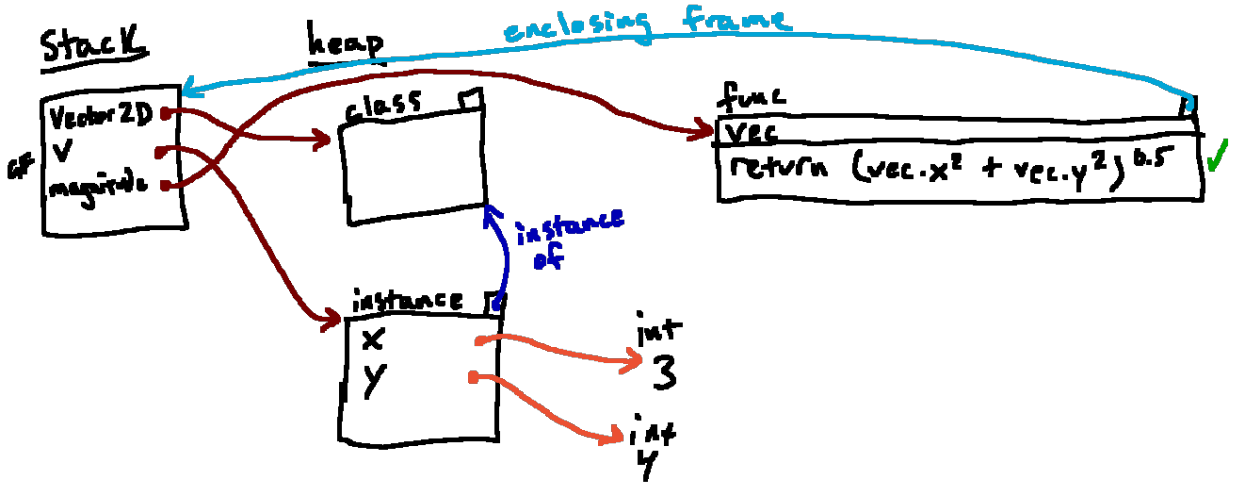
Now let's see what happens when we evaluate `magnitude(v)`, in terms of our environment diagram. As usual, we recommend working through the diagram step-by-step alongside the example below, trying to complete each step on your own before stepping ahead and comparing your result against ours.

<< First Step | < Previous Step | Next Step > | Last Step >>



STEP 1

Here we see the starting point repeated from above. Click through to the next step to get started.

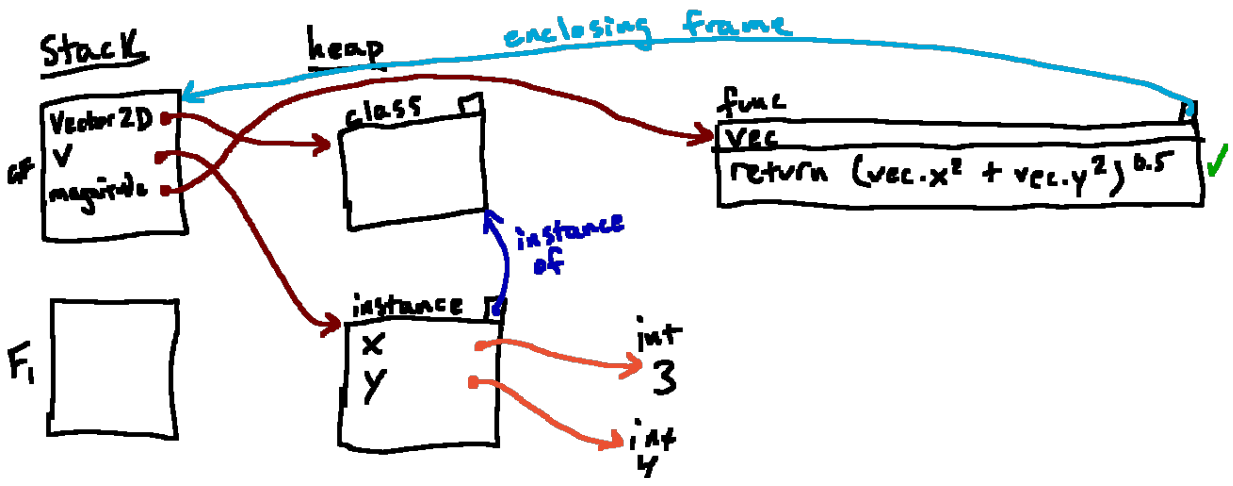


STEP 2

The call we're making here is `magnitude(v)`. This looks like a function call, so we start by evaluating the function we're going to call, as well as the arguments we're passing in. Here, evaluating the name `magnitude` gives us the function object in the upper-right corner of the diagram. That's the function we're going to call, and we've marked it with a little check mark just for reference.

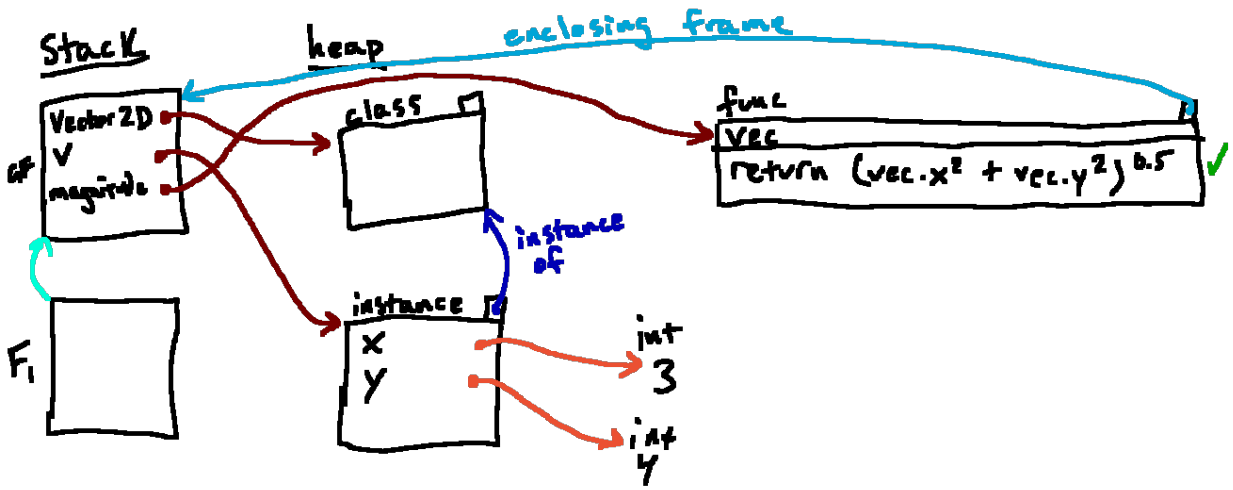
Evaluating `v` gives us the instance object we created (which has attributes `x` and `y`).

Now that we know what function we're calling and what arguments we're passing in, we can move ahead with the function call. The first step in actually calling the function is to make a new frame.



STEP 3

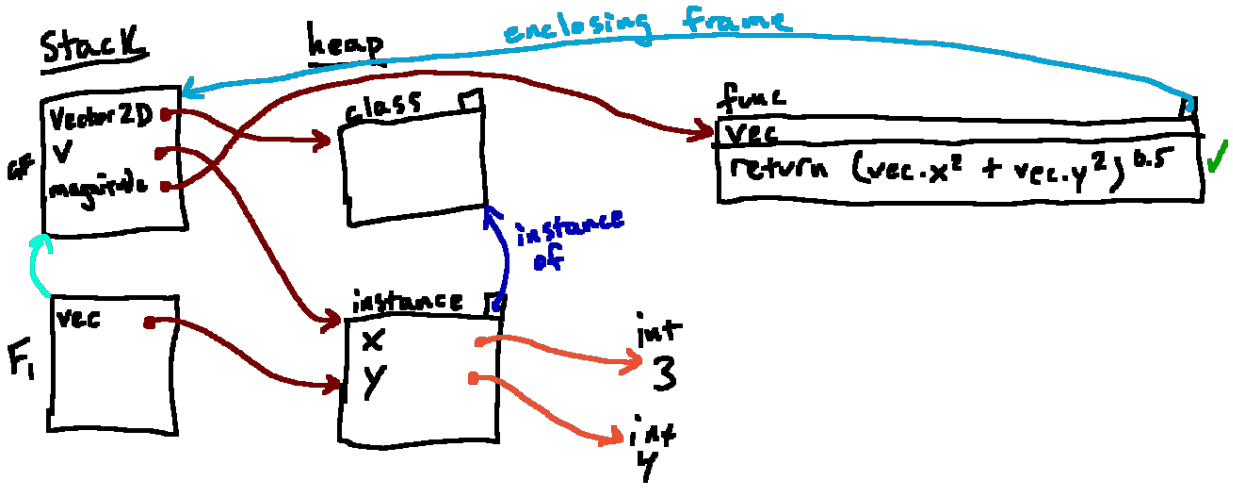
Here we have drawn our new frame, which will store the local variables for this call to `magnitude`. But we've left off one key piece: we've not yet drawn its parent pointer. What should **F1**'s parent be?



STEP 4

Since the function we're calling (the function object in the upper-right corner with the check mark next to it) has the global frame as its enclosing environment, the global frame will be **F1**'s parent.

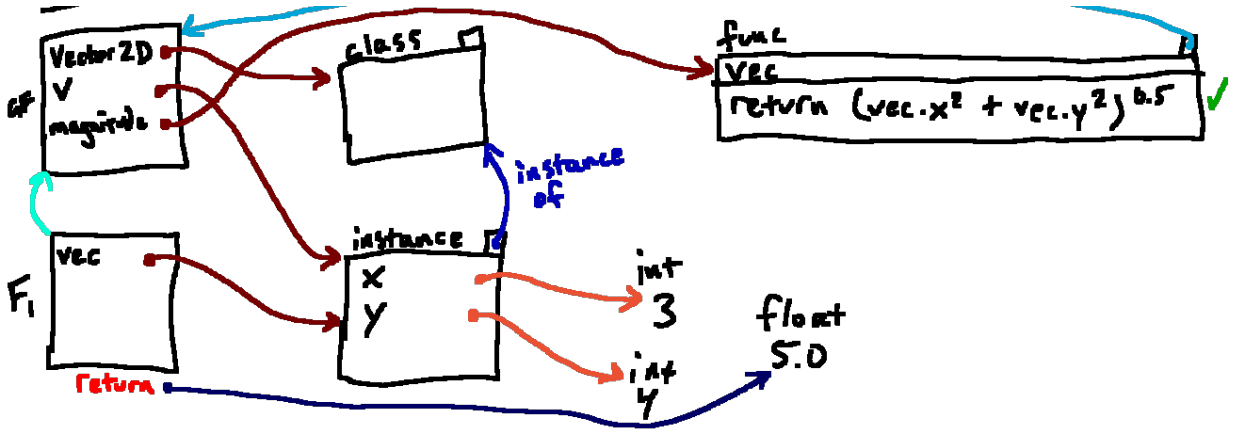
Now that we've set up our frame, the next step is to bind the parameters of the function to the arguments that were passed in, inside of our new frame. Go ahead and update your copy of the diagram based on this, before clicking through to the next step.



STEP 5

The function we're calling has a single parameter called `vec`, so that's the variable we're creating in our new frame; and, as usual, it points to the object that was passed in as argument (the instance that is called `v` in the global frame). After having done this, we're all set to run the body of the function. Think

through all of the steps that are going to happen when we run the body of the function in **F1**. What will the final result be?



STEP 6

Here we've actually skipped quite a few steps (most notably, the creation and subsequent garbage collection of all of the intermediate results of the calculations), but Python proceeds here by looking up `vec.x` (finding `3`) and squaring it to get `9`, looking up `vec.y` (finding `4`), squaring it to get `16`, adding those together to get `25`, and then finding the square root of that to get `5.0`. Note that the numbers `2` and `0.5` would also have been allocated as part of this computation.

3.1.2) Rearranging the Code

So far, we've seen an example of making a (small) class, making an instance of that class, and making a function designed to operate on instances of that class.

```
class Vector2D:
    pass

v = Vector2D()
v.x = 3
v.y = 4

def magnitude(vec):
    return (vec.x**2 + vec.y**2) ** 0.5
```

It's worth noting here that this function (`magnitude`) is not limited to operating only on instances of the `Vector2D` class; it will work for any object that has attributes `x` and `y` that both refer to numbers. This is a philosophy called "duck typing," which comes from the idea that if something looks like a duck and it quacks like a duck, then it's fine to treat it like a duck; and in this case, if something looks like a vector and quacks like a vector (in the sense of having attributes `x` and `y` that both contain numbers), then it's fine to use this function with it. There are different perspectives on this approach compared to others (some languages might be more strict about types), but Python is generally very flexible about things like this.

However, it's pretty clear that this function was *intended* to be used with `Vector2D` classes. But as it stands right now, there is nothing actually connecting them together in terms of the code. From a code-clarity perspective, it would be nice for the `magnitude` function (which was written to work with `Vector2D` instances) to be somehow associated with `Vector2D`.

We could perhaps make this connection clearer by calling our function something like `vector_magnitude`, but even then, it's a pretty weak association. However, Python gives us a much more explicit way to indicate that a function (or, indeed, any piece of data) is connected with a particular class: by including the code as part of the class's body. So let's make a small change to the code, putting the definition of the `magnitude` function inside the body of the `Vector2D` class (and we'll also add another piece for purposes of illustration, indicating the dimensionality of our vectors):

Show/Hide Line Numbers

```
1 class Vector2D:
2     ndims = 2
3
4     def magnitude(vec):
5         return (vec.x**2 + vec.y**2) ** 0.5
6
7 v = Vector2D()
8 v.x = 3
9 v.y = 4
```

Note that all we've done here is to take our `magnitude` function and move it into the body of the `Vector2D` class; we've not changed it in any other way. But this change is going to introduce the next wrinkle into this example: now we have a nontrivial body to our `Vector2D` class. Python will start the same way as before when we see the class definition, by making a new empty class object. But then, Python proceeds by running the body of the class definition, subject to some special rules:

- Any names that we bind in that definition are created as *attributes of the class* rather than as variables in a frame.
- When looking up a name as part of the class body, we first look for other class attributes we've already defined; if the name does not exist there, we proceed by looking it up in whatever frame we were running in when encountering the class definition.

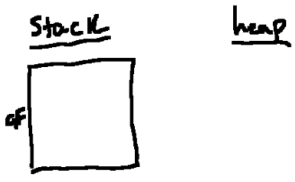
Note that these rules **only** apply when evaluating the body of the class as we're creating it; we'll have different name-lookup rules later on.

To make this more concrete, let's look at how Python works in the example above by walking through a new environment diagram for this code. Again, we'll go step-by-step:

Show/Hide Line Numbers

```
1 class Vector2D:
2   ndims = 2
3
4   def
  magnitude(vec):
5     return
  (vec.x**2 +
  vec.y**2) ** 0.5
6
```

<< First Step | < Previous Step | Next Step > | Last Step >>



```
v = Vector2D()
v.x = 3
v.y = 4
```

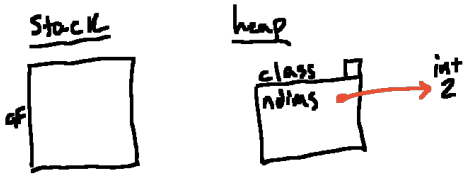
STEP 1

As always, we start by creating a global frame, in which we'll start running. The first thing we see when running is the class definition. As before, Python starts by making a new class object, which we'll show on the next step.

STEP 2

Now that we've got our empty class object, we start by running the body of the class subject to the special rules we mentioned above. The first thing we see in the body of the class is `ndims = 2`.

What will be the result of executing this line of code?



STEP 3

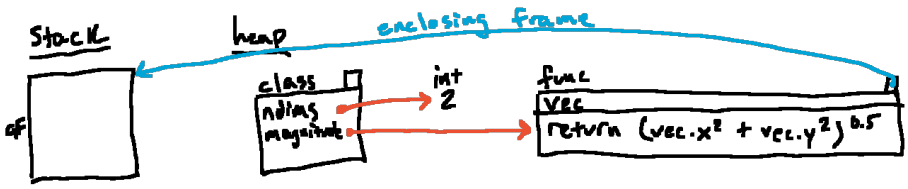
`ndims = 2` looks like an assignment statement, so, as always, we start by evaluating the expression on the right-hand side of the equals sign, which, in this case, gives us a new `int` object representing `2`.

Then, we associate that object with the name `ndims`. But because this is happening inside of a class definition, we bind that name not as a variable in the global frame but as an attribute in the class object we're creating, resulting in the diagram shown above.

Next (while still in the class's body) comes the definition of `magnitude`. This one is a little bit tricky, but we can think through it. What should that function object look like? What are its parameters, and what is its body? And, importantly, what is its enclosing frame?

Try thinking through those questions, and then click through to the next step to

see the result.



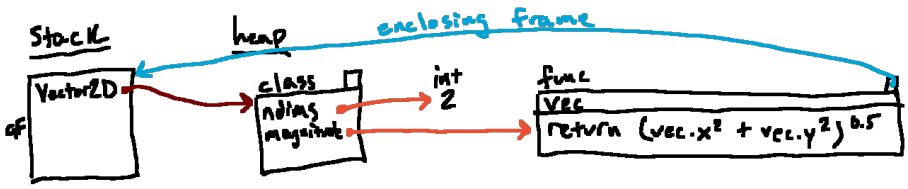
STEP 4

Here we see the result of defining that function. A couple of things are worth paying attention to here:

- The name `magnitude` was bound as an attribute in the class object, not as a variable in the global frame.
- *However*, that function's enclosing frame (the frame in which we were running when we defined it) is still the global frame.

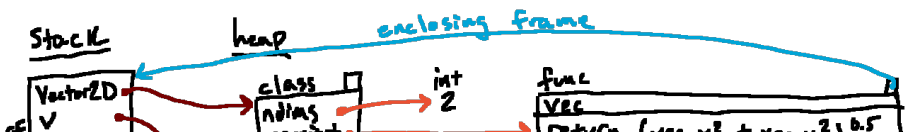
This may feel somewhat strange, but it's critically important that things work this way, and we'll see some examples of why things need to work this way later on.

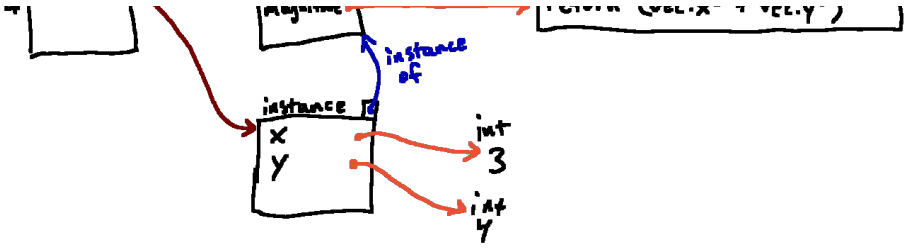
Now that we've got this function object created, we reach the bottom of the class definition, and we are done with constructing the class object. The next step is to associate that class with the name `Vector2D` in the global frame, which we'll show on the next step.



STEP 5

Now that we have our class object, we can proceed with running the remaining code (making an instance and setting some attributes of that instance). These steps will proceed exactly as they did in our initial example, so try adding them to your diagram before clicking through to the next step to see the result.





STEP 6

Here we see the end result of running the code.

3.2) Calling Methods

From some perspective, this is really nice! We've still made our function as before, but now, as part of the `Vector2D` class, it is a little bit clearer that the `magnitude` function is intended to operate on `Vector2D` objects. To introduce another small piece of jargon: when a function is defined inside of a class like this, we'll refer to it as a *method*.

Now let's try to call this function the same way we did before.

If we were to run `print(magnitude(v))`, what would happen?

- We would see `5.0` printed to the screen.
- We would see `5` printed to the screen.
- We would get a `NameError` exception.
- We would get a `TypeError` exception.
- We would get some other kind of exception.

The answer/explanation for the question above shows one way that we could call this method. But that example, for better or for worse, is not super idiomatic, i.e., you will very rarely come across Python code that uses that syntax for calling a method (using the class as a way to look up a method and then passing in an instance of that class explicitly). But there is something quite nice about that form: it's very explicit; we tell Python where to find the function we want to call and what arguments to pass in, and everything proceeds according to rules we've already seen.

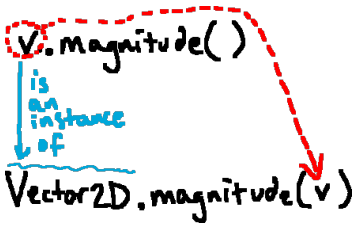
However, it's far more common to see a different way of calling this method, using the following syntax:

```
v.magnitude()
```

Calling the method in this way produces the same result as our earlier invocation of `Vector2D.magnitude(v)`, but something is strange: even though the method we've defined has a single parameter, we've just (seemingly) called it with no arguments. So what is going on here?

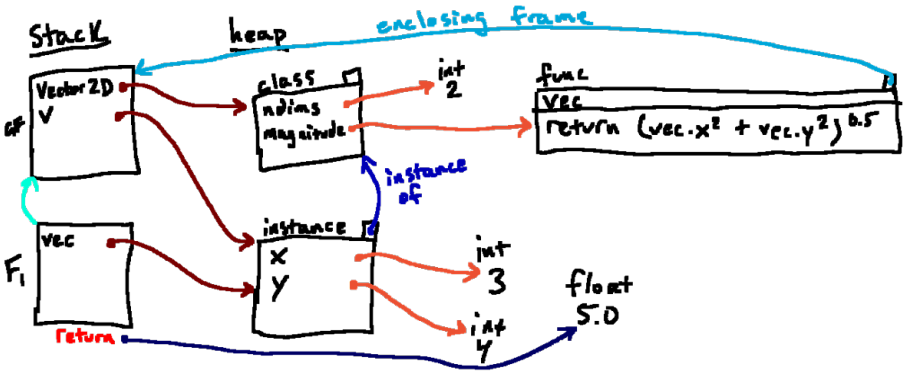
It turns out that Python is doing some magic to make things work out this way. The actual details are somewhat complicated, but one way that we can think about what Python is doing is: when we look up a method by way of an instance, Python figures out what class that's an instance of, finds the given method within that class, and then *implicitly passes in that instance as the first argument*. So we can think of this as a transformation where the call

`v.magnitude()` is turned into `Vector2D.magnitude(v)`:



It's perhaps worth mentioning that this is not *precisely* what's actually happening behind the scenes. But it's a close-enough approximation for most applications (and the actual details are complicated enough!) that we'll stick with this model for now; and in general, we'll say that if we look up a class method by way of an instance of that class, then the instance we used to do the lookup will be implicitly passed in as the first argument.

Since this process happens *before* the method is actually called, the environment-diagram representation of the process for actually calling the given method is identical regardless of whether we used `Vector2D.magnitude(v)` or `v.magnitude()`; they both result in a diagram like the following (showing the state just before the `5.0` is returned):



3.2.1) A Common Error

It's worth pausing here for a moment to mention a common error message, regarded by some as the most confusing error message in all of Python.

Consider evaluating the following expression using the definitions above:

```
print(v.magnitude(v))
```

In fact, it may be useful not just to consider this code, but to run it. What happens? Why?!

Show/Hide

The error message we see here is:

```
TypeError: Vector2D.magnitude() takes 1 positional argument but 2 were
given
```

Until you get used to it (and even for a while afterwards...), this can be extremely confusing (*What do you mean, Python? I only gave it one argument!!*).

What's going on here, though, is that Python is implicitly providing `v` (the instance we used to look up the method) as the first argument; and then whatever arguments we provide are passed in afterwards. So even though what we wrote is `v.magnitude(v)`, thinking about the transformation we mentioned above, that's equivalent to writing `Vector2D.magnitude(v, v)`, where the second `v` is the one we provided explicitly, and the first was provided implicitly by Python.

3.3) The Nature of Self

Here is where our code for the `Vector2D` class left off in the last section:

```
class Vector2D:

    ndims = 2

    def magnitude(vec):
        return (vec.x**2 + vec.y**2) ** 0.5
```

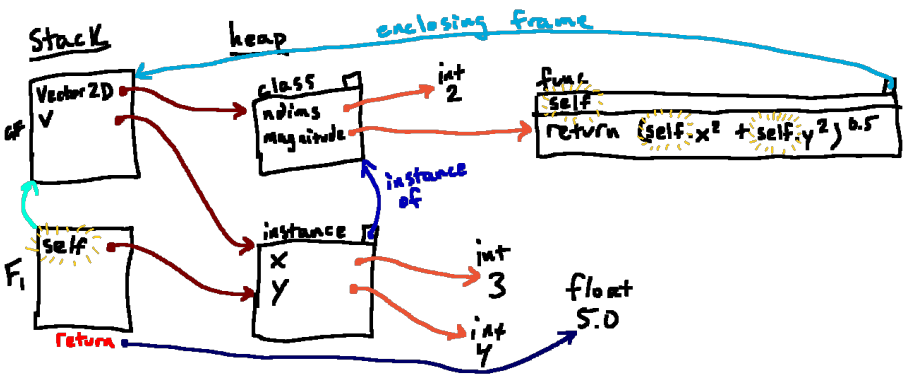
Those of you who have worked with classes in Python before might have been surprised by the lack of a common word here. Typically, when you encounter Python code "in the wild," so to speak, class methods will have a first parameter called `self`. But here, the word `self` is absent (perhaps conspicuously so!). In this section, we'll take a brief moment to talk about the nature of self. Not in an existential sense, of course. Countless great philosophers have spent their lifetimes considering *that* question, and we'll not be illuminating all of life's great mysteries in this reading; but what we *can* do is to try to remove some of the mystery around `self` in Python.

So what is `self`? As it so happens, it's not a keyword, nor a built-in object, or anything like that. It's just a really, really, really strong convention for the name that we give to the first parameter of a method (whose argument will be provided implicitly by the common way of invoking the method).

So it would be far more common to write this code as follows:

```
class Vector2D:

    ndims = 2

    def magnitude(self):
        return (self.x**2 + self.y**2) ** 0.5
```

But *this has no practical effect on the program*!. In the same way that changing the name of a variable from `x` to `y` in any other function you write doesn't affect the way the function behaves, changing the name of this parameter from `vec` to `self` doesn't change anything about how the function behaves. What it *does* change is one inner detail of our environment diagrams: in the frame we create when calling the `magnitude` method, the instance we're working with will be called `self` instead of `vec`:



Again, that change is purely internal and has no meaningful effect on the computation. But this is such a strong convention that it is worth following. And moving forward, we'll follow that convention and always call that first parameter `self`.

Another reason to like the name `self`, other than that everyone is doing it, is that it is consistent with a shift in perspective that often comes with the introduction of classes: a shift toward "object-oriented" thinking. With the shift from `Vector2D.magnitude(v)` to `v.magnitude()` comes a shift in perspective:

- In the expression `Vector2D.magnitude(v)`, it feels like the *function* is the active entity here. We're saying: "Hey, `Vector2D.magnitude`, here's a vector whose magnitude I want you to compute."
- The details of the computation don't change when we instead write `v.magnitude()`, but in this form, it feels like `v` is now the active entity. This feels more like saying: "Hey, `v`, tell me *your* magnitude!"

When we take this second perspective, the name `self` starts to feel like a nice choice of name. It's the instance about which we're asking these questions, and that instance is answering those questions about *itself*.

3.4) Variable Lookup vs Attribute Lookup

Before we move on, let's talk just a bit more about `self`. It's common, especially when just starting out, to try to debug issues involving classes by randomly putting "`self.`" in front of things (or deleting "`self.`" from in front of things) until the error messages go away. That's a fine place to start, of course; but one of the things we're trying to do in 6.101 is to move beyond that kind of "guess-and-check" style of programming and toward more principled ways of thinking about things.

One of the things we've been talking about since we first started with environment diagrams is *name resolution*: when I ask Python to look up a name, where and how does it look for that name? One of the things that we've introduced here is another, distinct, set of rules for name resolution. Here, we'll try to make the rules for those two kinds of name resolution a bit clearer and a bit more concrete, and hopefully use that as a way to start thinking about how and when to use `self` in your own programs.

The rules we established much earlier in the course were about looking up variables inside of frames. That process was summarized as follows:

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`

What we've introduced here is something separate, which we call *attribute lookup*. The idea is that we've already found an object and, instead of looking in a frame for a variable, we want to look *within that object* for something. This process proceeds a little bit differently:

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`

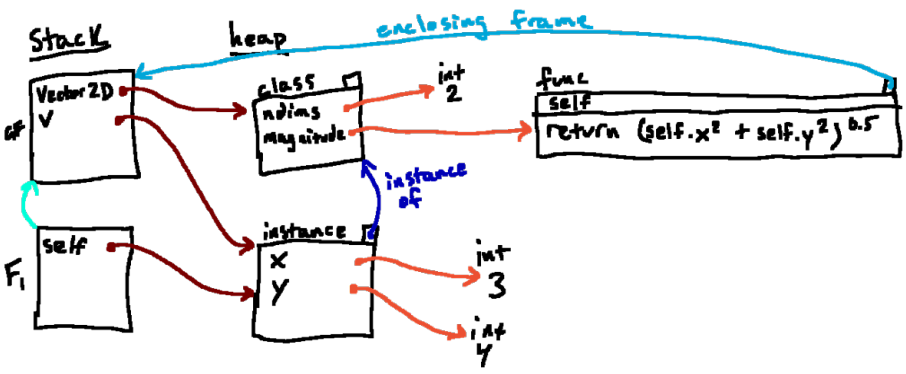
Since we're saving our conversation about superclasses until the next reading, the most important steps for us now are the first two. It's also important to notice that this process of attribute lookup *never crosses over to start looking in frames*; it's a completely separate process.

3.4.1) An Example

As a brief concrete example, let's look back at our `magnitude` example from before:

```
class Vector2D:

    ndims = 2

    def magnitude(self):
        return (self.x**2 + self.y**2) ** 0.5

v = Vector2D()
v.x = 3
v.y = 4
print(v.magnitude())
```

What we want to think about here is the difference between saying, for example, `self.x` inside the body of `magnitude`, versus just saying `x`. At the point where we start evaluating the body of the `magnitude` function, our environment looks like:



Check Yourself:

If the body of our `magnitude` function had been `return (x**2 + y**2) ** 0.5`, what would have happened?

Show/Hide

In this case, we would need to evaluate `x` as part of evaluating the overall expression. Looking up `x` in **F1**, we don't find it. So we look for `x` in the global frame. We don't find it there, either! And there's no built-in called `x`, so we end up with a `NameError`!

Check Yourself:

How is this different from how things evolve with the code as written above (which uses `self.x` instead of `x`)?

Show/Hide

With the code as written, we never look up a variable called `x`. We *do* look up a variable called `self` in **F1**, and we immediately find it; evaluating `self` in **F1** gives us the `Vector2D` instance we created earlier. Then, *inside of that object*, we look for an attribute called `x`, which we find right away; so `self.x` ultimately evaluates to `3`.

As another example, if we had `self.ndims` somewhere in the function body, that lookup would proceed in the same way. We would start by looking up the name `self`, and we would find that same `Vector2D` instance. Then we would look in that object for an attribute called `ndims`. Not finding it, we would look in that instance's class (following the dark blue arrow in the diagram), where we would find the value of `2`.

It also turns out that this distinction provides some motivation for why our magnitude function's enclosing frame was the global frame instead of the `Vector2D` class. Having things set up that way makes sure that we can access information both from instances we're working with and from our usual frame structure. We can look up attributes by way of `self` (or whatever we called that first parameter), and we can look up variables from elsewhere in the environment in the same way that we could in any other function (by referring to those things by name, without `self.`).

3.5) `__init__`

Let's wrap up this example by taking one more step to make our `Vector2D` class more conventional (and also more convenient to use). As of right now, it's somewhat painful for us to make instances of our `Vector2D` class. Here's the code we wrote to make a single instance:

```
v = Vector2D()
v.x = 3
v.y = 4
```

If we want to make multiple instances of vectors within my program, it's kind of a pain to type out those three lines each time. It's also error-prone! What if we typo the name `x` or `y`? Or what if we forget to set one of those attributes for one of our instances (suddenly `magnitude` won't work for that instance!)?

Conveniently, Python gives us a way to canonize the way in which instances of a class are set up, by way of a "magic" method called `__init__`. We'll define `__init__` like any other method, and we'll be able to call it like any other method. But calling it `__init__` will cause Python to invoke it implicitly whenever we create a new instance of `Vector2D`, and any arguments we provide when calling `Vector2D` will be passed in to the `__init__` method. Let's adjust our code to include an `__init__` method:

```
class Vector2D:
    ndims = 2

    def __init__(self, x, y):
        self.x = x
        self.y = y

    def magnitude(self):
        return (self.x**2 + self.y**2) ** 0.5
```

When we've done this, we can make a new instance with, for example, `v = Vector2D(6, 8)`, which will cause `v` to be a new instance whose `x` and `y` attributes are set to `6` and `8`, respectively.

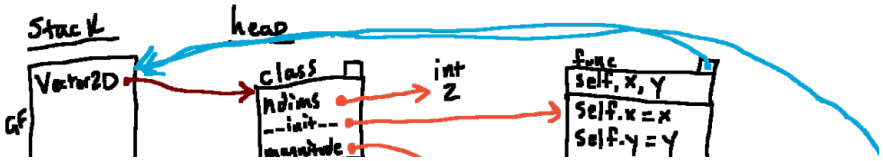
This process follows the steps we outlined earlier for creating an instance of a class, with one notable exception: if the class we're making an instance of contains an `__init__` method, then after creating the instance, Python will automatically call that `__init__` method with the new instance passed in as the first argument (followed by any arguments we explicitly provided).

Let's walk through this process as well, using an environment diagram:

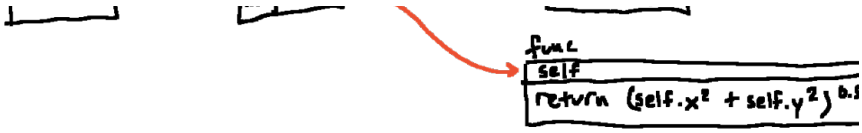
Show/Hide Line Numbers

```
1 class Vector2D:
2     ndims = 2
3
4     def __init__(self, x,
```

<< First Step | < Previous Step | Next Step > | Last Step >>



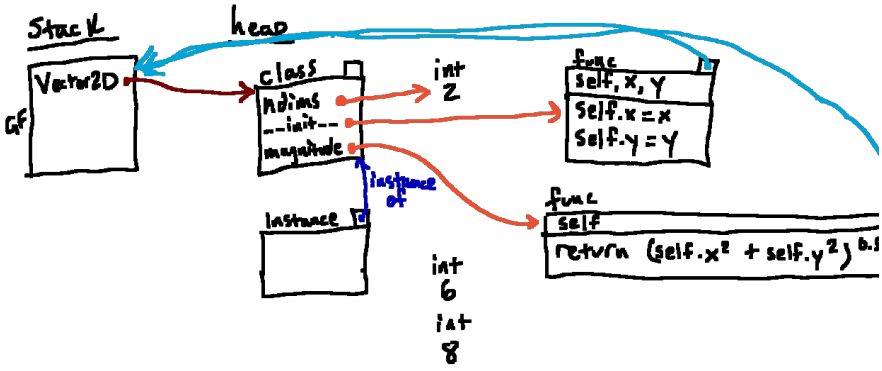
```
y):
    self.x = x
    self.y = y

  def magnitude(self):
    return (self.x**2 + self.y**2) ** 0.5

v = Vector2D(6, 8)
```



STEP 1

This picture shows the diagram after running all of the code in the class definition. Note that, compared to earlier versions, we now have two meth defined inside of the `Vector2D` class.

Now we're going to take a look at running the code: `v = Vector2D(6,`

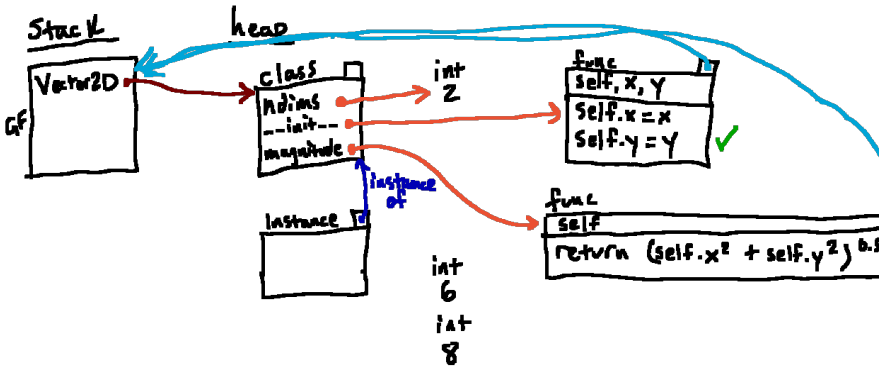


STEP 2

The code `v = Vector2D(6, 8)` first evaluates its arguments `6` and `8` then makes a new empty instance, shown here.

Next, since there is an `__init__` method in the attribute-lookup chain h Python is going to call that function implicitly with our new (currently emp instance passed in as a first argument.

In order to do that, we need to know what function we're calling. In the ne we'll show the results of evaluating those arguments.

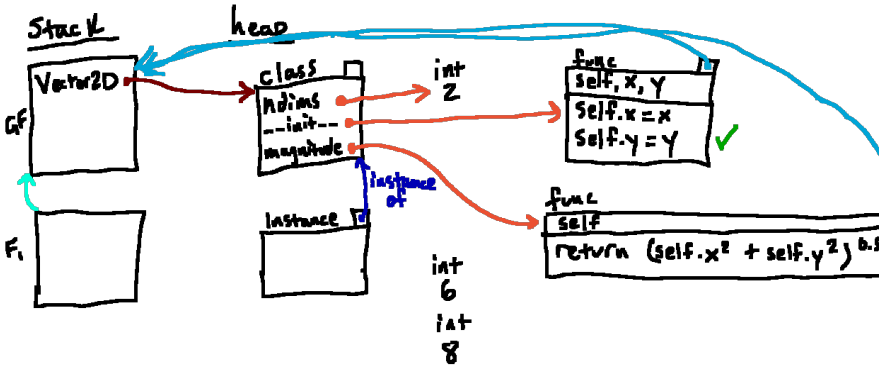


STEP 3

Here, we've marked `Vector2D.__init__` as the function we're going t and we need to figure out the arguments to it as well. The first (which Pyt going to pass in implicitly) is the instance object we made before. The se and third are the objects we got from evaluating `6` and `8`, respectively.

From this point on, this process will proceed just like any other function c

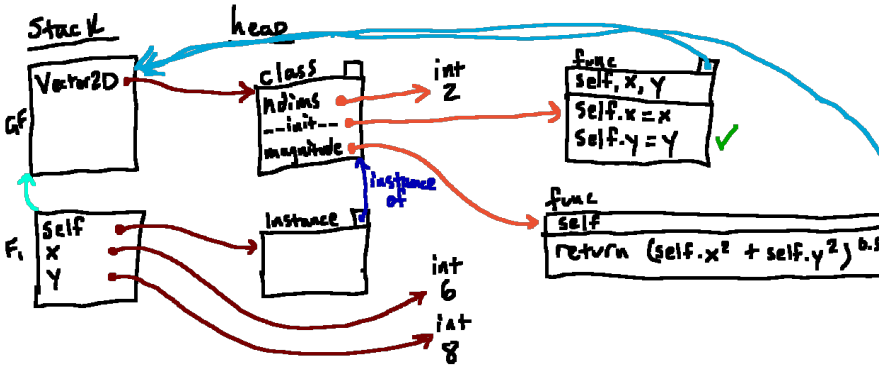
Now that we know what function we're calling and what arguments we're passing in, we can proceed with calling the function. Our next step is goir making a frame for this function call. Before clicking through: what will its pointer be?



STEP 4

Here we see our new frame, with its parent pointer going to the global fra enclosing frame of the function we're calling).

Our next step will be to bind the parameters of the function to the argume that were passed in, inside of our new frame **F1**. Think through how the di will change as a result of making those bindings before clicking through t next step.

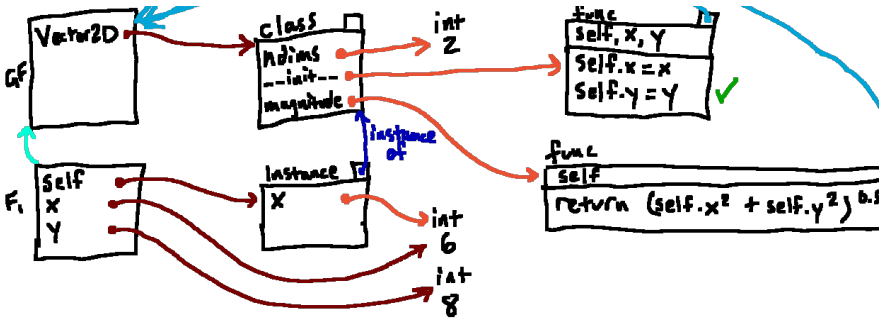


STEP 5

Here, we see the result of that setup. The function we're calling had three parameters: `self`, `x`, and `y`. So here we've bound those names to the th arguments that were passed in.

Now we're ready to run the body of the function. Clicking through to the r step will only show the result of executing the first line of the body, `self x`. Think carefully about how that will affect the diagram, before clicking t

to the next step.

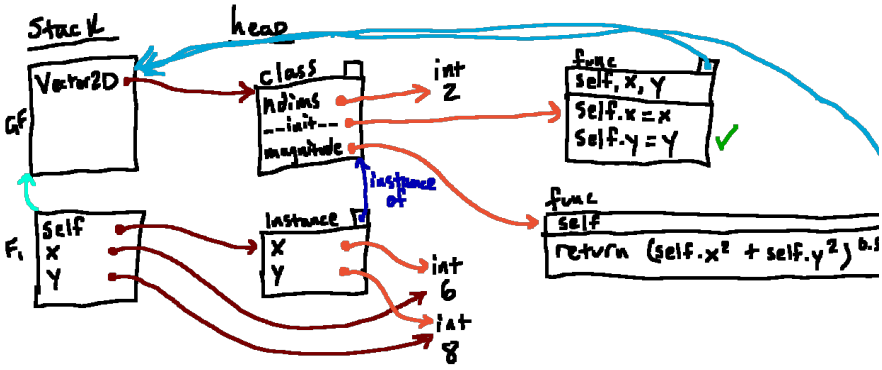


STEP 6

Here we see the result of executing `self.x = x` inside of **F1**. This looks assignment statement, so, as always, we start by evaluating the right-har of the equals sign, where we see the name `x`. Looking up the name `x` in **F** us the `int` object representing `6`.

Then, where do we put that? Well, we're putting it in `self.x`. We start by evaluating `self` in **F1**, and following the arrow we end up at the new inst we're making. So inside of that instance, we make a new attribute called pointing to the `6` object we found a moment ago.

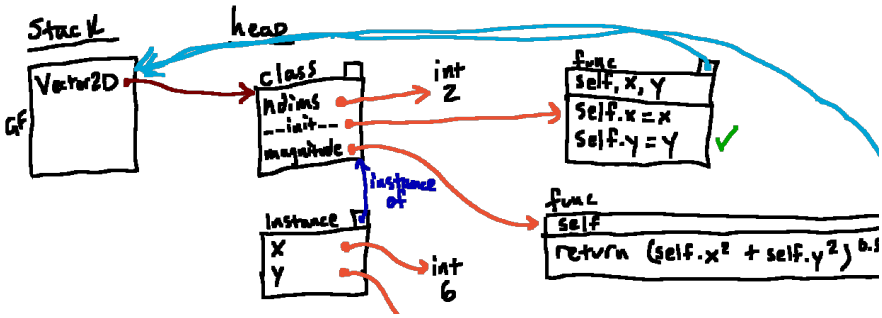
Clicking through to the next step will show the result of running the next l the body of this function.



STEP 7

Given the process we followed on the last step, hopefully this part is not t surprising! The end result is that our new instance now has an attribute c that is bound to the `int` object representing `8`.

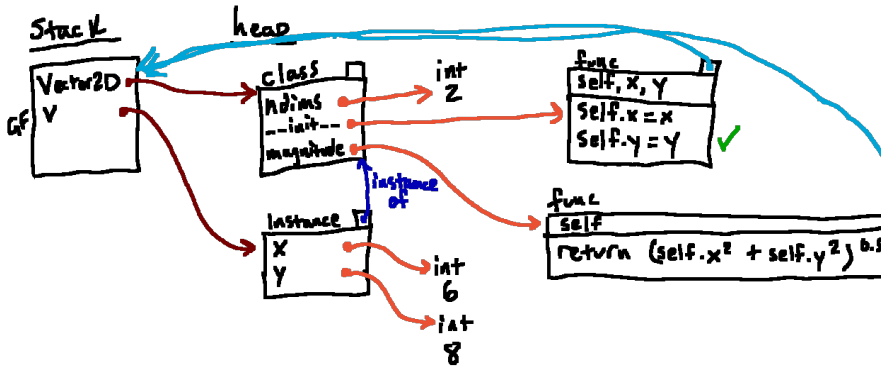
At this point, we've reached the end of the body of the `__init__` functi we're all done, and we're ready to clean up **F1**. Clicking through to the ne will show the result.



STEP 8

Now that we've cleaned up **F1**, we can clearly see the end result of this w process. Our brand-new instance has its attributes appropriately set!

Finally, let's recall how we got here. Our original goal was to execute `v = Vector2D(6, 8)` in the global frame. And we've evaluated the right-ha of the equals sign (the result is this new instance!), so all that's left to do associate this instance with the name `v` in the global frame.



STEP 9

Here we see the end result of this process, with our new instance bound t name `v`.

`__init__` is a really powerful tool that helps us avoid repetition when making multiple instances of the same class. With this implemented, we can make new instances much more easily and rest assured that each one is going to have been set up properly.

3.5.1) Dunder Methods

`__init__` is not the only method that Python will invoke implicitly. Python gives us a number of ways to integrate our custom type more tightly into the language, by way of other "magic" methods that we can implement if we want. We can think of most of these as being translations. For example:

- `print(x)` is translated implicitly to `print(x.__str__())`
- `abs(x)` is translated implicitly to `x.__abs__()`
- `x + y` is translated implicitly to `x.__add__(y)`
- `x - y` is translated implicitly to `x.__sub__(y)`
- `x[y]` is translated implicitly to `x.__getitem__(y)`
- `x[y] = z` is translated implicitly to `x.__setitem__(y, z)`

So by defining these kinds of methods, we allow for using normal Python operations with instances of our classes. For example, if our class defines the `__add__()` method, then we will be able to use the Python `+` operator on its instances.

These methods are also sometimes called "dunder" methods, short for "double underscore," because of the

naming convention that they all follow.

We'll see several examples of these over the coming readings, recitations, and labs; but for a complete list, see here.

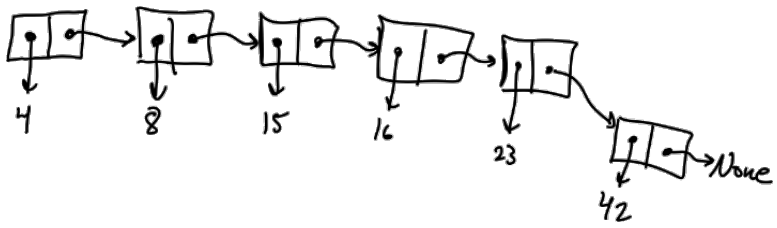
4) Example: Linked Lists

Now that we've talked about some of the machinery at play here, we'll devote the rest of this reading to walking through an example in detail. You're encouraged to follow along with our examples and try things out on your own machine as you work through this section!

The example we'll work with here will involve implementing a structure called a *linked list*, which is a different way of implementing lists (ordered collections of elements) than the way that Python's lists are implemented internally. Logically, we will be implementing an ordered collection of elements, and the operations that we'll be able to perform on it will be similar to those that we can perform on lists, but how the data are stored and how those operations are performed will be different.

We'll start by implementing some basic operations, and then we'll take advantage of the "dunder"-method idea to make our linked lists feel like they're really a part of Python.

As a running example, let's consider a list containing 4, 8, 15, 16, 23, and 42 (in that order). In our linked-list representation, each location in the list is represented by a *node*, which is some kind of object that stores two things: an element and a reference to the next node in the list. So graphically, we might think of our example linked list in the following way:



This is a little bit of shorthand designed to show the structure of a linked list, but in the code that we'll write below, each of the nodes will be represented by an instance of a class. Notice that there are several different nodes here, each of which has a value and a reference to the next node in the list. In our representation, we will know that a node represents the *end* of a linked list if it has no next node (if that reference points to `None` instead of to another node).

> How many nodes would we need to represent a linked list containing `-7`, `2`, `'hello'`, and `'cat'`?

We call this thing a linked list because, in some sense, it is not only the nodes but also the *linkages between them* that tell us what elements are in the list, governing how we perform various operations on the list.

Below, we'll write code to represent this kind of structure in Python. Interestingly, our code will not have an explicit representation for the entire list; rather, we'll make a representation for an individual node, and the overall list will in some sense be represented by the connections between those nodes.

Here is our initial representation, including a full `__init__` method as well as some small skeletons for a few

methods we'll implement below and an instantiation of our example list.

```
class LinkedList:
    def __init__(self, element, next_node=None):
        self.element = element
        self.next_node = next_node

    def get(self, index):
        pass

    def set(self, index, value):
        pass


x = LinkedList(4,
        LinkedList(8,
            LinkedList(15,
                LinkedList(16,
                    LinkedList(23, LinkedList(42))))))
```

Note that every time we make a new instance of this class, we specify an element and a "next node," which should either be a `LinkedList` instance that follows this node in the list or `None` if the node we're making represents the end of the list.

> Write an expression that uses `LinkedList` to construct a linked list containing the elements `9`, `7`, and `8` (in that order). *Hint*: how many `LinkedList` instances should there be in such a list?

4.1) Getting an Element

Let's start our implementation with a common operation: getting the element at a given index in a linked list. There are multiple ways that we could go about doing this, so let's take a look at a couple ways below.

4.1.1) Iteratively

Let's start by trying an iterative approach:

```
    def get(self, index):
        for _ in range(index):
            self = self.next_node
        return self.element
```

Take a moment to think about this code and then answer the following questions about it:

Using the example linked list `x` above, what will happen when we run `print(x.get(2))`? Select all that apply.

- [ ] It will run to completion and give us the correct answer.
- [ ] It will run to completion and give us an incorrect answer.
- [ ] It will change the value of `x` in the global frame such that subsequent calls will fail.
- [ ] It will enter an infinite loop.
- [ ] An exception will be raised.

4.1.2) Recursively

It's also the case, though, that linked lists have a nice, natural recursive structure that we can leverage. The key signal here that recursion might be a nice choice is that each linked list's `next_node` attribute is itself a linked list!

We can take advantage of this to write a recursive version of the `get` method. Try your hand at implementing this yourself before looking at our solution below:

Show/Hide

```
def get(self, index):
    if index == 0:
        return self.element
    if self.next_node is None:
        raise IndexError('index out of range!')
    return self.next_node.get(index - 1)
```

4.2) Setting an Element

Next, let's move on to another common operation on lists: mutating the list by changing a single element. Here, we'll focus on a recursive implementation (though you may wish to try to write this iteratively as well, for extra practice!).

If we start by copy and pasting our recursive version of `get` from above, we can change our code a little bit to implement `set`:

```
def set(self, index, value):
    if index == 0:
        self.element = value  # formerly: return self.element
    if self.next_node is None:
        raise IndexError('index out of range!')
    self.next_node.set(index - 1, value)  # formerly: return
```

```
        self.next_node.get(index - 1)
```

Using the example linked list `x` from above, what will happen when we run `x.set(2, 'cat')`? Think carefully about the code and select all that apply.

- ☐ It will successfully change `x` so that the element at index `2` is now `'cat'`.
- ☐ It will enter an infinite loop (or infinite recursion).
- ☐ Our code will raise an `IndexError`.
- ☐ A different exception will be raised.

4.3) Refactoring

Hopefully at this point in the semester, some things about that last example (`set`) raised some red flags not only in terms of correctness but also in terms of style. For one, we said the words "copy and pasting" 🚩. But even if we hadn't written that code with copy/paste, we still ended up with code that was very similar between the two functions. There's a definite possibility for refactoring here to avoid some of that repetitious code.

Check Yourself:

What functionality is similar between the `get` and `set` methods? What is different? How can we rearrange things to avoid repeating those pieces that are shared between the two functions?

Show/Hide

There is a lot of similarity here. In particular, the code for working our way recursively through a linked list to find the node at a given index (including raising an exception at the appropriate time) is repeated almost exactly in both functions!

What's different, of course, is what we do when we reach the right node. In the case of `get`, we want to return that node's `element` attribute; and in the case of `set`, we want to modify that node's `element` attribute.

So we could write a helper that returns the *node* (not the element) at a given index, and then both `get` and `set` could be implemented in terms of that helper.

Try your hand at writing this code before looking at our solution below:

Show/Hide

Here is one possible implementation:

```
def _get_node(self, index):
    if index == 0:
        return self
    elif self.next_node is None:
        raise IndexError('index out of range!')
    else:
        return self.next_node._get_node(index - 1)

def get(self, index):
    return self._get_node(index).element

def set(self, index, value):
    self._get_node(index).element = value
```

4.4) Leveraging Dunder Methods

At this point, we've got some pieces working. We've got a linked-list representation that we can use to create new linked lists, mutate existing linked lists, and get elements out of linked lists. But in order to do so, we need to

explicitly invoke the `get` and `set` methods, for example:

```
print(x.get(3))
x.set(3, 'cat')
print(x.get(3))
```

It would be nicer if we could use the syntax we're more familiar with (from working with other collections) for getting and setting elements:

```
print(x[3])
x[3] = 'cat'
print(x[3])
```

This is where those "dunder" methods come in! And it only takes a tiny change to make this work. In particular, if we change the *names* of these methods to `__getitem__` and `__setitem__`, respectively, then the example above will work without any other changes.

This change results in methods that can still be called like normal (we could, for example, still explicitly call `x.__getitem__(3)`). But Python will also implicitly invoke those methods in other situations. Python will, for example, translate `x[3]` to `x.__getitem__(3)`; and `x[3] = 'cat'` to `x.__setitem__(3, 'cat')`.

4.5) Displaying a Linked List

As we've seen over the course of the semester, one of the most useful tools we have for debugging is the `print` statement. But as it stands now, if we `print` an instance of our `LinkedList` class, we see something that is not particularly useful:

```
<__main__.LinkedList object at SOME_MEMORY_LOCATION>
```

This is Python's best attempt at displaying a `LinkedList` object for us, but it's really hard to make use of that when debugging. Luckily, though, Python lets us control this behavior through the use of another dunder method, `__str__`. Whenever we `print` an instance of a class we've made, if a `__str__` method is present, Python will call that method and print the result.

So in this case, we could make debugging easier for ourselves by defining `__str__`.

For example, if we define `__str__` as follows, we get a much nicer representation:

```
def __str__(self):
    return f"LinkedList({self.element}, {self.next_node})"
```

Check Yourself:

Add this code to your `LinkedList` class and try printing out `x` and/or some other linked lists of your own devising.

We generally recommend implementing `__str__` early on whenever you're making a new class, since it makes debugging so much more approachable!

4.6) Looping Over a Linked List

As our last example for this reading, let's add one last operation to our linked-list class.

Another common operation we might want to perform on an ordered collection of elements involves looping over it. For example, it would be great if we could say:

```
for elt in x:
    print(elt)
```

Conveniently, Python has a dunder method for that purpose, too! The above code is equivalent to:

```
for elt in x.__iter__():
    print(elt)
```

So if we define a new method `__iter__` (which should be a generator that yields elements from the linked list, in order), then we'll be able to use the `for i in x` syntax without explicitly calling `x.__iter__()`.

Try your hand at writing adding the `__iter__` method (a generator) to your `LinkedList` class before looking at our implementation(s) below. (You may also want to take this moment to refresh your knowledge about generators from an earlier reading.)

Show/Hide

There are a number of ways to implement `__iter__`. Here, we show one iterative and one recursive solution.

The iterative version is similar to the iterative version of `get` that we wrote early on. It uses a loop to work its way through the linked list, yielding each element until we've gone through the whole list:

```
def __iter__(self):
    current = self
    while current is not None:
        yield current.element
        current = current.next_node
```

The recursive version is a little bit different, but it follows the familiar "first/rest" recursive formulation we've seen in the past, where each call yields one element, and then the remaining elements (if any) are yielded from recursive calls:

```
def __iter__(self):
    yield self.element
    if self.next_node is not None:
        yield from self.next_node  # same as: yield from
self.next_node.__iter__()
```

4.7) Other Operations

What we've got so far is definitely a good start, but there are many operations that we could implement. If you're looking for extra practice, you could consider implementing one or more of the following:

- A method for deleting a node from a linked list by way of mutation
- A method for appending an element to a linked list by way of mutation
- A method for concatenating two linked lists together (without any aliasing)
- A method for inserting an element at an arbitrary location in the linked list by way of mutation (for an extra challenge, make sure it can work to insert a new element at the very start of the linked list, too!)

And there are certainly more you could work on, too! If you do try any of these and you need some help (or you want us to take a look once you're done), feel free to ask on the mailing lists or at open lab hours or office hours.

5) Summary

As usual, we've covered a fair amount of territory in this reading. We started by introducing the rules by which classes and instances operate in terms of our environment model by building up from minimal examples to something that demonstrated a fair number of features of classes, often leaning on our prior experience with name resolution and with functions.

We also took particularly close looks at a couple of pieces that are unique to classes, in particular the fact that Python will, in certain situations, implicitly pass an argument to a method (and we talked about the conventional name for it, `self`).

We then introduced "dunder" methods (also called "magic" methods), which allow us to use nicer, more "Pythonic" syntax for invoking certain methods (for example, allowing the normal syntax for subscripting or addition to work with instances of our custom types).

Finally, we saw an example that put a number of these pieces together, where we got started on an implementation of a class to represent linked lists; this provided an opportunity to think about linked data structures but also to see some of the ways that we can make use of classes.

In this week's lab, you'll get some experience with a different kind of linked data structure called a prefix tree, which we'll also use as an opportunity to get more practice with the material from this reading. In next week's reading, we'll expand on these ideas by introducing inheritance (which allows for sharing of structure between multiple classes), and we'll also talk more about *object-oriented design*, i.e., how we can use some of these ideas to help us manage the complexity of our code.