Intro

Functions are useful, but they're not always intuitive. Today we're going to learn about a different way of programming, where instead of functions we will deal primarily with objects. This school of thought is meant to make programming more intuitive, by letting you define tangible things — like rocks, or dogs, or trees, or even databases. Let's get right to it.

Classes & Inheritance

Recall how we need to use the keyword `def` in order to define a new function. There's also a keyword that you use to define a new type of object: it's `class`. As convention, we capitalize the names of classes and write them in CamelCase instead of using underscores. Within a class, we can define variables that describe the members of that class.

```
class Dog:

    scientific_name = 'doggo'
```

Above, we have a `class` that describes what it means to be a `Dog`. It's a start, at least. But what if, for example, we need to know the fur color of a dog? This is the first solution that comes to mind:

```
class Dog:

    scientific_name = 'doggo'
    fur_color = 'gold'
```

But this is a problem. Not every dog has gold fur, so we shouldn't specify that in the `Dog` class. In order to get more specific, we can use a handy dandy thing called class inheritance. This is the idea that one class can build on top of another one. For example, here's some code where we define a new `Retriever` class that inherits from `Dog`. `scientific_name` is defined for every `Dog`, including every `Retriever` since every `Retriever` is a `Dog`. But `fur_color` is defined only for a `Retriever`.

```
class Dog:

    scientific_name = 'doggo'

class Retriever(Dog):

    fur_color = 'gold'
```

Notice the parentheses. This tells us `Retriever` inherits from `Dog`, or in other words, it builds on top of the `Dog` class.

Let's see how to do classes in environment diagrams.

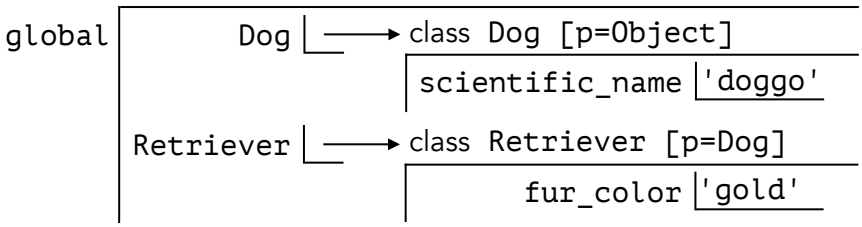
Classes in Environment Diagrams

Here's the code from the previous page again, for ease of reference.

```
class Dog:
    scientific_name = 'doggo'
class Retriever(Dog):
    fur_color = 'gold'
```

Notice the parentheses. This tells us `Retriever` inherits from `Dog`, or in other words, it builds on top of the `Dog` class.

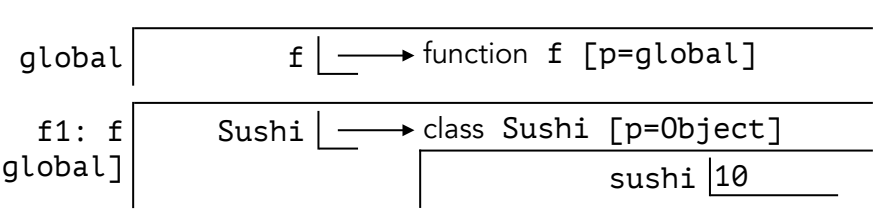
What happens when we look up `scientific_name` inside `Retriever`? It's not defined, but `Retriever` inherits from `Dog` and we can find `scientific_name` in `Dog`, so we're fine. This might remind you of how a function can look up variables in its parent frame, and that's exactly what's going on here! `Retriever` is basically a frame whose parent frame is `Dog`, and that's how we'll represent it.



The variable `Dog` gets bound to a class called `Dog`, and the variable `Retriever` gets bound to a class called `Retriever`. Each class can have its own variables, rather like the frames we've seen used for function calls in the past. Each class has a parent frame, too. For `Retriever`, that parent frame is `Dog`; for `Dog`, that parent frame is `Object`. You don't have to know much about the `Object` class, but you can think of it sort of like the global frame for objects instead of functions. If a class doesn't inherit from anything, it really inherits from `Object`.

Like any other variable, we can also define classes within functions.

```
def f():
    class Sushi:
        sushi = 10
f()
```



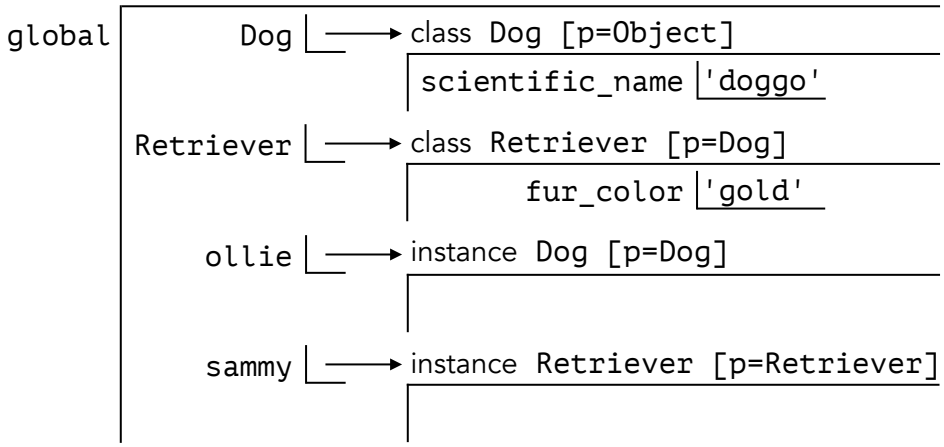
Check that this makes sense so far, before moving on.

Instances

You can think of a class like a template. The `Dog` class describes every `Dog`, and the `Retriever` class describes every `Retriever`. But what if we want to get even more specific than describing *every* `Dog`, or *every* `Retriever`? What if we want to talk about one very specific dog, like Sammy? This is where instances come in. Look at the code below. By "calling" a class like you would call a function, we can make a specific instance of that class. For example, in the code below `ollie` is a `Dog` and `sammy` is a `Retriever`.

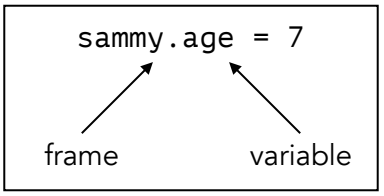
```
ollie = Dog()
sammy = Retriever()
```

When we draw them in an environment diagram, we label each individual `Dog` or `Retriever` as an instance, rather than a class.



Notice, `ollie` and `sammy` both start out with no variables defined inside their respective frames. The difference between them is just that `ollie`'s parent frame is `Dog`, whereas `sammy`'s parent frame is `Retriever`. That means we could look up `fur_color` or `scientific_name` in `sammy`, but we can't look up `fur_color` in `ollie` because that variable isn't defined in any of his parent frames. This is exactly the same as variable lookup in function frames, like you've seen before.

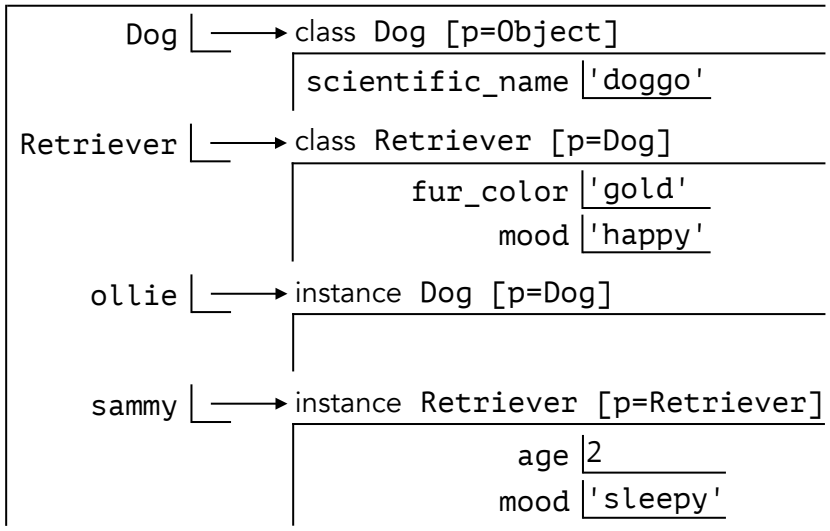
But this is not very useful yet. We should be able to describe each instance in more detail. How can we specify `sammy`'s age, or mood? For this, we need dot notation. Dot notation gives us the ability to talk about variables within classes or instances.



Put this on your cheat sheet. It is important, and we will use it repeatedly throughout the chapter.

In dot notation, the name before the dot specifies what frame we should be looking at. The name after the dot specifies the variable we're talking about within that frame.

```
ollie = Dog()
sammy = Retriever()
sammy.age = 2
Retriever.mood = 'happy'
sammy.mood = 'sleepy'
```



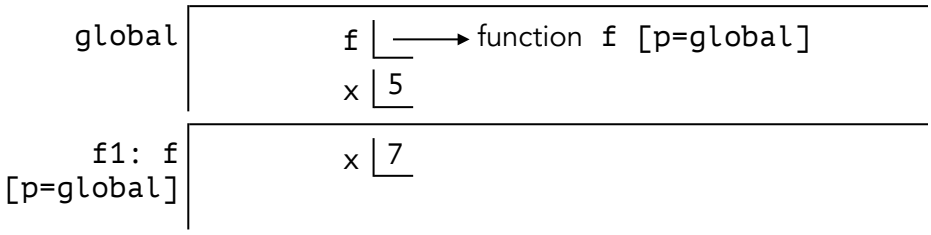
When we write `Retriever.mood`, that specifies the variable `mood` within the frame `Retriever`. When we write `sammy.mood`, that specifies the variable `mood` within the frame `sammy`. So, when we say `sammy.mood = 'happy'`, we get the variable `mood` in the frame `sammy`, bound to the string `'happy'`.

The variables in the `sammy` frame are called *instance attributes*. The variables in the `Retriever` frame are called *class attributes*. I put a low priority on terminology, so don't worry about memorizing this, but you may hear these words thrown around here and there.

Quick Aside on Assignment and Lookup

This is no different from what you've been doing all semester long, but it's worth re-iterating now. Imagine you have two frames open, like in the diagram below.

```
x = 5
def f():
    x = 7
f()
```



Notice that when we assign `"x = 7"` in frame `f1`, we don't change the variable `x` in `f1`'s parent frame. Rather, we make a new variable within `f1`, also called `x`. We have seen this throughout the semester. Variable assignment only happens in the current frame, *not* any of the parent frames.

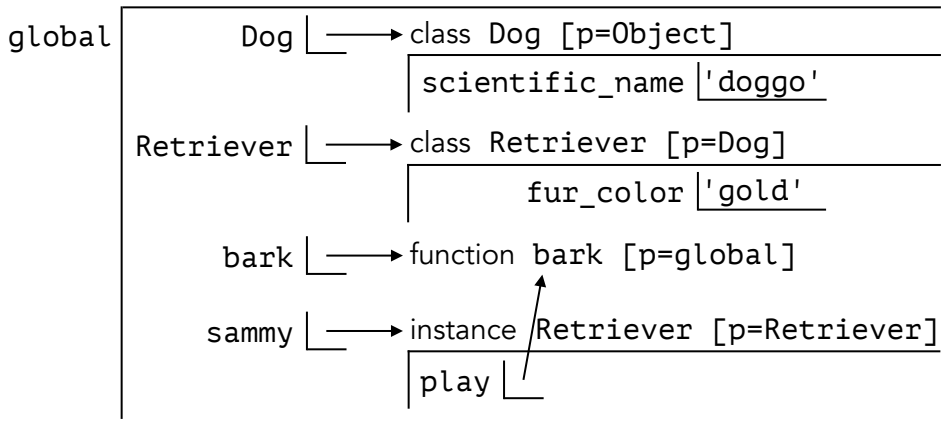
The same idea extends to object oriented programming. Assigning `sammy.mood` doesn't affect `Retriever.mood`. Instead we just make a new variable called `mood` in the frame `sammy`.

Functions in Classes

Now we have seen that classes and instances can have variables inside them. Well, variables can be bound to functions too, can't they? This is exciting. It means we can have classes and instances, with functions inside them. Here's an example:

```
def bark(x):
    print('woof!', x)

sammy.play = bark
```



Notice the parent of `bark` is `global`, since it was defined in the `global` frame.

Ok, but let's go a little further. What happens if we put a function inside a class definition?

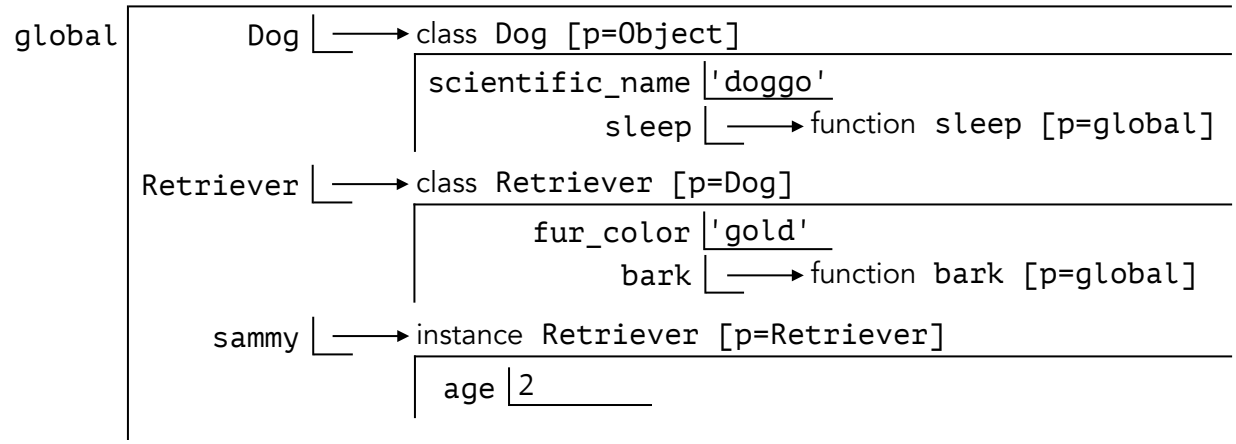
```
class Dog:

    scientific_name = 'doggo'
    def sleep():
        print('zzz')

class Retriever(Dog):

    fur_color = 'gold'
    def bark(x):
        print('woof!', x)

sammy = Retriever()
sammy.age = 2
```



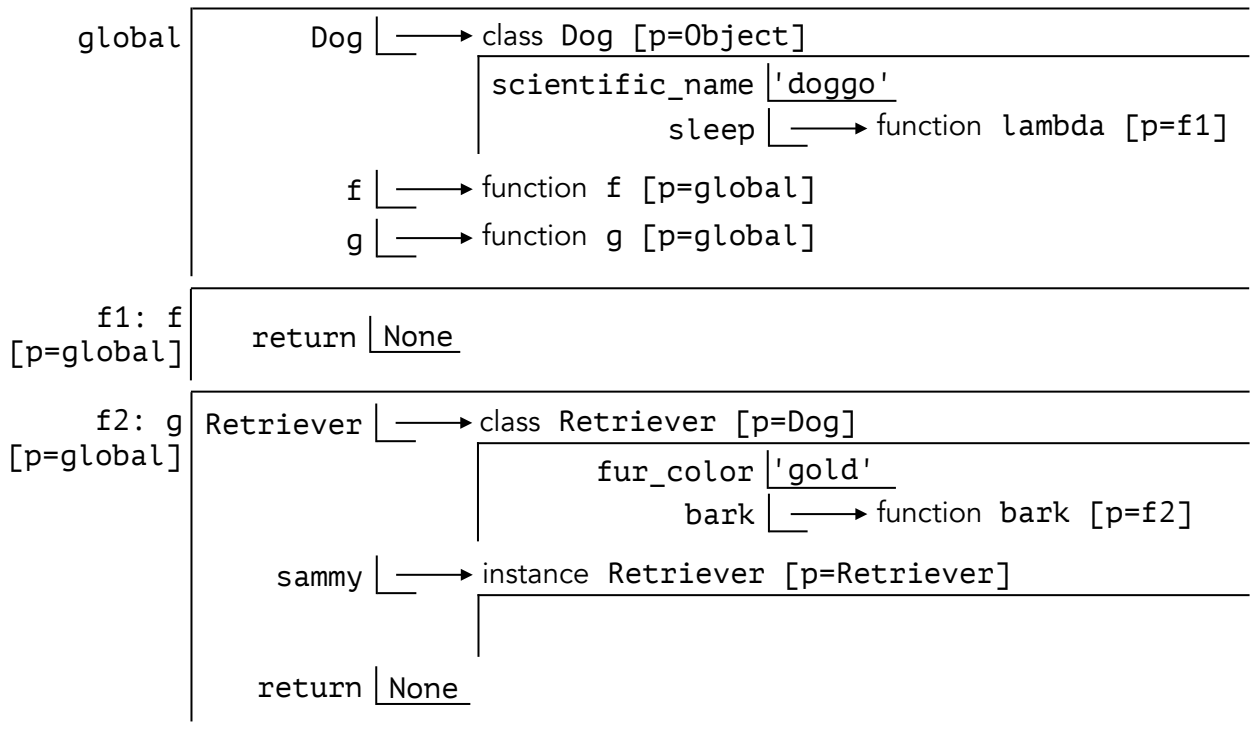
Look at the environment diagram above. `bark` gets defined inside the `Retriever` class, so we keep it inside the `Retriever` frame. But notice its parent is `global`, not `Retriever`! **This is because the parent of a function has to be a function frame, and the parent of an object has to be an object frame!** What does that mean, exactly?

The kind of frames we have seen so far in this course have all been *function frames*. These include `global`, and any frames that get opened when you call a function. We usually name them things like `f1`, `f2`, `f3`, and so on. On the other hand, *object frames* are the ones that correspond to a class or an instance. In the diagram above, `Dog`, `Retriever`, and `sammy` are all bound to object frames.

The sentence in bold says that the parent of a function has to be a function frame. For instance, the parent of `bark` is `global`. Note, the parent of `bark` is not allowed to be `Retriever` because `bark` is a function, and `Retriever` is an object frame not a function frame. Similarly, the parent of an object has to be an object frame. So, the parent of `sammy` is the object frame `Retriever`, and the parent of `Retriever` is the object frame `Dog`.

```
class Dog:
    scientific_name = 'doggo'
def f():
    Dog.sleep = lambda: print('zzz')
def g():
    class Retriever(Dog):
        fur_color = 'gold'
        def bark(x):
            print('woof!', x)
    sammy = Retriever()
f()
g()
```

`global` isn't always the parent of a function inside a class. In this example, the parent of `Dog.sleep` is `f1` because it is bound to a `lambda` function created in `f1`. The parent of `Retriever.bark` is `f2` because `Retriever` was made in the frame `f2`, and since `bark` is defined within `Retriever`, that means `bark` was made in `f2` as well.



There's a lot of information on the previous page, so make sure you understand it before moving on. Read it multiple times if that helps. The important part is just this:

> **The parent of a function has to be a function frame, and the parent of an object has to be an object frame.**

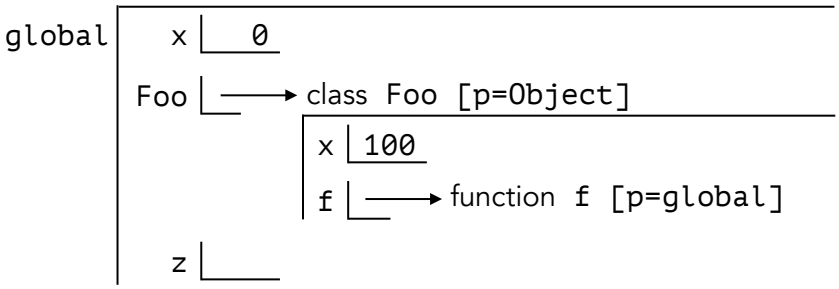
Also, you might hear a function inside a class referred to as a *method*. Terminology isn't very important but you can learn it if you want to.

Practice: Variable Lookup in Classes

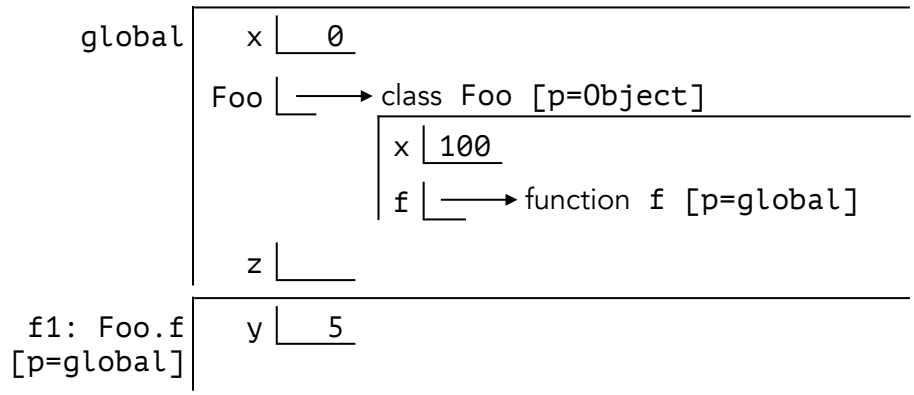
Take a look at the code below.

```
x = 0
class Foo:
    x = 100
    def f(y):
        return x + y
z = Foo.f(5)
```

The goal is to figure out what `z` is. Let's get started by drawing the `global` frame.

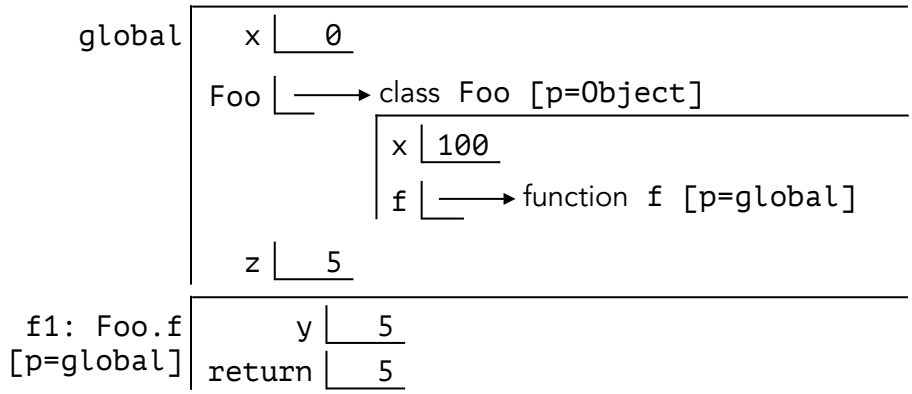


Next we call the function `f` within the frame `Foo`. Using dot notation, which we learned a few pages ago, we express this as `Foo.f`. We pass in the value 5 for its parameter `y`.



The return value of `Foo.f` is `x+y`, so let's look up those variables from our current frame. `x` is not defined within `Foo.f`, so we look at the parent frame, which is `global`. There, we see `x` is bound to `0`. Then we look up `y`, and see it's bound to 5. We return `0+5`, which is 5.

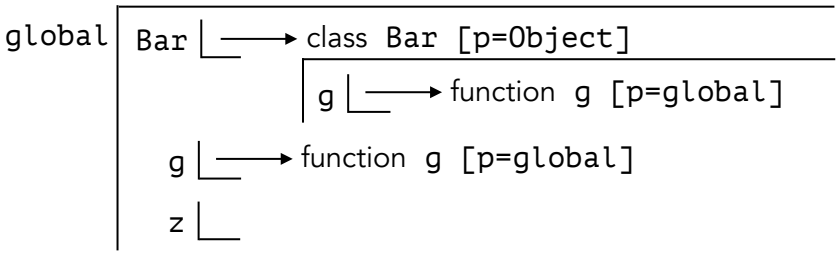
The final environment diagram is below. In the end z is bound to 5, *not* 105. This is because the parent of Foo.f is global, so we used the version of x we found in global instead of the version we would find in Foo.



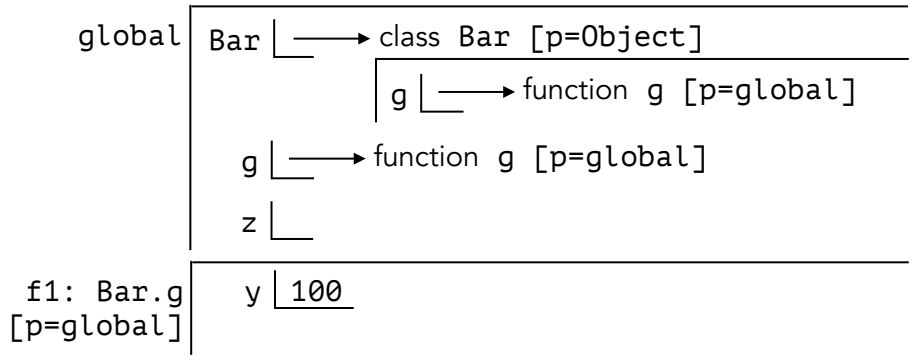
Let's do another example. Try doing this one on your own, before reading the answer.

```
class Bar:
   def g(y):
        return g(y)
def g(y):
    return 0
z = Bar.g(100)
```

As always, let's start by drawing the global frame.

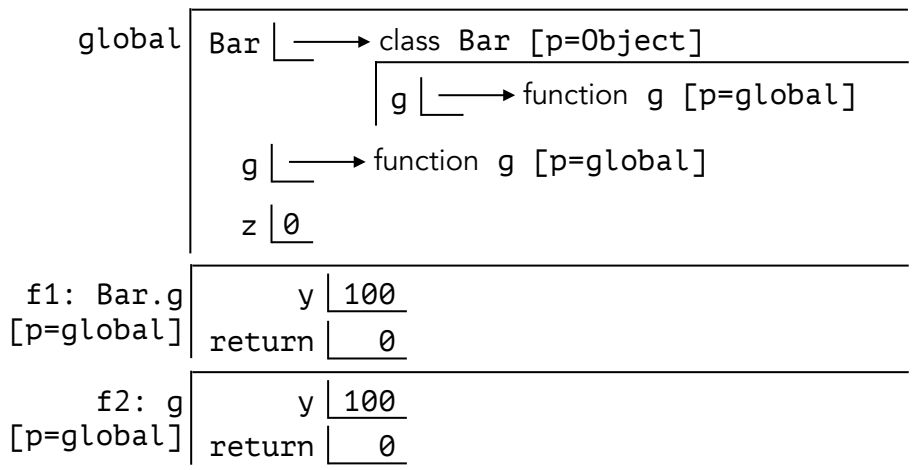


Next we call the function g within the frame Bar. Using dot notation, we express this as Bar.g. We pass in the value 100 for its parameter y.



Bar.g just returns a call to g. When we look up g from frame f1, we don't see it so we have to search the parent frame, which is global. We use the version of g that we find there.

Since we're calling the version of g inside `global`, and not the version inside `Bar`, we will open `f2` for `g` instead of `Bar.g`. If you look at the version of `g` inside `global`, you'll see it just returns `0`.



In the end, `z` gets bound to `0`. We don't get a `RecursionError` when `Bar.g` calls `g`, because these are two different functions! Check both of these examples make sense before continuing.

Built-in Functions

There are some functions that are so commonly useful, the creators of Python wrote them for us. We've seen a few so far: `str`, `len`, and even < are a few examples.

```
>>> str(4)
'4'
>>> len('alien overlords')
15
>>> 1 < 2
True
```

It's nice that we can use these built-in functions on primitives like integers and strings. But can we use them for objects too? Let's try it.

```
>>> sammy = Retriever()
>>> beaux = Retriever()
>>> sammy.age = 2
>>> beaux.age = 1
>>> sammy < beaux
TypeError: '<' not supported between instances of Retriever
```

Yikes. But lucky for us, there's a way to implement these built-in operators for classes that we write. We do it using special functions that have two underscores on either side of their name. For example, in order to be able to use the < operator with instances of the `Retriever` class, we need to write a function called `__lt__` (which stands for "less than") inside `Retriever`.

More specifically, when we write something like `dog1 < dog2`, Python will automatically convert it to `Retriever.__lt__(dog1, dog2)`. Don't worry about memorizing any specifics here, but make sure you understand the example below.

```
>>> class Retriever:
...     def __lt__(dog1, dog2):
...         return dog1.age < dog2.age
>>> sammy = Retriever()
>>> beaux = Retriever()
>>> sammy.age = 2
>>> beaux.age = 1
>>> sammy < beaux
False
```

But < isn't he only built-in operator we can use in classes that we write. In order to for us to use the function `str`, for example, we can also define the function `__str__` inside `Retriever`. Then, when we call the built-in function `str` on an instance of `Retriever`, Python will automatically convert it to a call on `Retriever.__str__`. Below, `str(sammy)` automatically gets turned into `Retriever.__str__(sammy)`.

```
>>> class Retriever:
...     def __str__(dog):
...         return 'happy puppy'
>>> sammy = Retriever()
>>> str(sammy)
'happy puppy'
```

In fact, for *every* Python built-in there's a corresponding function that you can define inside a class that you write. If you want to support `len`, then define `__len__`; if you want to support `bool` then define `__bool__`; the list goes on and on.

```
>>> class Retriever:
...     def __len__(dog):
...         return dog.head - dog.tail
>>> sammy = Retriever()
>>> sammy.head = 10
>>> sammy.tail = -5
>>> len(sammy)
15
```

Again, you don't have to know all these by heart or anything like that. Just know that if you want to use a built-in operator on an instance of a class you write, then there's a corresponding function that you need to define within that class. If you want a full list of the Python built-ins and what function corresponds to each of them, then you can google search "list of all Python magic methods". Only keep reading, once you understand this.

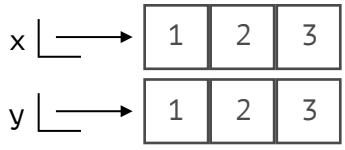
A Few Important Built-ins to Know

Now that we have seen how to handle built-in functions with classes, there are a few common ones that you should probably keep in the back of your mind.

1. `__eq__`
This function corresponds to the == operator. For example, `sammy == beaux` automatically gets converted into `Retriever.__eq__(sammy, beaux)`.

```
>>> class Retriever:
...     def __eq__(dog1, dog2):
...         return dog1.age = dog2.size
>>> sammy = Retriever()
>>> beaux = Retriever()
>>> sammy.age = 2
>>> beaux.size = 2
>>> sammy == beaux
True
```

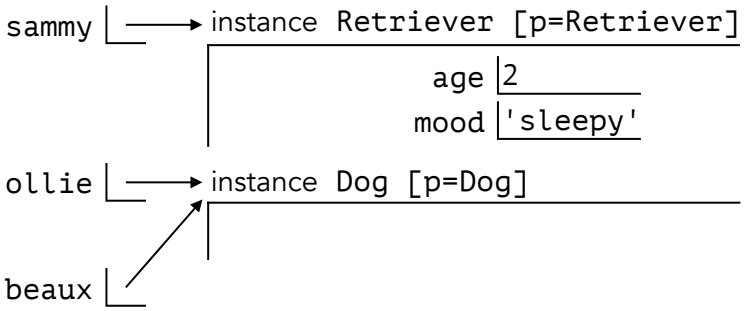
`__eq__` is a little special, because it is implemented in every class by default. If you don't redefine it on your own, then it will behave the same as the built-in function `is`, which is like ==, but instead of comparing values it compares whether two arrows point to the same thing.



In this example, x and y point to lists that have the same values, but really x and y are pointing to *different* lists, even if those lists are very similar.

```
>>> x == y
True
>>> x is y
False
```

In this example, `sammy` and `ollie` point to two totally separate instances, so `sammy is not ollie`. However, `ollie` and `beaux` both point to the same exact instance so `ollie is beaux`.




```
>>> sammy is ollie
False
>>> beaux is ollie
True
```

2. `__str__`

Like we saw before, we can define `__str__` in order to use the built-in function `str`. In fact, this also changes the way our class gets printed! This is because `print(x)` basically displays the result of `str(x)`, just without the quote marks shown on the sides.

```
>>> class Retriever:
...     def __str__(dog):
...         return 'stormageddon, dark lord of all'
>>> sammy = Retriever()
>>> str(sammy)
'stormageddon, dark lord of all'
>>> print(sammy)
stormageddon, dark lord of all
```

If you try to use `str` or `print` on an instance of a class that doesn't have a `__str__` function, then Python will use the `__repr__` function instead.

3. `__repr__`

This corresponds to what gets displayed when you evaluate something in the terminal.

For example, when you write this:
`>>> sammy`
the terminal will display, without quotes, the result of `Retriever.__repr__(sammy)`.

```
>>> 5
5
```

← Here, we evaluate the number 5. Since `repr(5)` returns `'5'`, we display 5 without quotes.

```
>>> class Retriever:
...     def __repr__(dog):
...         return 'cyberdog'
>>> sammy = Retriever()
>>> sammy
cyberdog
```

← Here, we evaluate `sammy`, which is an instance of `Retriever`. Since `Retriever.__repr__(sammy)` returns `'cyberdog'`, we display `cyberdog` without quotes.

Initializing New Objects with `__init__`

So far, we have been making new instances by first creating them, and then one-by-one assigning their variables. For example:

```
>>> sammy = Retriever()    # Create sammy
>>> sammy.age = 2          # Assign sammy an age
>>> sammy.mood = 'sleepy'  # Assign sammy a mood
```

There's a very important function that lets us wrap this all into one step. It's called `__init__`.

When you make a new instance of a class, Python will automatically call `__init__` on the instance being made. For example, consider this code:

```
>>> sammy = Retriever()
```

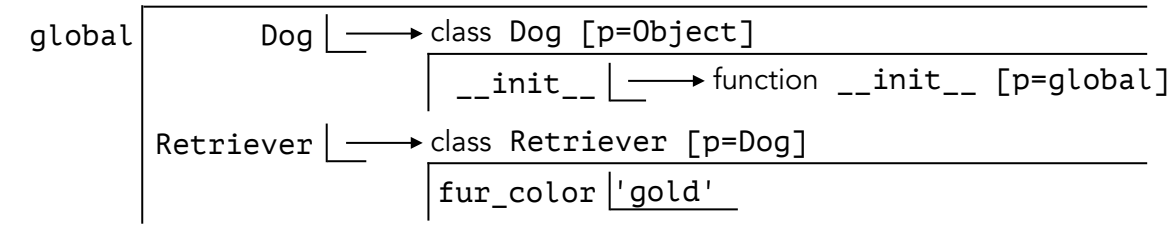
What *really* happens is this, if `Retriever` has an `__init__` function:

```
>>> sammy = Retriever()
>>> Retriever.__init__(sammy)
```

Usually we use `__init__` to assign a bunch of variables, like `age` and `mood`, without having to do it explicitly ourselves. Let's do an example in an environment diagram, to see how `__init__` works in more detail.

```
class Dog:
    def __init__(dog):
        dog.age = 2
        dog.mood = 'sleepy'
class Retriever:
    fur_color = 'gold'
sammy = Retriever()
```

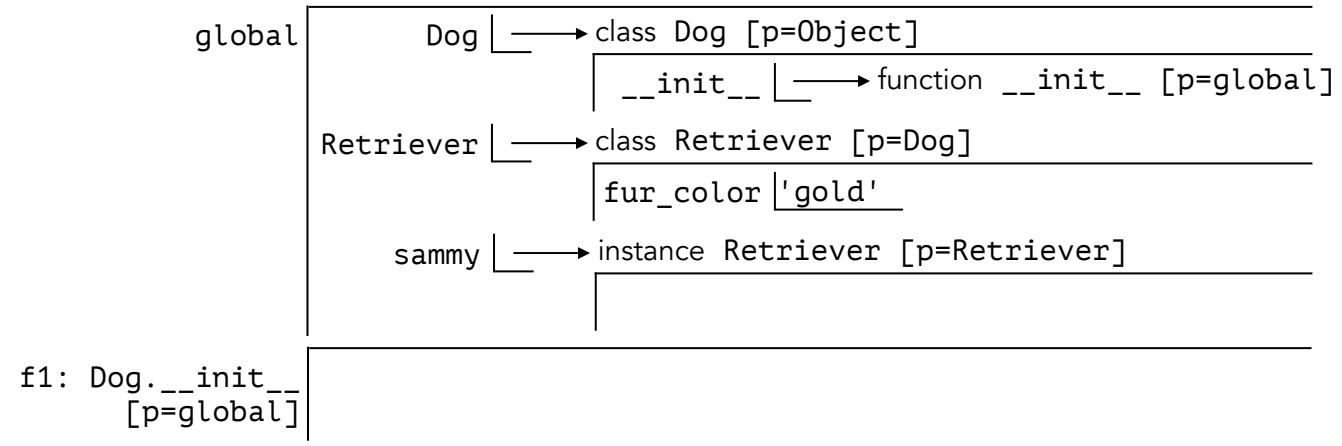
First things first, we should set up the `global` frame.



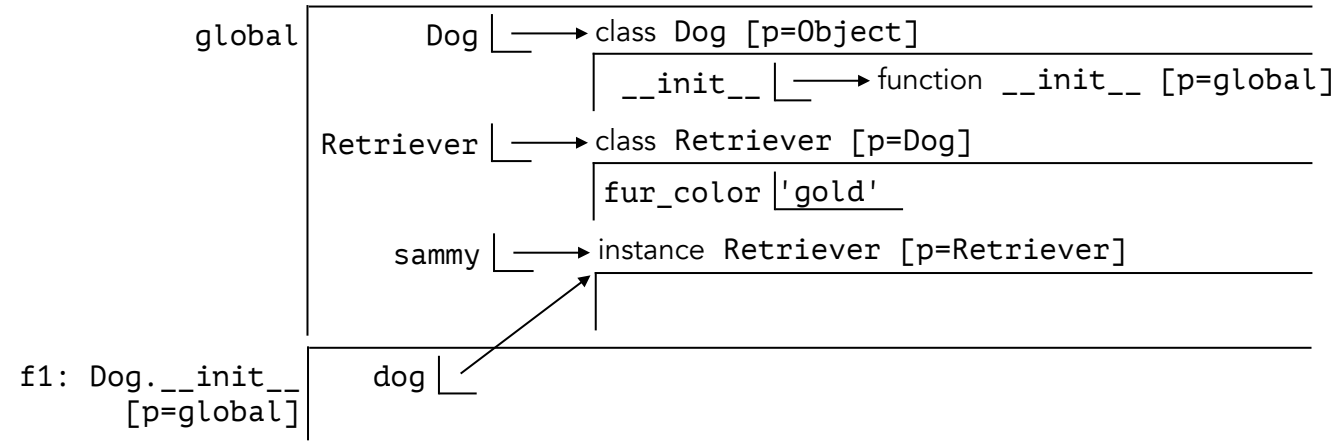
The next line to execute is `sammy = Retriever()`. Remember, what *really* happens is this:

```
sammy = Retriever()
Retriever.__init__(sammy)
```

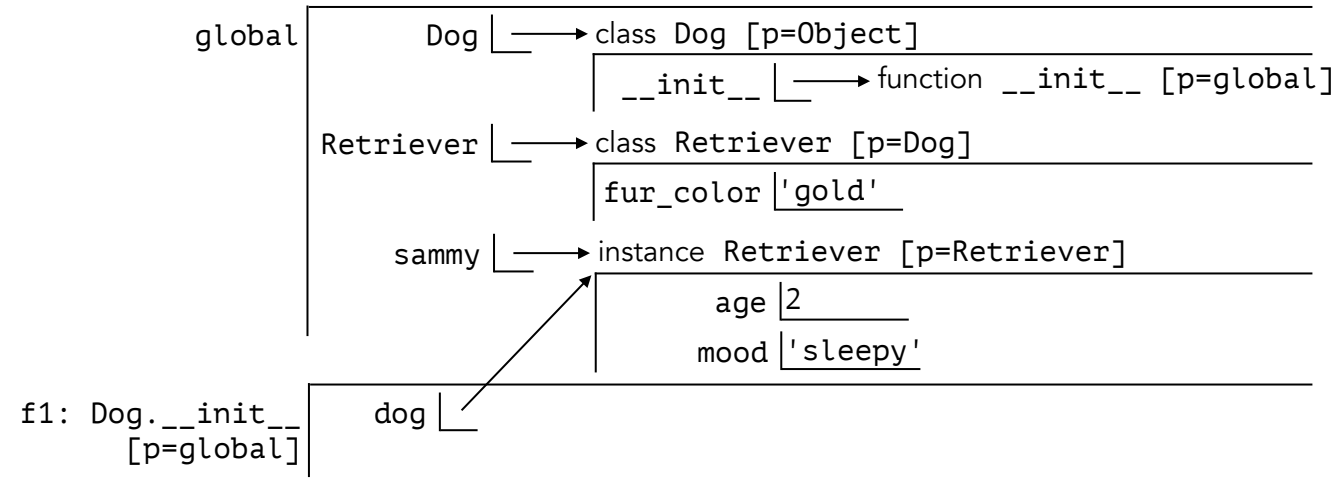
Since `__init__` isn't defined in the `Retriever` frame, we'll use the version from `Retriever`'s parent frame, `Dog`.



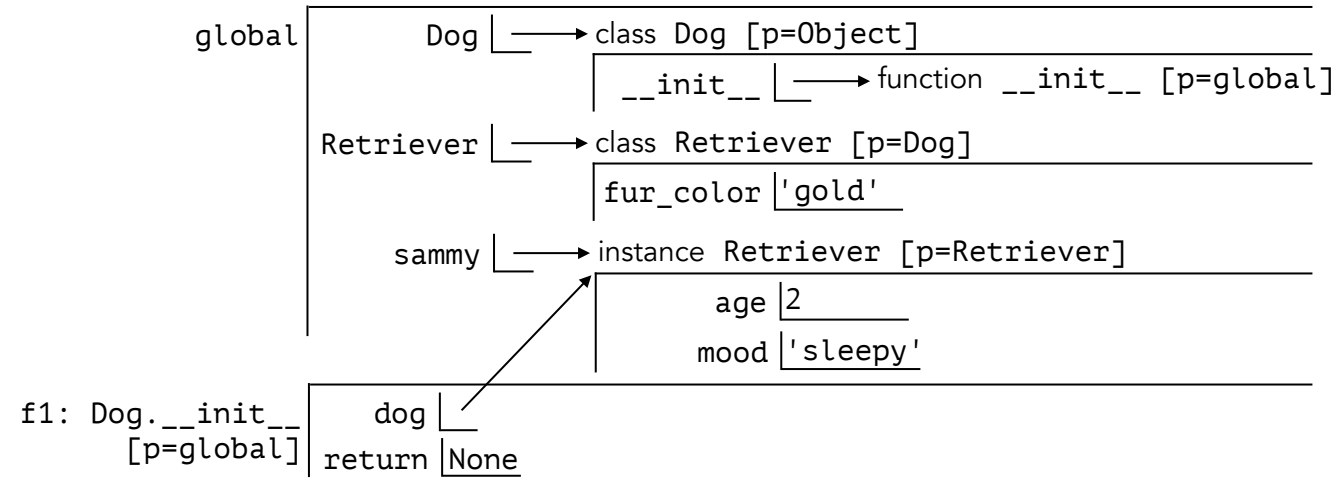
`Retriever.__init__` takes one parameter called `dog`, and we passed in `sammy`.



Then, inside frame `f1`, we see `dog.age = 2` and `dog.mood = 'sleepy'`. Recalling dot notation from earlier, that means we are assigning two variables `age` and `mood`, inside the frame that `dog` refers to.



And last of all, since `Retriever.__init__` doesn't specify a return value, it will return `none`.



In the end, the result is the same as what we had before:

```
sammy = Retriever()    # This is the same.
sammy.age = 2          # This happens in Dog.__init__ now.
sammy.mood = 'sleepy'  # This happens in Dog.__init__ now.
```

But still, this isn't very useful yet. In the example above, every new `Dog` would have its `age` set to 2, and its `mood` set to `'sleepy'`. We can make `__init__` more versatile by giving it more parameters. The first one will still be the new dog that we're initializing, but we can add as many other parameters as we like after that.

```
class Retriever:
    def__init__(dog, age, feeling):
        dog.age = age
        dog.mood = feeling
sammy = Retriever(2, 'sleepy')
```

`sammy` gets passed in for `dog`.

2 gets passed in for `age`.

`'sleepy'` gets passed in for `feeling`.

Make sure you understand everything so far, before reading on.

Calling Functions from Instances and Classes

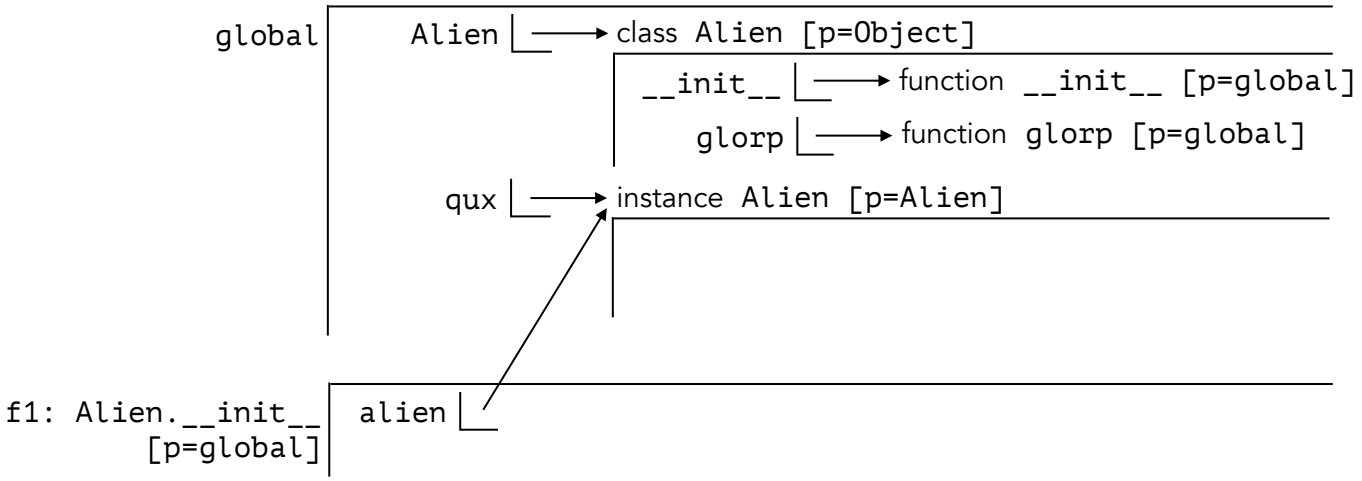
In fact, functions behave differently depending on whether we call them from a class or an instance. **If an instance calls a parent class' function, then that instance will pass itself in as the first argument to the function. Otherwise, everything works like normal and you have to pass in all the arguments manually.**

Write this down somewhere. It's important.

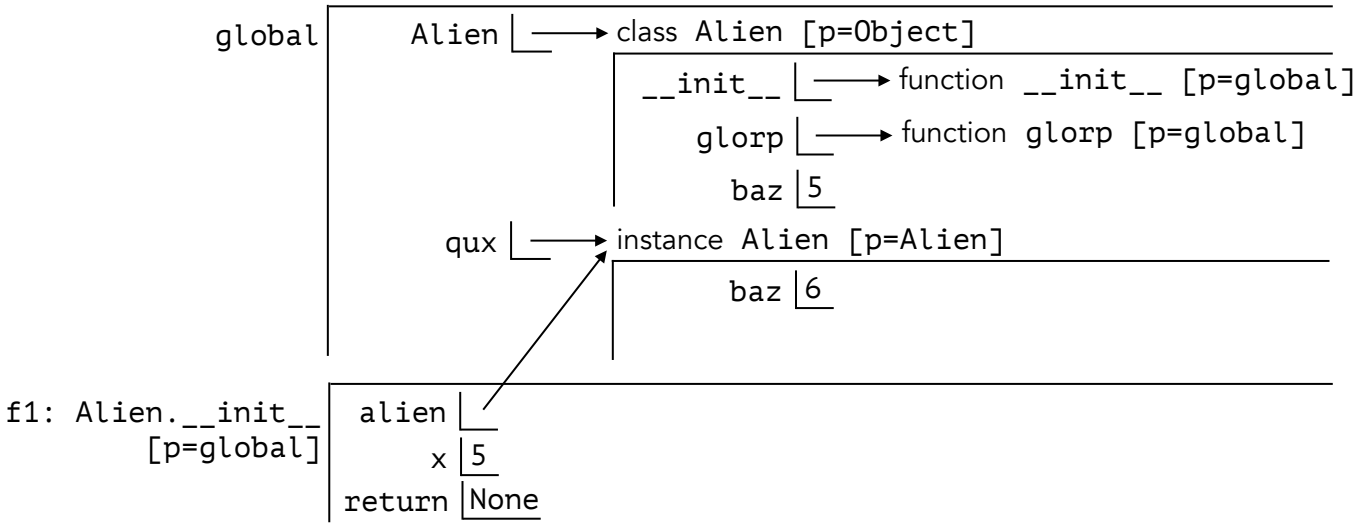
For example, let's consider the code below.

```
class Alien:
    def __init__(alien):
        x = 5
        Alien.baz = x
        alien.baz = x+1
    def glorp(self, fizz):
        return self or fizz
qux = Alien()
x = Alien.glorp(0, 7)
y = qux.glorp(0)
qux.glorp = lambda s: Alien.glorp
z = qux.glorp(0)
Alien().glorp(8)
```

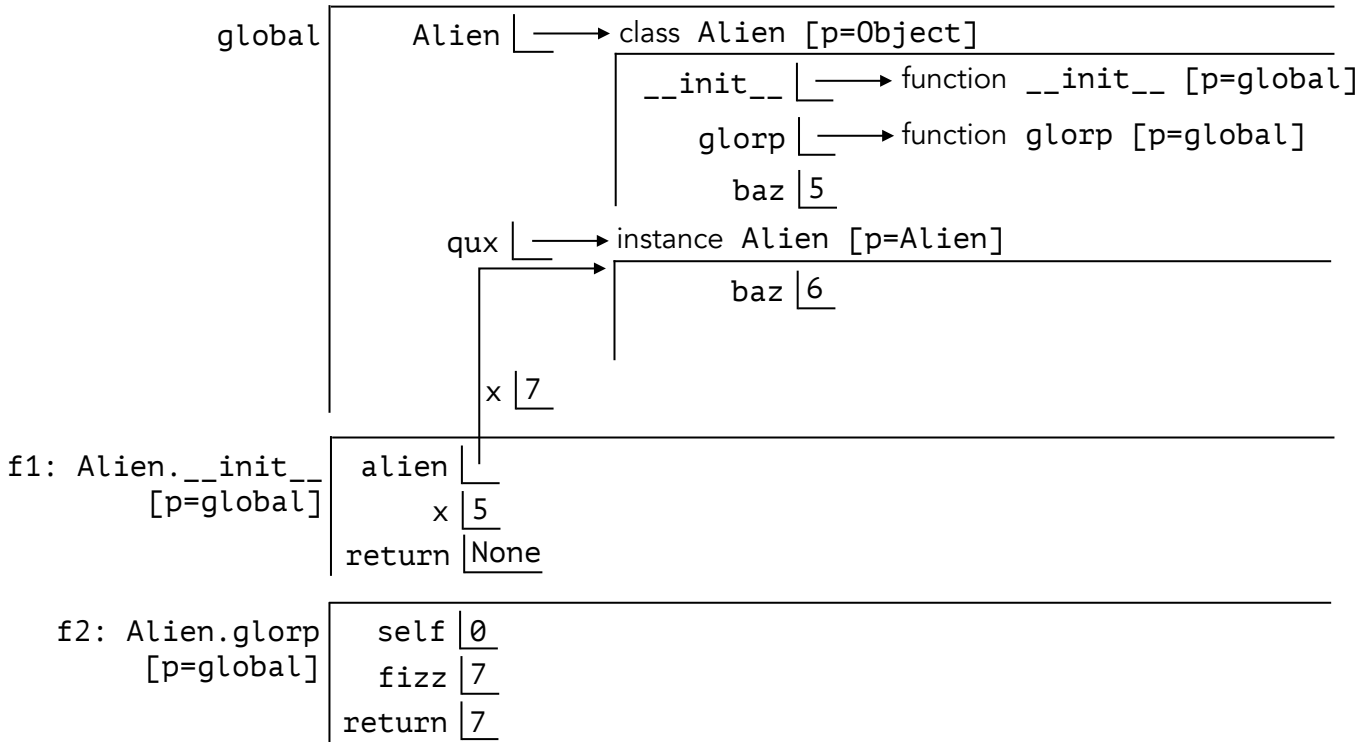
As always, the first step is to make the `global` frame. We'll start by defining the `Alien` class, and then making the instance `qux`, and finally calling `Alien.__init__` on `qux`.



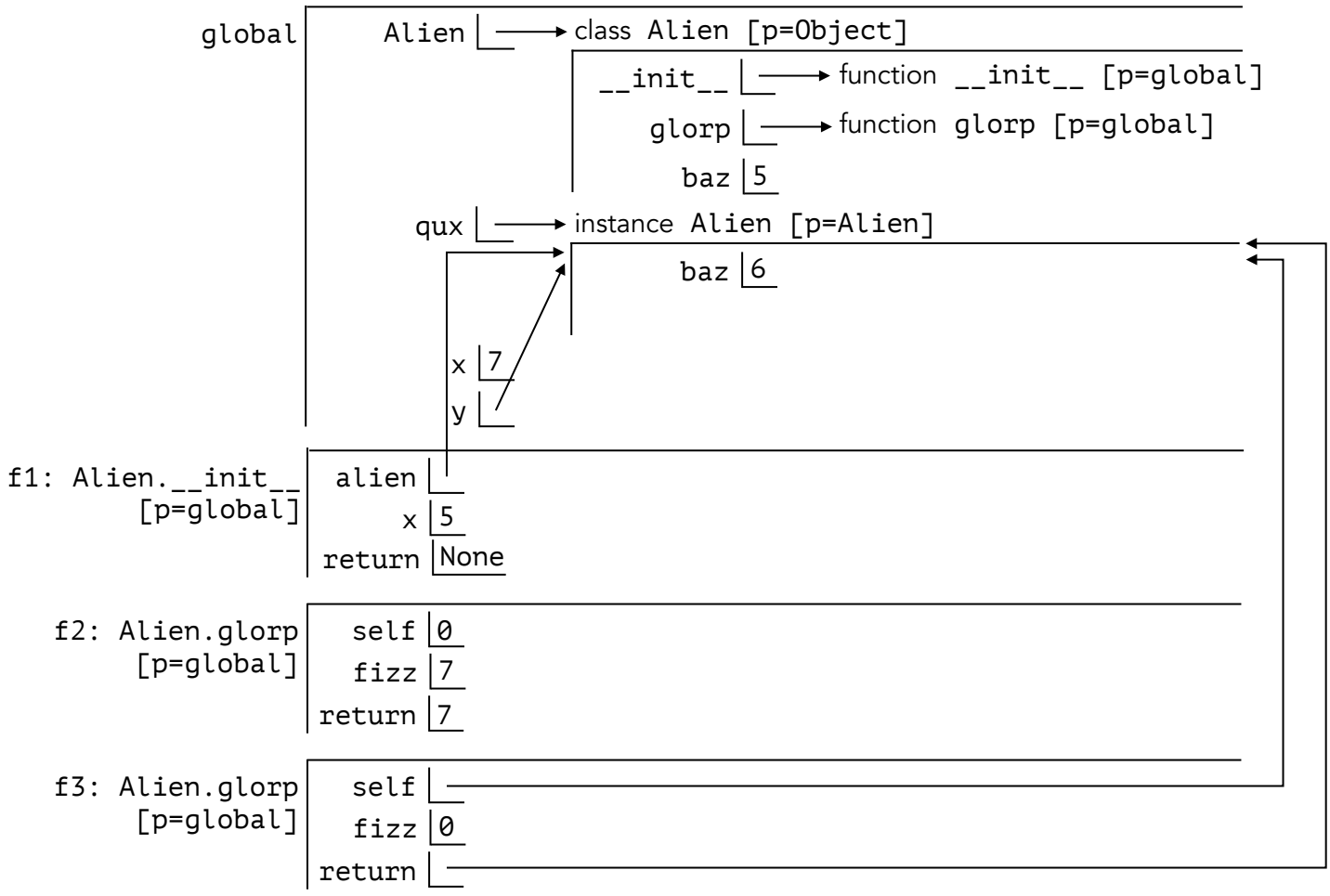
In frame `f1`, the first thing we do is assign the variable `x` to `5`. Just like with any function call, this happens within `f1` since there's no dot notation telling us to assign `x` in a different frame. Then we assign `Alien.baz` (which will happen in the frame `Alien` refers to) as well as `alien.baz` (which will happen in the frame `alien` refers to).



Now we see the line `x = Alien.glorp(0, 7)`. Remember the bold part from earlier, the one you should have written down by now: **If an instance calls a parent class' function, then that instance will pass itself in as the first argument to the function. Otherwise, everything works like normal and you have to pass in all the arguments manually.** Since `Alien.glorp` is not an instance calling a parent class' function, `Alien` does not pass itself in as the first argument and we treat it like a normal function call.



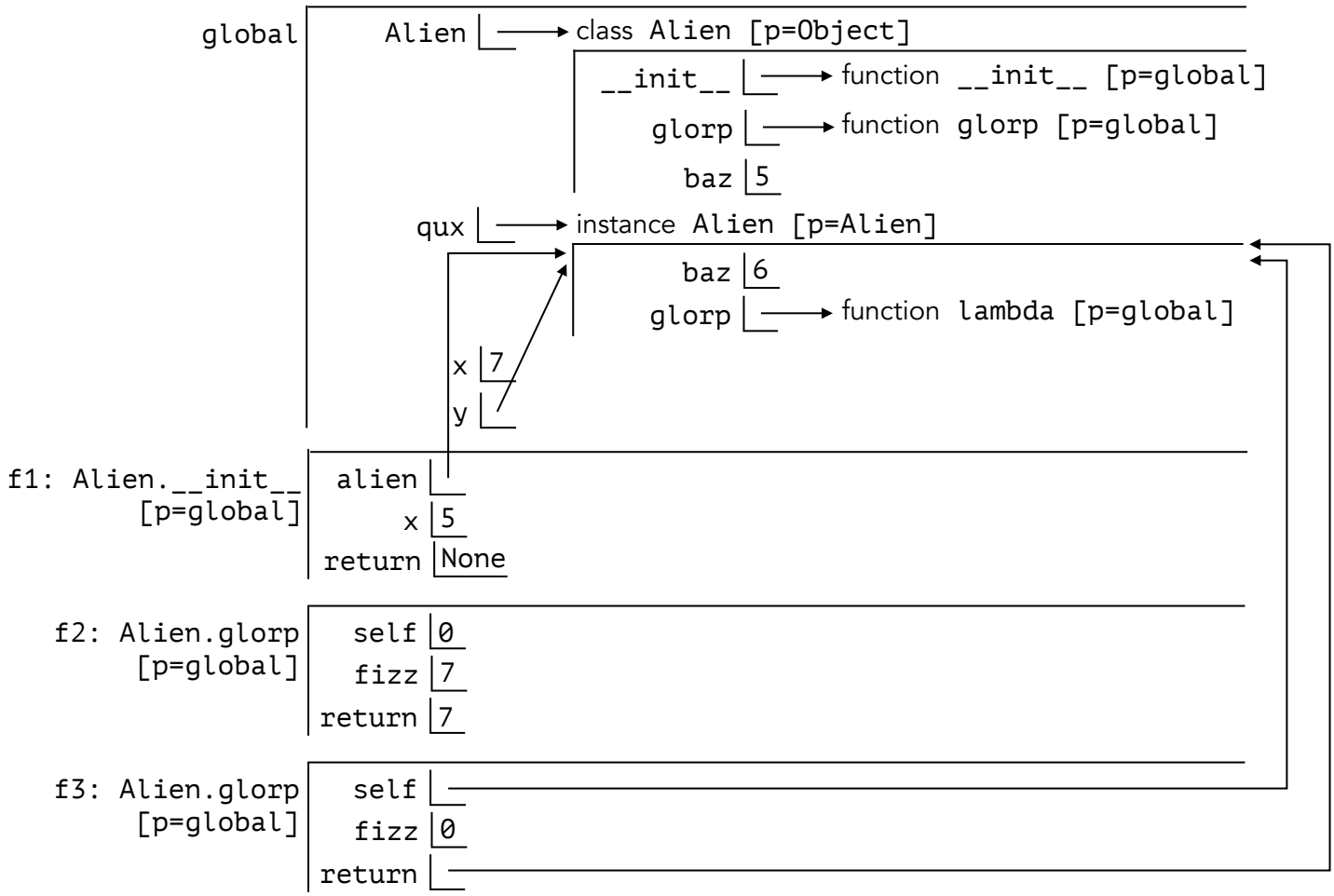
Next, we see `qux.glorp(0)`. The instance qux is calling `Alien.glorp`, which is in a parent class of qux! Referring back to the bold text on the previous page, that means qux will pass itself in as the first argument to `Alien.glorp`. The second argument, `0`, is provided as usual.



For convenience, here is the code again.

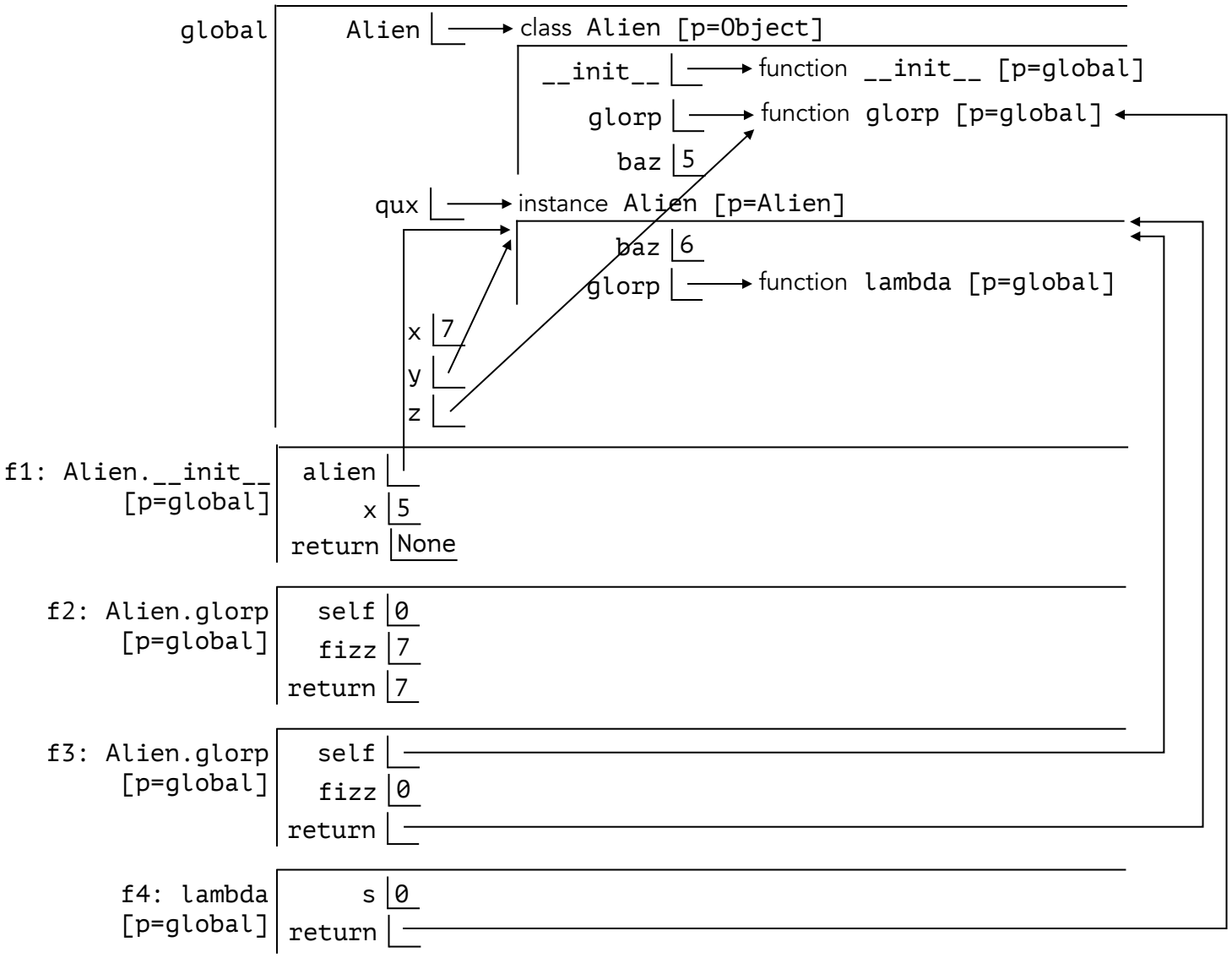
```
class Alien:
    def __init__(alien):
        x = 5
        Alien.baz = x
        alien.baz = x+1
    def glorp(self, fizz):
        return self or fizz
qux = Alien()
x = Alien.glorp(False, 7)
y = qux.glorp(0)
qux.glorp = lambda s: Alien.glorp
z = qux.glorp(0)
Alien().glorp(8)
```

We left off on `qux.glorp = lambda s: Alien.glorp`.

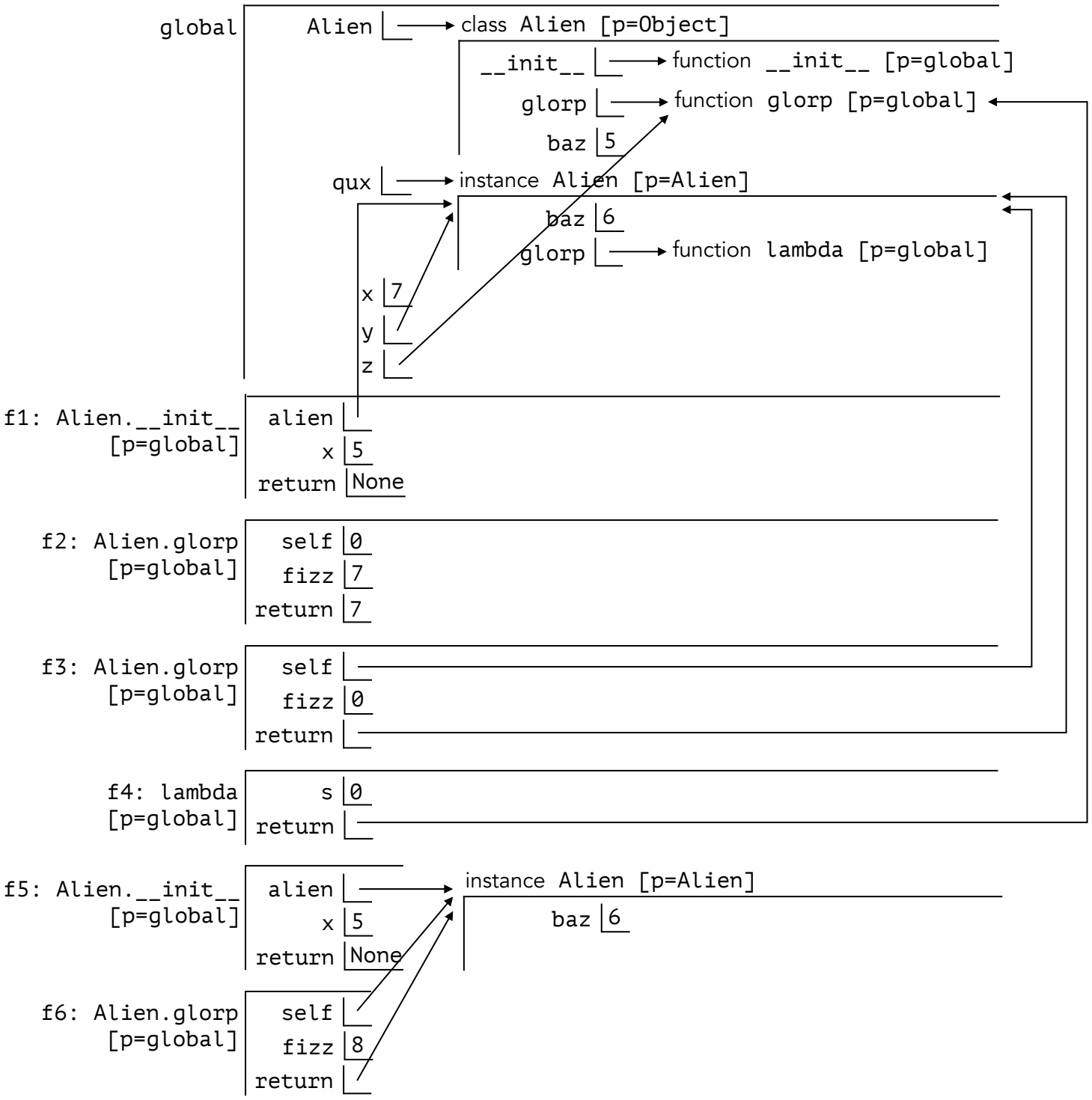


This line did something very important. Now, `qux.glorp` no longer refers to a function in a parent class of `qux`! Referring back to the bold text from before, that means `qux` will no longer pass itself in as the first argument to `qux.glorp`.

When we write `z = qux.glorp(0)`, we're now referring to the `lambda` function inside `qux`'s instance frame. Since this is *not* in a parent frame of `qux`, we will *not* pass `qux` in as the first argument to the function call.



But next, we have the line `Alien().glorp(8)`, a very different thing from `Alien.glorp(8)` which would throw an error because it doesn't have the right number of parameters. Rather, `Alien().glorp(8)` does two things. First we have the `Alien()` part, which makes a new instance of the `Alien` class. Then, from that new instance, we call the function `glorp`, which refers back to the version in the `Alien` class because the `lambda` version of `glorp` is specific to the instance `qux`, and `qux` alone. Again, refer back to the bold text from a few pages ago. Since the new instance is calling the version of `glorp` in its parent class, that means it will pass itself in as the first argument.



This was a pretty involved problem, and it involves a lot of edge cases. Review it and make certain you know why everything happens as it does. If you can do that, then you're in good shape. That said, there are a few more slippery details that aren't covered here but they can wait for later. We'll go over them in the chapter about bound methods.

Accessing Object Types

With all that out of the way, there's one very last thing to learn about. Sometimes we need to write programs that handle different types of objects uniquely. For example, Python secretly uses an `Integer` class to represent whole numbers and a `Float` class to represent decimal numbers. If you wanted to write a snazzy maths function that behaves differently depending on the type of its input, then you would have to figure out first whether you're dealing with an `Integer` or a `Float`. In Python, we have two handy functions that can help us in these situations.

`1. type`
This one tells you what class an object belongs to. In the example below, `type(sammy)` literally evaluates to `Retriever`, so writing `beaux = type(sammy)()` is the same exact thing as writing `beaux = Retriever()`.

```
>>> class Retriever:
...     fur_color = 'gold'
>>> sammy = Retriever()
>>> type(sammy) == Retriever
True
>>> beaux = type(sammy)()
>>> beaux.fur_color
'gold'
```

`2. isinstance`
This one evaluates to `True` or `False`, depending on whether the first argument is an instance of the second argument. In the example below `beaux` is an instance of `Dog` but not an instance of `Retriever`, while `sammy` is an instance of both `Dog` and `Retriever`.

```
>>> class Dog:
...     scientific_name = 'doggo'
>>> class Retriever(Dog):
...     fur_color = 'gold'
>>> beaux = Dog()
>>> sammy = Retriever()
>>> isinstance(beaux, Dog)
True
>>> isinstance(beaux, Retriever)
False
>>> isinstance(sammy, Retriever)
True
>>> isinstance(sammy, Dog)
True
```

That's all! This was a long chapter, but also an important one. Review the examples when you can and double check all the concepts make sense. Objects are a powerful programming tool.