Environment Model

You are not logged in.

Please Log In for full access to the web site.
Note that this link will take you to an external site (`https://shimmer.mit.edu`) to authenticate, and then you will be redirected back to this page.

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

How To Use These Readings

When learning to program, it is not enough just to listen or to read; you need to practice! In this course, readings will not only introduce new ideas; they will also ask you to participate.

There will be times on this page when you are asked to try things on your own (some of these may be "checked" for correctness by our system, but some are just suggested activities). Either way, it is

important to do (and understand) all of these activities. If you are having trouble understanding why something behaved as it did (even if you were able to produce the expected result), please ask for help! We will try to provide you with all the materials and support you need, but it is up to you to make use of them (and also to let us know how best we can help you!).

At several points throughout the 6.101 readings, we will ask you to stop and make a guess about something before trying it out using Python. We ask you to do this because research[1] shows that the process of thinking through a problem and making an educated guess helps you remember the answer better when you come across it, compared to simply being told the answer. So when we ask you to make a guess about something, we encourage you to think through the problem and commit to an educated guess before continuing on.

Another common pattern in these readings will involve asking you to run code examples on your own machine to observe the results. In these cases, we strongly enourage you to type the code out character-by-character into your Python environment of choice rather than using copy/paste. While it may seem tedious, typing out the examples yourself and making sure they match requires engaging with them on a deeper level than copy/paste does.

Finally, you will see several questions on the page that allow you to check your understanding. We strongly encourage you to work through these problems earnestly. Some of them are easily game-able (multiple-choice questions that allow infinitely many submissions, for example), and it can be tempting to click through a bunch of options mostly at random in order to make the little green checkmark (which indicates correctness) show up; but we're asking you to resist that temptation and *only submit answers to questions when you're reasonably well convinced that they're correct* and to keep going until you get each question right. Even then, once you have the right answer, it's worth taking the time to reflect and make sure you understand *why* it's correct.

For many questions, once you have submitted a correct answer, a new button will appear labeled "View Answer;" clicking that button will show you an answer and, in many cases, an explanation as well. These explanations are considered part of the reading, and you should definitely make use of them; but in order to maximize your learning, it is important that you get the question right on your own and try to make sure you understand why it's right before looking at our answers and explanations (because there is a big difference between watching someone solve a problem and being able to solve it yourself).

1) Introduction

As we mentioned in the previous readings, our goals in 6.101 involve helping you to improve your skills in several different areas related to computation:

- **programming**: analyzing problems, breaking them down, and forming plans
- **coding**: translating those plans into working code

- **debugging**: finding, understanding, and fixing errors

New concerns arise in each of these areas as we move to writing bigger and more complex programs. As such, a central theme of 6.101 is about *controlling complexity*: as our programs grow to a point where the code no longer fits on one screen, and as the big ideas no longer fit in our head all at once, how can we manage things so that we can still reason about those programs?

A critical part of managing complexity involves understanding the system we're working with. This generalizes to any kind of engineering. For example, people building bridges or designing airplanes need to know the relevant physics and materials science for the problems they're trying to solve. Without a deep knowledge of the underlying processes, there's no realistic hope of designing the kinds of big, complex systems that those engineers deal with.

In that sense, software engineering is no different from those other disciplines: in order to effectively engineer solutions to problems involving computation, we need to know something about how the computer works and how it responds to the various inputs we can give to it. In 6.101, we're going to focus on developing a model of Python specifically, but many pieces of this framework will generalize to other languages. This reading will introduce some of the basics of this model, as well as some of the rationale for the particulars of the model. We'll expand on these ideas throughout the term, fleshing out our model piece-by-piece so that as we learn new features and syntax of Python, we'll also learn how they fit into this model (i.e., what's happening "behind the scenes" when we use those things).

2) A Motivating Example

Before we get too far, let's start by looking at an example Python program:

Show/Hide Line Numbers

```
functions = []
for i in range(5):
    def func(x):
        return x + i
    functions.append(func)

for f in functions:
    print(f(12))
```

Without running this code, take a few moments to predict what is going to happen when it is run. Which of the following do you think is going to happen?

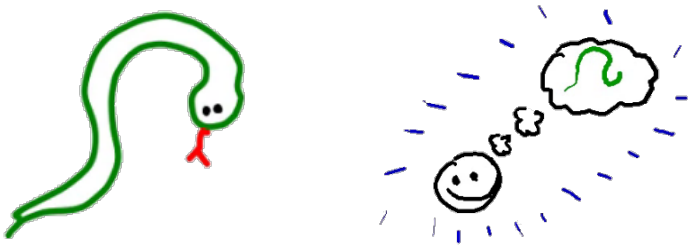
(you will not be graded on correctness for this question, just go ahead and mark your best guess)

- ○ It prints `12`, then `13`, then ..., then `16`
- ○ It prints `13`, then `14`, then ..., then `17`
- ○ It prints `16`, then `15`, then ..., then `12`
- ○ It prints `17`, then `16`, then ..., then `13`
- ○ A Python error occurs
- ◉ Something else

Now, only once you have made an educated guess above, type this code into your favorite text editor or IDE and run it. Does the result match your expectation?

If you're anything like the vast majority of 6.101 students in the past, the actual result of running this code will be surprising! And even if this example wasn't surprising, there will be times in the future where unexpected things happen, where code you've written does not behave in the way you expected. It's those moments where it's most helpful to have a mental model of what's going on "behind the scenes," which we can then use to explain the surprising behavior.

A major goal of 6.101 is helping each of you develop a coherent, consistent model that describes Python's behavior, which can then be used to help you make better plans, turn those plans into code, and debug that code more effectively. Our model will (necessarily) be a simplification. This idea is summed up in the following (highly scientific) diagram:



On the left, we have an actual Python, representing the real Python interpreter running in your machine. It's got a lot of things going on, in far more detail than we could realistically hope to keep in our heads at one time. On the right, however, is a depiction of our goal for a mental model of Python: it doesn't contain all of the details of the real Python interpreter, but it's still able to accurately predict (and explain) the behavior of the real interpreter.

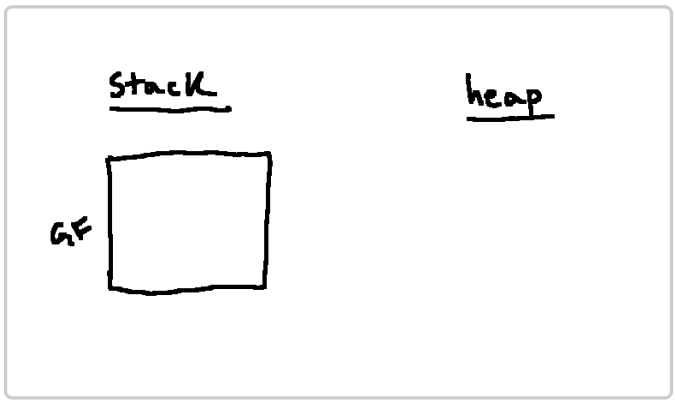
Our main tool for this purpose in 6.101 is called an *environment diagram* (which some people refer to as a

memory diagram), a pictorial way to keep track of what is going on inside your computer as we run our code, and specifically as a way to keep track of variable bindings and how they evolve over time. In this reading, we'll introduce some of the basic rules for environment diagrams, largely by way of small examples; and we'll expand on these ideas in future readings.

3) Environment Diagrams

Whenever Python needs to work with an object, that object is stored in memory; and, additionally, Python also needs a way to associate names with the objects it has stored in memory. And so there will be two important categories of things for us to keep track of in our diagrams: we'll need to keep track of the objects that are in play, and we'll also need to keep track of the names that we can use to refer to those objects. In our diagrams, we'll keep track of these in two distinct regions, which correspond to logically separate regions in memory: the **heap** (where objects are stored) and the **stack** (where we keep track of names). So an empty environment diagram will typically start by separating a drawing into those two regions.

The stack is organized into structures called **frames**, which we use to store mappings from names to objects. We'll represent each frame with a box in our diagram, and we will always *start* execution with a single frame called the **global frame**, which we'll usually label as GF as a kind of shorthand. So whenever we're setting up a new environment diagram, things will start out looking something like this:



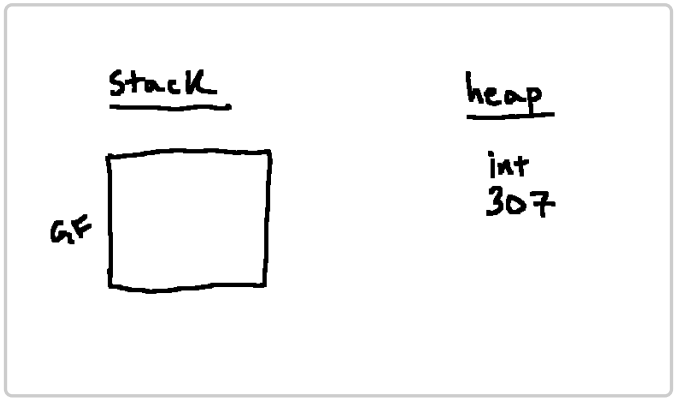
There isn't much going on the diagram yet, but let's go ahead and fix that. We'll start by looking at a very small example and then build from there. Let's start with the following program:

Show/Hide Line Numbers

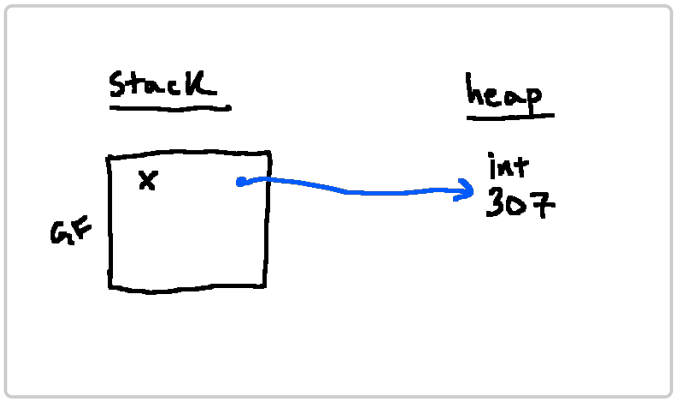
```
1 x = 307
2 x = 308
3 y = x
4 y = 342
```

Python is going to work its way through this one line at a time, and the first piece we encounter is the statement `x = 307`. This kind of statement, with the equals sign `=`, is called an *assignment statement*, and Python executes this statement by first evaluating the expression on the right-hand side of the equals sign and then associating that result with the name on the left.

In this case, the expression on the right-hand side of the equals sign is `307`; and, evaluating that, Python creates a new object to represent `307`. So we'll need to update our diagram to account for this. In this class, we'll generally denote an object on the heap with both a type and a value; so after evaluating `307` (but before associating it with a name), our diagram has evolved and looks like this, showing our new object on the heap:



Next, we associate this object with the name `x` in the frame in which we're currently running (in this case, the global frame). In our diagrams, we'll show this relationship with an arrow pointing from the name of a variable to the object referenced by that variable, so in this case our diagram becomes:

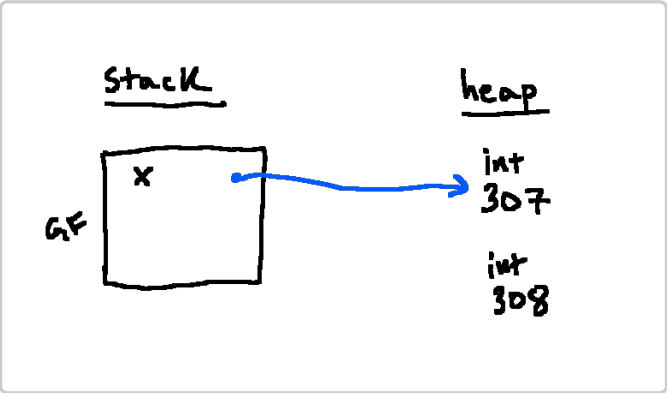


and this is how things look after we're done executing the assignment statement on line 1. Just to clarify some terminology here: in this case, we'll refer to `x` as a **variable**, which is **bound** to the object on the heap representing 307; we'll also refer to the arrow itself (representing the linkage between the variable `x` and the value to which it is bound) as a **reference**.

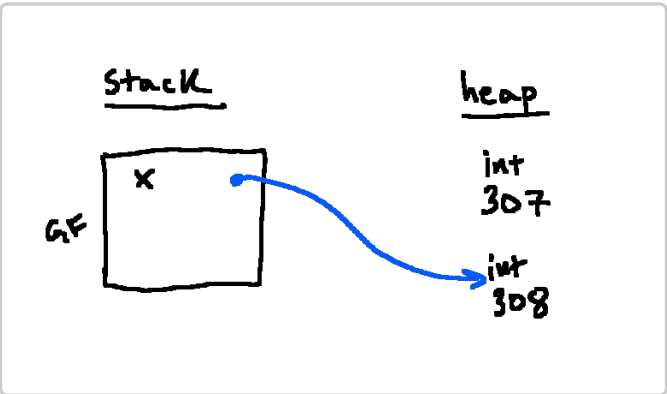
In order to evaluate the variable `x`, we follow the arrow to find whatever object it points to. For example, running `print(x)` would cause Python to evaluate `x`, finding the `int` object representing 307 (importantly, we find *exactly* that object, not a copy or anything like that) and display a representation of it to the screen.

It's worth mentioning that what we've done with the diagram above is not an arbitrary choice; it's a close approximation of what happens inside of your machine when we run that same code in the actual Python interpreter. Python does store an object representing 307 someplace in memory, and it separately keeps track of where that object is in memory (and that when we look up `x`, that's the spot where we will find the associated value).

Now let's take a look at what happens when we run the following line (line 2, `x = 308`). Here we see another assignment statement, so we proceed in exactly the same way. We start by evaluating the expression on the right-hand side of the equals sign, which gives us a new integer object on the heap representing 308:



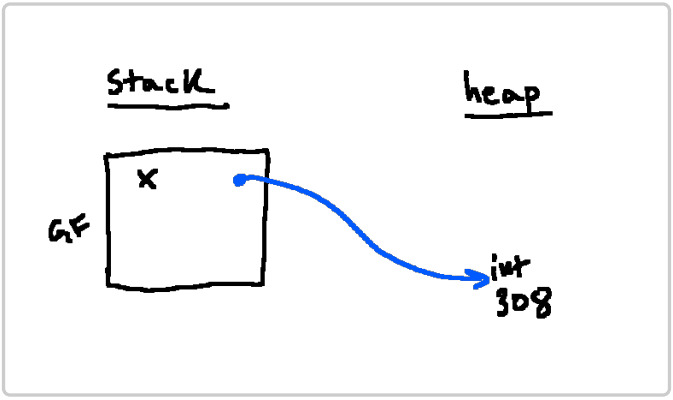
And then we bind the name `x` to that value `308`. But wait, `x` already has an arrow pointing from it to someplace else, so what happens? Well, each variable can only reference one location in memory, so Python proceeds by removing the existing binding and instead binding `x` to this new object. In our diagram, we'll represent that by erasing the existing arrow that we had drawn from `x` to `307` and drawing a new arrow from `x` to the new `308` object we created, resulting in a diagram like the following, which represents the result after we've finished running line 2:



We should also say something about that object representing 307, which is now left all alone on the heap with nothing pointing to it. No arrows pointing to that object means that we no longer have any way to access it from our program! And everyone's computer has a finite amount of memory, and so it generally doesn't make sense to waste valuable space in memory on objects that we can no longer use. Through a process called "garbage collection," Python will free up that memory for later use by effectively deleting objects that can no longer be reached from our program.

The actual internals of Python's garbage-collection process are quite complex[2], but in our diagrams, we'll approximate that by replicating a core aspect of Python's garbage collection: reference counting. Python does this by keeping track of the number of references to each object on the heap (in terms of our diagrams, the numbers of arrows pointing to each object); if and when an object reaches 0 references, Python removes it and frees up the associated memory for later use. In our diagrams, we will mimic this by erasing any object on the heap that has no arrows pointing to it after each line is done running.

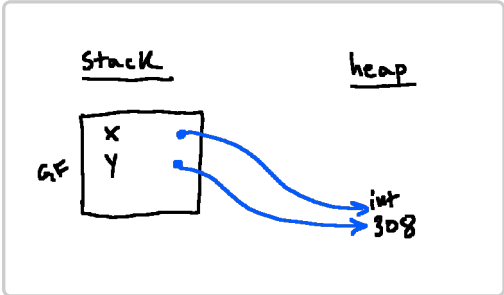
In this case, we've followed the rules to finish up line 2, and that object `307` is left with no arrows pointing to it. So we'll "garbage collect" it by erasing it from our diagrams (and just like your computer can use that region of memory for other purposes in the future, erasing that object from our diagram gives us more room to draw other objects later on if we need to!). After our `307` has been garbage collected, our diagram looks like this:

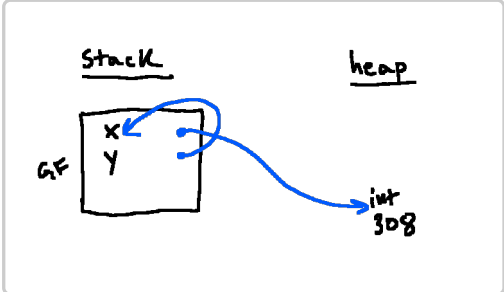


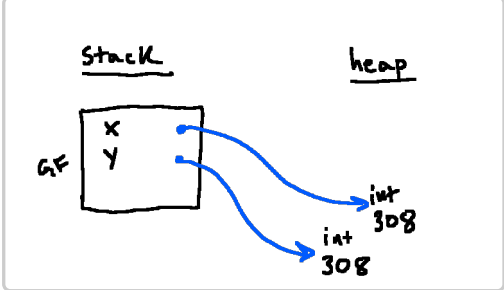
Note that at this point, evaluating `x` would cause us to follow the arrow to find the `308` object on the heap.

3.1) Check Yourself

At this point, our diagram accurately reflects the state of our program's execution after lines 1 and 2 have been evaluated. Now it's your turn to give the next step a shot! Starting from the diagram above, how will things change when we run `y = x`? Try following the rules we've laid out so far to draw what the diagram will look like after executing line 3. Which of the following diagrams best represents the state of the program after running line 3?







The program runs to completion, but the diagram doesn't match any of the options above.

The program does not run to completion but rather raises an exception.

Answer the question above and then view its answer to see the prompt for this question. Enter your answer below:

Returning from the hypothetical scenario represented by the question we just finished, let's finish up evaluating the program we started with:

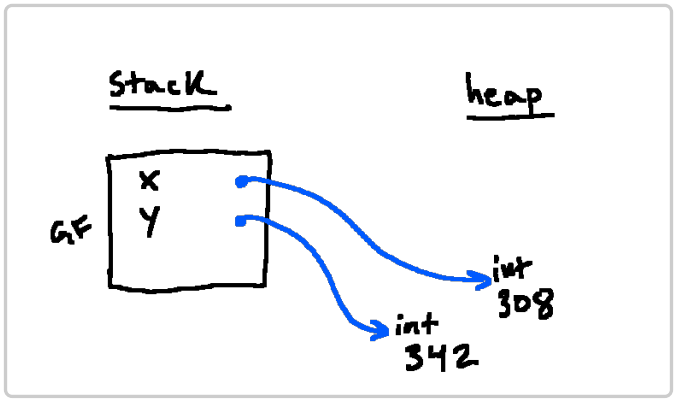
Show/Hide Line Numbers

```
1 x = 307
2 x = 308
3 y = x
4 y = 342
```

and at this point, we know what the diagram looks like after executing the first three lines (it matches the

correct answer to the multiple-choice question above). Take a moment now to finish drawing the diagram before looking at the answer below (click "Show/Hide"):

Show/Hide

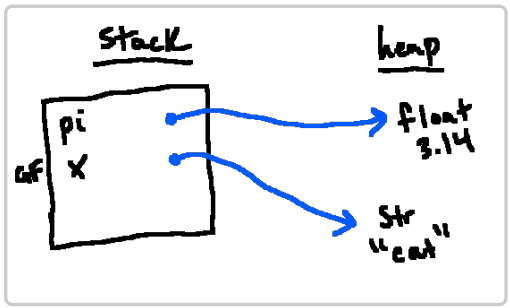


Importantly, setting `y = 342` only affected `y`'s binding, *not* `x`'s!

4) Lists

Of course, we're not just limited to integers in our programs; we can use all kinds of other types as well, and we'll need ways to draw those things in our diagrams as well. For many kinds of things, we will just draw them in a similar way to how we drew our `int`s in the example above, i.e. by writing both the type and value of the object. So, for example, the following code and diagram match:

```
pi = 3.14
x = "cat"
```

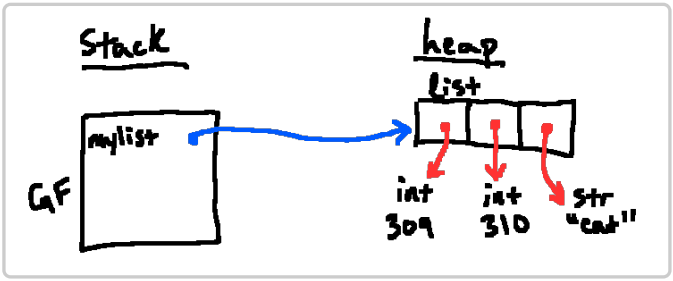


But other types of things require a little bit more care to represent faithfully in our diagrams. One such category of objects is *collections*, i.e., objects that contain other objects, like `lists`, `sets`, `tuples`, etc., as we'll want to be careful to represent those objects in a way that captures some of the detail (and subtlety) of the way those objects are constructed, so that we can use our diagrams to explain the behavior of these kinds of objects in our programs.

Today's readings are going to focus almost exclusively on lists, but we'll introduce lots of other types of objects in future readings.

As a small example, let's consider the following code and the corresponding diagram:

```
mylist = [309, 310, "cat"]
```



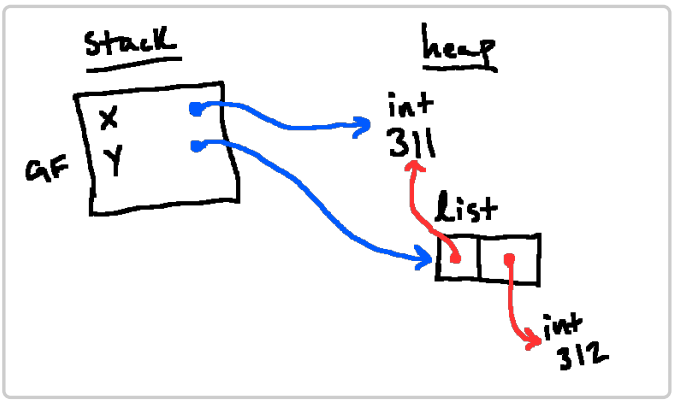
One thing to note is that Python's handling of assignment statements is always the same, regardless of the form of the expression on the right-hand side of the equals sign and regardless of the type of object that we get when evaluating that expression. So here, as before, we start by evaluating the expression on the right-hand side of the equals sign, and whatever object results from this is what we will bind to the name `mylist`.

We will draw a list in our diagrams as a box labeled `list`, with a little internal box for each element in the list. Importantly, when Python makes a list like this, the list's contents (items) are explicitly stored in memory as *references* to other Python objects, not the objects themselves. To ensure that we are accurately describing the way actual Python lists work, the lists we draw in our diagrams reflect that relationship. Here we've drawn the references in the list in a different color from those in the global frame, but that is purely a visual distinction to make the diagram easier for us to look at and interpret; the red and blue arrows mean exactly the same thing.

So in this case, Python sees a literal representation of a list containing three elements. So we get a new list object, and the references inside of that list point to the results of evaluating each of the subexpressions (`309`, `310`, and `"cat"`, respectively). In this case, since each of those is a literal expression, evaluating each one gives us a brand-new object on the heap.

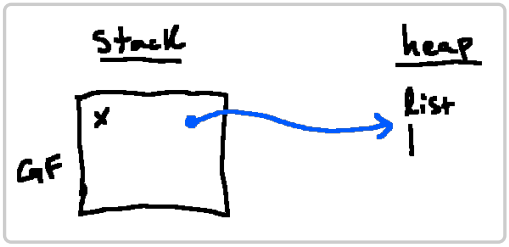
If one or more elements inside the square brackets had not been literals (for example, if we had referred to a variable in one of those spaces), we would follow our normal evaluation rules to figure out where the references in the list should point. For example, in the following piece of code, we end up with both the variable `x` and the first spot (index `0`) in the list pointing to the same object:

```
x = 311
y = [x, 312]
```



We will also want a way to draw *empty* lists in our programs. Since each box in our list representation contains an element, it does not make sense to draw an empty list as a single box. Instead, we'll draw it as a single vertical line (representing the left-hand side of a box), for example:

```
x = []
```



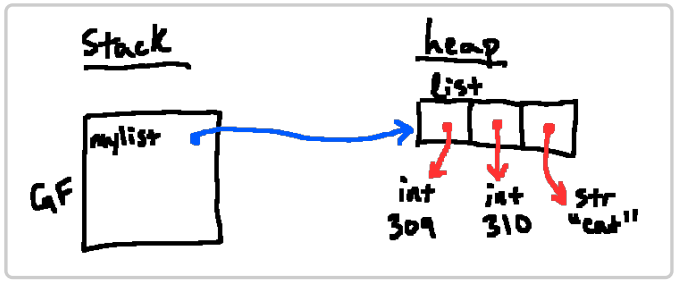
4.1) Operations on Lists

Now that we have a way to represent lists, let's talk about some common operations on lists and see how they work in our diagrams. We obviously won't be able to cover everything here, but we'll try to hit some of the highlights and talk about how these operations work in terms of our environment diagrams, as well as describing some common kinds of error messages we might see when working with lists. If you want to know all of the details, they are documented in the official Python documentation entry for `list` objects, though it takes some clicking to see all of the supported behaviors.

4.1.1) Indexing and Item Assignment

One of the most common operations we'll perform on lists is looking into the list to get one of the objects contained therein. This operation is referred to by several different names: we might say we are "indexing into" the list, or "getting an item" from the list, or "subscripting" the list. All of these refer to the same operation, which we denote with square brackets. As an example, let's start with one of our earlier examples:

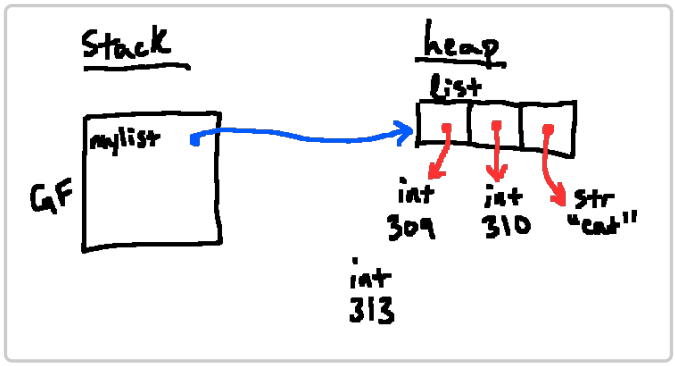
```
mylist = [309, 310, "cat"]
```



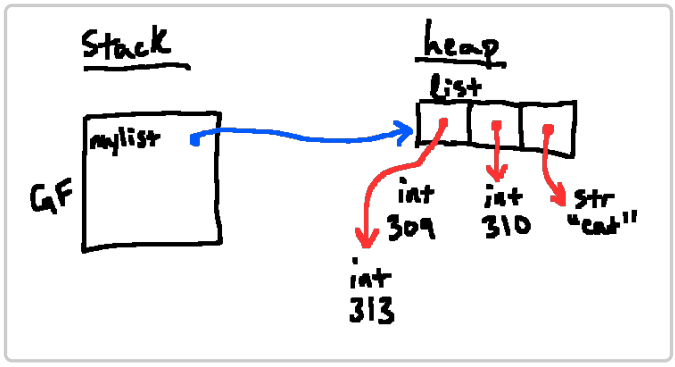
When we index into the list using `mylist[0]`, for example, Python starts by evaluating the thing to the left of the square brackets to figure out what object we're going to look in. In this case, we evaluate the `mylist` by following the reference from the global frame to the list object in memory. Then the part in the square bracket tells us to look at index `0`, so we follow the first reference in the list and find the object representing the integer `309` (and as before, since we followed a reference to get here, we don't make a copy or anything like that; we find exactly this `309` object).

If we were being really pedantic about drawing every step of the process, it's worth noting that Python would also have made a new object to represent the `0` in that expression and used it to index into the list. But since the `0` would be immediately garbage collected (there are no arrows pointing to it), we will often omit things like that in the diagrams.

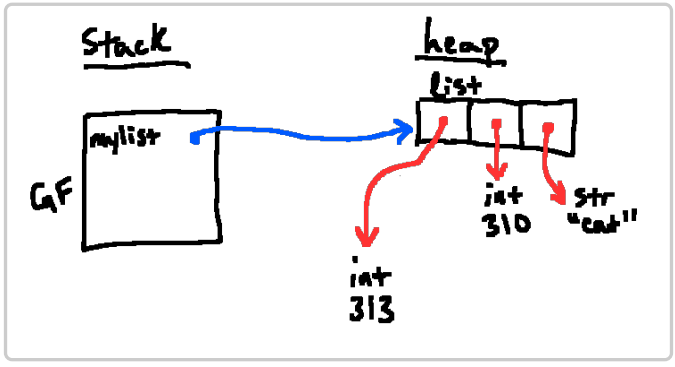
Lists are also *mutable*, in the sense that we are able to modify them (by changing what references are stored in them). One way that we can mutate a list is through an operation called "item assignment," using familiar syntax with square brackets. For example, let's look at how our diagram would evolve if we say `mylist[0] = 313`, starting from the diagram above. This looks like an assignment statement, and so we'll begin the same way we have done in the other examples of assignment statements above: we'll evaluate the expression on the right side of the equals sign, which makes us a new integer representing the value 313:



Then Python will change the list such that the first reference contained therein (the one at index `0`) now points to that new object:



And since the `309` object now has no references left pointing to it, it will be garbage collected, resulting in the diagram below:



Note that our variable `mylist` still refers to the exact same list it started out pointing to (i.e., to the same spot in memory), but that list has been mutated such that it now contains a different set of objects.

Common Error Messages

Lots of things can go wrong when indexing into a list or assigning a new value to an item in a list. Some of these will just result in bizarre behaviors, but some will result in exceptions. Let's just take a quick look at some common kinds of error messages that can arise from these operations (and note that information can be gleaned both from the type of exception we see (`IndexError` versus `TypeError` versus ...) and from the details of the message):

- **`IndexError: list index out of range`**

  This error message occurs when we try to access an invalid index in a list. For example, if we had `x = [9, 8, 7]`, the valid indices are `0`, `1`, and `2` (or, we can index from the other end with `-1`, `-2`, and `-3`). And so we would see this error message if we tried to do, for example, `x[8]` or `x[-4]` or any other integer value that isn't a valid index.

- **`TypeError: list indices must be integers or slices, not ...`**

  This error occurs when we try to use a value as an index that is not an integer. For example, if we had `x = [9, 8, 7]` and we tried to do `x[6.101]`, we would see `TypeError: list indices must be integers or slices, not float`.

- **`TypeError: '...' object is not subscriptable`**

  This error occurs when we try to use square brackets to index into (to "subscript") an object that doesn't support it. For example, if we tried to do `8.2[7]`, we would see `TypeError: 'float' object is not subscriptable`. This kind of error rarely comes about by directly writing an expression like that, though (it's kind of silly), but rather when indexing into an object referred to by a variable name. For example, if we did `x[7]` instead, we would still get this kind of error if the variable `x` referred to a float.

Keep an eye out for those kinds of error messages in your own programs. Once you know what they mean, the error messages can be a helpful tool to help us find, understand, and fix the errors that caused them!

Practice Problems

Consider the following piece of code:

Show/Hide Line Numbers

```
x = ["baz", 302, 303, 304]
x_copy = x

x[0] = 342

print(x)
print(x_copy)
```

What will be printed to the screen when we run this program? Try to use what we've seen so far to draw an environment diagram and use it to predict the behavior of the program, using Python only as a fallback; then enter your answer in the box below.

In the example above, how many total Python `list` objects were created while that program was running?

In the example we just saw, `x` and `x_copy` are *aliases* for the same list (they are two names for the same object in memory; calling it `x_copy` was just wishful thinking...). While aliasing has its uses, *unintentional* aliasing is a very common cause of bugs. When we accidentally have two separate names for the same object but think we have two separate objects, it can be surprising when changes we make in one place show up in the other. The more we understand about how these things work behind the scenes, though, the better able we'll be to recognize and fix these kinds of issues when they occur and to plan so as to avoid them in the first place!

Now consider the following:

Show/Hide Line Numbers

```
a = [301, 302, 303]
b = [a, a, a]

b[0][0] = 304

print(a)
print(b)
```

What will be printed to the screen when we run this program? Try to use what we've seen so far to draw an environment diagram and use it to predict the behavior of the program, using Python only as a fallback; then enter your answer in the box below.

In the example above, how many total Python `list` objects were created while that program was running?

4.1.2) List Slicing and Copying

We can also find *sublists* of a given list using a cool idea called *slicing*. The typical slicing syntax looks like `x[start:stop]`, where `x` is a list, and `start` and `stop` are integers representing indices into `x`. This operation will create a new list that contains a subset of the references from `x`, starting at the index given by `start` (inclusive) and ending at the index given by `stop` (exclusive). So, for example, if we had `x = [9, 8, 7, 6, 5, 4, 3, 2]` and computed `x[2:5]`, the result would be `[7, 6, 5]`. Note that this is a *new list* but that it references the same items that `x` references.

Leaving off one or the other of the given indices provides a shorthand way to grab a sublist from the beginning or end of the list. `x[:N]` gives us a new list containing the first `N` items from `x`, and `x[N:]`

gives us a new list containing the items from index `N`, `N+1`, `N+2`, ..., all the way to the end of `x`. We can also use negative indices here, so `x[-N:]` is a way to get the *last* `N` elements in `x`.

Leaving off both arguments, like `x[:]`, gives us a copy of `x`. Importantly, though, it is what we call a *shallow* copy of `x`, in that it is a new list containing all of the same references that `x` contained.

We can also provide a *third* piece when using this syntax, like `x[start:stop:step]`. In this form, the third value represents the number of values we should skip in between each element we include in our new list. So `x[0:7:3]`, for example, will go from index `0` (inclusive) to index `7` (exclusive), but counting by 3's instead of taking every element.

This gives us yet another cute shorthand, `x[::-1]`, which makes a new list with all of the references from `x` but in the reverse order.

Practice Problems

After running `x = [9, 8, 7, 6, 5, 4, 3, 2]`, what is the value of `x[1:4]`? Enter your answer as a Python list in the box below:

After running `x = [9, 8, 7, 6, 5, 4, 3, 2]`, what is the value of `x[:3]`? Enter your answer as a Python list in the box below:

After running `x = [9, 8, 7, 6, 5, 4, 3, 2]`, what is the value of `x[:-3]`? Enter your answer as a Python list in the box below:

After running `x = [9, 8, 7, 6, 5, 4, 3, 2]`, what is the value of `x[4:]`? Enter your answer as a Python list in the box below:

After running `x = [9, 8, 7, 6, 5, 4, 3, 2]`, what is the value of `x[-4:]`? Enter your answer as a Python list in the box below:

Another Slicing Example

Consider the following piece of code:

Show/Hide Line Numbers

```
x = [[9, 8], [7, 6, 5]]
y1 = x
y2 = x[:]

x[0] = 20
x[1][0] = 30

print(x)
print(y1)
print(y2)
```

Before trying to run any code or draw any diagrams, try to predict what will be printed to the screen when we run this code. Now type this program into Python and run it. Do the results match your expectation?

Regardless of whether you correctly predicted the output or not, let's spend some time to try to make sure that we understand what happened, since there is a lot going on in that little program!

We'll talk through this example step-by-step in detail below. We encourage you to follow along on your own, trying to draw the diagram on your own for each step before clicking through the interactive diagram below to see our work.

Show/Hide Line Numbers

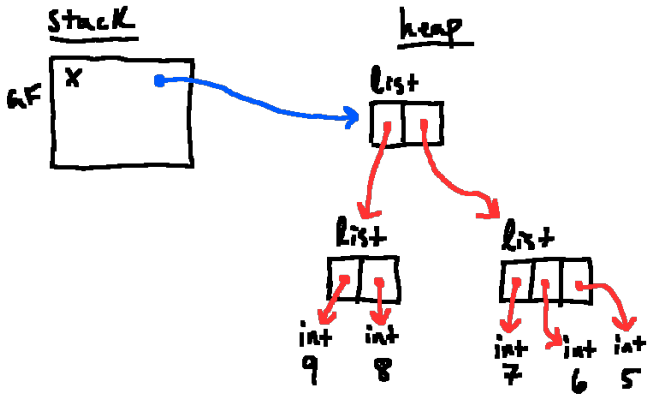
```
x = [[9, 8], [7, 6,
5]]
y1 = x
y2 = x[:]

x[0] = 20
x[1][0] = 30

print(x)
print(y1)
print(y2)
```

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

STEP 1

Before we start running this program, we start in the usual place, setting aside regions of the diagram for the stack and the heap, and starting with a single frame (the global frame, **GF**).

The next step to be executed is the assignment statement on line 1. What should the diagram look like after that statement has been executed? Try drawing it out yourself before moving on to the next step.

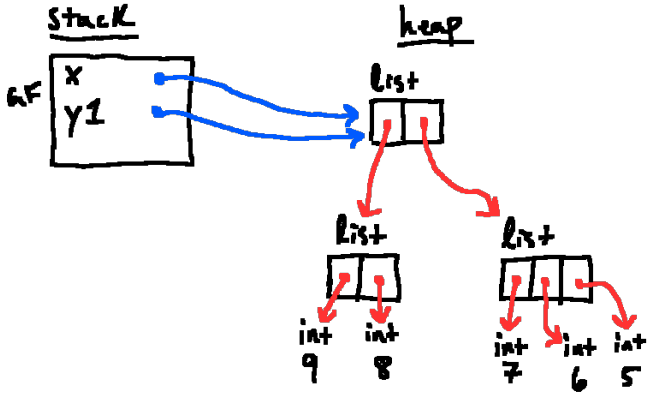


STEP 2

There were actually several steps involved in that one line of code! And after we're done, we end up with `x` pointing to a somewhat complex structure in memory. During that process, we created a total of 5 new `int` objects and 3 `list` objects.

Double-check that your diagram matches the above before we move on.

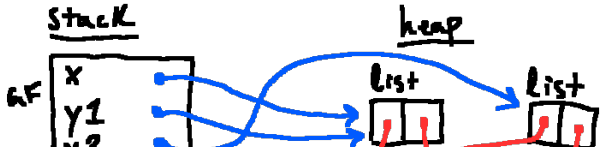
The next statement we're going to run is on line 2: `y1 = x`. Try making the relevant updates to your diagram before moving on to the next step.



STEP 3

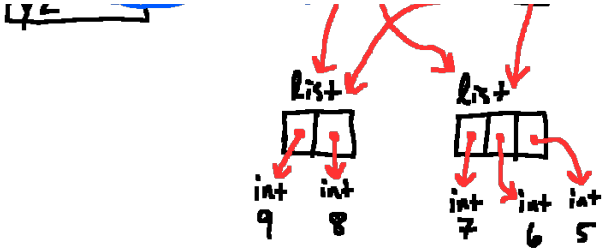
When evaluating the `x` on the right-hand side of the equals sign, we follow the arrow pointing from `x`, and we find the top-most `list` object we drew on the heap, so `y1` is going to refer to that same object.

The next step is going to be to run line 3: `y2 = x[:]`. This line looks very similar to line 2, but it's different in an important way, the usage of the slicing operator. Remember that this will make a new list object containing all the same references as `x`. Try to update your diagram before moving on to the next step.



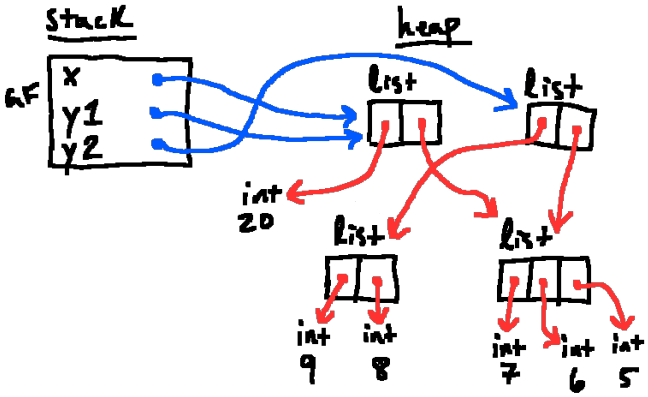
STEP 4

Here is how things should look at this point.



We have our new list object called `y2`, and it contains the same two references as the list object called `x` (which is also called `y1`).

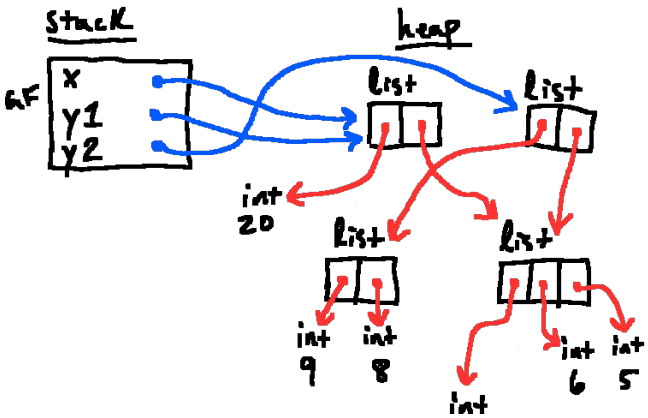
Next, we're going to run line 5. Try to update your diagram based on this before moving on. Which arrows (references) in the diagram are going to change? Which of our variables (`x`, `y1`, and/or `y2`) will be affected?



STEP 5

Notice that the reference we changed was the one at index `0` in the list called `x`. This only affects that one object and not any of the other objects. But despite the fact that only one *object* was changed, both `x` and `y1` will 'notice' that change, in the sense that if were were to `print(x)` and `print(y1)`, we would see that `20` in both of the outputs.

Now that that's done, we're ready to execute line 6. Again, try to update your diagram before moving on. Which reference(s) will change, and which of our variables will be affected?



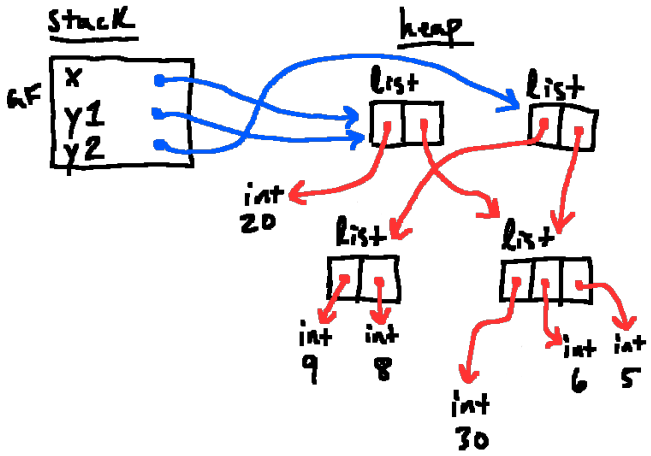
STEP 6

This command caused us to change index `0` of the bottom-right-most list object in the diagram (which had previously pointed to an integer `7` that we also garbage-collected as part of the

30

last step).

One question remains: which `print` statements will have a `30` in them now as a result of this change? Try to think through that before moving on to the next step.



STEP 7

The result of the three `print` statements is:

```
[20, [30, 6, 5]]
[20, [30, 6, 5]]
[[9, 8], [30, 6, 5]]
```

Importantly, since the list we mutated in the last step is an element in both top-level lists, *all* of the print statements notice this change!

The main takeaway here is that if you notice similar kinds of behaviors in your own programs ("I only changed that in `x`! Why is it showing up in `y`?"), be on the lookout for accidental aliasing of this form!

4.1.3) Adding Elements to Lists

Python also gives us ways to mutate lists by adding additional items to them. Here, we'll introduce three built-in list methods that add items to lists:

- `append`

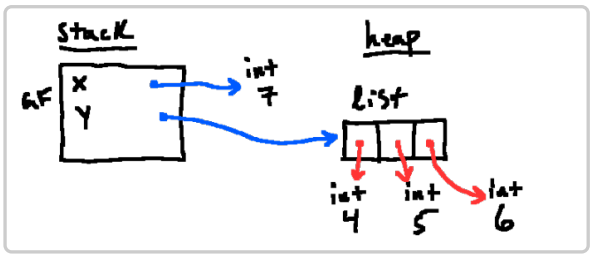
  The `append` method allows us to add an item to the end of a list. If we have a list called `x` that has m elements in it and we call `x.append(v)`, `x` will be mutated such that its length is now $m + 1$; it still contains all of the items it contained before, but it now has one more item on the end: whatever object we specified when calling `append`.

  For example, let's look at the following code:

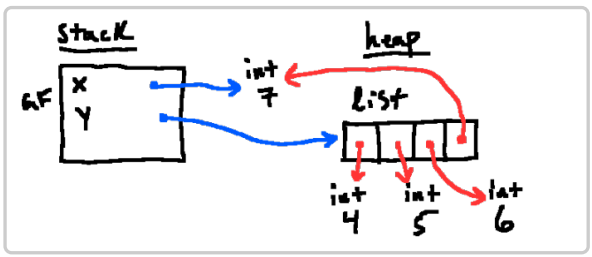
```
x = 7
```

```
y = [4, 5, 6]
y.append(x)
```

After the first two lines (before calling `append`), we have something like the following:



And after running `y.append(x)`, we see that we have mutated `y` so that it contains one additional reference, pointing to the object that was passed in when calling `append`.



One thing to be careful of when calling `append` is that it works by mutating the list we invoke it on, but it always returns `None`. Consider the following small example:

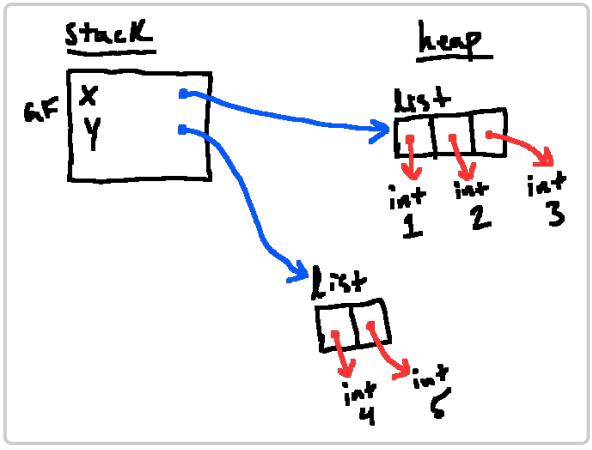
```
x = [4, 5, 6]
y = x.append(7)
```

In this example, `x` will be mutated so that it has a `7` on the end, but `y` will be `None`.

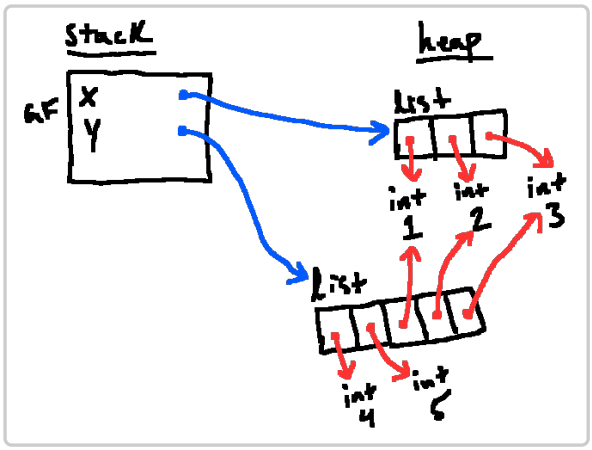
- `extend`

  The `extend` method gives us a way to add multiple elements to the end of a list with a single method call. `x.extend(items)` takes an *iterable object* (a list or some other structure that can be looped over), and it appends each element to `x`.

  For example, consider two lists created with `x = [1, 2, 3]` and `y = [4, 5]`, respectively:



Calling `y.extend(x)` will add each reference from `x` to `y`, in order, resulting in the following diagram:



- `insert`

  Both of the methods described above add elements to the *end* of the list in question. But when we want to add elements elsewhere in the list (at the beginning or in the middle), they won't help us. But another method will: `insert`!

  `x.insert(N, val)` takes an index `N` in addition to the value we want to add to the list. It adds a new reference to the list, such that `val` will now be at index `N`, and everything that was previously at `N` or later in the list is now one index higher.

Practice Problems

Consider a list defined by `x = [7, 7, 7, 7, 7]`. After running `x.insert(N, 8)` (for some non-negative value of `N`), printing x shows the following value: `[7, 7, 8, 7, 7, 7]`

What value of `N` was used?

Consider a list defined by `x = [1, 2, 3]`. We perform some operation `x._____([4, 5])`, and then, when printing `x`, we see `[1, 2, 3, 4, 5]`. What operation did we perform, `append` or `extend`?

--

From the question above, what value would `x` have had if we had used the other of these two methods instead? Enter a Python list in the box below:

4.1.4) Removing Elements from Lists

Python also provides a couple of ways to *remove* items from lists. Among other ways, the `pop` and `remove` methods allow for mutating a list to remove elements.

- `pop`

  The `pop` method allows us to remove an item from a list based on its index. `x.pop(N)` takes an index `N` as input. It removes the associated element from the list and returns it. Everything that was at index `N+1` or later in the list will then be one index lower.

  `x.pop()`, with no argument specified, is equivalent to `x.pop(len(x)-1)`, i.e., it removes the last element in the list and returns it.

- `remove`

  The `remove` method allows us to remove an item from a list based on its *value* rather than an index. `x.remove(v)` takes an arbitrary object as its input. If that object exists as an item in `x`, it will be removed. If `v` occurs more than once in `x`, only the earliest occurrence will be removed. If `v` does not exist as an item in `x`, an exception will be raised.

Common Error Messages

- **`ValueError: list.remove(x): x not in list`**

This error occurs when we try to use `remove` to remove an element from a list, but that element is not present in the list.

- **`IndexError: pop index out of range`**

  This error is similar to the one we saw with regular list indexing earlier in this reading. It occurs when the index given to `pop` is not a valid index into the relevant list.

- **`IndexError: pop from empty list`**

  As you might expect, this error happens when we try to call `pop` with an empty list.

Practice Problems

Consider a list defined by `x = [6, 2, 8, 0, 7, 3, 9, 1, 4, 5]`. After running `x.pop(N)` (for some non-negative value of `N`), printing x shows the following value: `[6, 2, 8, 0, 7, 3, 1, 4, 5]`

What value of `N` was used?

Consider a list defined by `x = [6, 2, 8, 0, 7, 3, 9, 1, 4, 5]`. After running `x.remove(N)` (for some non-negative value of `N`), printing x shows the following value: `[6, 2, 8, 0, 3, 9, 1, 4, 5]`

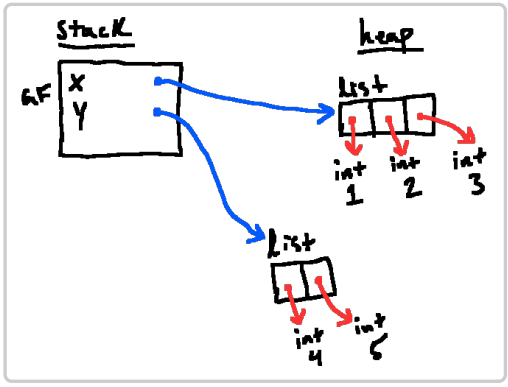
What value of `N` was used?

Consider a list defined by `x = [1, 2, 3, 4, 5, 4, 3, 2, 1]`. After running `x.remove(3)`, what is the new value of `x`? Enter your answer as a Python list in the box below:

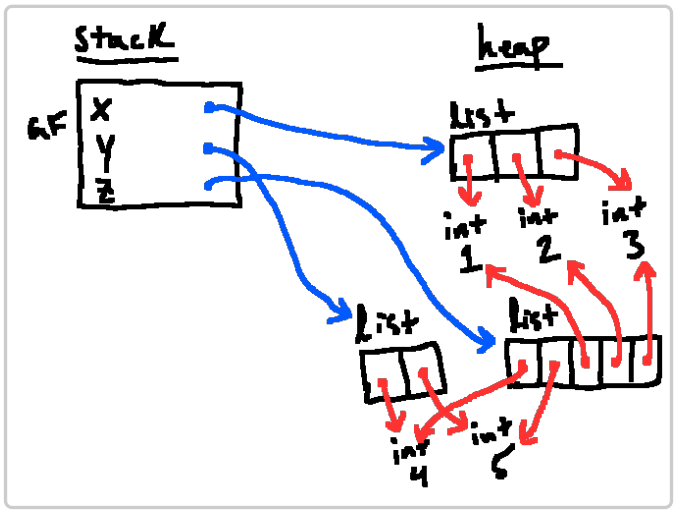
4.1.5) Concatenation

Python also supports using the `+` operator on lists, to implement an operation called *concatenation*. Concatenating two lists using `+` makes a *new* list containing all of the references from the left-side operand, followed by all of the references from the right-side operand. For example, consider the following:

```
x = [1, 2, 3]
y = [4, 5]
```



Running `z = y + x` starting from this point would produce a new list containing 5 elements (the 2 elements from `y`, followed by the three elements from `x`).



The end result here is similar to what we would have gotten from using `extend`, except that instead of modifying `y` in-place and mutating it to add the new elements, we instead made a brand-new list.

Note also that, unlike the `+` operator for numbers (addition), the `+` operator for lists (concatenation) is not commutative; while `y + x` above gave us `[4, 5, 1, 2, 3]`, `x + y` would instead give us `[1, 2, 3, 4, 5]`.

Python does support the `+=` operator on lists, but it behaves differently than you might expect: it mutates the left-side list in-place. That is, `a += b` is *exactly* equivalent to `a.extend(b)`; it mutates `a` rather than making a new list.

Common Error Messages

- **`TypeError: can only concatenate list (not "...") to list`**

  This error occurs when we try to use `+` when the left-side operand is a list but the right-side operand is not. For example, if we tried `[1, 2, 3] + 8`, we would see `TypeError: can only concatenate list (not "int") to list`.

  Note, though, that the exact error message we see depends on the order of the operands. The error

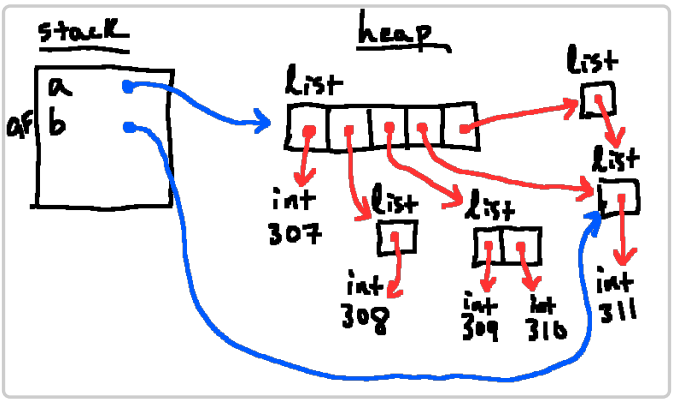
message shown above is what happens if we try `[1, 2, 3] + 8`; if we were to try `8 + [1, 2, 3]` instead, we would see a different error message: `TypeError: unsupported operand type(s) for +: 'int' and 'list'`.

- **`TypeError: '...' object is not iterable`**

  This error occurs when we try to use the `+=` when the left-side operand is a list but the right-side operand is not. For example, if we tried `x = [1, 2, 3]` followed by `x += 8`, we would see `TypeError: 'int' object is not iterable`.

5) More Practice: Matching Environment Diagrams

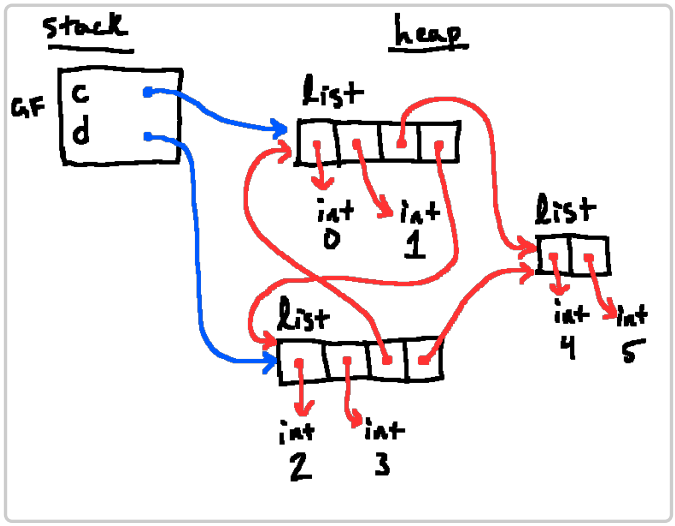
Consider the following environment diagram:



In the box below, write Python code that, when run, would result in a diagram matching this one. You may use as many lines as you need, and you may create extra variables or objects, but when the code is done running, the structures of `a` and `b` should match those shown in the diagram above.

```
1 pass  # Your code here
```

Consider the following environment diagram:



In the box below, write Python code that, when run, would result in a diagram matching this one. You may use as many lines as you need, and you may create extra variables or objects, but when the code is done running, the structures of `c` and `d` should match those shown in the diagram above.

```
1 pass  # Your code here
```

6) Tuples

We're nearing the end of this reading, but it's worth spending just a little bit of time to introduce a data type closely related to lists: tuples. Like lists, tuples are ordered collections of arbitrary Python objects. Like lists, the items in tuples are stored as references to other objects on the heap. The main place where they differ is that tuples are *immutable*; that is to say, once we have built a tuple, we cannot change any of the references therein, nor can we add or remove items.

Syntactically, a tuple is represented by a comma-separated list of expressions that is not surrounded by square brackets `[]` or squiggly brackets `{}`. You will often see tuples surrounded by round brackets `()`, but that turns out not to be necessary. For example, both of the following lines of Python create tuples:

```
x = (1, 2, 3)
y = 7, 8, 9
```

Parentheses on their own do not make a tuple. They are just used for grouping.

In particular, `(7)` is still an integer, not a tuple. If you do want to make a tuple with a single element in it,

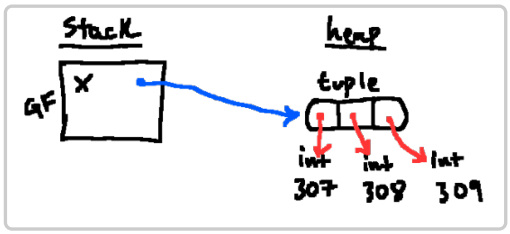
you can follow that element with a comma, for example `(7,)` or `7,`. Trailing commas are ignored, not only in tuples but in lists and dictionaries too.

Parentheses are necessary for grouping when the tuple is being created as an element of another list or tuple, like `[1, (2,3), 4]`, or passed as an argument to a function, like `print((1,2,3))`.

One place where parentheses *do* represent a tuple is an *empty* tuple, which you can create with `()`. You can also create an empty tuple with its constructor, `tuple()`.

Graphically, we'll depict tuples similarly to how we depict lists, with two main differences: obviously, we'll write `tuple` instead of `list` above the object; and we'll also usually round the edges of the tuple object in our drawings to mimic the way they often show up in code (surrounded by round brackets). For example, consider the following piece of code and the associated diagram:

```
x = (307, 308, 309)
```



6.1) Check Yourself

Consider the following variable definitions:

```
a = ()
b = 6.101
c = 6.101,
d = 6,1,0,1
e = 6,1,0,1,
f = (6.101)
g = (6.101,)
h = (6, 1, 0, 1)
i = (6, 1, 0, 1,)
```

Which of these are tuples? Enter your answer below as a sequence of characters representing the variable names corresponding to tuples (e.g., `xyz`), or enter "none" if none of the expressions evaluate to tuples. Make your best guess before using Python, but feel free to use Python to help out once you have a guess.

Consider the two tuples created with the code below:

```
x = ("baz", [1, 2], (3, 4), 5)
y = (7, 8)
```

Which of the following statements would be able to run without raising an exception? Check the boxes associated with the statements that will run successfully, and leave those that would raise exceptions unchecked.

- ☐ `x[1] = 20`
- ☐ `x[1][0] = 20`
- ☐ `x[2] = 20`
- ☐ `x[2][0] = 20`
- ☐ `x.append(y)`
- ☐ `x.extend(y)`
- ☐ `y = x`

7) When To Use Environment Diagrams

While we spent some time focusing on the specifics of Python lists, our real main focus in this reading has been on starting to develop a mental model of Python, using environment diagrams as a technique for keeping track of what's going on "behind the scenes" as our programs run. But it's reasonable to be asking at this point, just to what extent we're expecting you to use them in 6.101.

We believe that environment diagrams have tremendous explanatory power (there is a reason you'll often see us draw these diagrams during recitations to explain some unexpected result, after all!) and that having an understanding of the computer's internals on this level is critical to being an effective programmer, coder, and debugger. That said, drawing the diagrams can be a little bit slow and tedious.

Our ultimate goal is that drawing the diagrams by hand will eventually be completely unnecessary; we expect this to happen eventually, not because the diagrams are not useful, but because your brain is subconsciously thinking through these steps on its own. However, it takes time to internalize the way these things work, and drawing things out carefully in the meantime will help you internalize things more quickly. And having a specific, consistent way of drawing these things means that we all have a shared language that we can use when discussing ideas about the *internals* of Python (beyond just syntax). So we will draw

a lot of these diagrams over the course of the term, and we will ask you to draw and interpret diagrams as well (as you work through the readings, during recitations, and on exams).

Outside of those contexts, there will be times when drawing these diagrams makes sense and times when it doesn't. When we get near the end of the semester and are writing programs with thousands of lines and tens of functions, for example, we probably don't want to draw a diagram for the *whole* program because it would take a long time and be very complex. But we think that it can still be useful to draw diagrams for small programs of your own or, perhaps even more usefully, for small portions of bigger programs.

8) Summary

We've covered a lot of ground in this reading! Firstly, we introduced the idea of an *environment diagram* as a graphical model for keeping track of how things evolve behind the scenes as we run a program. Then we saw a couple of examples of fitting different kinds of Python objects into our model, with a focus on lists. We also introduced several common operations on lists and demonstrated how they fit into our environment model, as well as some common error messages that can arise when performing those operations.

In next week's reading, we'll expand this model to include one of the most powerful tools we have in our programming toolkit: functions. We'll explain the rules by which function objects operate behind the scenes, and we'll explore some of the neat higher-level behaviors that arise from those rules, in addition to exploring the use of functions as a powerful organizational tool in our programs.

For now, we're going to need to leave you on the edges of your seats about the first example from today's reading (why did it print `16` five times instead of doing something else?), but by the end of the next reading, we'll have all of the tools necessary both to understand the behavior of that program and also to write a version that does something more like what the author of that code probably intended!

Footnotes

[1] here is one example paper

[2] you can read about them here if you're so inclined