

```
B id(3)
Out[4]: 140736101680104
id(4)
Out[5]: 140736101680136
id('32933923')
Out[6]: 2317423028720
id('32933923')
Out[7]: 2317423031024
id('32933923')
Out[8]: 2317423116464
id('a')
Out[9]: 140736101744608
```

in my trials, the memory address of intergers seem to be constnant, while those of strings seem to be consntant changing. do u believe I'm submitting a real scenario? if so, Can u explain this to a oython beginner and generalize it to all types?
Show thinking

<thinking> Let me analyze what the user is showing me and determine if this is a real scenario in Python.

The user is showing the output of the `id()` function in Python when applied to various objects:

1. `id(3)` returns 140736101680104
2. `id(4)` returns 140736101680136
3. Multiple calls to `id('32933923')` return different addresses: 2317423028720, 2317423031024, 2317423116464
4. `id('a')` returns 140736101744608

In Python, the `id()` function returns the memory address (identity) of an object. This behavior reflects Python's object model and memory management.

For small integers (typically -5 to 256 in CPython), Python pre-allocates these objects during initialization and reuses them throughout the program's execution. This is an optimization called "integer interning" - these small integers are cached and reused rather than being created anew each time they're needed. That's why the memory addresses for 3 and 4 appear constant.

For strings, Python's behavior is more complex:

1. Some strings may be interned (especially short strings, identifiers, etc.)
2. But string literals that appear in different places in code typically create new string objects
3. When the user calls `id('32933923')` multiple times in a session, each call is creating a new string literal object, which gets a new memory address

This is indeed a real scenario that accurately reflects Python's memory management. The user's observation that integers have consistent memory addresses while strings (particularly longer

or more complex ones) get different addresses is correct and aligns with Python's implementation.

So yes, I believe the user is submitting a real scenario, and I'll explain this behavior and generalize it to other types.

</thinking>

Yes, you're observing a real Python behavior! This is related to how Python manages memory and implements object interning.

What you're seeing is:

- The integers (3, 4) have consistent memory addresses
- The string '32933923' gets a new memory address each time
- This reflects important details about Python's object model

What's happening here

Python uses object interning as an optimization technique. For certain immutable objects that are frequently used, Python pre-allocates and reuses them instead of creating new objects each time.

Small integers

- Python pre-allocates integers in the range [-5, 256]
- When you use these small integers, Python references the same object in memory
- That's why `id(3)` always returns the same value in a Python session

Strings

- String interning is more selective
- Python doesn't automatically intern all strings
- String literals in code may be interned, but:
 - When you call `id('32933923')` repeatedly in the interactive shell, each call creates a new string object
 - That's why you see different memory addresses each time

Generalizing to other types

This behavior varies by type:

Interned (consistent id):

- Small integers (-5 to 256)
- Some short strings and identifiers
- Boolean values (True, False)
- None

Not interned (changing id):

- Large integers (outside the -5 to 256 range)
- Most strings
- Lists, dictionaries, sets
- Custom objects

This behavior is an implementation detail of CPython (the standard Python interpreter) and helps with memory efficiency. Understanding this can help you write more efficient code and avoid subtle bugs related to object identity vs equality.

Does this explanation help? Would you like me to clarify anything specific about how Python manages memory for different types?

```
B id('a')
Out[9]: 140736101744608
id('a')
Out[10]: 140736101744608
id('a')
Out[11]: 140736101744608 its constant. voila!
```

You're absolutely right! I missed that important detail in my explanation.

You've discovered another aspect of Python's string interning behavior: single-character strings like 'a' are indeed interned! That's why `id('a')` returns the same memory address each time.

Python's string interning is more selective than I initially described:

1. Single-character strings are reliably interned
2. String literals that appear in source code are often interned
3. Strings that look like identifiers (alphanumeric with underscores) are more likely to be interned
4. Longer strings or strings with special characters typically aren't interned

So Python's interning is actually quite nuanced:

- Small integers (-5 to 256): Always interned
- Single character strings: Always interned
- Identifier-like strings: Often interned
- Longer/complex strings: Usually not interned

This behavior helps Python optimize memory usage and performance for commonly used values. Good catch on the consistent behavior of `id('a')`!

B You are a computer science professor hosting your weekly office hours. I'm reading the intro to memory model part of my intro to CS textbook that you authored. I'm confused about this paragraph: "Since we did not write the code for the class that defines the `str` type, we know nothing about what data members it uses to store its contents. So we just write the value `'bonjour'` inside the box. This is a perfectly fine abstraction." Can you explain what this means to a 15-year-old beginner in your class who so happens to be your favorite student? For context, the textbook just elaborated on the meanings of and differences between variables, objects, IDs and memory addresses.

Professor leans forward with a warm smile

Ah, that's a great question! Let me break this down for you.

When you create a string like `'bonjour'` in Python, there's a lot happening behind the scenes that we don't need to worry about.

Think of it like this: imagine you have a smart speaker that plays music. You don't need to understand all the electronics inside it to use it - you just say "play music" and it works! That's abstraction - hiding complexity so we can focus on using something.

In our textbook, when we draw memory diagrams, we represent strings as simple boxes with the characters inside, like this:



```
graph LR; A["'bonjour'"]
```

But in reality, the `str` class in Python is doing a lot of complicated things to store those characters. It might be tracking:

- The individual characters
- The length of the string
- Special encoding information
- Methods for operations like searching or replacing

We don't draw all those details because:

1. We didn't write the `str` class ourselves
2. It would make our diagrams too complex
3. We don't need those details to understand how variables reference strings

This simplified representation is an "abstraction" - it hides unnecessary complexity while showing what's important for our current level of understanding.

Does that help explain the passage? It's basically saying we can use a simplified model of strings in memory without worrying about their internal structure.

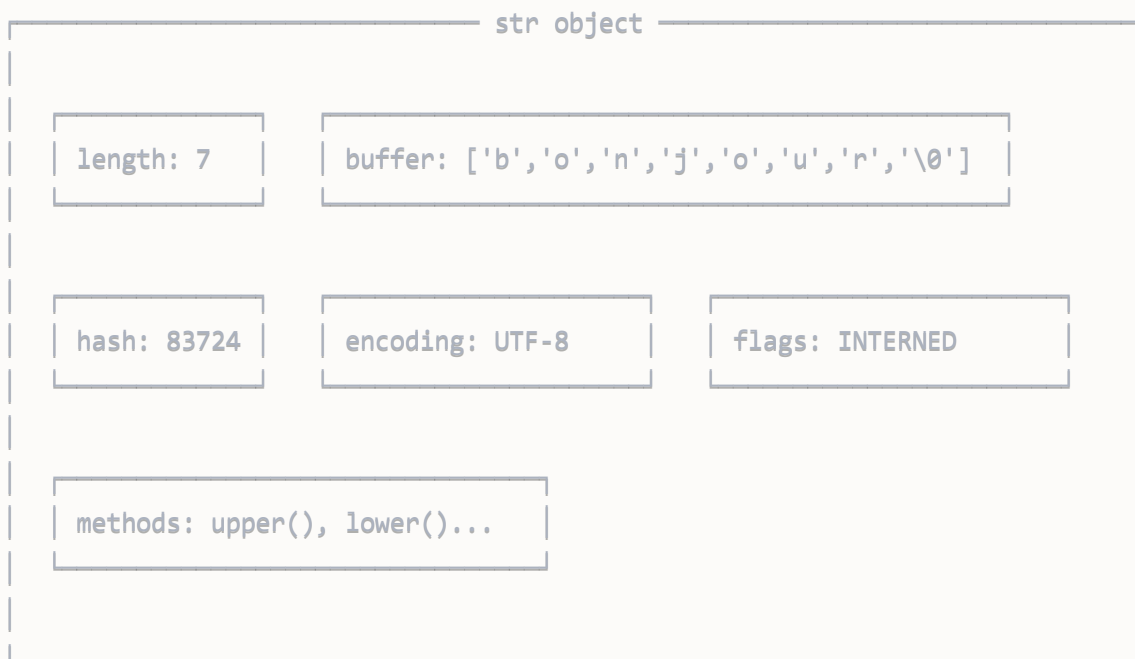
B yes, thanks a lot sir! that's really helpful! Can u eleaborate on what "data members" are in this paragraph, and what this object of "bonjour" really should look like without the abstraction?

Professor's eyes light up at the follow-up question

Great question! I'm happy to dig deeper into this.

"Data members" refers to the internal variables that a class uses to store its information. Think of them as the "behind the scenes" variables that make an object work.

Without our simplified abstraction, the string `'bonjour'` is actually a complex object with multiple data members. Here's what it might really look like in memory:



The actual string implementation includes:

1. **Length field** - Keeps track of how many characters are in the string
2. **Character buffer** - The actual array of characters with a null terminator
3. **Hash value** - For quick dictionary lookups
4. **Encoding information** - Whether it's UTF-8, ASCII, etc.
5. **Flag bits** - Special indicators (like if the string is interned)
6. **Method pointers** - References to all the string methods like `.upper()`, `.find()`, etc.

The implementation details vary by Python version and are optimized for performance and memory usage. CPython (the standard Python implementation) uses a sophisticated structure with many optimizations.

But here's the beauty of abstraction: you don't need to understand all these details to use strings effectively! Just like you don't need to understand how a car engine works to drive a car. We abstract away these complex details so you can focus on solving problems.

When you're ready to learn more about how things work under the hood, we'll cover those implementation details in our advanced classes!

B Thanks a lot sir! My next confusion is about the following section: "**Aliasing**

When two variables refer to the same object, we say that the variables are **aliases** of each other.[3]

Consider the following Python code:

```
>>> x = [1, 2, 3]
>>> y = [1, 2, 3]
>>> z = x
```

`x` and `z` are aliases, as they both reference the same object. As a result, they have the same id. You should think of the assignment statement `z = x` as saying “make `z` refer to the object that `x` refers to.” After doing so, they have the same id.

```
>>> id(x)
4401298824
>>> id(z)
4401298824
```

In contrast, `x` and `y` are not aliases. They each refer to a list object with `[1, 2, 3]` as its value, but they are two different list objects, stored separately in your computer’s memory. This is again reflected in their different ids.

```
>>> id(x)
4401298824
>>> id(y)
4404546056
```

I wonder if we input `x=2, z=x`, would they still refer to the same objects? given that 2 is immutable? also, for the different objects `x` and `y` refer to, do the intergers inside the lists use the same object for that particular interger? Or do we have different interger objects with identical values for different identical lists? FInally, can u give me a group of code I could paste in the console, run and see for myself? Thanks!

Professor smiles, impressed by the thoughtful questions

Excellent questions! You're really thinking deeply about how Python's memory model works. Let's explore these concepts.

Integers and Aliasing

For your first question about `x=2, z=x` - yes, they would still refer to the same object! Even though integers are immutable, the variables would be aliases of each other. Let's see why:

```
>>> x = 2
>>> z = x
>>> id(x)
140736101680072 # Your actual ID will vary
>>> id(z)
140736101680072 # Same as x's ID
```

This happens because the assignment `z = x` makes `z` reference the same object that `x` references, regardless of whether that object is mutable or immutable.

Integers inside Lists

For your second question about integers inside different lists - this is fascinating! Since small integers are interned in Python, the individual integers inside different lists will indeed reference the same integer objects if they have the same value.

Let's create some code you can run to see all of these concepts in action:

```
python

# Part 1: Aliasing with integers
x = 2
z = x
print("Part 1: Aliasing with integers")
print(f"id(x): {id(x)}")
print(f"id(z): {id(z)}")
print(f"x is z: {x is z}") # 'is' checks if they reference the same object
print()

# Part 2: Aliasing with Lists
a = [1, 2, 3]
b = [1, 2, 3] # Different List with same values
c = a        # Alias to the first List
print("Part 2: Aliasing with lists")
print(f"id(a): {id(a)}")
print(f"id(b): {id(b)}")
print(f"id(c): {id(c)}")
print(f"a is b: {a is b}") # Different objects
print(f"a is c: {a is c}") # Same object
print()

# Part 3: Elements inside Lists
print("Part 3: Elements inside lists")
print(f"id(a[0]): {id(a[0])}") # The '1' in first List
print(f"id(b[0]): {id(b[0])}") # The '1' in second List
print(f"a[0] is b[0]: {a[0] is b[0]}") # Same object (interned)

# Part 4: Modifying Lists to see aliasing effects
print("\nPart 4: Modifying lists")
```

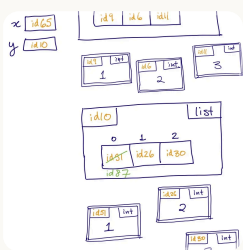
```
print(f"Before: a = {a}, b = {b}, c = {c}")
a[0] = 99
print(f"After changing a[0]: a = {a}, b = {b}, c = {c}")
# Notice c changes when a changes, but b stays the same
```

When you run this code, you'll see:

1. `x` and `z` are aliases (same ID)
2. `a` and `c` are aliases, but `b` is a different list
3. The integers inside `a` and `b` have the same IDs (they're interned)
4. When you modify `a`, `c` also changes because they're aliases

This is a key insight: while the lists themselves are different objects, the small integers they contain are the same objects in memory thanks to Python's interning mechanism!

Feel free to modify this code and experiment further. This kind of hands-on exploration is the best way to deepen your understanding!



B thank u. that is fascinating. However, realize that in the textbook, the professor who drew the illustration seems to believe even small integers require separate objects for their presence in different list objects : https://www.teach.cs.toronto.edu/~csc148h/notes/python-recap/memory_model_part1.html do u think u could elaborate on why?

Professor examines the diagram carefully

Ah, I see what's happening here! You've spotted something really interesting in the textbook illustration. Let me explain what's going on.

You're right - the diagram shows different ID values for the same integers in different lists. For example, the integer 1 appears with id9 in the first list and id51 in the second list, even though we'd expect small integers to be interned.

This is an important teaching moment! There are a few possible explanations:

1. **Teaching simplification:** The most likely explanation is that the diagram was simplified for teaching purposes. When introducing memory models, it's sometimes clearer to show each value as a separate object to help students understand the general concept of references without the additional complexity of interning.
2. **Implementation detail:** While Python does intern small integers (-5 to 256), this is an implementation detail of CPython that could potentially change. The author might be

deliberately showing a more general model that doesn't rely on this optimization.

3. **Visual clarity:** Drawing all integers with the same value pointing to the same object would require more complex arrows crossing throughout the diagram, which might make it harder to read.
4. **Context-specific behavior:** In some rare contexts (like multithreaded environments or certain implementations), Python's interning behavior might not be as straightforward as we typically describe it.

What's really important here is understanding the fundamental concepts:

- Variables refer to objects
- Lists contain references to other objects, not the objects themselves
- Multiple variables can refer to the same object (aliasing)

The implementation detail of whether small integers are interned is secondary to these key concepts. In practice, if you run the code I gave you earlier, you'll see that Python does indeed intern small integers, just as we discussed.

Great observation! This kind of critical thinking - noticing discrepancies between the model and reality - shows you're developing a deeper understanding of how Python actually works.

B Thank you for your encouragement! My next questions refers to the following section from ur textbook :**"Lists**

Mutating and non-mutating list methods

Lots of list methods, such as `count` and `index` return a value and do not mutate the list they are called on.

```
>>> lst = [9, 0, 5, 8, 9, 1]
>>> id(lst)
140623854885120
>>> lst.count(9)
2
>>> lst.index(8)
3
>>> id(lst) # lst is still referencing the same list object.
140623854885120
>>> lst
[9, 0, 5, 8, 9, 1] # And the contents of that object are unchanged.
```

Calls to these methods are easy to trace in the memory model, since nothing is mutated. Other list methods, such as `insert` do mutate the list they are called on.

```
>>> lst.insert(3, 999)
>>> id(lst)
140623854885120 # lst still references that same list object.
>>> lst
[9, 0, 5, 999, 8, 9, 1] # But the contents of it have been changed.
```

To trace calls to mutating methods using the memory model, we simply have to change what's inside the list object." I am particularly confused about the last sentence "To trace calls to mutating methods using the memory model, we simply have to change what's inside the list object." I 'm not sure what it means and how it connects to the context at all. isnt tracing about knowing and keeping track of whats going on? how is that "change what's inside the list object.", which likelt refers to the method itself?

Professor nods thoughtfully

That's an excellent question about that final sentence! I can see why it's confusing, as it's not phrased as clearly as it could be.

Let me clarify what's meant by "To trace calls to mutating methods using the memory model, we simply have to change what's inside the list object."

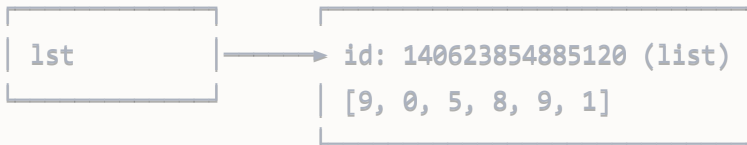
When we "trace" code in computer science, we're essentially simulating what happens during execution, step by step. When we trace code using a memory model (like the box diagrams we've been discussing), we have to update our diagram as the code runs.

What the textbook is trying to say is:

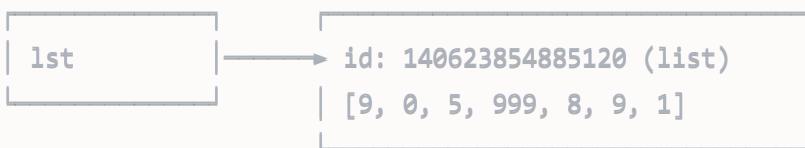
1. When tracing non-mutating methods (like `.count()` or `.index()`), our memory model diagram doesn't change at all - the list object stays exactly the same.
2. But when tracing mutating methods (like `.insert()` or `.append()`), we need to update our memory model diagram by modifying the contents inside the list object's box - while keeping the list object's ID the same.

Here's what it would look like when tracing:

Before `lst.insert(3, 999)`:



After `lst.insert(3, 999)`:



Notice how:

- The variable `lst` still points to the same object (same ID)
- Only the contents inside the list have changed

So "changing what's inside the list object" refers to updating our memory model diagram to reflect the changes to the list's contents - it's about how we document the mutation, not about the method itself.

Does that help clarify what the textbook meant by that sentence?

B I'm looking at the following problem u offered: "`lst1 = [1, [2, 3], 4]` >>> `lst2 = [5, [6, 7, 7]]` >>> `lst1.extend(lst2)` >>> `lst1 [1, [2, 3], 4, 5, [6, 7, 7]]` >>> `lst2.append(10)` >>> `lst2 [5, [6, 7, 7], 10]` >>> `lst1` >>> `lst2[0] = 15` >>> `lst2 [15, [6, 7, 7], 10]` >>> `lst1` >>> `lst2[1].append(100)` >>> `lst2 [15, [6, 7, 7, 100], 10]` >>> `lst1`" I think when we input `lst2[1].append(100)`, we mutated the list object that `lst2[1]` refers to. It so happens to be the the same list object that `lst1[4]` refers to, for when we input `lst1.extend(lst2)`, we appended the same pointer from `lst 2` to `lst 1` (along with other items in `lst1`). Am i Correct? am I missing any nuances? Would love ur insights

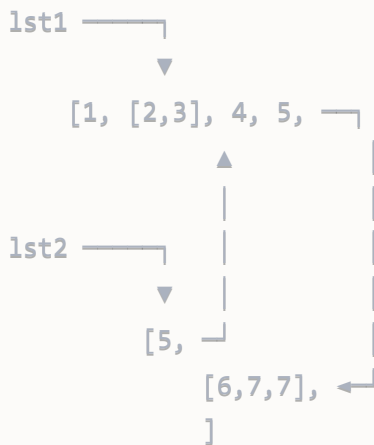
Professor's eyes light up with enthusiasm

You've got it exactly right! Your analysis is spot-on and shows a deep understanding of how Python's memory model works with nested lists.

Let me confirm your understanding and highlight the key insight:

When you call `lst1.extend(lst2)`, Python doesn't make copies of the objects in `lst2` - it just adds references to those same objects into `lst1`. So both `lst1[4]` and `lst2[1]` end up pointing to the same inner list object `[6, 7, 7]`.

Here's the memory model after `lst1.extend(lst2)`:



That's why when you execute `lst2[1].append(100)`, you're modifying the inner list that both variables reference. The change is visible through both `lst1[4]` and `lst2[1]` because they're aliases - different references to the same object.

The only nuance worth mentioning is that this behavior only happens with mutable objects (like lists). If you had primitive values or immutable objects, modifications would create new objects rather than changing existing ones.

This is exactly the kind of aliasing relationship that makes understanding the memory model so important! Mutations to shared objects can create effects that seem mysterious if you don't realize the objects are shared.

Your understanding is excellent - you've identified precisely how the nested references work and why the modification to `lst2[1]` affects what you see in `lst1`.

B "If you had primitive values or immutable objects, modifications would create new objects rather than changing existing ones."hmm. are u sure, professor? No offense, but I'm under the impression that with `.extend()`, u always add the original ids to the end of the list directly, regardless of the type of object the id is

Professor gives a thoughtful smile

You're absolutely right to question me, and I appreciate your critical thinking! I need to clarify what I said because it was imprecise.

You're correct - when using `.extend()`, Python always adds references to the original objects to the end of the list, regardless of whether those objects are mutable or immutable. The list gets extended with references to the existing objects, not copies of them.

Let me be more precise:

1. When you do `lst1.extend(lst2)`, Python adds references to the exact same objects from `lst2` into `lst1`, whether they're mutable or immutable.
2. The distinction I was trying to make is about what happens with subsequent modification attempts:
 - With mutable objects (like lists), you can modify the object directly, and those changes are visible through all references to it
 - With immutable objects (like integers or tuples), you can't modify them - any "modification" actually creates a new object and updates the reference

Let me demonstrate with code:

```
python

# With immutable objects
a = [1, 2, 3]
b = [4, 5, 6]
a.extend(b)  # a is now [1, 2, 3, 4, 5, 6]
              # a[3] and b[0] reference the same int object (4)

# If we try to "modify" b[0]
b[0] = 99    # This doesn't modify the int object 4
              # It creates a new int object 99 and makes b[0] refer to it
              # a[3] still refers to 4
```

So you're right - `.extend()` always works with the original object references. The difference is in how subsequent modifications behave, not in how `.extend()` itself works.

Thank you for catching that! This is exactly why I love teaching - students who think critically help me be more precise and accurate.

B "For example, this code:

```
>>> junk = [[1, 2], 3, [4, 5]]
>>> result = junk[1:]
>>> result
[3, [4, 5]]
```

" my guess is that mutating `result[1]` would make changes to `junk`, while `result[0]` won't. Also, a simple way to tell whether a method mutates the mutable is to see if it returns `None`. is there anything you'd like to add or comment? did I miss any nuances? Thanks!

Professor nods approvingly

Your understanding is excellent! Let's examine both your insights:

First, regarding `result = junk[1:]`:

- You're absolutely right that mutating `result[1]` would affect `junk[2]` since both reference the same list object `[4, 5]`
- And correct that changing `result[0]` (which is just the integer `3`) won't affect `junk[1]` because:
 1. Integers are immutable, so you'd be creating a new integer object
 2. You'd only be changing what `result[0]` references, not what `junk[1]` references

Second, about methods that return `None`:

That's a very clever observation! In Python, there is indeed a convention that methods which mutate an object in-place typically return `None`. Examples include:

- `list.append()`, `list.sort()`, `list.extend()`
- `dict.update()`
- `set.add()`, `set.update()`

While methods that don't mutate typically return a new value:

- `sorted(list)` vs `list.sort()`
- `str.upper()` (strings are immutable)
- `dict.get()`

This isn't an absolute rule (there are exceptions), but it's a very good rule of thumb that will serve you well!

One small nuance I'd add: slice operations like `junk[1:]` create shallow copies of lists - meaning the outer list structure is new, but the contained objects are the same ones. This is why changes to nested mutable objects propagate across both copies.

Your understanding of Python's memory model is really strong! You're grasping both the technical details and the practical patterns - exactly what I hope for in my students.

B Yes, that clears it up. Thanks! I'm now looking at the following paragraph: "**How function calls are tracked**"

Python must keep track of the function that is currently running, and any variables defined inside of it. It stores this information in something called a **stack frame**, or just "frame" for short.

Every time we call a function, the following happens:

1. A new frame is created and placed on top of any frames that may already exist. We call this pile of frames the **call stack**.
2. Each parameter is defined inside that frame.
3. The arguments in the function call are evaluated, in order from left to right. Each is an expression, and evaluating it yields the id of an object. Each of these ids is assigned to the corresponding parameter.

Then the body of the function is executed.

In the body of the function there may be assignment statements. We know that if the variable on the left-hand-side of the assignment doesn't already exist, Python will create it. But with the awareness that there may be a stack of frames, we need a slightly more detailed rule: If the variable on the left-hand-side of the assignment doesn't already exist *in the top stack frame*, Python will create it *in that top stack frame*". I'm specifically confused about "If the variable on the left-hand-side of the assignment doesn't already exist *in the top stack frame*, Python will create it *in that top stack frame*." does it mean if we are calling a function, Python will always create the variable when it 's not in the "function" stack frame? how exactly does the "the top stack frame" relate to "the function frame", and how does it serve the purpose of the function call? can u give some examples? Thanks a lot!

Professor leans forward, drawing a diagram on a notepad

Great question about stack frames! Let me clarify this with some examples.

The "top stack frame" refers to the frame of the function that's currently executing. When we say "top," we're referring to the stack data structure, where new items are added to the top. So:

1. The "function frame" and "top stack frame" are the same thing when you're inside a function
2. When a function calls another function, the caller's frame is below, and the callee's frame is on top

Let me illustrate with an example:

python

```
def outer():
    x = 10
    print(f"In outer: x = {x}")

    def inner():
        y = 20
        print(f"In inner: y = {y}")
        # What happens if we use x here?
        print(f"In inner: x = {x}") # This works! (LEGB rule)
```

```
# What if we try to modify x?  
# x = 30 # This would create a new local x, not modify outer's x
```

```
inner()  
# y is not accessible here  
# print(y) # This would cause an error
```

```
outer()
```

Let's trace through what happens with stack frames:

1. When `outer()` is called:

- A new frame is created for `outer` and becomes the top frame
- Variable `x` is created in this frame with value 10

2. When `inner()` is called from inside `outer`:

- A new frame is created for `inner` and becomes the top frame
- `outer`'s frame is still there, just below `inner`'s frame
- Variable `y` is created in the `inner` frame with value 20
- When `print(x)` runs in `inner`, Python doesn't find `x` in the current frame, so it looks in enclosing scopes (following the LEGB rule)

The key insight about "If the variable doesn't already exist in the top stack frame, Python will create it in that top stack frame" is this:

python

```
def example():  
    a = 5          # Creates variable 'a' in example's frame  
    a = a + 1      # Modifies the existing 'a' in the current frame  
    b = 7          # Creates variable 'b' in example's frame  
  
    # But what about this?  
    global_var = 100 # This creates a NEW variable in example's frame  
                    # It does NOT modify any global variable with the same name  
                    # unless you use the 'global' keyword
```

If you wanted to modify a global variable instead of creating a new local one, you'd use:

python

```
global_var = 42 # Global variable  
  
def example():  
    global global_var # This tells Python to use the global variable  
    global_var = 100 # Now this modifies the global variable
```


The stack frame system ensures that:

1. Each function has its own private variables
2. Variables with the same name in different functions don't interfere with each other
3. Function calls can be nested without variables getting mixed up

Does this help clarify how the "top stack frame" relates to function calls and variable creation?

B yes, it does! thanks prof, always learning a lot from u. in the following section, u talked about how to check for preconditions: ""Enter PythonTA

The `python_ta` library we use in this course has a way to automatically check preconditions for all functions in a given file. Here is an example:

```
from python_ta.contracts import check_contracts # NEW

@check_contracts # NEW
def max_length(strings: list[str]) -> int:
    """Return the maximum length of a string in the given list of strings.

    Preconditions:
        - strings != []
    """
    max_so_far = -1
    for s in strings:
        if len(s) > max_so_far:
            max_so_far = len(s)

    return max_so_far
```

Notice that we've kept the function docstring the same, but removed the assertion. Instead, we are importing a new module (`python_ta.contracts`), and then using the `check_contracts` from that module as a... what? 🤔

The syntax `@check_contracts` is called a **decorator**, and is technically a form of syntax that is an *optional part of a function definition* that goes immediately above the function header. We say that the line `@check_contracts` *decorates* the function `max_length`, which means that it adds additional behaviour to the function beyond what is written the function body.

So what is this “additional behaviour” added by `check_contracts`? As you might imagine, it reads the function's type contract and the preconditions written in the function docstring, and causes the function to check these preconditions every time `max_length` is called. "" When I tried this myself, I noticed that `@check_contracts` always works no matter how many empty lines are inserted between this argument and the header. I wonder then, how would I know that `@check_contracts` is associated with this function exclusively? and at what number of increasing lines between `@check_contracts` and the function header are they no longer appreciated? Don't be stressed to provide an answer - it's fine to say ur not sure or u dont have access to the disired information. thanks!

Professor leans back, considering the question

That's a fascinating question about decorators! Let me share what I know about the syntax rules.

In Python, a decorator must appear immediately before the function definition it decorates, with no intervening code. However, you're right that you can have blank lines between the decorator and function header.

The official Python syntax is quite specific about this:

```
python

@decorator
def function():
    pass
```

Blank lines (whitespace) are generally ignored by Python's parser in this context. This means you could technically have many blank lines between the decorator and the function header, and it would still work:

```
python

@check_contracts

# This still works even with blank lines here!

def max_length(strings: list[str]) -> int:
    # ...
```

However, if you put any actual code or even a comment between the decorator and the function, it will break the association:

```
python

@check_contracts
x = 5 # This would break the decorator association
def max_length(strings: list[str]) -> int:
    # ...
```

To your specific question about "how many blank lines" - there isn't a hard limit in the Python syntax. The decorator will remain associated with the function regardless of how many blank lines you put between them. However, I would strongly advise against relying on this in practice!

Following PEP 8 (Python's style guide), the recommended style is to have the decorator immediately above the function definition with no intervening blank lines, as this makes the code much more readable and makes the association between the decorator and function explicit.

While I don't have access to the exact implementation details of the Python parser, this is a case where the language syntax is permissive but good coding practices are much more restrictive.

It's a great question that shows you're really thinking about how Python syntax works under the hood!

B Thanks a lot prof, thats very cencouraging! I have some follow up regarding contents in ur txtbook:1."Since we need to be aware of the types we are using at any point in our code, it is good practise to document this. In this course, we will document the types of all *functions* and class *instance attributes*. We'll use Python's type annotation syntax to do so." in this paragraph, is it actually talking about the paramteters associated with funtiond and classess that need to have their type annoatated? and what does "*instance attributes*" mean? for ur reference, this is the context:"**1.5 Python Type Annotations**

In many programming languages, we cannot use a variable until we have declared its type, which determines the values that can be assigned to it; furthermore, a variable's type can never change. Python takes a very different approach: only *objects* have a type, not the variables that refer to those objects; and in fact, a variable can refer to any type of object. Nonetheless, we can't use a Python variable unless we know what type of object it refers to at the moment—how would we know what we can do with it?" 2."def split_numbers(numbers: list[int]) -> tuple[list[int], list[int]]:" in this type contract, what if the function returned a tuple with 3 elements, all of them being a list of ints? would that violate the contract? does tuple[list[int], list[int]] restrict the length of treturn valuer of 2, even though a tuple could be as long as we want? Thanks!

Professor adjusts glasses and addresses the questions

Excellent follow-up questions! Let me address them both:

Question 1: Type Annotations

You're partially right. When the text mentions documenting types of "functions and class instance attributes," it's referring to:

1. Function type annotations: This includes both the parameters AND the return value, like:

python

```
def calculate_area(length: float, width: float) -> float:
    return length * width
```

2. Instance attributes: These are the variables that belong to each individual object (instance) created from a class. For example:

python

```
class Student:
    def __init__(self, name: str, age: int):
        self.name: str = name # instance attribute
        self.age: int = age    # instance attribute
```

"Instance attributes" are different from "class attributes" which are shared by all instances of a class. Each object created from a class has its own copy of instance attributes, but shares class attributes with all other instances.

The paragraph is highlighting that while Python itself is dynamically typed (variables can refer to objects of any type), we still need to know what type we're working with at any given time to

use it correctly. Type annotations help document this for humans reading the code (and for tools like type checkers).

Question 2: Tuple Type Constraints

You've asked a really insightful question! In Python's type annotation system:

```
python
```

```
def split_numbers(numbers: list[int]) -> tuple[list[int], list[int]]:
```

This annotation DOES specify that the function should return a tuple of exactly two elements, each being a list of integers. It constraints both the types AND the structure.

If the function actually returned a tuple with three lists of integers, this would violate the type contract. The correct annotation for that would be:

```
python
```

```
def triple_split(numbers: list[int]) -> tuple[list[int], list[int], list[int]]:
```

For a tuple with variable length but all the same type, you would use:

```
python
```

```
from typing import Tuple
# For any number of lists of integers
def variable_split(numbers: list[int]) -> Tuple[list[int], ...]:
```

The `tuple[list[int], list[int]]` notation is very specific about both types and structure - it must be exactly two elements of the specified types.

Great questions! These kinds of details are exactly what separate a basic understanding from a deeper mastery of Python's type system.

- B** 1. **Instance attributes:** These are the variables that belong to each individual object (instance) created from a class. For example:

```
python
```

```
class Student:
    def __init__(self, name: str, age: int):
        self.name: str = name # instance attribute
        self.age: int = age    # instance attribute
```

"Instance attributes" are different from "class attributes" which are shared by all instances of a class. Each object created from a class has its own copy of instance attributes, but shares class attributes with all other instances. Thanks, but I'm confused. I mentioned that instance attributes are individual objects created from a class, but it's obvious from the example that they are created within a function within the class. You also mentioned that "class attributes" are what's shared across the entire class, not "instance attributes". "Instance attributes" are exclusive to the function they are defined in. Finally, you said "Each object created from a class has its own copy of instance attributes, but shares class attributes with all other instances.", while specifying that attributes are variables. What does it mean for each object to have its own copy of "attributes"? and how can an object share a variable with an "instance"? I take instance to mean a function defined in the class. How can an object and an instance "share anything"? Clearly I'm missing something, and I could really use my professor's enlightenment. Thanks!

Professor smiles warmly, recognizing an important conceptual confusion

I see where the confusion is coming from! Let me clarify these concepts more carefully:

First, let's clear up what an "instance" is: An instance is not a function defined in a class - it's an individual object created from that class. When you do something like `student1 = Student("Alice", 20)`, you're creating an instance of the Student class.

Let me break down the terminology more precisely:

Class vs. Instance

- A **class** is like a blueprint (e.g., `Student`)
- An **instance** is an individual object created from that class (e.g., `student1`, `student2`)

Instance Attributes vs. Class Attributes

Instance attributes:

- Belong to each individual object created from the class
- Usually defined inside `__init__` using `self.attribute_name`
- Each instance has its own separate copy

Class attributes:

- Defined at the class level (outside any method)
- Shared by all instances of the class

Here's a clearer example:

python

```
class Student:
    # This is a class attribute - shared by all Student objects
    school_name = "Python University"

    def __init__(self, name: str, age: int):
        # These are instance attributes - each Student object has its own
        self.name = name
        self.age = age
```

Now if we create two students:

python

```
student1 = Student("Alice", 20)
student2 = Student("Bob", 22)

# Each student has their own name and age (instance attributes)
print(student1.name) # Alice
print(student2.name) # Bob

# Both share the same school_name (class attribute)
print(student1.school_name) # Python University
print(student2.school_name) # Python University

# If we change the class attribute, it changes for all instances
Student.school_name = "Code Academy"
print(student1.school_name) # Code Academy
print(student2.school_name) # Code Academy
```

The `self` parameter in methods refers to the specific instance that the method is being called on. When you see `self.name`, it's referring to the name attribute of that specific instance.

Does this help clarify the distinction between instances, classes, and their respective attributes?

B Thanks! Finally, I am a bit confused about this part: "A type or `None`"

One of the most common uses of the `|` is to say that a value could be a certain type, or `None`. For example, we might say that a function returns an integer or `None`, depending on some success or failure condition. For all type expressions `<T>`, we can write `<T> | None` to express this. Here is an example:

```
def find_pos(numbers: list[int]) -> int | None:
    """Return the first positive number in the given list.

    Return None if no numbers are positive.
    """
    for n in numbers:
        if n > 0:
            return n
```

`Any`

Sometimes we want to specify that the type of a value could be anything (e.g., if we're writing a function that takes a list of any type and returns its first element). We annotate such types using `Any`, which is imported from the `typing` module." my question is, what is the difference between `<T>` and `Any`? and why does `<T>` NOT include `None`? thanks!

Professor nods thoughtfully

That's an excellent question about type annotations in Python! The distinction between `<T>` and `Any` is subtle but important.

`<T>` in this context is not actual Python syntax - it's a placeholder notation the textbook is using to represent "some specific type." In actual code, you'd replace `<T>` with a concrete type like `int`, `str`, or `list[float]`.

When we write `<T> | None`, we're saying "either this specific type OR None" - for example, `int | None` means "either an integer or None."

Now, about `Any` vs a specific type:

1. Specific type (like `int`, `str`, etc.):

- Restricts the value to that particular type
- Type checkers will flag errors if you try to use it incorrectly
- Doesn't include `None` by default

2. `Any` (from `typing` module):

- Effectively opts out of type checking for that value
- Can be assigned any type, including `None`
- Can be used in any operation
- Doesn't provide safety guarantees

Here's a practical example showing the difference:

```
python

from typing import Any

def func1(x: int) -> int:
    return x + 1 # Type checker ensures x is an integer

def func2(x: Any) -> Any:
    return x + 1 # No type checking - x could be anything
                # Will fail at runtime if x doesn't support addition
```

`Any` doesn't mean "any type except None" - it truly means ANY type, including None. But a specific type like `int` doesn't include `None` by default, which is why we need `int | None` to explicitly allow for that possibility.

The reason we often need to explicitly include `None` is that many functions in Python return `None` in special cases (like when they don't find a match), and we want our type annotations to accurately reflect that possibility.

1.5 Python Type Annotations In many programming languages, we cannot use a variable until we have declared its type, which determines the values that can be assigned to it. Here, a variable's type can never

PASTED

B Thanks! I have finished Chapter one of the textbook. As a professor, you are designing a couple of leetcode style practice questions, that tests my mastery of the following contents. They are fully automated python programs that provides a very leetcode-analogous experience, I read all the descriptions and requirements that are clearly given, design the function at a specified place in the code, and when I click run, a full set of testing on all relevant cases(basic, edges, etc.) will be automatically implemented, and returning feedback on each test example (expected vs actual, etc.) the first couple of questions will be based on 1.5 Python Type. Thanks!

I'd be happy to design some practice questions based on Python type annotations! Here are three leetcode-style problems that will test your understanding of this topic.

Python Type Annotation Practice Problems
Code



These practice problems are designed to test your understanding of Python's type annotation system from Chapter 1.5. Each problem focuses on different aspects:

1. **Basic Type Annotations:** Working with the `Any` type for generic functions

2. **Union Types:** Using the `|` operator to indicate multiple possible types
3. **Callable Types:** Working with function parameters
4. **Optional Types:** Using the `| None` pattern (or `Optional` from typing)
5. **Complex Type Annotations:** Combining multiple types in complex data structures

To use this practice set:

1. Copy the code into a Python file
2. Implement each function where indicated by "YOUR CODE HERE"
3. Run the file to automatically test your implementations

Each problem builds on the concepts from the textbook and challenges you to apply them in practical scenarios. The automated tests will verify your solutions work correctly for various inputs, including edge cases.

Would you like me to explain any of these problems in more detail, or would you prefer another set of practice questions focusing on different aspects from the chapter?



   **Retry** 

Claude can make mistakes. Please double-check responses.