

CS 131 Discussion

# Week 6: Spooky Stackframes

matt wang (he/him)

 [matt@matthewwang.me](mailto:matt@matthewwang.me)

 [mattxw.com/131slides](https://mattxw.com/131slides)

 [mattxw.com/131feedback](https://mattxw.com/131feedback)

# Discussion Agenda

1. feedback from last week
2. warmup (light/medium/heavy)
3. selected hw answers
4. person of the week

~ break ~

5. language of the week
6. project concept: call stacks
7. project 2 extra tips
8. extra time: project 2 OH



snack of the week

# Feedback / Iteration

**Thank you for giving mid-quarter feedback!**

- good topics: hw review, project tips, concept explanations
- attitude: welcoming, enthusiastic

Summarizing some actionables:

- more project tips, especially concepts (this week: call stacks)
- more practice problems with increasing difficulty
- splitting up required/optional problems in HW (layout)
- **balancing internship experience and impostor syndrome\***
- less third-person narration in lecture notes / materials
- test case distribution (for Project 3)

# **warmup problems**

(light, medium, heavy)

# light, medium, heavy?

Different types of questions:

- light: simple, straightforward applications of concepts, no tricks
  - you should see the question and immediately think of the answer!
- medium: a couple of moving parts in one concept; includes tricky cases
  - the types of questions we'd ask you a homework or exam
- heavy: joining multiple concepts together, including from other CS fields
  - not fair game for an exam; but, **test mastery of material!**

## light: **flimsy functions**

```
def spook(*args):  
    for arg in args:  
        print(f'Hey {arg}, boo!')
```

With reference to function parameter concepts, what is `*args`?

And, how would we use this function?

## light: **flimsy functions**

```
def spook(*args):  
    for arg in args:  
        print(f'Hey {arg}, boo!')
```

With reference to function parameter concepts, what is `*args`?

**spook's variadic parameters/arguments!**

And, how would we use this function?

## light: **flimsy functions**

```
def spook(*args):  
    for arg in args:  
        print(f'Hey {arg}, boo!')
```

With reference to function parameter concepts, what is `*args`?

**spook's variadic parameters/arguments!**

And, how would we use this function?

```
spook("Youngbo", "Matt", "Zian")
```



## light: **flimsy functions**

```
def spook(*args):  
    for arg in args:  
        print(f'Hey {arg}, boo!')
```

What if we instead did...

```
spook(["Youngbo", "Matt", "Zian"])
```

## light: **flimsy functions**

```
def spook(*args):  
    for arg in args:  
        print(f'Hey {arg}, boo!')
```

What if we instead did...

```
spook(["Youngbo", "Matt", "Zian"])  
  
> Hey ['Youngbo', 'Matt', 'Zian'], boo!
```

## medium: **parameter pontificating**

```
Grade gradeQuestion(Student s, Response r, Question q){  
    var points = q.getPointsFromResponse(r);  
    q = updateQuestionStats(q, score);  
    s.score = s.score + points;  
    r.setGraded(true);  
}  
  
gradeQuestion(s1, r1, q1);
```

Which lines are affected by parameter passing techniques? How so?

## medium: **parameter pontificating**

```
Grade gradeQuestion(Student s, Response r, Question q){
```

```
    var points = q.getPointsFromResponse(r);
```

```
    q = updateQuestionStats(q, score);
```

```
    s.score = s.score + points;
```

```
    r.setGraded(true);
```

```
}
```

```
gradeQuestion(s1, r1, q1);
```

← No arg assignment here  
(probably!), so no effect!

Which lines are affected by parameter passing techniques? How so?

## medium: **parameter pontificating**

```
Grade gradeQuestion(Student s, Response r, Question q){
```

```
    var points = q.getPointsFromResponse(r);
```

```
    q = updateQuestionStats(q, score);
```


```
    s.score = s.score + points;
```

```
    r.setGraded(true);
```

```
}
```

```
gradeQuestion(s1, r1, q1);
```

**This is assignment to an object.** In pass by reference, this would change q1; otherwise, no change!



Which lines are affected by parameter passing techniques? How so?

## medium: **parameter pontificating**

```
Grade gradeQuestion(Student s, Response r, Question q){
```

```
    var points = q.getPointsFromResponse(r);
```

```
    q = updateQuestionStats(q, score);
```

```
    s.score = s.score + points;
```

```
    r.setGraded(true);
```

```
}
```

```
gradeQuestion(s1, r1, q1);
```

Which lines are affected by parameter passing techniques? How so?

It's assignment, but, we're accessing a **field of an object**: it changes s1! for both pass by ref/obj ref!

(but, it wouldn't change s1 for pass by value)

## medium: **parameter pontificating**

```
Grade gradeQuestion(Student s, Response r, Question q){
```

```
    var points = q.getPointsFromResponse(r);
```

```
    q = updateQuestionStats(q, score);
```

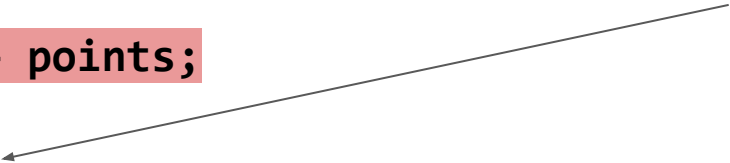
```
    s.score = s.score + points;
```

```
    r.setGraded(true);
```

```
}
```

```
gradeQuestion(s1, r1, q1);
```

This is calling a method of an object. It would change r1 for pass by ref/obj ref, but not value.



Which lines are affected by parameter passing techniques? How so?

## heavy: **refereeing races**

Python has something called a **Global Interpreter Lock (GIL)**, which means that only one CPU thread can control execution at once.

Specific to memory management, what problem does Python's GIL prevent?



## heavy: **refereeing races**

Python has something called a **Global Interpreter Lock (GIL)**, which means that only one CPU thread can control execution at once.

Specific to memory management, what problem does Python's GIL prevent?

Answer: Python uses **reference counting** for garbage collection. Incrementing and decrementing references to shared objects could introduce a race condition! The GIL prevents this problem by only ever letting one thread run at a time (i.e. assign/reassign, which change reference count).

**last week's homework!**

*(super abridged)*

# People seemed to struggle with...

For the homework, honestly ... it seems like y'all nailed everything??\*

I'll make a quick nit on 2C and move on.

**Though, you should check your own answers!**

\* (sampling bias since it's only people who chose to submit, and we gave you the answers beforehand ... but I'll stay proud)

## Question 2C

Kevin is writing some systems software for a GPS. He has to frequently allocate and deallocate arrays of lat and long coordinates. Each pair of coordinates is a fixed-size tuple, but the number of coordinates is variable (you can think of them as random).

He's trying to decide between using C# (has a mark-and-compact GC) and Go (has a mark-and-sweep GC). What advice would you give him?

```
struct Coord {  
    float lat;  
    float lng;  
};  
  
function frequentlyCalledFunc(count) {  
    Array[Coord] coords = new Array[Coord](count);  
}
```

## Question 2C

Kevin is writing some systems software for a GPS. He has to frequently allocate and deallocate arrays of lat and long coordinates. Each pair of coordinates is a fixed-size tuple, but the number of coordinates is variable (you can think of them as random).

He's trying to decide between using C# (has a mark-and-compact GC) and Go (has a mark-and-sweep GC). What advice would you give him?

### Mark-and-compact!

Observations:

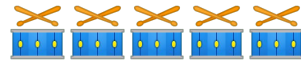
- arrays are contiguous
- frequent deallocation & reallocation

```
struct Coord {  
    float lat;  
    float lng;  
};  
  
function frequentlyCalledFunc(count) {  
    Array[Coord] coords = new Array[Coord](count);  
}
```

can't go over midterm stuff today :(

(since we don't have discussion next week, we'll do a post or something?)

# person of the week



# Carol / Kipply (200?-)

- optimized TruffleRuby implementation at Shopify ([blog post](#))
- has done a lot of AI work at startups (Cohere, Anthropic)
- one of my friends from hs :)
  - though it's been a while :'(

## Why Carol?

- she's done some *stellar* blog posts on interpreters & compilers
  - [Intro to JITs](#), [JIT comparisons](#), [History of Python](#)
- she's done some *super cool* work:
  - while super young, and
  - **without a college degree**

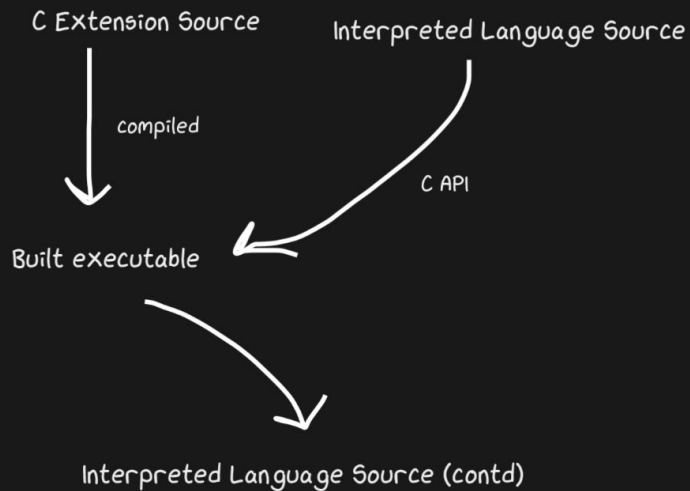
Hopefully – a little inspiring and a little reassuring.



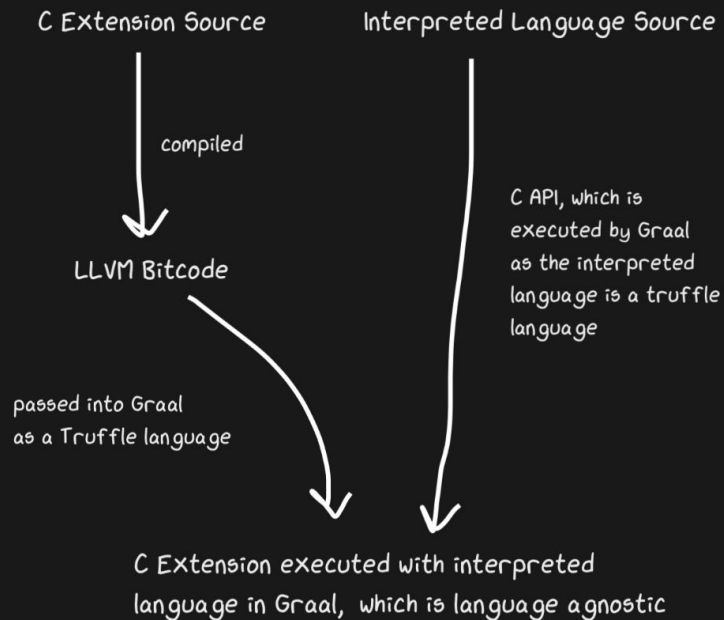
Carol and a pink shark  
(from [her website](#))

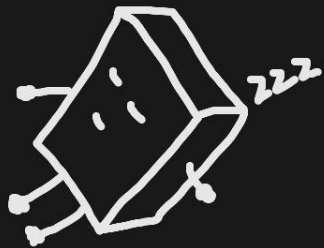


## CLASSIC C EXTENSIONS (such as CRuby)

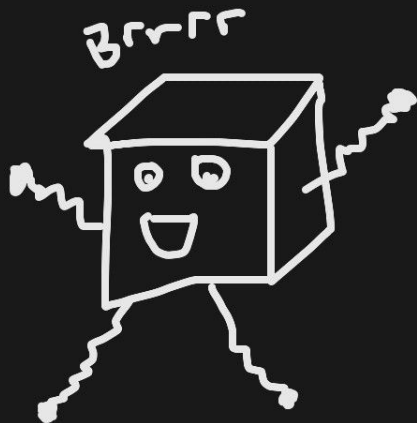


## C EXTENSIONS WITH GRAAL



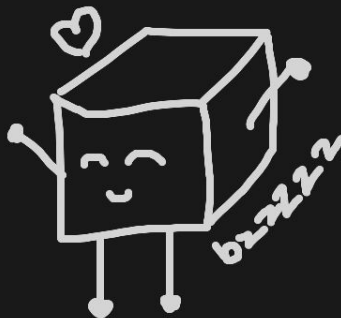


Sleepy interpreting JIT



OMG I CAN COMPILE FAST CODE  
I AWAKE NOW

cute vms



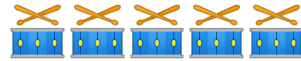
- syntax checking before runtime!
- runs on all machines!
- helps you jit with less overhead!



~ **break** ~

discussion will resume at 11:05

# language of the week



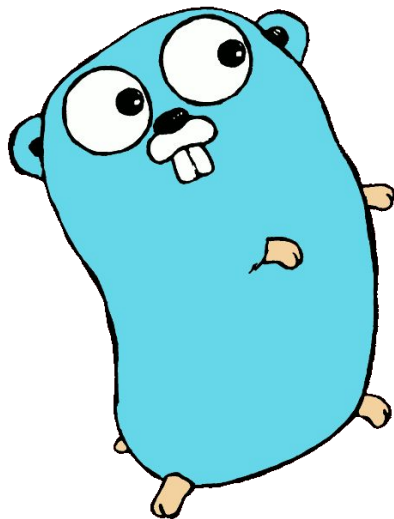
# Go

*"We hated C++ so much, we made a new language..."*

- made by a team at Google because they *really, really, really* hated C++
- **first-class primitives for parallelism**
- **"simple enough to hold in your head"**

Why Go?

- we've learned about weird C++ behavior!
- it's come up in exams and hw problems!



```
func f(from string) {  
    for i := 0; i < 3; i++ {  
        fmt.Println(from, ":", i)  
    }  
}  
  
func main() {  
    f("direct")  
  
    go f("goroutine")  
  
    go func(msg string) {  
        fmt.Println(msg)  
    }("going")  
  
    time.Sleep(time.Second)  
    fmt.Println("done")  
}
```

```
$ go run goroutines.go  
direct : 0  
direct : 1  
direct : 2  
goroutine : 0  
going  
goroutine : 1  
goroutine : 2  
done
```

goroutines with go ([go by example](#))

# opinionated languages

Go is part of a larger trend of opinionated / “restrictive” languages:

- like Rust and Python, there is a “correct” style & formatting
- furthermore, Go does not have: unions, assertions, exceptions, inheritance, overloading, implicit type coercion, ...
- Go doesn’t even allow you to have unused variables!!

Many languages *restrict* you from doing things that you can do in C and C++.

Core argument: what we’ve learned in class from C & C++’s weak typing, and similar arguments to memory safety, threads, and more!

**project 2 stuffs**



# quick tour: call stacks & stack frames

Almost all lexically-scoped languages implement **call stacks**.

- related concepts: **stack frames, activation records**

Core idea: save lexical environments with a stack.

- when calling a function, push onto stack
- when returning from a function, pop off stack

You need to decide: **what goes in the stack?**

# refresher: ip stack

```
00| x = 3
01| def a():
02|     print(x)
03|     b()
04|     print(x)
05|
06| def b():
07|     x -= 1
08|     if x > 0:
09|         a()
10|
11| a()
```

**x = 3**

**output:**

**ip stack**

# refresher: ip stack

```
00| x = 3
01| def a():
02|     print(x)
03|     b()
04|     print(x)
05|
06| def b():
07|     x -= 1
08|     if x > 0:
09|         a()
10|
11| a()
```

x = 3

output:  
3

ip stack

# refresher: ip stack

```
00| x = 3
01| def a():
02|     print(x)
03|     b()
04|     print(x)
05|
06| def b():
07|     x -= 1
08|     if x > 0:
09|         a()
10|
11| a()
```

x = 3

output:  
3

4  
ip stack

# refresher: ip stack

```
00| x = 3
01| def a():
02|     print(x)
03|     b()
04|     print(x)
05|
06| def b():
07|     x -= 1
08|     if x > 0:
09|         a()
10|
11| a()
```

x = 2

**output:**

3

4  
**ip stack**

# refresher: ip stack

```
00| x = 3
01| def a():
02|     print(x)
03|     b()
04|     print(x)
05|
06| def b():
07|     x -= 1
08|     if x > 0:
09|         a()
10|
11| a()
```

x = 2

output:

3

10  
4  
**ip stack**

# refresher: ip stack

```
00| x = 3
01| def a():
02|     print(x)
03|     b()
04|     print(x)
05|
06| def b():
07|     x -= 1
08|     if x > 0:
09|         a()
10|
11| a()
```

x = 2

10  
4  
**ip stack**

**output:**

3  
2

# refresher: ip stack

```
00| x = 3
01| def a():
02|     print(x)
03|     b()
04|     print(x)
05|
06| def b():
07|     x -= 1
08|     if x > 0:
09|         a()
10|
11| a()
```

x = 2

4  
10  
4  
**ip stack**

**output:**

3  
2





A FEW  
MOMENTS LATER

# refresher: ip stack

```
00| x = 3
01| def a():
02|     print(x)
03|     b()
04|     print(x)
05|
06| def b():
07|     x -= 1
08|     if x > 0:
09|         a()
10|
11| a()
```

4  
10  
4  
10  
4  
**ip stack**

**x = 0**

**output:**

3  
2  
1

# refresher: ip stack

```
00| x = 3
01| def a():
02|     print(x)
03|     b()
04|     print(x)
05|
06| def b():
07|     x -= 1
08|     if x > 0:
09|         a()
10|
11| a()
```

4  
10  
4  
10  
4  
**ip stack**

**x = 0**

**output:**

3  
2  
1

# refresher: ip stack

```
00| x = 3
01| def a():
02|     print(x)
03|     b()
04|     print(x)
05|
06| def b():
07|     x -= 1
08|     if x > 0:
09|         a()
10|
11| a()
```

4  
10  
4  
10  
4  
**ip stack**

**x = 0**

**output:**

3  
2  
1  
0

# refresher: ip stack

```
00| x = 3
01| def a():
02|     print(x)
03|     b()
04|     print(x)
05|
06| def b():
07|     x -= 1
08|     if x > 0:
09|         a()
10|
11| a()
```

4  
~~10~~  
4  
10  
4  
**ip stack**

x = 0

**output:**

3  
2  
1  
0

# refresher: ip stack

```
00| x = 3
01| def a():
02|     print(x)
03|     b()
04|     print(x)
05|
06| def b():
07|     x -= 1
08|     if x > 0:
09|         a()
10|
11| a()
```

4  
~~10~~  
4  
10  
4  
**ip stack**

x = 0

**output:**

3  
2  
1  
0  
0

# refresher: ip stack

```
00| x = 3
01| def a():
02|     print(x)
03|     b()
04|     print(x)
05|
06| def b():
07|     x -= 1
08|     if x > 0:
09|         a()
10|
11| a()
```

4  
~~10~~  
4  
~~10~~  
4  
**ip stack**

x = 0

**output:**

3  
2  
1  
0  
0

# refresher: ip stack

```
00| x = 3
01| def a():
02|     print(x)
03|     b()
04|     print(x)
05|
06| def b():
07|     x -= 1
08|     if x > 0:
09|         a()
10|
11| a()
```

4  
~~10~~  
4  
~~10~~  
4  
**ip stack**

x = 0

**output:**

3  
2  
1  
0  
0  
0



# some thoughts

This is a tried-and-true approach to managing instruction pointers! It handles many complicated things for you.

Things to think about:

- when you return, and the stack is already empty, what does that mean?
- can you ever push or pop an invalid item onto the stack?
- **can we extend this to hold other “local” information?**

## extension: call stack

```
01| def cd(x):  
02|     if x == 0  
03|         return  
04|     print(x)  
05|     cd(x-1)  
06|     print(x)  
07|  
08| cd(2)  
09| # end
```

**call stack**

**x =**

**output:**

## extension: call stack

```
01| def cd(x):  
02|     if x == 0  
03|         return  
04|     print(x)  
05|     cd(x-1)  
06|     print(x)  
07|  
08| cd(2)  
09| # end
```

(09, \_\_)  
**call stack**

**x =**

**output:**

## extension: call stack

```
01| def cd(x):  
02|     if x == 0  
03|         return  
04|     print(x)  
05|     cd(x-1)  
06|     print(x)  
07|  
08| cd(2)  
09| # end
```

(09, \_\_)  
**call stack**

**x = 2**

**output:**  
2

## extension: call stack

```
01| def cd(x):  
02|     if x == 0  
03|         return  
04|     print(x)  
05|     cd(x-1)  
06|     print(x)  
07|  
08| cd(2)  
09| # end
```

(09, \_\_)  
**call stack**

**x = 2**

**output:**  
2

## extension: call stack

```
01| def cd(x):  
02|     if x == 0  
03|         return  
04|     print(x)  
05|     cd(x-1)  
06|     print(x)  
07|  
08| cd(2)  
09| # end
```

(06, x = 2)  
(09, \_\_)  
**call stack**

x = 1

**output:**  
2

## extension: call stack

```
01| def cd(x):  
02|     if x == 0  
03|         return  
04|     print(x)  
05|     cd(x-1)  
06|     print(x)  
07|  
08| cd(2)  
09| # end
```

(06, x = 2)  
(09, \_\_)  
**call stack**

**x = 1**

**output:**  
2  
1

## extension: call stack

```
01| def cd(x):  
02|     if x == 0  
03|         return  
04|     print(x)  
05|     cd(x-1)  
06|     print(x)  
07|  
08| cd(2)  
09| # end
```

(06, x = 1)  
(06, x = 2)  
(09, \_\_)  
**call stack**

**x = 1**

**output:**  
2  
1



## extension: call stack

```
01| def cd(x):  
02|     if x == 0  
03|         return  
04|     print(x)  
05|     cd(x-1)  
06|     print(x)  
07|  
08| cd(2)  
09| # end
```

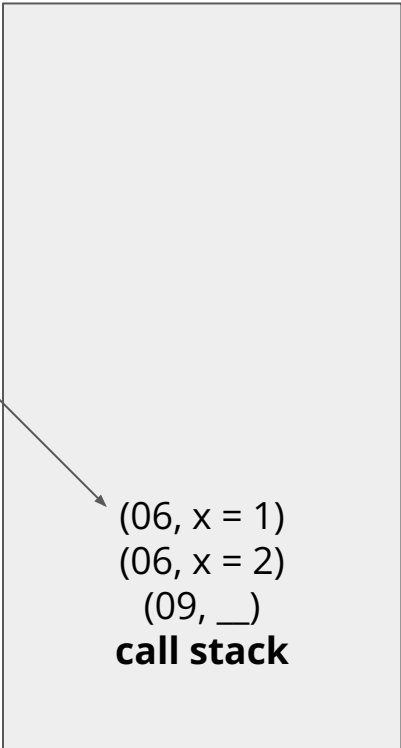
(06, x = 1)  
(06, x = 2)  
(09, \_\_)  
**call stack**

**x = 0**

**output:**  
2  
1

## extension: call stack

```
01| def cd(x):  
02|     if x == 0  
03|         return  
04|     print(x)  
05|     cd(x-1)  
06|     print(x)  
07|  
08| cd(2)  
09| # end
```



(06, x = 1)  
(06, x = 2)  
(09, \_\_)  
**call stack**

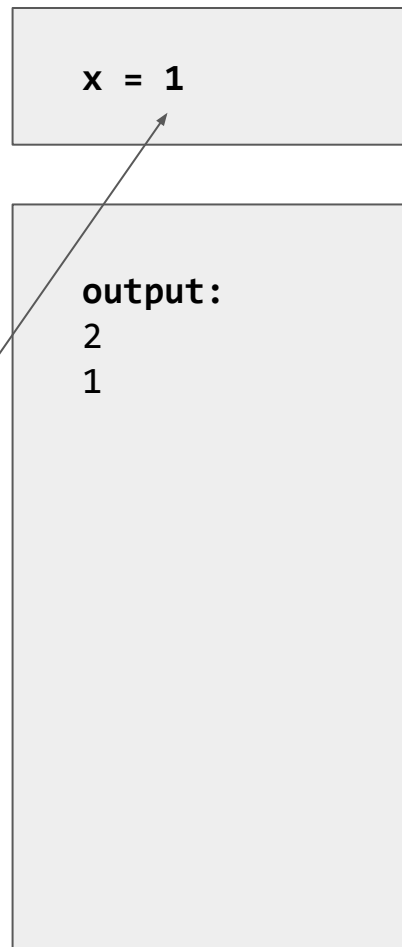
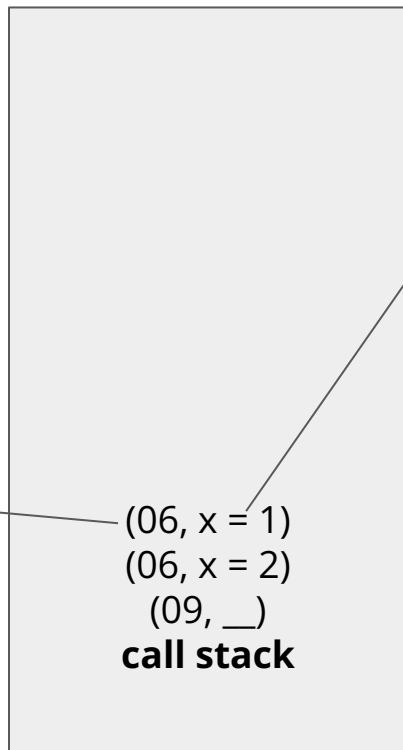
**x = 0**

**output:**

2  
1

## extension: call stack

```
01| def cd(x):  
02|     if x == 0  
03|         return  
04|     print(x)  
05|     cd(x-1)  
06|     print(x)  
07|  
08| cd(2)  
09| # end
```



## extension: call stack

```
01| def cd(x):  
02|     if x == 0  
03|         return  
04|     print(x)  
05|     cd(x-1)  
06|     print(x)  
07|  
08| cd(2)  
09| # end
```

~~(06, x = 1)~~  
(06, x = 2)  
(09, \_\_)  
**call stack**

**x = 1**

**output:**

2  
1  
1

## extension: call stack

```
01| def cd(x):  
02|     if x == 0  
03|         return  
04|     print(x)  
05|     cd(x-1)  
06|     print(x)  
07|  
08| cd(2)  
09| # end
```

~~(06, x = 1)~~  
(06, x = 2)  
(09, \_\_)  
**call stack**

**x = 1**

**output:**

2  
1  
1

## extension: call stack

```
01| def cd(x):  
02|     if x == 0  
03|         return  
04|     print(x)  
05|     cd(x-1)  
06|     print(x)  
07|  
08| cd(2)  
09| # end
```

~~(06, x = 1)~~  
~~(06, x = 2)~~  
(09, \_\_)  
**call stack**

**x = 2**

**output:**  
2  
1  
1

## extension: call stack

```
01| def cd(x):  
02|     if x == 0  
03|         return  
04|     print(x)  
05|     cd(x-1)  
06|     print(x)  
07|  
08| cd(2)  
09| # end
```

~~(06, x = 1)~~  
~~(06, x = 2)~~  
(09, \_\_)  
**call stack**

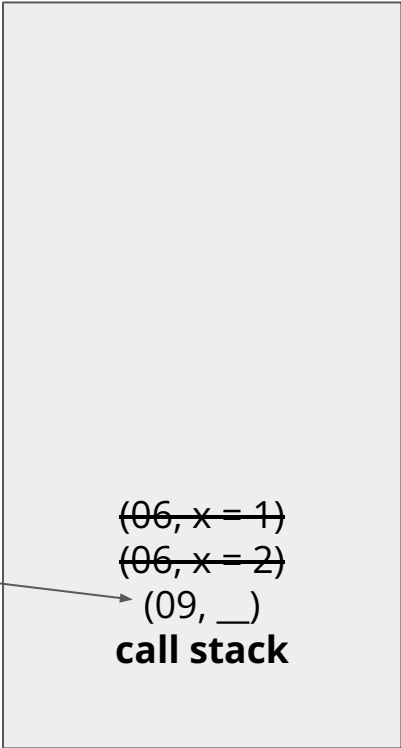
**x = 2**

**output:**

2  
1  
1  
2

## extension: call stack

```
01| def cd(x):  
02|     if x == 0  
03|         return  
04|     print(x)  
05|     cd(x-1)  
06|     print(x)  
07|  
08| cd(2)  
09| # end
```



~~(06, x = 1)~~  
~~(06, x = 2)~~  
→ (09, \_\_)  
**call stack**

**x = 2**

**output:**

2  
1  
1  
2



## extension: call stack

```
01| def cd(x):  
02|     if x == 0  
03|         return  
04|     print(x)  
05|     cd(x-1)  
06|     print(x)  
07|  
08| cd(2)  
09| # end
```

~~(06, x = 1)~~  
~~(06, x = 2)~~  
~~(09, \_\_)~~  
**call stack**

x = \_\_

**output:**

2  
1  
1  
2

# observations

Call stacks neatly solve:

- function calls, including recursion
- local *and* global variables

Our example of a call stack does *not* solve:

- function argument names
- nested scope (if, while, ...) & shadowing
- pass-by-reference
- maybe some other things ...

However, this is a **good mental model**, and the above problems can be solved relatively neatly!

# an overall note on incremental dev

This project is set up for you to do incremental work:

- you have a fully-working local test suite
- you have a subset of test cases, and can write your own
- you get your mark back in real-time

## Use this to your advantage!

Don't binge-code / stare at the project for hours; pick a small feature, write some test cases, implement it, and repeat!

(this advice is very helpful *outside of this class*)

# if you're just starting...

First, it's okay! **Don't panic!**

Some overall tips:

1. build a mental model of what needs to be done, in what order
  - a. ex: lexical scoping is *necessary* to implement shadowing, etc.
  - b. ex: function arguments are *necessary* for pass-by-ref, but nothing needs pass-by-ref
2. while you still have time:
  - a. pick one feature (ideally, the most necessary one)
  - b. write a few test cases
  - c. implement what you need to get them to pass
  - d. repeat

## example of a trivial test case

```
func main
  funccall foo
  print a # should be an error!
end func

func foo
  assign a 10
  print a # should print 10
end func
```

# example of a trivial test case

```
func main
  funccall foo
  print a # should be an error!
end func

func foo
  assign a 10
  print a # should print 10
end func
```

Some notes:

1. **This isn't valid Brewin++!**
2. But, it's the *minimum* example to test lexical scoping:
  - no static types, function parameters, return
  - no shadowing, nested blocks
  - very simple code!
3. Suggestion: write test cases like these *incrementally*

# if you're not yet at 25/25 on public test cases...

First, it's okay! **Don't panic!**

Some overall tips:

1. each test case contains only a handful of concepts: **do you know which ones?**
2. for the test cases that don't work, isolate the problem
3. suggestion: think about your data structure (ex: stacks)
4. still stuck?
  - a. **Ask for help!** (but, maybe before Monday night?)
  - b. Bandaid fix and move on!

# if you are at 25/25 on public test cases

First, nice job :) that's the hard part!

You should:

- identify what language features public test cases don't explicitly test
- look for *interaction* between features:
  - ex: how do you handle shadowing when the variables have different types?
  - ex: pass by reference and recursion
- write more test cases!!



# things I would do

Help yourself by:

- writing reasonable error messages :)
  - remember: we're not grading you on the description, so add what you need!
- writing an error message when something we *won't* test you on happens
  - indicative of a bug in your implementation!
  - ex: wrong number of arguments to function
- saving your code frequently!
  - ex: use git; make your commit message the # of cases you've passed!

# things I would not do

These are mostly around **overcomplication** and **early optimization**.

- **do not worry about time or space optimization (within reason).**
  - it is *so unlikely* that your program will take >5s or use all the RAM on the grader
  - “inefficient” algorithms that are  $O(n^2)$ ,  $O(2^n)$ , etc. are fine!
- **don't do X because Python/other language implements X.**
  - read the spec!
  - real-life interpreters are much more complicated than what's needed for your project!
- **you don't need more libraries for this part of the project!**
  - (just my opinion!)

## **some more, general advice**

Some seemingly-simple constructs can be broken up. Ex, declaration is:

- reserving a variable name
- read type annotation + set default value
- possibly: deal with shadowing
- and ... more?

These bite-sized operations can be tested independently!

## some more, general advice

And, some more questions for you:

- now that functions can have arguments, are the builtins (print, input, strtoint) still “special cases”? Why or why not?
- are void and the ref types true “types”? Why or why not?

If you haven't already, review my [slides last week](#) on Proj 2 too!

also, I have **two office hours on Monday**

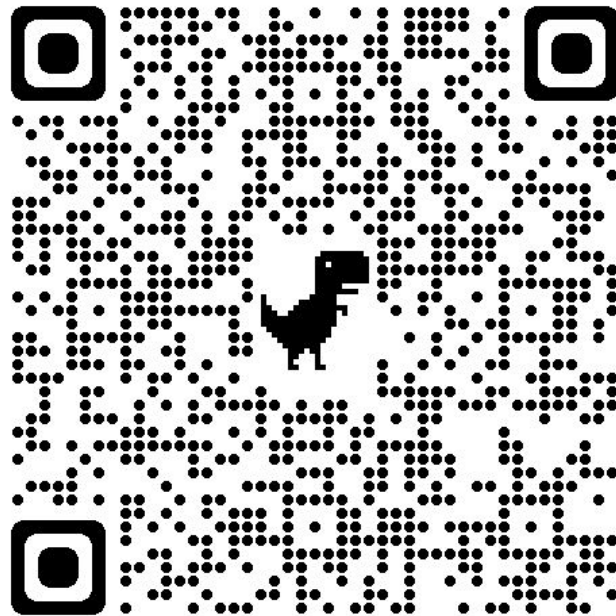
1. **11-12:30**, in-person (Boelter 3256S)
2. **5-6**, online ([mattxw.com/zoom](https://mattxw.com/zoom))

## ~ **post-discussion survey** ~

always appreciate the feedback!

- good warmup problem format?
- how was the midterm?
- was the call stack helpful?

see you in ... two weeks :((



<https://forms.gle/33gPkKDfajrrQrZ88>

**midterm / project q&a**

(feel free to leave!)

## Useful Project Links:

- [Spec](#)
- [Gradescope](#)
- [Template](#) + [Autograder](#) (optional)



appendix / slide graveyard

## **bonus content: tail call optimization**

\*matt is only running this section if he ends super early

## **bonus content: tail call optimization**

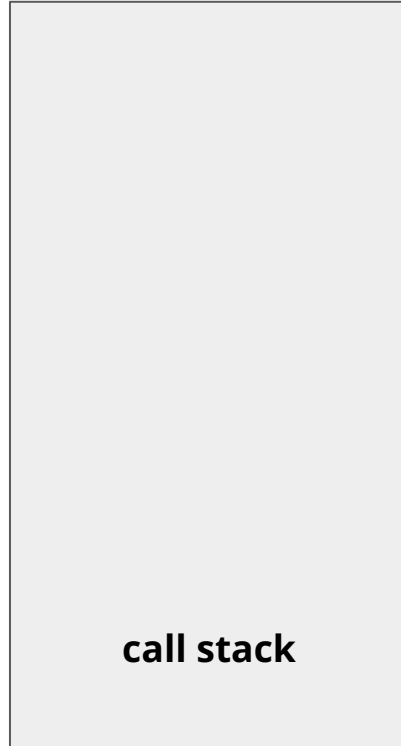
You might be wondering: why is Python's recursion limit so low? And is it practical to use recursion for *everything* in most languages, including to replace loops?

### **Stellar question!**

To answer that, let's return to our call stack model.

# call stack model

```
01| def is_even(x):  
02|     if x == 0:  
03|         return True  
04|     return is_odd(x-1)  
05| def is_odd(x):  
06|     if x == 0:  
07|         return False  
08|     return is_even(x-1)  
09| is_even(100)
```



Q: When we call `is_even(100)`, how many items go on the call stack?

# call stack model

```
01| def is_even(x):  
02|     if x == 0:  
03|         return True  
04|     return is_odd(x-1)  
05| def is_odd(x):  
06|     if x == 0:  
07|         return False  
08|     return is_even(x-1)  
09| is_even(100)
```

...  
(04, x=90)  
(08, x=91)  
(04, x=92)  
(08, x=93)  
(04, x=94)  
(08, x=95)  
(04, x=96)  
(08, x=97)  
(04, x=98)  
(08, x=99)  
(04, x=100)  
**call stack**

Q: When we call  
`is_even(100)`, how many  
items go on the call stack?

A: 100!

# call stack model

```
01| def is_even(x):  
02|     if x == 0:  
03|         return True  
04|     return is_odd(x-1)  
05| def is_odd(x):  
06|     if x == 0:  
07|         return False  
08|     return is_even(x-1)  
09| is_even(100)
```

```
...  
(04, x=90)  
(08, x=91)  
(04, x=92)  
(08, x=93)  
(04, x=94)  
(08, x=95)  
(04, x=96)  
(08, x=97)  
(04, x=98)  
(08, x=99)  
(04, x=100)  
call stack
```

Q: When we call `is_even(100)`, how many items go on the call stack?

A: 100!

But, that feels ... really inefficient. Could we redesign the call stack to fix this problem?

Hint: what's the last thing we do in `is_even`?

# call stack model

```
01| def is_even(x):  
02|     if x == 0:  
03|         return True  
04|     return is_odd(x-1)  
05| def is_odd(x):  
06|     if x == 0:  
07|         return False  
08|     return is_even(x-1)  
09| is_even(100)
```

```
...  
(04, x=90)  
(08, x=91)  
(04, x=92)  
(08, x=93)  
(04, x=94)  
(08, x=95)  
(04, x=96)  
(08, x=97)  
(04, x=98)  
(08, x=99)  
(04, x=100)  
call stack
```

These functions are what we call **tail recursive**: the very last thing they do is return a function call (a **tail call**).

In that case, we don't even need the entire stack frame, since we'll never use those values again!

# call stack model

```
01| def is_even(x):  
02|     if x == 0:  
03|         return True  
04|     return is_odd(x-1)  
05| def is_odd(x):  
06|     if x == 0:  
07|         return False  
08|     return is_even(x-1)  
09| is_even(100)
```



...  
(04)  
(08)  
(04)  
(08)  
(04)  
(08)  
(04)  
(08)  
(04)  
(08)  
(04)  
**call stack**

These functions are what we call **tail recursive**: the very last thing they do is return a function call (a **tail call**).

In that case, we don't even need the entire stack frame, since we'll never use those values again!

This could still work! In fact, we may not even need the IP stack anymore ...



# but...

there's a gotcha!

**not all functions are tail recursive!**

```
fib :: (Int n) => n -> n  
fib 1 = 1  
fib 2 = 1  
fib n = fib(n-1) + fib(n-2)
```



this addition is the last thing we do!

**but...**

there's a gotcha! **not all functions are tail recursive!**

however, you can usually rewrite them to be tail recursive :)

```
fibAux n result previous
| n == 0 = result
| otherwise =
    fibAux (n - 1) (result + previous) result
```

```
fib n
| n == 0 = 0
| otherwise = fibAux n 1 0
```

# tail call optimization

Most functional-first languages implement **tail call optimization (TCO)**: they smartly eliminate stack frames with tail calls.

- Ex: Haskell, OCaml, many Lisp dialects, Erlang

However, no “mainstream” languages fully implement TCO:

- most, like Python and JavaScript\*, don't even try
- some, like Scala and Rust, can do some tail calls but not all
  - ex [sibling call optimization in LLVM](#); mutual tail calls are notoriously hard

# bigger picture

Bottom line: limitations to using recursion *instead of loops* in most languages.

However, **recursive strategies still work**, and can be **more readable, maintainable, and testable!**

Tail call optimization may not always be good! Guido van Rossum argues that it makes debugging harder, which is a priority for Python.

# Fun reading on Python & Tail Recursion

(by Guido van Rossum, the *creator of Python!*)

- [Tail Recursion Elimination](#)
- [Final Words on Tail Calls](#)

# Julia

*Better, faster Python for math and science!*

Julia in a nutshell:

- Python-like abstractions, dynamic typing, REPL support (the **Ju** in **Ju**pyter Notebooks)
- but, C-like speed!!

And some neat things:

- great support for arbitrary-precision math (midterm wasn't lying!)
- **“multiple dispatch”** –

