Happy Thanksgiving, eh!
Interface vs. Implementation
Abstract data types: a common language

Set
Multiset
List
Map
Iterable
Stacks and Queues

- Push
- Pop
- Enqueue
- Dequeue
- Back
- Front
Simplicity is powerful

In Python, frames for function calls form a stack.

In Assignment 1, the people on each floor form a queue.
Key phrases for communicating errors

Abstract behaviour
“When this function runs, we expect X to happen, but Y happens instead.”

Concrete prediction
“We can check that Y happens by doing…”

Code-centered explanation
“The problem is at Line n, because…”
Raising exceptions

How can we immediately report to client code that one of our functions was called incorrectly?
Recall: strategies for handling bad inputs

Preconditions (“it’s the user’s fault”)

Do nothing (“fail silently”)

Input processing (“fix the problem for them”)
Exceptions

An exception is a special object in Python that represents some kind of error.

Raising an exception is a way to interrupt the normal execution of a program. The exception object is used to report the type of error, and relevant details.
Stack.pop, again
Get to know each other!

If you could instantly become an expert in something, what would it be?
Diversion: balancing parentheses
A string is *balanced* when...

Every ( is followed by a matching ). Nested is allowed.

Balanced
- ( )
- (1 2 (4 5))
- (((4)))
- ()()()()()

Not balanced
- (1 )
- ( )
- bla()a()ah
(a(s(d(sdf)safd(s(gdas)))(qwerqwer)sadf)((werqew()df(asd(asdfa sdfasf))))()sdfs(d(((sfsfsdad)sdfsdfasdfas))(wer(wer(w)erwe(r))()wqwqqeww(()(d(fsdfadssdfs)(()()dfsdfsdfsdfsa)dfsa(dass(fdasqwerwd(d(fas)dfas dfas))))))
Key ideas

Ignore all characters except ( and ).

Keep track of when you see a (, but forget about it when you’ve seen the matching ).
START
END
(1 - (5 * 8) + 6
START
END
START
6
END
4 + 6
Now design your function!
Evaluating efficiency of implementations

Given multiple implementations of the same interface, what are different ways we can compare them?
Consider another stack implementation
A timing experiment

A common technique used to gain evidence about the efficiency of some code is to run a **timing experiment** that simply runs the code and see how long it takes to run.

Such experiments often are repeated multiple times for different **sizes of data** (in our case, stack sizes).
Two fundamental questions

1. Why do Python lists behave this way?

2. How can we talk about running time more precisely, without relying on timing experiments?