CSC 148

Week 4

Sadia ‘Rain’ Sharmin

Classes begin 10 minutes after the hour

This Week's Cool Programmer(s) Feature: *Joy Buolamwini*

Week 4

Abstract Data Types

SUBTOPICS:

- ADTs: Stacks
- Writing Client Code
- Handling Exceptions
- Efficiency of Stack Implementations

Announcements

Assignment 0

It's due this Friday!

ADTs: Stacks

Abstraction

Why it matters

Client code can be written without knowing anything about the implementation.

Reduces cognitive load for the programmer of client code.

Modern, complex software would be impossible otherwise.

Implementation can change with no effect on client code.

We call this plug-out, plug-in compatibility.

Abstract Data Types

ADTs

ADTs take abstraction a step further

An Abstract Data Type such as a stack, queue, set, etc. is beyond any particular implementation, beyond even any particular programming language!

It is part of a common language used by programmers everywhere.

Abstract Data Types

A Common Language

In CS, we recycle our intuition about the outside world as ADTs. We abstract data and operations, and suppress the implementation.

For example:

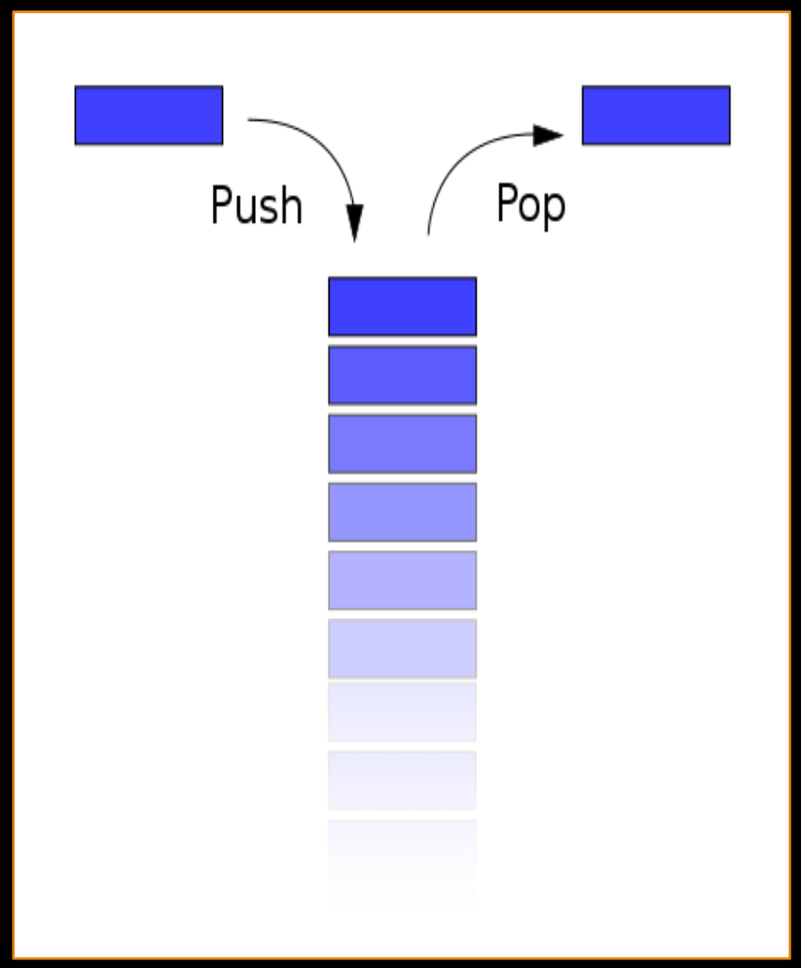
Sequences of items; can be added, removed, accessed by position, etc.

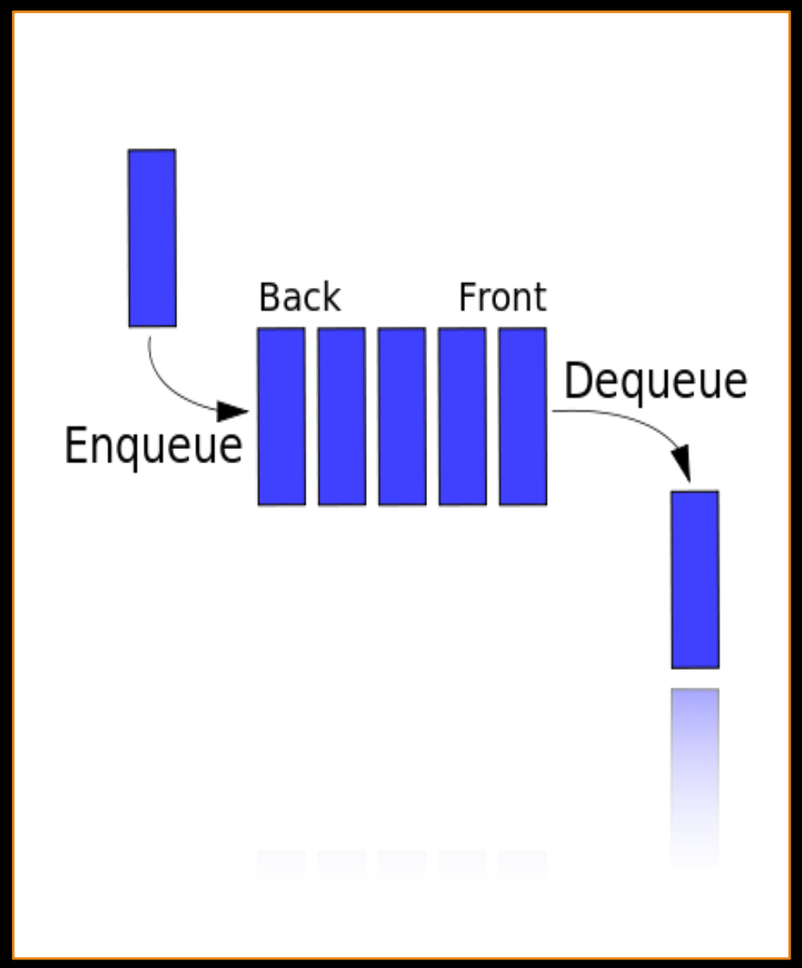
Specialized collection of items where we only have access to most recently added item

Collection of items accessed by their associated keys

Two ADTs we'll discuss

Stacks and Queues





Stacks Application

Balanced Parentheses

Let's think about this

Stacks

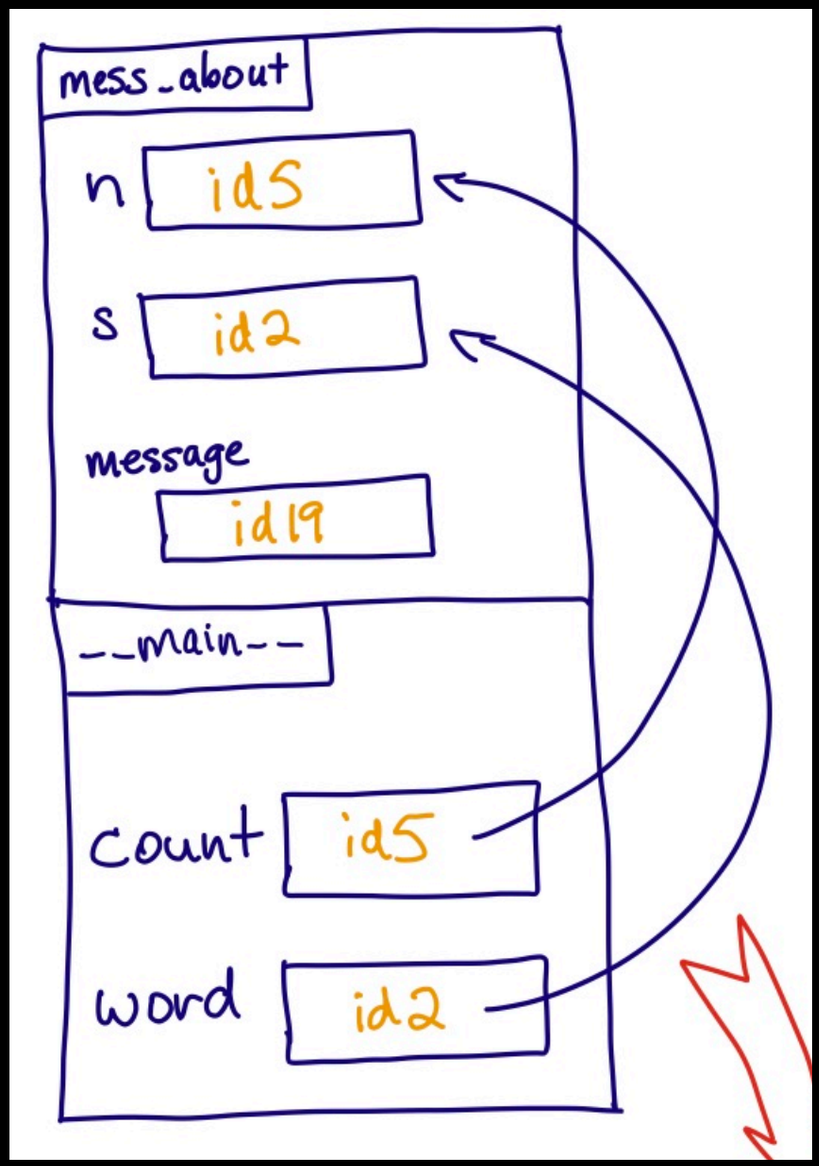
Where have we seen a stack being used within our coding practices before?

Stacks

Let's think about this

Where have we seen a stack being used within our coding practices before?

In Python, frames for function calls form a stack. Remember: *call stack*



Stacks

Let's think about this

Where have we seen a stack being used within our coding practices before?

In Python, frames for function calls form a stack. Remember: *call stack*

What does a stack store?

What are the operations?

Stack Application

Parentheses Checking

In some situations, it is important that opening and closing parentheses, brackets, braces match

(a * b) + c good

a *) b + (c bad

Why is this useful?

e.g., your IDE checks for well-formed code

compilers, interpreters, calculators, etc.

Stack Application

Define balanced parentheses

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

Balanced

() (1 2 (4 5)) (((4))) ()()()()()

Not balanced

(1)((() bla()a()ah)

Stack Application

Define balanced parentheses

A string with no parentheses is balanced

A string that begins with a left parenthesis “(“, ends with a right parenthesis “)”, and in between has balanced parentheses.

The concatenation of two strings with balanced parentheses is also balanced: (...) (...)

Stack Application

Your task

We will only deal with regular parentheses (for now) – i.e. we only care about round bracket characters (and).

We'll give you 4 examples of expressions, one character at a time – Remember, the computer only “sees” one character at a time

Use the stack on the worksheet, to determine if the parentheses are balanced!

Ready?

WORKSHEET

Balancing Parentheses

Stack Application

Use the stacks!

We want you to get practice using stacks today, so make sure you think carefully about how the stacks can be used to solve this problem.

START

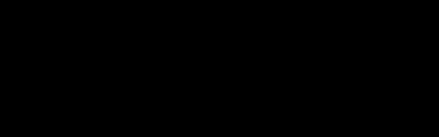
(

1

(

9

8



6

END

$$(1 - (5 * 8) + 6$$

How to use the stacks

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

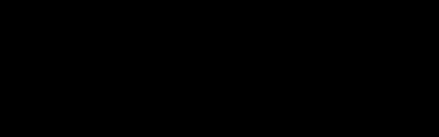
(

a

(

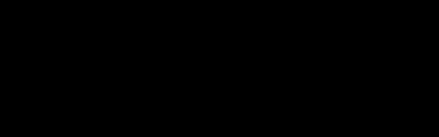
(

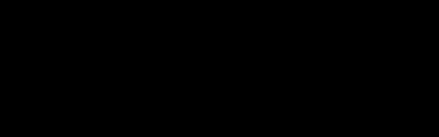
3

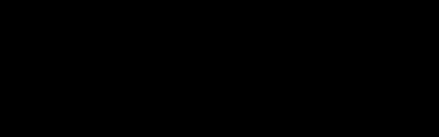


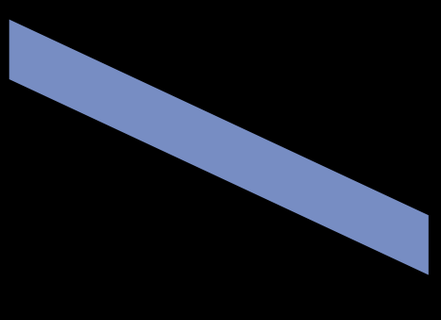
(

b





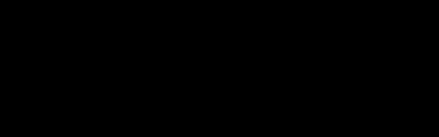




(

2

b

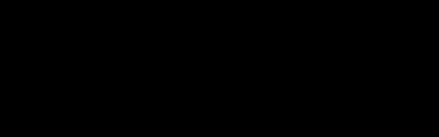


END

(a * ((3) + (b))) / (2 - b)

START

4



6

(

END

)9(+ 4

We'll give you the fourth expression all in one go.

Take the same approach.

Ready?

(a+((b-(c/(d*e))-(f+g))))

WORKSHEET

Balancing Parentheses

Q2: Discuss in breakout rooms

The Algorithm

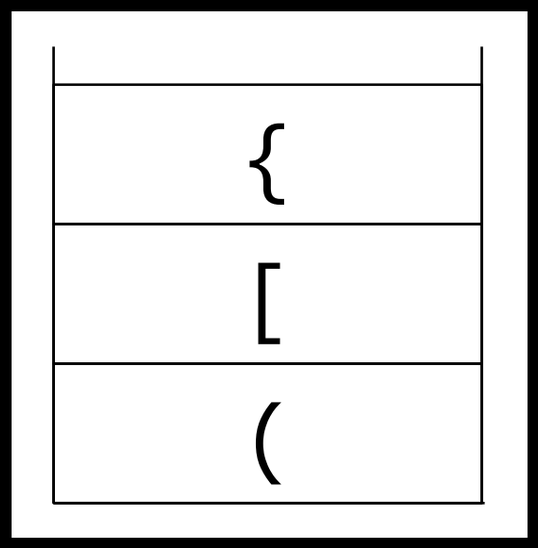
Why do it like this?

The computer only “sees” one character at a time

(1 + [2 * { 9 / 3 }] – 4)

Stack can be used to check balanced parentheses as we've seen in the previous examples

Push when it’s a “leftie”, pop and compare when it’s a “rightie”



} ✓

] ✓

) ✓

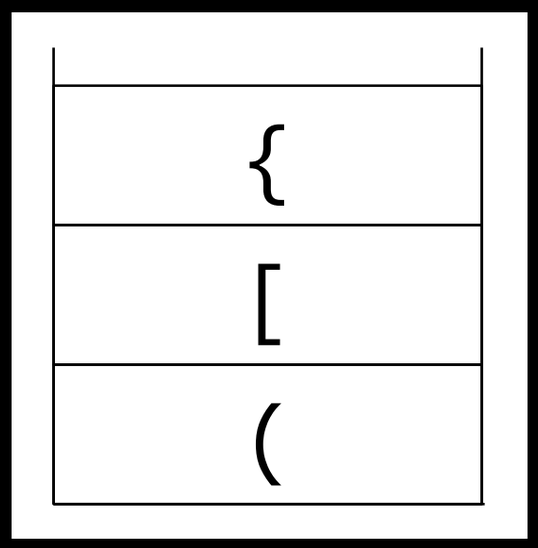
The Algorithm

Let's see what happens if imbalanced:

(1 + [2 * { 9 / 3 }) – 4]

Stack can be used to check balanced parentheses

Push when it's a "leftie", pop and compare when it's a "rightie"



}

)

WORKSHEET

Balancing Parentheses

Q3: Implement the Function

Writing Client Code

WORKSHEET

Stack Size

Handling Exceptions

Exceptions

pop()

Pop the item off the top of the stack

What happens when the stack is empty?

Code should raise an exception

But, client code should not crash

Handle this gracefully => handling exceptions

Raising Exceptions

Side note:

How can we immediately report to client code that one of our functions was called incorrectly?

Remember this?

Strategies for handling bad inputs

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

Do nothing (“fail silently”)

Input processing (“fix the problem for them”)

Exceptions

What are they?

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

Exceptions

Types of Exceptions

Generic Exception class

Other predefined exceptions

e.g., NotImplementedError, IndexError, etc.

May define our own custom exceptions

Can subclass Exception to define new custom exceptions

Let's create our own exception EmptyStackError and use it when we try to pop() from an empty stack

DEMO

Raising Exceptions

See stack.py

Exceptions

Handling Exceptions

Exceptions are useful for developers but you will often want to prevent a program from crashing when an exception occurs for the sake of your end-users.

You can use try...except to handle exceptions in a neater way rather than just letting the program crash.

DEMO

Handling Exceptions

See except.py and no_except.py

Efficiency of Stack Implementations

Efficiency

Stack implementations

Given multiple implementations of the same interface, what are different ways we can compare them?

Let's consider a different stack implementation.

WORKSHEET

Considering a different implementation of class Stack

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)

Efficiency

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?

Efficiency

Python lists

How are Python lists implemented, and what are the implications for the running time of list operations?

Efficiency

A Python list in memory

A Python list stores the ids of its elements in a **contiguous block of memory.**

A Python list is actually a dynamic array based implementation.

Read more here: https://www.laurentluce.com/posts/python-list-implementation/

Efficiency

A Python list in memory

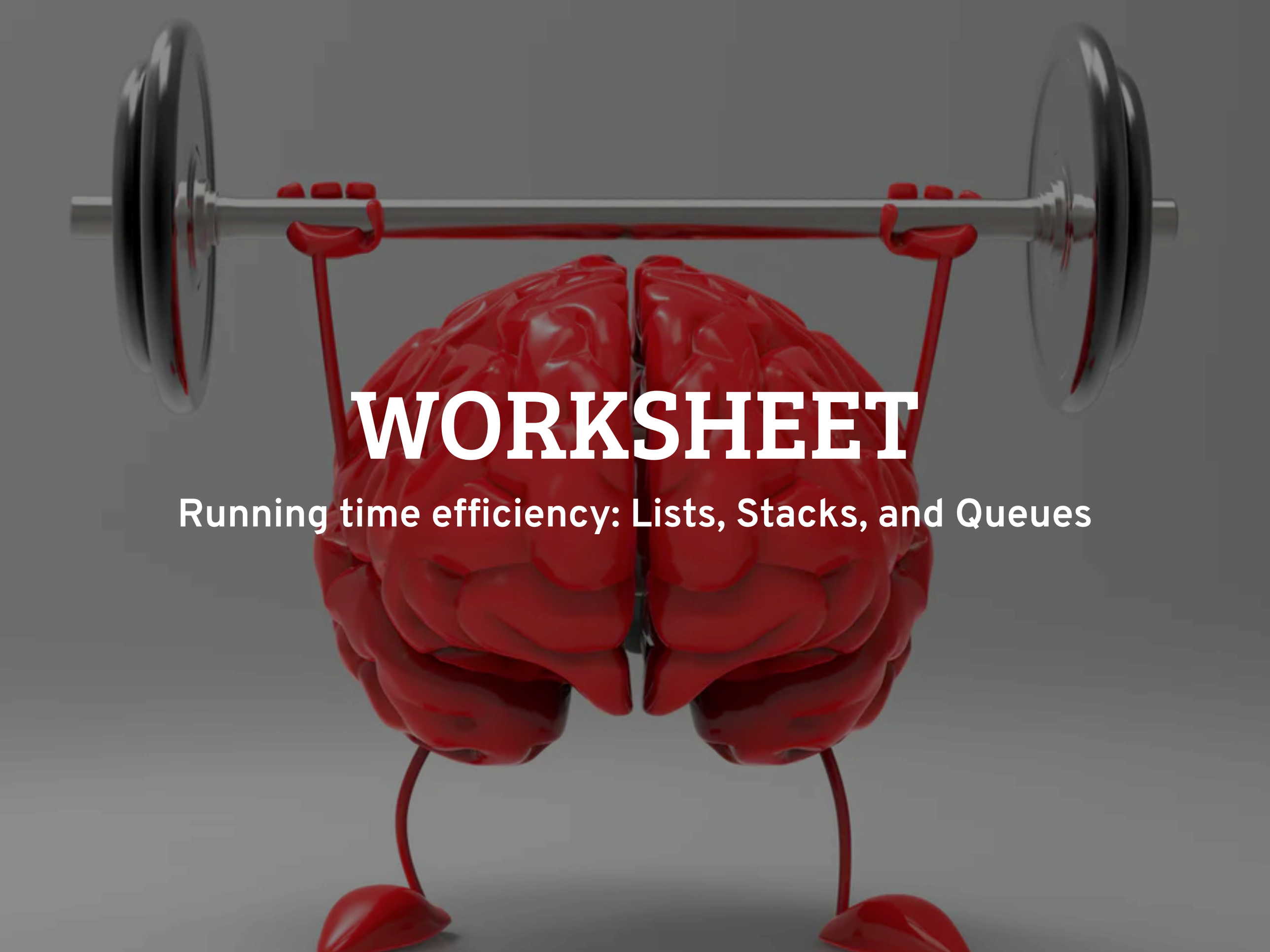
This makes list indexing take constant time: the running time doesn't depend on the length of the list.

Efficiency

A Python list in memory

Insertions and deletions must preserve the contiguity of the element ids: when we're inserting/removing from anywhere but the end of the list, other elements must be *shifted over*.

Key thing to remember: the front of the list is fixed, while the back can “expand” to take up more or less space.



WORKSHEET

Running time efficiency: Lists, Stacks, and Queues

Efficiency

Communicating about running time

Goal: communicate how long a function/program takes to run as it is given larger and larger inputs.

In other words, we want to describe running time as a function of input size.

Efficiency

Communicating about running time

We want a way of talking about running time that doesn't depend on timing experiments or exact step counts.

What we care about is the *type of growth*:

logarithmic, linear, quadratic, exponential, etc.

Efficiency

Big-Oh notation

$O(n)$, $O(n^2)$, $O(\log n)$, $O(2^n)$, ...

See Section 3.4 of the Lecture Notes.