

# Problem Solving for Technical Interviews

Interviewing Skills from the Ground Up  
Junior Dev Struggle Bus  
January 2020

# Introductions

# About You

I want to know:

- Your name
- What you hope to learn in this series
- Your favorite technology / language / tech stack
- A little-known fact about you



# About Me

- Name: Lizzy
- Background:
  - B.A. in Music (a long time ago, never finished degree)
  - Extra generic administrative experience (2010-2018)
  - B.S. in Computer Science (current, graduating in June)
- Favorite CS subject: Graphs
- Other tidbits:
  - My first career goal was to be a stegosaurus
  - I love karaoke
  - I make a really dank carrot cake



# What This Workshop Is

- A deep dive into **data structures** and the **algorithms** we use to navigate and modify them
- A bridge between bootcamp curriculum and CS academia
  - We do not focus on frameworks, UI components, or teach syntax specifics
  - Instead, we discuss data representation in an abstract and language-agnostic way, and learn syntax as a means to the end of implementing data structures
- A safe environment to review technical concepts and learn new ones
- Programming concepts demonstrated in Python 3



# What This Workshop Is *Not*

- A magic bullet
  - The topics and problems we will discuss here are only an introduction
  - It takes more than 2 hours to internalize core concepts that are completely foreign to us
  - The iceberg of expertise is 10% learning and 90% practice
- A competition or race
  - Everyone benefits from lifting each other up, as is the JDSB way :)
- A behavioral interview / resume workshop
  - Links to external sources will be provided, but we will not spend meaningful time in this workshop preparing elevator pitches or discussing how to beat an ATS





# Interviews Introductions

# Why Do Technical Interviews Exist?

For interviewers, technical interviews demonstrate the candidate's:

- Existing technical skills and (sometimes) domain knowledge
- Problem-solving strategies
- Communication style

*...In order to assess...*

**“Would I want to work with this person on real problems?”**

**“Would they be a good fit on our team?”**

To summarize: **Interviews are a conversation.**

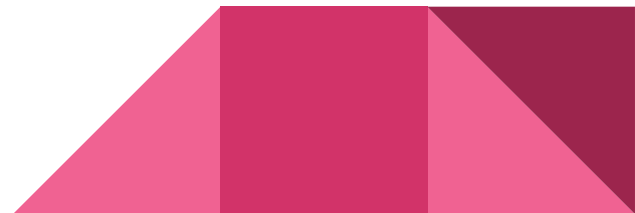




# What Do Interviewers Look For?

An ideal candidate:

- Demonstrates deep understanding of problem statement and solution mechanics
- Effectively communicates thought process
- Demonstrates deep understanding of relevant or useful data structures
- Approaches problems with creativity and shows flexibility in approach
- Discusses solution efficiency and trade-offs of various approaches
- Discuss testing methodologies for solution & test the solution they wrote
- Demonstrate competence in language of choice
  - Is this an afterthought?



# The Most Important Questions

Your interviewer will be asking themselves:

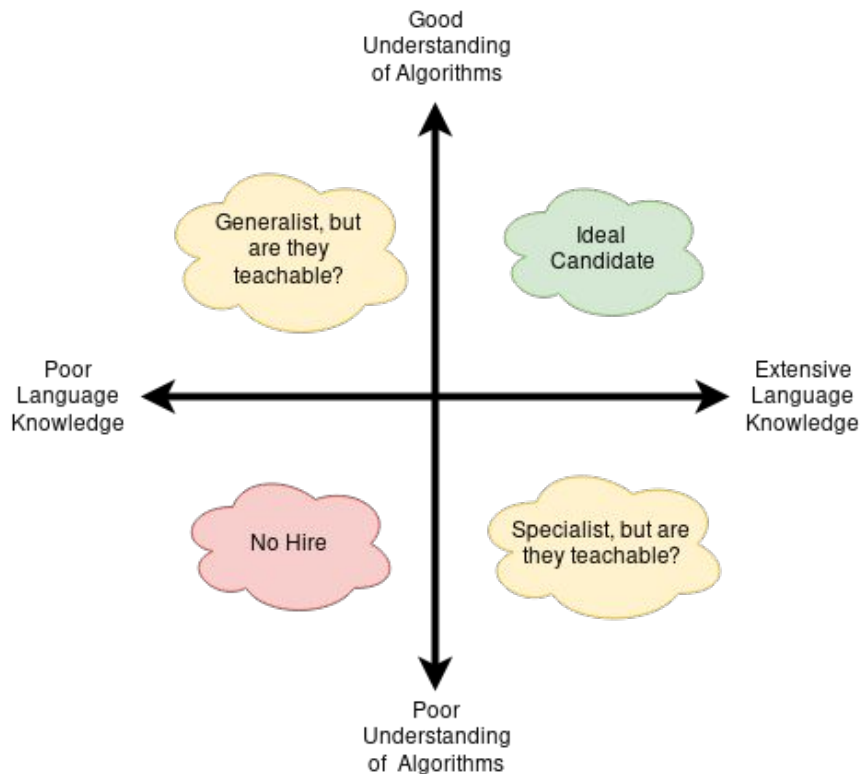
***“Would this person be a good fit for our team?”***

***“Would I want to collaborate with this person and solve real problems together?”***

Yes, this was on a previous slide. It's so important that it deserves to be repeated.



# Does the Programming Language Matter?



- In most technical interviews:
  - You may pick your preferred programming language
  - Interviewer will ask the same technical question regardless of your language choice
- There are two axes of evaluation:
  - Strategy of solution
  - Language knowledge

# Why Whiteboards?

- There are lots of good reasons to move away from whiteboard interviews (like [this](#) and [this](#) and [this](#) and...)
- Whiteboard interviews are the existing norm, so for the time being, we will have to get comfortable with it.
- They are useful because:
  - Complex solutions take thought and idea exploration; often times, we find ourselves sketching out ideas in notebooks, on napkins, or on *whiteboards*
  - On the job, you will inevitably collaborate with colleagues using this exact interview format: drawing designs and writing code snippets *on a whiteboard*
  - If you are capable on a whiteboard, you will rock at writing the actual code in the IDE of your choice.

# Whiteboarding Resources

These are some good articles on the topic.

- <https://medium.com/@emailbivas/programmers-fail-interviews-because-of-the-whiteboard-161df28ad74b>
- <https://www.offerzen.com/blog/tips-improve-the-things-you-control-in-interview>



# Introduction to Python

# Language of Choice

- This workshop will discuss data structures and algorithms, and implement these structures in Python
- We use Python because it is:
  - Easy to learn
  - Perfect for prototyping and trying things out
  - A concise, expressive language which is great for technical interviews
  - Everywhere in industry
  - Fun!



# The Basics

- Python is an ***interpreted*** scripting language
  - Interpreted: values are computed as lines of code are read by interpreter
- There are built-in data types and functions
  - Data types: bool, int, str, float
  - Collection types: list, set, dict, tuple
  - Functions: len(), type(), str(), repr(), sorted(), and many more
- Data is dynamically typed
- Code blocks are identified by indentation instead of braces
  - True for `if` statements, `for` loops, functions, and more!





# A First Look: Python vs Javascript

```
if this_is_true:
    do_a_thing()
else:
    pass          # do nothing
```

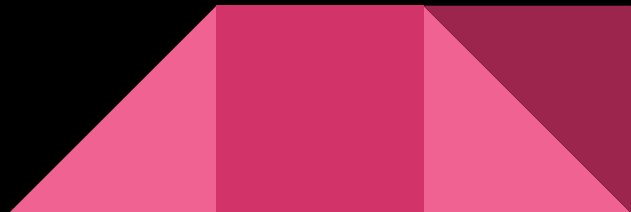


```
if (thisIsTrue == true) {
    doAThing();
} else {
    // do nothing
}
```

```
stuff = [1, 'a', 3.14159]
for item in stuff:
    print(item)
```



```
var stuff = [1, 'a', 3.14159]
for (int i = 0; i < stuff.length; i++) {
    console.log(stuff[i])
}
```



# Functions and Variables

# Variable Syntax

- Variables are implicitly declared (no var keyword)
- Memory is allocated on value assignment
- Anything can be assigned to a variable
  - Data
  - Function labels
  - Data type labels

```
x = 3
y = x
foo = print
foo(3)      # prints 3 to console
bar = int
bar is int  # True
```



# Function Syntax

- Functions are methods
  - Routines that can be called repeatedly
  - Can receive data as argument(s)
  - Can return data
- Defined using the def keyword

```
def triple(x):  
    return x * 3  
  
s = triple('hi') # 'hihihi'  
i = triple(3)   # 9
```

# Data Types

# The Basics

- Integers
- Floats
- Booleans
  - Capitalized: True and False
- Strings
  - No chars - a one-letter string is a string
  - Can be declared with single or double quotes ('Hello' and "World")



# Lists

- A **mutable**, array-like structure
- Raw declarations use square brackets around comma-delimited entries
- Elements are accessed by zero-based index
- List elements can be of any type

```
my_list = [1, 2, 3, 'a', 'b', 'c']  
my_list.append([4, 5, 6]) # add an inner list as an element  
print(my_list[6])         # [4, 5, 6]  
my_list.extend([4, 5, 6]) # concatenates argument to my_list
```

# Tuples

- An **immutable** array-like structure
- Raw declarations use square brackets around comma-delimited entries
- Elements are accessed by zero-based index
- Accepts duplicate data and list entries can be of any type

```
my_tuple = ('a', 'b', 'c', 'd', 'e')  
my_tuple.append('f') # ERROR
```



# Sets

- A collection of unordered, unique elements
- Raw declarations use curly braces around comma-delimited entries
- For the math geeks: supports common set operations (union, difference, intersection)

```
alphabet = {'a', 'b', 'c', 'd', 'e'}  
alphabet.add('f')      # adds to set  
alphabet.add('a')      # does nothing  
print(alphabet)        # {'a', 'b', 'c', 'd', 'e', 'f'}
```

# Dictionaries

- A map-like structure
  - Maps a unique key to a provided value
- Raw declarations use curly braces around comma-delimited entries. Each key-value pair is designated by a colon between the two
- Values can be directly accessed by their keys
- Keys must be immutable Python types (lists cannot be used as keys)
- Keys are inherently unique

```
my_dict = {'a': 1, 'b': 2}
my_dict['c'] = 3      # adds to dictionary
print(my_dict['a'])   # prints 1
print(my_dict)        # {'a': 1, 'b': 2, 'c': 3}
```

# Classes

# What Is A Class?

A class is an in-memory structure which has defined data members and routines.

A class is **instantiated** to hold real data at run-time. A class instance being held in memory is commonly called an **object**.

Classes allow the client to use complex data management patterns with relative ease. They are an indispensable tool for developers and users.

We will talk more about classes in the next sessions.





# Development Tools

# Scripts are Neat...

Python code does not get compiled into a machine-readable executable; instead, the source code gets **interpreted** at runtime. This means the code's instructions are executed *directly and freely*, and syntax or logical errors are detected at runtime.

Like most code we write, the source code can live in a file.

Scripts are invoked and interpreted by typing `python3 my_script.py` in the console.



## ...and REPL is Neater!

- **Read Eval Print Loop**
- It allows for fast prototyping and testing ideas very quickly
- You can write anything in REPL that you would in a script
- It also has help modules!
  - In REPL, type `help(datatype)` to see documentation for that datatype
- Invoke by simply running `python3` in the console (no filenames)



# A Quick Demo

REPL & Python Data Structures



---





# Problems

# FizzBuzz

We begin with a classic interview problem.

For each number 1 through 100, print the number, unless:

- The value is a multiple of 3; print “Fizz” instead
- The value is a multiple of 5; print “Buzz” instead
- The value is a multiple of 3 and 5; print “FizzBuzz” instead

Write the Python code to do this.



# FizzBuzz Solution

```
def fizzbuzz():  
    for i in range(1, 101):  
        s = ''  
        if i % 3 == 0:  
            s += 'Fizz'  
        if i % 5 == 0:  
            s += 'Buzz'  
        if not s:  
            s = i  
        print(s)
```

```
fizzbuzz()
```



# Fibonacci

The Fibonacci sequence uses the sum of 2 most recent values to determine the next value, and begins with 0 and 1.

The first 10 values are: 0 1 1 2 3 5 8 13 21 34

Implement a function which calculates and prints the Fibonacci sequence up to arbitrary maximum value.

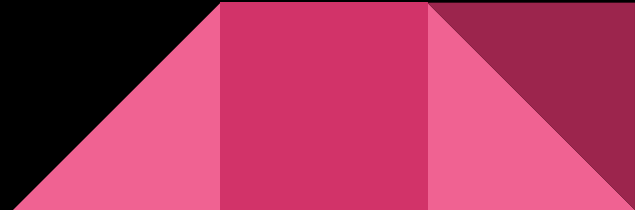


# Fibonacci Solution

```
def fibonacci(max_val):  
    a = 0  
    b = 1  
    while a < max_val:  
        print(a)  
        a, b = b, a + b
```

```
fibonacci(10)    # prints 0 1 1 2 3 5 8
```

```
fibonacci(40)    # prints 0 1 1 2 3 5 8 13 21 34
```



# Fibonacci, Continued

*Implement a function which calculates and prints the Fibonacci sequence up to arbitrary maximum value.*

Can you do it iteratively and recursively\*? Why or why not?

What if you had to calculate the first  $n$  terms of the sequence?

What if you had to calculate the  $n$ th term?

\*Note: we will discuss recursion in great depth in other sessions.



# Fibonacci, Continued

*Can you do it iteratively and recursively? Why or why not?*

An iterative solution is an easy implementation; a recursive solution, on the other hand, is difficult when we are asked to calculate the sequence up to a maximum value.

A recursive approach would **work backwards** based on the input given; we have no way of knowing whether the maximum value is a valid member of the sequence, or determining the previous values in the sequence.

It makes more sense to solve this problem iteratively.



# Fibonacci, Continued

*What if you had to calculate the first  $n$  terms of the sequence?*

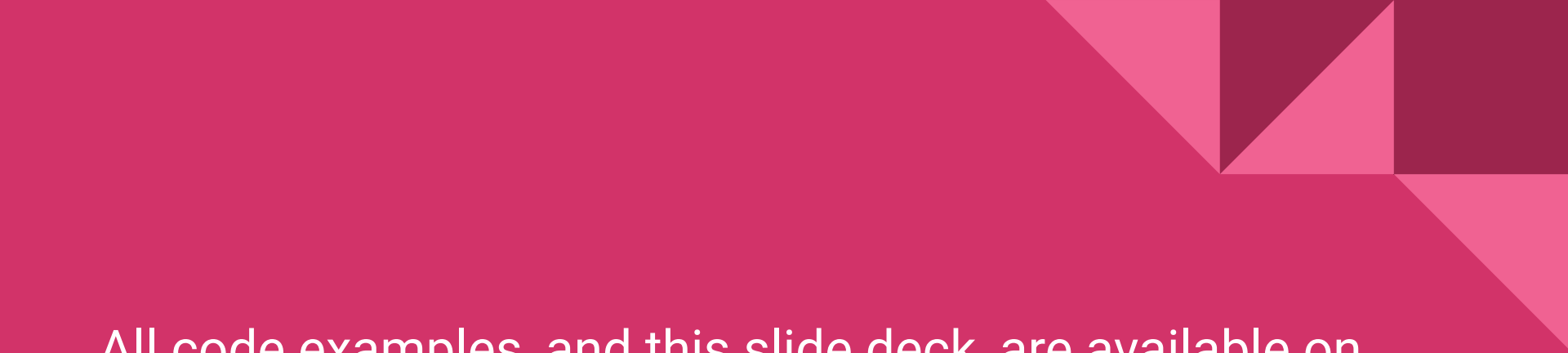
We can alter the iterative solution to stop after a specified number of iterations instead of comparing the most recently calculated value in the sequence. We can also implement a recursive solution.

*What if you had to calculate the  $n$ th term?*

Finding the  $n$ th term is a problem variation which has a very elegant recursive solution as well as an iterative solution.







All code examples, and this slide deck, are available on  
GitHub:

<https://github.com/junior-dev-struggle-bus/interviews>