6AMB Telegram Bot Workshop

Who am I? (Again!)

- I'm CPL Siddhartha Bose from 62FMD
- Received my bachelor degrees in Computer Science and Economics in 2019 from UCLA
- Never coded before 2015
- Failed my first programming course and wanted to quit
- At the very least my journey should show you that programming may seem daunting but it's a level playing field, and anyone can succeed with time

Workshop Objectives

- By the end of today we will have created 2 telegram bots!
- The goal is to get more comfortable with the basics of python while understanding the products we can create with the language
- We'll go through a recap of the introduction course before creating a dummy bot and then hopefully we can create a bot which can be more useful in your lives

Recap Syllabus

1) Installing Python and using the terminal
2) Recap of Intro Workshop
    1) Python variables
    2) Python data types
    3) Logical operators
    4) Strings
    5) Booleans
    6) Lists, Sets, Tuples and Dictionaries
    7) Conditional Statements and Loops
    8) Functions

How to Download Python

- Unlike the first course, today we will have to download python!
- The steps are pretty simple but need to be followed precisely for your given operating system

Downloading Python contd

1. Visit https://www.python.org/downloads/
2. Download and install Python 3.9.7 for your given operating system

Windows:

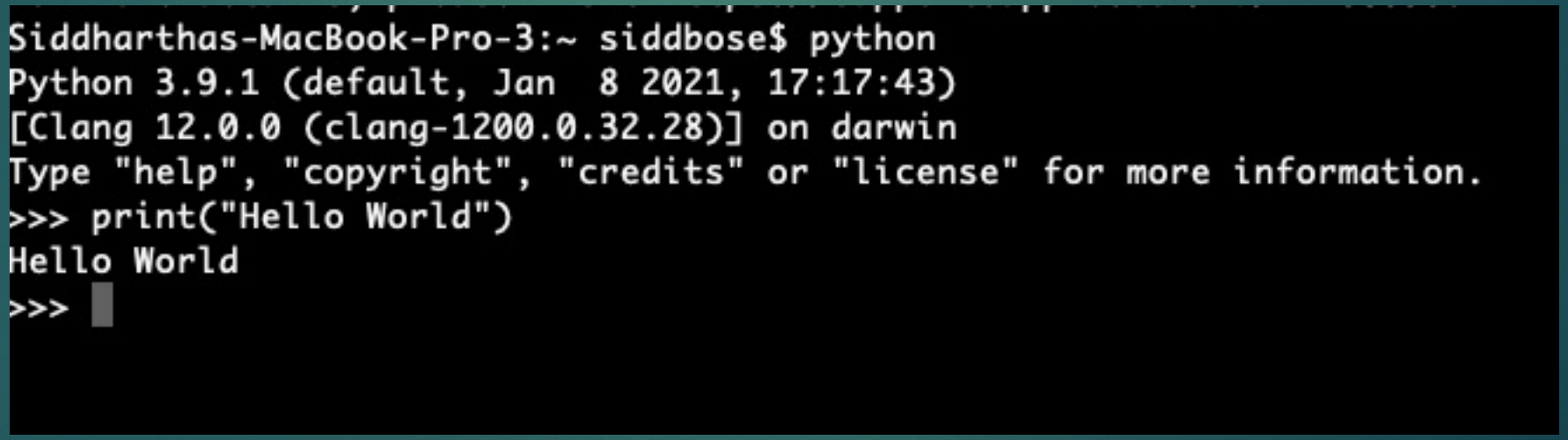
Click the option that asks if you want to add Python to path

Mac:

1. If you are using a Mac, open up Terminal using finder
2. type in 'sudo nano /etc/paths'
3. type in your password
4. go to the bottom of the file using arrow keys
5. Add the line '/usr/local/bin/python3'
6. Press control-x and then 'Y' to exit

Hello World

- So, to start we will create a “hello world” program in python
- Type in “python” into your CLI
- Then type in print(“hello world”)



Downloading Visual Studio Code

- In order to write our code, today we will be using program called Visual Studio Code (created by Microsoft)
- Visit this link to download the appropriate version for your computer
  - https://code.visualstudio.com/download
- Once downloaded let's install the python extension using the extensions tab on the left hand side
- Lastly, let's open up a terminal and get started

Understanding CLI

- Command Line Interface is essentially a window into the heart of your computer. It allows you to manipulate files and directories through simple text.
  - Some useful commands include:
    - cd – change directory: you can move into and out of folders like this
      - cd Documents: this will move you into your documents folder
    - mkdir – make directory: you can create folders like this
    - touch [filename] – you can create files like this
    - python3 file.py - you can run python files like this

Python Introduction Recap

- Now that we have downloaded Python let's do a recap of the last workshop
- This recap should take the majority of the morning session, followed by the bot creation in the afternoon
- Let's start with variables
  - How to use them
  - How to name them
  - How to output them

Variables Recap

- Variables are the building block for python
  - You can use them to store and manipulate data
- Since Python is “dynamically typed” we can initialize variables without explicitly telling the program what type they are

```
x = 5
y = "hello world"
z = 3.758
```

- As show above, x is an integer, y is a string and z is a float

Python Data types

- We have seen a few data types but let's go through all of the important ones here

1. Text: string ("hi")
2. Numeric: int (5), or float (5.0)
3. Sequences:
    1. List: [1,2,3]
    2. Tuple: (1,2,3)
4. Mapping:
    1. Dictionary: {"name": "Sidd Bose", "rank": "CPL"}
5. Boolean:
    1. True, False (capitalization matters here!)

Casting Variables

- Although variables are dynamically typed in Python, we also have the concept of casting
- Using casting we can ”tell” the program what type we want a variable to be

```
x = int(5) #x will be 5
y = str(5) #y will be "5"
z = float(5) #z will be 5.0
```

- As shown above, 5 can be represented in different ways if we cast it

Naming Variables

- Naming variables is relatively simple in python but we must follow a few rules:
  - The variable must start with a letter or underscore
  - The variable can only contain alphanumeric characters and underscores (A-z, 0-9, and _)
  - Capital letters and lowercase letters refer to different variable

Inputting and Outputting Variables

- Outputting variables is simple using the print() command

```
a = 1
b = 2

print (a + b) #this prints out 3
```

- Similarly, inputting variables uses the input(prompt) command as follows, where prompt is the user prompt for input

```
val = input("what is your age? ")

print("age: " + str(val))
```

Logical, Arithmetic and Comparison Operator Recap

- Having variables is one part of the battle, but often we want to compare, contrast or compute!
- In order to do so we can leverage different operators
  - Logical: Checking if two or more statements is true
  - Arithmetic: Using variables to do math
  - Comparison: Comparing the magnitude or equality of variables

Logical Operators

- These operators show how python leverages English
- **and**, for example x < 5 and x < 10
  - Returns True if both statements are True, False otherwise
- **or**, for example x < 5 or x < 4
  - Returns True if either statement is True, False if both are False
- **not**, for example not(x < 5)
  - Returns True if the inside statement is False, True otherwise

Arithmetic Operators

- Most of these should be very intuitive for all of us
- These operators work between two numeric values
- Addition (+), for example x+y
- Subtraction (-), for example x-y
- Multiplication (*), for example x*y
- Division (/), for example x/y
- Modulus (%), for example x % y (where x % y gives the remainder)
- Exponentiation (**) for example x**y (which is x^y)

Comparison Operators

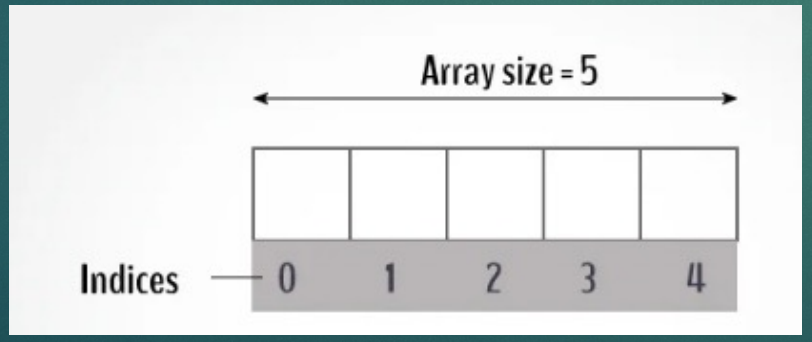
- These are also intuitive, but the syntax is key
- **Equal (==)**, for example 5 == 5 is True
- **Not equal (!=)**, for example 5 != 5 is False
- **Greater than (>)**, for example 8 > 7 is True
- **Greater than or equal to (>=)**, for example 6 >= 7 is False
- **Lesser than (<)**, for example 7 < 8 is True
- **Lesser than or equal to (<=)**, for example 7 <= 6 is False

- In each case, the result of this operator will tell you if the comparison is True or False
  - For example, x = 2, y = 3, then x == y returns False and y >= x returns True

Strings Recap

- As we know, a string is a word or character encapsulated in quotations: “var” like this
- Since a string is an array of characters, we can access elements by indexing as shown below

- For example, if x = “hello”, then x[0] is “h”



String Methods

- The most important string methods are as follows (using x = “abc” and y = “defg” as examples):
  - len(x)-> will return the length of x, or 3
  - x[1:3] -> will return “bc” as a sliced string (format is [first index: last index +1])
  - x+y-> will return “abcdefg”
- For more string methods, visit:
  - https://www.w3schools.com/python/python_strings_methods.asp

F Strings

- In Python F Strings allow us to insert variables into a string
- The syntax is as follows: f"string {var} string"

```
age = 35
depot = 62

fstring = f"Soldier x is {age} years old from {depot}FMD"
```

- This allows us to concatenate a string more naturally

Booleans and how to use them

- If you recall, Booleans are simple True and False values
- When we use logical and comparison operators, Booleans are the result
  - You can evaluate statements and have them print out as True or False

```
print(2 < 5) #prints True
print(5 != 5) #prints False since 5 is actually equal to 5!
print(125 % 5 == 0) #prints True since 125/5 has no remainder
```

Lists, Sets and Tuples

- Lists, sets and tuples are all ways to hold multiples pieces of data at the same time
- The table below explains the difference between the 3 data types:
  - Mutable = changeable
  - Ordered = objects stay in a certain order
  - Indexing = you can access a value like in a string
  - Duplicates = you can have multiple of the same value
- Examples:
  - List: [1,1,2,3,4], we can add and remove values
  - Set: {1,2,3,4} we can add and remove values but we can't access via indexing
  - Tuple: (1,2,3,4,4), we cannot add or remove values but we can index

	Mutable	Ordered	Indexing / Slicing	Duplicate Elements
List	✓	✓	✓	✓
Tuple	✗	✓	✓	✓
Set	✓	✗	✗	✗

Dictionaries Recap

- Perhaps the most powerful data type mentioned thus far, dictionaries use ‘keys’ to index into them
- For example, a dictionary could be instantiated as follows:
- This is an example of a nested dictionary
- Essentially the dictionary ”soldiers” contains:
  - Dictionaries for each soldier and those contain:
    - Name, age, rank
- Soldier1’s rank can be obtained as such
  - soldiers[“soldier1”][“Rank”]

```
soldiers = {
    "soldier1":{
        "Name": "Sidd Bose",
        "Age": 24,
        "Rank": "LCP"
    },
    "soldier2":{
        "Name": "Malcolm Hoo",
        "Age": 40,
        "Rank": "ME2"
    }

}
```

Conditional Statements Recap

- If you recall, conditional statements use logical operators to check certain conditions and execute different blocks of code
- The format is:
  - If, elif ….., else
- As we see on the right, if a condition is met, a certain block of code is executed
- If no conditions are met, only the else block is executed
- We need to ensure that indentation is correct in if else blocks otherwise python will not run

```
fitnessTestScore = 0
fitnessTestValue = ""
fitnessTestCash = 0
fitnessTestOffs = 0

if fitnessTestScore >= 85:
    fitnessTestValue = "gold"
    fitnessTestCash = 300
    fitnessTestOffs = 3
elif fitnessTestScore < 85 and fitnessTestScore >= 75:
    fitnessTestValue = "silver"
    fitnessTestCash = 200
    fitnessTestOffs = 2
elif fitnessTestScore < 75 and fitnessTestScore >= 60:
    fitnessTestValue = "pass"
    fitnessTestCash = 0
    fitnessTestOffs = 1
else:
    fitnessTestValue = "FAIL"
    fitnessTestCash = 0
    fitnessTestOffs = 0
```

Loops: While and For

- Looping is the best way to run a certain piece of code repeatedly
- In python the two most common forms of loops are while and for loops
- For the most part, both can be used interchangeably and are up to the programmer to choose which one suits them or the task best

While Loops

- Looping is critical in control flow because it allows us to do the same set of tasks repeatedly without having to explicitly write them out
- A while loop is much like it sounds
  - While a certain condition is met do a job
  - For example, in English we might say:
    - While you are in school, study hard and get good grades
- The following is an example of this sentiment

```
studentAge = 5
graduationAge = 18
def studyHard():pass

def getGoodGrades(): pass

while studentAge <= graduationAge:
    studyHard()
    getGoodGrades()
    studentAge = studentAge + 1
```

For Loops

- For loops are very similar to while loops but in python they allow you to write code in a more concise manner
- For example, if we want to print out the list from the previous section, we would write it as such

```
x = [1, 2, 3]

for elements in x:
    print(elements)
```

- You can obviously also use for loops just like while loops where you manually iterate (the range function iterates through values for you):

```
x = [1, 2, 3]

for i in range(3):
    print(x[i])
```

Functions

- The last part of today's recap is functions
- Functions allow for a task, or a block of code to be encapsulated
- This code is only run when it is "called"
- Functions can take inputs, usually as "parameters" and can output either as print statements or as return values as we mentioned earlier

Functions Contd.

- In order to declare a function the syntax is as follows:

```
def functionName(parameter1, parameter2):
    #code
    #code
    #code
    return ValueError


functionName(3, 5)
```

- A function can have zero or more arguments/parameters
- This information can then be used or manipulated and then returned
- In order to ”call” a function, we simply type the name of the function with the relevant parameters as shown above

Recap Over!

- Okay, recap was a lot of information but it will be crucial for the next part of the day
- For the second half of the workshop we will first work on first creating a simple bot that we can interact with
- Hopefully by the end you will have the tools to use code from our last workshop to build a slightly more useful bot

Classes

- An important concept in "object-oriented programming" is classes
- Classes essentially define an object and its characteristics
- For example: A human has a name, age and gender
- In a class we can write it as follows:

```
class Human:
    def __init__(self, name):
        self.name = name
        self.age = 0
        self.gender = ""
```

- Def __init__ is a function called a constructor. It will create a new object

Classed Contd

- In order to create a new class using the def init function we do the following

```
class Human:
    def __init__(self, name):
        self.name = name
        self.age = 0
        self.gender = ""

sidd = Human("Sidd")
```

- Since the init function requires a name variable to set the name characteristic, Human(“Sidd”) is the way to create the class

Creating a Telegram Bot

- Telegram is an app most of us are familiar with
- Like whatsapp, it is a free instant messaging platform but it has the unique benefit of allowing customized “bots” to run on the application
- These bots are lightweight applications that have quite a range of use-cases

Telegram Bot Contents

1. Importing telegram and telegram.ext
2. Getting the API Key
3. Understanding the parts of the bot
    1. MessageHandler, CommandHandler, ConversationHandler, Keyboards
4. Interacting with our first bot
5. Creating a dummy bot

Goal of Dummy Bot

- The goal of our dummy bot is to take our name, age and gender as input and output the stored information
- We will use all the techniques that will be useful to create more interesting and complicated bots
- Let's begin!

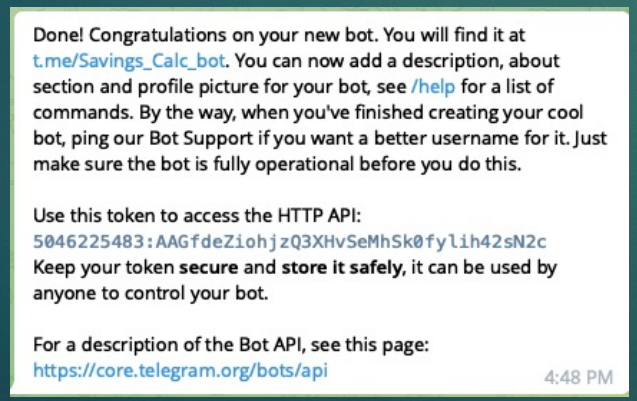
Installing and Importing Libraries

- In most python projects, the first step will be to install and import all relevant libraries
- To install a library we can use the 'pip' command in our terminal
  - pip3 install python-telegram-bot
- Now let's create a new python file
  - touch dummy.py
  - Open the file
- Now to import the libraries we need, type the following:

```
import telegram
from telegram.ext import Updater, ConversationHandler, CommandHandler, MessageHandler
```

Getting the API key from Telegram

- This part is less programming, more logistics
- On telegram, we can access the BotFather account
- By using the /newbot command, we can follow a couple of steps and get a new bot API Key
- An API key is essentially a way to access and identify our bot
- The final message looks something like this



Message Handler

- Before we get into the crucial stuff, we need to talk about message handlers
- Essentially these are objects that allow us to:
  - 1. choose what kind of information we want our functions to accept (images, locations, text)
  - 2. Define which function will trigger based on a given input
- A Message handler looks like this:

```
MessageHandler(Filters.text, function_name)
```

Command Handler

- Much like the message handler, the command handler allows us to create functions that trigger when a telegram command is used
- Commands in telegram are usually in the /command format
- To create a command handler we can do the following:

```
CommandHandler('start', function_name)
```

  - Where ‘start’ is the command, and function_name is the function

Conversation Handler

- To build the bot we need to understand one fundamental concept: the conversation handler

- Essentially, it is an object (like the class we created earlier) that holds the following information
  - Entry points: how we can access the bot (CommandHandler)
  - States: steps within a bot's functions (MessageHandler)
  - Fallbacks: what to do when things go wrong (CommandHandler)

Conversation Handler Contd

- To create a command handler is a bit specific, so let's look at the general format

```
ConversationHandler(
        entry_points=[list of command handlers],
        states={
                Dictionary of steps and message Handlers
                Step: MessageHandler
        },
        fallbacks=[List of command handlers],
        )
```

Conversation Handler Contd

- More specifically we can have the following:

```
FIRSTSTEP, SECONDSTEP, THIRDSTEP = range(3)

ConversationHandler(
        entry_points=[CommandHandler('start', start),CommandHandler('help', help)],
        states={
                FIRSTSTEP: [MessageHandler(Filters.text, function_name)],
                SECONDSTEP: [MessageHandler(Filters.text, function_name1)],
                THIRDSTEP: [MessageHandler(Filters.text, function_name2)],
        },
        fallbacks=[CommandHandler('cancel', end)],
        )
```

- Where FIRSTEP, SECONDSTEP and THIRDSTEP are simply mapped to 0,1,2 using the range function

Human Class

- So let's start by creating a Human class
- We need the human class to have:
  - Name
  - Gender
  - Age
- So we define it as shown in the class slide

```
class Human:
    def __init__(self, name):
        self.name = name
        self.age = 0
        self.gender = ""
```

Humans dictionary

- The method I like to use to hold information in telegram bots is a dictionary
- As we learned, we can index into a dictionary however we want and in this case every user has a unique identifier (their chat id)
- Therefore, if we create a dictionary, we can use chat_id to store information for every user

```
humans = {}
```

- The dictionary can be created as above

Setting Up the API Key

- Before we write our functions, we need to let Telegram know what code to run using our API key
- We do this by setting up an "Updater" as follows

```
# The API Key we received for our bot
API_KEY = '5062654174:AAEg0b-JUF46drA87A4jnnmRohav0LyUBYA'
updater = Updater(API_KEY)
dispatcher = updater.dispatcher
```

- Again, ensure that the API key is what you got from Telegram

Creating Our First Functions

- In order to start using the bot we must have at least 2 functions.
  - Start
  - Main
- The start function will allow us to have some functionality and the main function will let us actually run the code!

Writing the Start Function

- For each ‘state’ we will initialize functions as follows:
  - Def function_name(update_obj, context)
  - Update_obj gives us message meta data (text, chat id etc) and context is the state of the bot at any time

```
# The entry function
def start(update_obj, context):
    update_obj.message.reply_text("Hello there, welcome to the bot")
```

- Although we can’t run it yet, here we see an important function
  - Update_obj.message.reply_text(“text”) will send a message to the user!

Writing the main function

- The main function is essentially the engine of this project. It takes the Conversation handler and allows us to run the bot!
- In most python projects you will notice this main() and if __name__ format. Essentially it allows us to run the main function even if we have many files
- In this function we see:
  - ConversationHandler
  - Dispatcher.add_handler
  - Updater.start_polling
  - Updater.idle

```
def main():

    handler = ConversationHandler(
        entry_points=[CommandHandler('start', start)],
        states={
        },
        fallbacks=[],
        )
    # add the handler to the dispatcher
    dispatcher.add_handler(handler)

    updater.start_polling()

    updater.idle()


if __name__ == '__main__':
    main()
```

Breaking Down the Main Function

- We've spoken about the ConversationHandler already but let's look at the other couple of lines
- Dispatcher.add_handler:
  - This adds our conversation handler to the bot Ensures that when we run the bot all the relevant functions are present
- Updater.start_polling():
  - Polling is a way to access the bot via the internet
- Updater.idle():
  - Tells the bot to basically wait for input

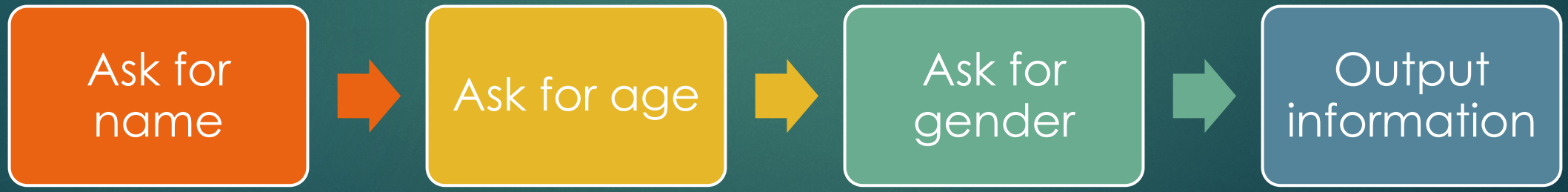
```
def main():

    handler = ConversationHandler(
        entry_points=[CommandHandler('start', start)],
        states={
        },
        fallbacks=[],
        )
    # add the handler to the dispatcher
    dispatcher.add_handler(handler)

    updater.start_polling()

    updater.idle()


if __name__ == '__main__':
    main()
```

Running the bot

- Let's run the bot now
- If we type in "python3 dummy.py" and click enter the bot should start
- You can then go on telegram desktop or the application and type in /start
- This should lead to the bot replying!

Now What?

- This process that we just went through is important in every project; creating a prototype
- Essentially it is the smallest version of the project that we can say "does something"
- Now that we have a function, let's make a few more that correspond to the following flowchart!



How do we move between states?

- This is where the concept of a return value and the ConversationHandler come into play
- For every state we want to move between, we need to add a message handler to the conversation handler
- A habit programmers have here is to assign each state a name and a number to access later on

```
FIRSTSTEP, SECONDSTEP, THIRDSTEP = range(3)
```

- This line is the same as

```
FIRSTSTEP = 0
SECONDSTEP = 1
THIRDSTEP = 2
```

Adding a second function

- Now let's add a second function that echoes our message
- As we mentioned before in our start function slide, we can use update.message.reply_text() to reply to the user
- In addition, we can use update.message.text to grab the information that the user has just typed in

```
def name_step(update_obj, context):
    msg = update_obj.message.text
    update_obj.message.reply_text("you just sent me: " + msg)
```

Moving from start to the second step

- Now that we have two functions we need to update the first function and the ConversationHandler
- We add a state and message handler to ConversationHandler as follows:

```
handler = ConversationHandler(
    entry_points=[CommandHandler('start', start)],
    states={
        FIRSTSTEP: [MessageHandler(Filters.text, name_step)]
    },
    fallbacks=[],
    )
```

- Then we add the line “return FIRSTSTEP” at the end of the start function

Third Function

- Let's finish the functionality by adding a third function that again simply echoes the text received!
- 1. Create the function

```
def gender_step(update_obj, context):
    msg = update_obj.message.text
    update_obj.message.reply_text("you just sent me: " + msg)
```

- 2. Create a state for the function

```
SECONDSTEP = 0
```

- 3. Add the state to the ConversationHandler

```
handler = ConversationHandler(
    entry_points=[CommandHandler('start', start)],
    states={
        FIRSTSTEP: [MessageHandler(Filters.text, name_step)],
        SECONDSTEP: [MessageHandler(Filters.text, gender_step)],

    },
    fallbacks=[],
    )
```

- 4. Add "return SECONDSTEP" to the name_step function

Taking stock of where we are

- So now we have 3 functions, they don't do much but they interact with the user
- Let's start asking questions we need answers to and storing information

Storing Information

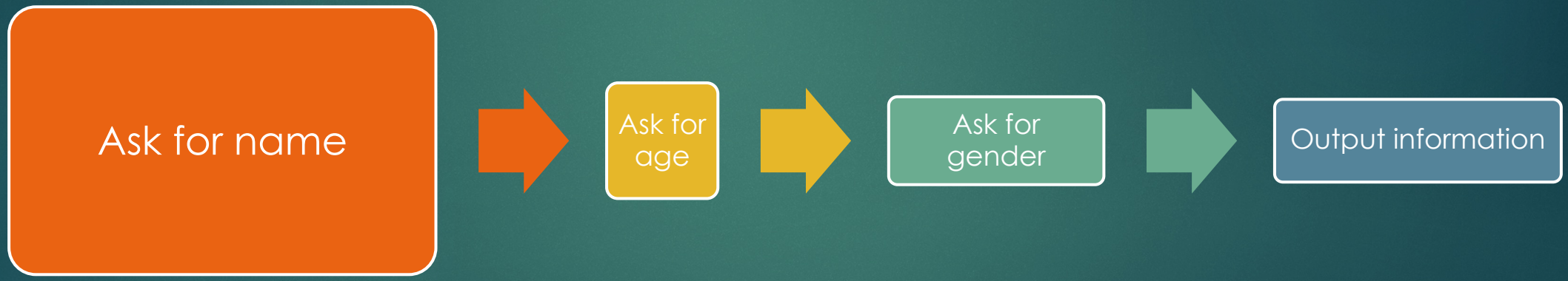
- Storing information is essential to using a bot, and this means we need data structures
- We mentioned two specific data structures earlier: classes and dictionaries
  - class Human:
  - humans = {}

Format of each function

- The common format for a function in a Telegram bot is:
  - Store the information we received from the previous step (if not first step)
  - Manipulate information if needed
  - Set up for next function
  - Reply to user
- As such let's redo the start function

Redoing Start Function

- Now that we have the basic format down, let's follow the flowchart and redo the start function
- First we need to ask the user for their name



- All we have to do is change the prompt!

```
# The entry function
def start(update_obj, context):
    update_obj.message.reply_text("Hello there, what is your name?")
    return FIRSTSTEP
```

Redoing the second function

- Let's follow the format more closely now
- Store:
  - We need to store the information (name) into our class and dictionary
- Manipulate:
  - There isn't any manipulation being done here
- Set up:
  - We don't need to set up for the next function either
- Reply:
  - We can now reply and ask for age!

Coding out the second function

- Store:
  - Here we grab the message text, create a Human object, store the object in the dictionary

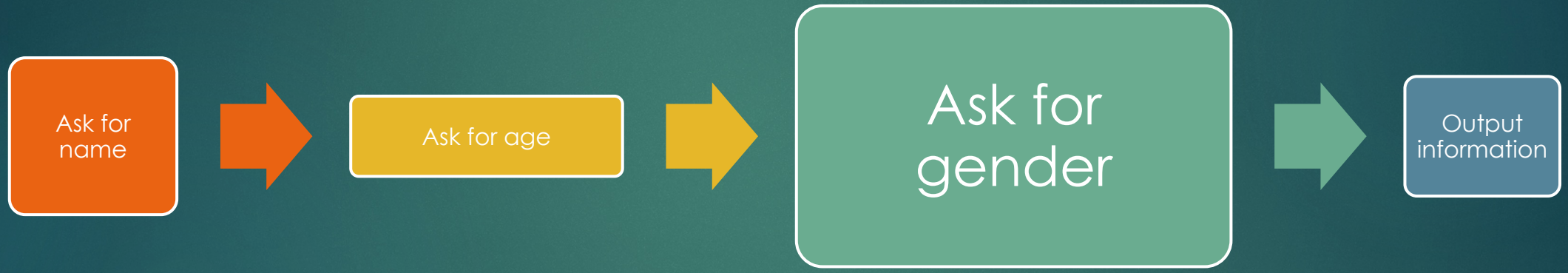
```
chat_id = update_obj.message.chat_id
msg = update_obj.message.text
humans[chat_id] = Human(msg)
```

- Manipulate: N/A
- Set Up: N/A
- Reply:

```
update_obj.message.reply_text("How old are you?")
return SECONDSTEP
```

Post Second Function

- Now that the second function is written, we have the third prompt to send to the user



- Here, since we have two genders we will use a Keyboard

Keyboards

- For those of us who have used Telegram, we are probably familiar with Keyboards
- Essentially they replace the QWERTY keyboard we are used to with custom buttons
- Each keyboard is a List of Lists!
- The format is as follows:

```
list1 = [[telegram.KeyboardButton(text="One")],[telegram.KeyboardButton(text="Two")]]
```

- Creating a keyboard involves a few steps so let's go through them one by one

Keyboards Contd

- Steps are as follows:
  - Create the list of buttons

```
list1 = [[telegram.KeyboardButton(text="Male")],[telegram.KeyboardButton(text="Female")]]
```

  - Create a keyboard markup

```
kb = telegram.ReplyKeyboardMarkup(keyboard=list1,resize_keyboard = True, one_time_keyboard = True)
```

  - Add the keyboard to the reply message

```
update_obj.message.reply_text("What is your gender?",reply_markup=kb)
```

Creating the third function

- Store:
  - Like the second function we need to store the information from the previous function
- Manipulate: N/A
- Set Up:
  - This is where we create the keyboard
- Reply:
  - Reply with the new keyboard

Coding up the third function

- Store:

```
chat_id = update_obj.message.chat_id
msg = update_obj.message.text
humans[chat_id].age = msg
```

- Manipulate: N/A
- Set up:

```
list1 = [[telegram.KeyboardButton(text="Male")],[telegram.KeyboardButton(text="Female")]]

kb = telegram.ReplyKeyboardMarkup(keyboard=list1,resize_keyboard = True, one_time_keyboard = True)
```

- Reply and return:

```
update_obj.message.reply_text("What is your gender?",reply_markup=kb)
return THIRDSTEP
```

The End Function

- Now that we have created our last function that prompts the user we need 1 more function to do the following
  - 1. Output the information
  - 2. End the Conversation

- First things first let's add a final step to the ConversationHandler

```
THIRDSTEP:[MessageHandler(Filters.text, end)]
```

The End Function Contd

- All we have to do now is access and output the information we have gathered
- Since all the information is held in the dictionary as a class, we can simply access the dictionary and then reply

```
def end(update_obj, context):

    chat_id = update_obj.message.chat_id
    msg = update_obj.message.text
    humans[chat_id].gender = msg

    update_obj.message.reply_text(
        f"Thank you {humans[chat_id].name}, you are a {humans[chat_id].gender} and {humans[chat_id].age} years
    )
```

- The last thing we have to do is add a return statement that ends the conversation

```
return ConversationHandler.END
```

Are We done?

- Yes and no
- So when it comes to applications where users are inputting information, we always want a backup plan
- This is where try/except blocks come into play
- Essentially at every function we want to “try” the code and if something unexpected happens we have to handle it in a predefined way
  - As I’ve mentioned before, there is nothing worse than undefined behavior!

Except Function

- We will just create a quick function called “Error” which will trigger if something goes wrong.
- All this function will do is reply “there was a problem, press /start to start again!” and then end the conversation
- In more complex fallback functions we can tell the user the actual issue but for now this is good enough

Coding the Except function

```
def error(update_obj, context):
    update_obj.message.reply_text("There was an error. Click /start to start again!")
    return ConversationHandler.END
```

How to use try/except blocks

- For every function we will put a ”try:” as the first line, and then after the code an except line
- For example here is the start function

```
# The entry function
def start(update_obj, context):
    try:
        update_obj.message.reply_text("Hello there, what is your name?")
        return FIRSTSTEP
    except Exception:
        error(update_obj, context)
```

Dummy Bot Complete!

- This brings us to the end of our second workshop and this time you all have homework (which you hopefully want to do!)
- I will give you all a copy of the Savings Calculator code from the first workshop and a format for how the bot should flow
  - https://github.com/siddbose97/pythonWorkshop/blob/master/savings_calculator.py
- You can find the code for my dummy bot at the following link as well (the API key is removed, so input your own!)
  - https://github.com/siddbose97/Telegram_Workshop/blob/master/sandbox.py

Savings Calculator Bot Format

Conclusion

- Today's workshop was much tougher than the first and I'll be the first to admit that applying programming knowledge is never easy
- Although Google is always your best friend on this I'll be available as well if you have questions so please reach out to me via Whatsapp or email at siddbose97@gmail.com!