Introduction to Python Statements

In this lecture we will be doing a quick overview of Python Statements. This lecture will emphasize differences between Python and other languages such as C++

There are two reasons we take this approach for learning the context of Python Statements:

- *1. If you are coming from a different language this will rapidly accelerate your understanding of Python.*
- *2. Learning about statements will allow you to be able to read other languages more easily in the future.*

Python vs Other Languages

Let's create a simple statement that says: "If a is greater than b, assign 2 to a and 4 to b"

Take a look at these two if statements (we will learn about building out if statements soon).

Version 1 (Other Languages)

```
if (a>b){
    a = 2;
    b = 4;
}
```

Version 2 (Python)

```
if a>b:
    a = 2
    b = 4
```

You'll notice that Python is less cluttered and much more readable than the first version. How does Python manage this?

Let's walk through the main differences:

_Python gets rid of () and {} by incorporating two main factors: a colon and whitespace. The statement is ended with a *colon, and whitespace is used (indentation) to describe what takes place in case of the statement.*

Another major difference is the lack of semicolons in Python. Semicolons are used to denote statement endings in many other languages, but in Python, the end of a line is the same as the end of a statement.

Lastly, to end this brief overview of differences, let's take a closer look at indentation syntax in Python vs other languages:

Indentation

Here is some pseudo-code to indicate the use of whitespace and indentation in Python:

Other Languages

```
if (x)
    if(y)
        code-statement;
else
    another-code-statement;
```

Python

```
if x:
    if y:
        code-statement
else:
    another-code-statement
```

Note how Python is so heavily driven by code indentation and whitespace. This means that code readability is a core part of the design of the Python language.

Now let's start diving deeper by coding these sort of statements in Python!

`Time to code!`

if, elif, else Statements

`if` Statements in Python allows us to tell the computer to perform alternative actions based on a certain set of results.

Verbally, we can imagine we are telling the computer:

"Hey if this case happens, perform some action"

We can then expand the idea further with `elif` and `else` statements, which allow us to tell the computer:

_"Hey if this case happens, perform some action. Else, if another case happens, perform some other action. Else, if none of *the above cases happened, perform this action."*

Let's go ahead and look at the syntax format for `if` statements to get a better idea of this:

```
if case1:
        perform action1
    elif case2:
        perform action2
    else:
        perform action3
```

First Example

Let's see a quick example of this:

In [2]:
```
if True:
    print('It was true!')
```

```
It was true!
```

Let's add in some else logic:

In [3]:
```
x = False

if x:
    print('x was True!')
else:
    print('I will be printed in any case where x is not true')
```

```
I will be printed in any case where x is not true
```

Multiple Branches

Let's get a fuller picture of how far `if`, `elif`, and `else` can take us!

We write this out in a nested structure. Take note of how the `if`, `elif`, and `else` line up in the code. This can help you see what `if` is related to what `elif` or `else` statements.

We'll reintroduce a comparison syntax for Python.

In [4]:
```
loc = 'Bank'

if loc == 'Auto Shop':
    print('Welcome to the Auto Shop!')
elif loc == 'Bank':
    print('Welcome to the bank!')
else:
    print('Where are you?')
```

```
Welcome to the bank!
```

Note how the nested `if` statements are each checked until a True boolean causes the nested code below it to run. You should also note that you can put in as many `elif` statements as you want before you close off with an `else`.

Let's create two more simple examples for the `if`, `elif`, and `else` statements:

In [6]:
```
person = 'Prasad'

if person == 'Prasad':
    print('Welcome Prasad!')
else:
    print("Welcome, what's your name?")
```

```
Welcome Prasad!
```

In [7]:
```
person = 'Matplotlib'

if person == 'Seaborn':
    print('Welcome Seaborn!')
elif person =='Matplotlib':
    print('Welcome Matplotlib!')
else:
    print("Welcome, what's your name?")
```

```
Welcome Matplotlib!
```

Indentation

It is important to keep a good understanding of how indentation works in Python to maintain the structure and order of your code. We will touch on this topic again when we start building out functions!

for Loops

A `for` loop acts as an iterator in Python; it goes through items that are in a sequence or any other iterable item. Objects that we've learned about that we can iterate over include strings, lists, tuples, and even built-in iterables for dictionaries, such as keys or values.

We've already seen the `for` statement a little bit in past lectures but now let's formalize our understanding.

Here's the general format for a `for` loop in Python:

```
for item in object:
    statements to do stuff
```

The variable name used for the item is completely up to the coder, so use your best judgment for choosing a name that makes sense and you will be able to understand when revisiting your code. This item name can then be referenced inside your loop, for example if you wanted to use `if` statements to perform checks.

Let's go ahead and work through several example of `for` loops using a variety of data object types. We'll start simple and build more complexity later on.

Example 1

Iterating through a list

```
In [8]:  # We'll learn how to automate this sort of list in the next lecture
         list1 = [1,2,3,4,5,6,7,8,9,10]
```

```
In [9]:  for num in list1:
             print(num)
```

```
1
2
3
4
5
6
7
8
9
10
```

Great! Hopefully this makes sense. Now let's add an `if` statement to check for even numbers. We'll first introduce a new concept here--the modulo.

Modulo

The modulo allows us to get the remainder in a division and uses the % symbol. For example:

```
In [10]:  17 % 5
Out[10]:  2
```

This makes sense since 17 divided by 5 is 3 remainder 2. Let's see a few more quick examples:

```
In [1]:  # 3 Remainder 1
         10 % 3
Out[1]:  1
```

```
In [2]:  # 2 Remainder 4
         18 % 7
Out[2]:  4
```

```
In [3]:  # 2 no remainder
         4 % 2
Out[3]:  0
```

Notice that if a number is fully divisible with no remainder, the result of the modulo call is 0. We can use this to test for even numbers, since if a number modulo 2 is equal to 0, that means it is an *even number!*

Back to the `for` loops!

Example 2

Let's print only the even numbers from that list!

In [10]:
```
for num in list1:
    if num % 2 == 0:
        print(num)
```

We could have also put an `else` statement in there:

In [ ]:
```
for num in list1:
    if num % 2 == 0:
        print(num)
    else:
        print('Odd number')
```

Example 3

Another common idea during a `for` loop is keeping some sort of running tally during multiple loops. For example, let's create a `for` loop that sums up the list:

In [ ]:
```
# Start sum at zero
list_sum = 0

for num in list1:
    list_sum = list_sum + num

print(list_sum)
```

Great! Read over the above cell and make sure you understand fully what is going on. Also we could have implemented a `+=` to perform the addition towards the sum. For example:

In [ ]:
```
# Start sum at zero
list_sum = 0

for num in list1:
    list_sum += num

print(list_sum)
```

Example 4

We've used `for` loops with lists, how about with strings? Remember strings are a sequence so when we iterate through them we will be accessing each item in that string.

In [ ]:
```
for letter in 'This is a string.':
    print(letter)
```

Example 5

Let's now look at how a `for` loop can be used with a tuple:

In [12]:
```
tup = (1,2,3,4,5)

for t in tup:
    print(t)
```

```
1
2
3
4
5
```

Example 6

Tuples have a special quality when it comes to `for` loops. If you are iterating through a sequence that contains tuples, the item can actually be the tuple itself, this is an example of tuple unpacking. During the `for` loop we will be unpacking the tuple inside of a sequence and we can access the individual items inside that tuple!

In [13]:
```
list2 = [(2,4),(6,8),(10,12)]
```

In [14]:
```
for tup in list2:
    print(tup)
```

```
(2, 4)
(6, 8)
(10, 12)
```

In [15]:
```
# Now with unpacking!
for (t1,t2) in list2:
    print(t1)
```

```
2
6
10
```

Cool! With tuples in a sequence we can access the items inside of them through unpacking! The reason this is important is because many objects will deliver their iterables through tuples. Let's start exploring iterating through Dictionaries to explore this further!

Example 7

In [16]:
```
d = {'k1':1,'k2':2,'k3':3}
```

In [17]:
```
for item in d:
    print(item)
```

```
k1
k2
k3
```

Notice how this produces only the keys. So how can we get the values? Or both the keys and the values?

We're going to introduce three new Dictionary methods: `.keys()`, `.values()` and `.items()`

In Python each of these methods return a dictionary view object. It supports operations like membership test and iteration, but its contents are not independent of the original dictionary – it is only a view. Let's see it in action:

In [18]:
```
# Create a dictionary view object
d.items()
```

Out[18]:
```
dict_items([('k1', 1), ('k2', 2), ('k3', 3)])
```

Since the .items() method supports iteration, we can perform dictionary unpacking to separate keys and values just as we did in the previous examples.

In [19]:
```
# Dictionary unpacking
for k,v in d.items():
    print(k)
    print(v)
```

```
k1
1
k2
2
k3
3
```

If you want to obtain a true list of keys, values, or key/value tuples, you can cast the view as a list:

In [20]:
```
list(d.keys())
```

Out[20]:
```
['k1', 'k2', 'k3']
```

Remember that dictionaries are unordered, and that keys and values come back in arbitrary order. You can obtain a sorted list using sorted():

In [21]:
```
sorted(d.values())
```

Out[21]:
```
[1, 2, 3]
```

Conclusion

We've learned how to use for loops to iterate through tuples, lists, strings, and dictionaries. It will be an important tool for us, so make sure you know it well and understood the above examples.

while Loops

The `while` statement in Python is one of most general ways to perform iteration. A `while` statement will repeatedly execute a single statement or group of statements as long as the condition is true. The reason it is called a 'loop' is because the code statements are looped through over and over again until the condition is no longer met.

The general format of a while loop is:

```
while test:
    code statements
else:
    final code statements
```

Let’s look at a few simple `while` loops in action.

In [22]:
```
x = 0

while x < 10:
    print('x is currently: ', x)
    print(' x is still less than 10, adding 1 to x')
    x += 1
```

```
x is currently:  0
 x is still less than 10, adding 1 to x
x is currently:  1
 x is still less than 10, adding 1 to x
x is currently:  2
 x is still less than 10, adding 1 to x
x is currently:  3
 x is still less than 10, adding 1 to x
x is currently:  4
 x is still less than 10, adding 1 to x
x is currently:  5
 x is still less than 10, adding 1 to x
x is currently:  6
 x is still less than 10, adding 1 to x
x is currently:  7
 x is still less than 10, adding 1 to x
x is currently:  8
 x is still less than 10, adding 1 to x
x is currently:  9
 x is still less than 10, adding 1 to x
```

Notice how many times the print statements occurred and how the `while` loop kept going until the True condition was met, which occurred once x == 10. It's important to note that once this occurred the code stopped. Let's see how we could add an `else` statement:

In [23]:
```
x = 0

while x < 10:
    print('x is currently: ', x)
    print(' x is still less than 10, adding 1 to x')
    x += 1

else:
    print('All Done!')
```

```
x is currently:  0
 x is still less than 10, adding 1 to x
x is currently:  1
 x is still less than 10, adding 1 to x
x is currently:  2
 x is still less than 10, adding 1 to x
x is currently:  3
 x is still less than 10, adding 1 to x
x is currently:  4
 x is still less than 10, adding 1 to x
x is currently:  5
 x is still less than 10, adding 1 to x
x is currently:  6
 x is still less than 10, adding 1 to x
x is currently:  7
 x is still less than 10, adding 1 to x
x is currently:  8
 x is still less than 10, adding 1 to x
x is currently:  9
 x is still less than 10, adding 1 to x
All Done!
```

break, continue, pass

We can use `break`, `continue`, and `pass` statements in our loops to add additional functionality for various cases. The three statements are defined by:

break: Breaks out of the current closest enclosing loop. continue: Goes to the top of the closest enclosing loop. pass: Does nothing at all.

Thinking about `break` and `continue` statements, the general format of the `while` loop looks like this:

```
while test:
    code statement
    if test:
        break
    if test:
        continue
else:
```

`break` and `continue` statements can appear anywhere inside the loop's body, but we will usually put them further nested in conjunction with an `if` statement to perform an action based on some condition.

Let's go ahead and look at some examples!

In [24]:
```python
x = 0

while x < 10:
    print('x is currently: ',x)
    print(' x is still less than 10, adding 1 to x')
    x+=1
    if x==3:
        print('x==3')
    else:
        print('continuing...')
        continue
```

```
x is currently:  0
 x is still less than 10, adding 1 to x
continuing...
x is currently:  1
 x is still less than 10, adding 1 to x
continuing...
x is currently:  2
 x is still less than 10, adding 1 to x
x==3
x is currently:  3
 x is still less than 10, adding 1 to x
continuing...
x is currently:  4
 x is still less than 10, adding 1 to x
continuing...
x is currently:  5
 x is still less than 10, adding 1 to x
continuing...
x is currently:  6
 x is still less than 10, adding 1 to x
continuing...
x is currently:  7
 x is still less than 10, adding 1 to x
continuing...
x is currently:  8
 x is still less than 10, adding 1 to x
continuing...
x is currently:  9
 x is still less than 10, adding 1 to x
continuing...
```

Note how we have a printed statement when x == 3, and a continue being printed out as we continue through the outer while loop. Let's put in a break once x == 3 and see if the result makes sense:

In [25]:
```
x = 0

while x < 10:
    print('x is currently: ',x)
    print(' x is still less than 10, adding 1 to x')
    x+=1
    if x==3:
        print('Breaking because x==3')
        break
    else:
        print('continuing...')
        continue
```

```
x is currently:  0
 x is still less than 10, adding 1 to x
continuing...
x is currently:  1
 x is still less than 10, adding 1 to x
continuing...
x is currently:  2
 x is still less than 10, adding 1 to x
Breaking because x==3
```

Note how the other `else` statement wasn't reached and continuing was never printed!

After these brief but simple examples, you should feel comfortable using `while` statements in your code.

A word of caution however! It is possible to create an infinitely running loop with `while` statements. For example:

In [ ]:
```
# DO NOT RUN THIS CODE!!!!
while True:
    print("I'm stuck in an infinite loop!")
```

A quick note: If you did run the above cell, click on the Kernel menu above to restart the kernel!

Useful Operators

There are a few built-in functions and "operators" in Python that don't fit well into any category, so we will go over them in this lecture, let's begin!

range

The range function allows you to quickly generate a list of integers, this comes in handy a lot, so take note of how to use it! There are 3 parameters you can pass, a start, a stop, and a step size. Let's see some examples:

In [26]:
```
range(0,11)
```

Out[26]:
```
range(0, 11)
```

*Note that this is a **`generator`** function, so to actually get a list out of it, we need to cast it to a list with **`list()`**. What is a generator? Its a special type of function that will generate information and not need to save it to memory. We haven't talked about functions or generators yet, so just keep this in your notes for now, we will discuss this in much more detail in later on in your training!*

In [27]:
```
# Notice how 11 is not included, up to but not including 11, just like slice n
list(range(0,11))
```

Out[27]:
```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

In [28]:
```
list(range(0,12))
```

Out[28]:
```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11]
```

In [29]:
```
# Third parameter is step size!
# step size just means how big of a jump/leap/step you
# take from the starting number to get to the next number.

list(range(0,11,2))
```

Out[29]:
```
[0, 2, 4, 6, 8, 10]
```

In [30]:
```
list(range(0,101,10))
```

Out[30]:
```
[0, 10, 20, 30, 40, 50, 60, 70, 80, 90, 100]
```

enumerate

enumerate is a very useful function to use with for loops. Let's imagine the following situation:

In [31]:
```
index_count = 0

for letter in 'abcde':
    print("At index {} the letter is {}".format(index_count,letter))
    index_count += 1
```

```
At index 0 the letter is a
At index 1 the letter is b
At index 2 the letter is c
At index 3 the letter is d
At index 4 the letter is e
```

_Keeping track of how many loops you've gone through is so common, that enumerate was created so you don't need to worry about creating and updating this index_count or loop*count variable*

In [32]:
```
# Notice the tuple unpacking!

for i,letter in enumerate('abcde'):
    print("At index {} the letter is {}".format(i,letter))
```

```
At index 0 the letter is a
At index 1 the letter is b
At index 2 the letter is c
At index 3 the letter is d
At index 4 the letter is e
```

zip

Notice the format enumerate actually returns, let's take a look by transforming it to a list()

In [33]:
```
list(enumerate('abcde'))
```

Out[33]:
```
[(0, 'a'), (1, 'b'), (2, 'c'), (3, 'd'), (4, 'e')]
```

It was a list of tuples, meaning we could use tuple unpacking during our for loop. This data structure is actually very common in Python , especially when working with outside libraries. You *can use the `zip()` function to quickly create a list of tuples by "zipping" up together two lists.*

In [34]:
```
mylist1 = [1,2,3,4,5]
mylist2 = ['a','b','c','d','e']
```

In [35]:
```
# This one is also a generator! We will explain this later, but for now let's
zip(mylist1,mylist2)
```

Out[35]:
```
<zip at 0x1b752578ac0>
```

In [36]:
```
list(zip(mylist1,mylist2))
```

Out[36]:
```
[(1, 'a'), (2, 'b'), (3, 'c'), (4, 'd'), (5, 'e')]
```

To use the generator, we could just use a for loop

In [37]:
```
for item1, item2 in zip(mylist1,mylist2):
    print('For this tuple, first item was {} and second item was {}'.format(ite
```

```
For this tuple, first item was 1 and second item was a
For this tuple, first item was 2 and second item was b
For this tuple, first item was 3 and second item was c
For this tuple, first item was 4 and second item was d
For this tuple, first item was 5 and second item was e
```

in operator

We've already seen the in keyword durng the for loop, but we can also use it to quickly check if an object is in a list

In [38]:
```
'x' in ['x','y','z']
```

Out[38]:
```
True
```

In [39]:
```
'x' in [1,2,3]
```

Out[39]:
```
False
```

min and max

Quickly check the minimum or maximum of a list with these functions.

In [40]:
```
mylist = [10,20,30,40,100]
```

In [41]:
```
min(mylist)
```

Out[41]:
```
10
```

In [42]:
```
max(mylist)
```

Out[42]:
```
100
```

random

Python comes with a built in random library. There are a lot of functions included in this random library, so we will only show you two useful functions for now.

In [43]:
```
from random import shuffle
```

In [44]:
```
# This shuffles the list "in-place" meaning it won't return
# anything, instead it will effect the list passed
shuffle(mylist)
```

In [45]:
```
mylist
```

Out[45]:
```
[30, 40, 20, 10, 100]
```

In [46]:
```
from random import randint
```

In [47]:
```
# Return random integer in range [a, b], including both end points.
randint(0,100)
```

Out[47]:
```
10
```

In [48]:
```
# Return random integer in range [a, b], including both end points.
randint(0,100)
```

Out[48]:
```
77
```

input

In [49]:
```
input('Enter Something into this box: ')
```

Out[49]:
```
Enter Something into this box: great job!
'great job!'
```

List Comprehensions

In addition to sequence operations and list methods, Python includes a more advanced operation called a list comprehension.

List comprehensions allow us to build out lists using a different notation. You can think of it as essentially a one line `for` loop built inside of brackets. For a simple example:

Example 1

In [50]:
```
# Grab every letter in string
lst = [x for x in 'word']
```

In [51]:
```
# Check
lst
```

Out[51]:
```
['w', 'o', 'r', 'd']
```

This is the basic idea of a list comprehension. If you're familiar with mathematical notation this format should feel familiar for example: x^2 : x in { 0,1,2...10 }

Let's see a few more examples of list comprehensions in Python:

Example 2

```
In [1]: # Square numbers in range and turn into list
        lst = [x**2 for x in range(0, 11)]
```

```
In [2]: lst
```

```
Out[2]: [0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

Example 3

Let's see how to add in `if` statements:

```
In [3]: # Check for even numbers in a range
        lst = [x for x in range(11) if x % 2 == 0]
```

```
In [4]: lst
```

```
Out[4]: [0, 2, 4, 6, 8, 10]
```

Example 4

Can also do more complicated arithmetic:

```
In [5]: # Convert Celsius to Fahrenheit
        celsius = [0,10,20.1,34.5]

        fahrenheit = [((9/5)*temp + 32) for temp in celsius ]

        fahrenheit
```

```
Out[5]: [32.0, 50.0, 68.18, 94.1]
```

Example 5

We can also perform nested list comprehensions, for example:

```
In [6]: lst = [ x**2 for x in [x**2 for x in range(11)]]
        lst
```

```
Out[6]: [0, 1, 16, 81, 256, 625, 1296, 2401, 4096, 6561, 10000]
```

Later on in the course we will learn about generator comprehensions. After this lecture you should feel comfortable reading and writing basic list comprehensions.

Guessing Game Challenge

Let's use `while` loops to create a guessing game.

The Challenge:

Write a program that picks a random integer from 1 to 100, and has players guess the number. The rules are:

1 If a player's guess is less than 1 or greater than 100, say "OUT OF BOUNDS"

2 On a player's first turn, if their guess is

- *within 10 of the number, return "WARM!"*
- *further than 10 away from the number, return "COLD!"*

3 On all subsequent turns, if a guess is

- *closer to the number than the previous guess return "WARMER!"*
- *farther from the number than the previous guess, return "COLDER!"*

4 When the player's guess equals the number, tell them they've guessed correctly and how many guesses it took!

First, pick a random integer from 1 to 100 using the random module and assign it to a variable

Note: `random.randint(a,b)` returns a random integer in range `[a, b]`, including both end points.

In [7]:
```
import random

num = random.randint(1, 100)
```

Next, print an introduction to the game and explain the rules

In [8]:
```
print("WELCOME TO GUESS ME!")
print("I'm thinking of a number between 1 and 100")
print("If your guess is more than 10 away from my number, I'll tell you you're
print("If your guess is within 10 of my number, I'll tell you you're WARM")
print("If your guess is farther than your most recent guess, I'll say you're ge
print("If your guess is closer than your most recent guess, I'll say you're ge
print("LET'S PLAY!")
```

```
WELCOME TO GUESS ME!
I'm thinking of a number between 1 and 100
If your guess is more than 10 away from my number, I'll tell you you're COLD
If your guess is within 10 of my number, I'll tell you you're WARM
If your guess is farther than your most recent guess, I'll say you're getting
COLDER
If your guess is closer than your most recent guess, I'll say you're getting W
ARMER
LET'S PLAY!
```

Create a list to store guesses

Hint: zero is a good placeholder value. It's useful because it evaluates to "False"

In [9]:
```
guesses = [0]
```

Write a `while` loop that asks for a valid guess. Test it a few times to make sure it works

In [11]:
```python
while True:

    guess = int(input("I'm thinking of a number between 1 and 100.\n  What is y

    if guess < 1 or guess > 100:
        print('OUT OF BOUNDS! Please try again: ')
        continue

    break
```

```
I'm thinking of a number between 1 and 100.
  What is your guess? 500
OUT OF BOUNDS! Please try again:
I'm thinking of a number between 1 and 100.
  What is your guess? 50
```

Write a `while` loop that compares the player's guess to our number. If the player guesses correctly, break from the loop. Otherwise, tell the player if they're warmer or colder, and continue asking for guesses.

Some hints:

- *it may help to sketch out all possible combinations on paper first!*
- *you can use the `abs()` function to find the positive difference between two numbers*
- *if you append all new guesses to the list, then the previous guess is given as guesses `[-2]`*

In [13]:
```python
while True:

    # we can copy the code from above to take an input
    guess = int(input("I'm thinking of a number between 1 and 100.\n  What is y

    if guess < 1 or guess > 100:
        print('OUT OF BOUNDS! Please try again: ')
        continue

    # here we compare the player's guess to our number
    if guess == num:
        print(f'CONGRATULATIONS, YOU GUESSED IT IN ONLY {len(guesses)} GUESSES
        break

    # if guess is incorrect, add guess to the list
    guesses.append(guess)

    # when testing the first guess, guesses[-2]==0, which evaluates to False
    # and brings us down to the second section

    if guesses[-2]:
        if abs(num-guess) < abs(num-guesses[-2]):
            print('WARMER!')
        else:
            print('COLDER!')

    else:
        if abs(num-guess) <= 10:
            print('WARM!')
        else:
            print('COLD!')
```

```
I'm thinking of a number between 1 and 100.
  What is your guess? 50
COLDER!
I'm thinking of a number between 1 and 100.
  What is your guess? 75
WARMER!
I'm thinking of a number between 1 and 100.
  What is your guess? 85
WARMER!
I'm thinking of a number between 1 and 100.
  What is your guess? 92
COLDER!
I'm thinking of a number between 1 and 100.
  What is your guess? 80
WARMER!
I'm thinking of a number between 1 and 100.
  What is your guess? 78
COLDER!
I'm thinking of a number between 1 and 100.
  What is your guess? 82
WARMER!
I'm thinking of a number between 1 and 100.
  What is your guess? 83
CONGRATULATIONS, YOU GUESSED IT IN ONLY 15 GUESSES!!
```

That's it! You've just programmed your first game!

Thank You

CONNECT WITH ME :

LinkedIn GitHub kaggle Medium

PRASADMJADHAV2