



# PYTHON FOR DATA SCIENCE

## UNIT 14

### LOOPS

&

### LIST COMPREHENSIONS



# MAIN TOPICS TO BE COVERED

- What are loops? And, why are they used?
- Two main kinds of loops:
  - ... **for** loops
  - ... **while** loops
- The range () function
- Using **break** statements to discontinue loop execution
- List comprehensions



# LOOP BASICS



# WHAT ARE LOOPS?

- Loops are pieces of code that repeat code several times before moving on to the statement that comes after the code block that the for loop repeats

- More technically: loops “iterate over a sequence”

*“do something over again or repeatedly”*

*“ordered list”*

- The different types of loops repeat differently
  - **for** loops repeat the code n times (i.e., a for loop will repeat the code a fixed number of times)
  - **while** loops repeat the code until a condition is satisfied



# “for” LOOPS



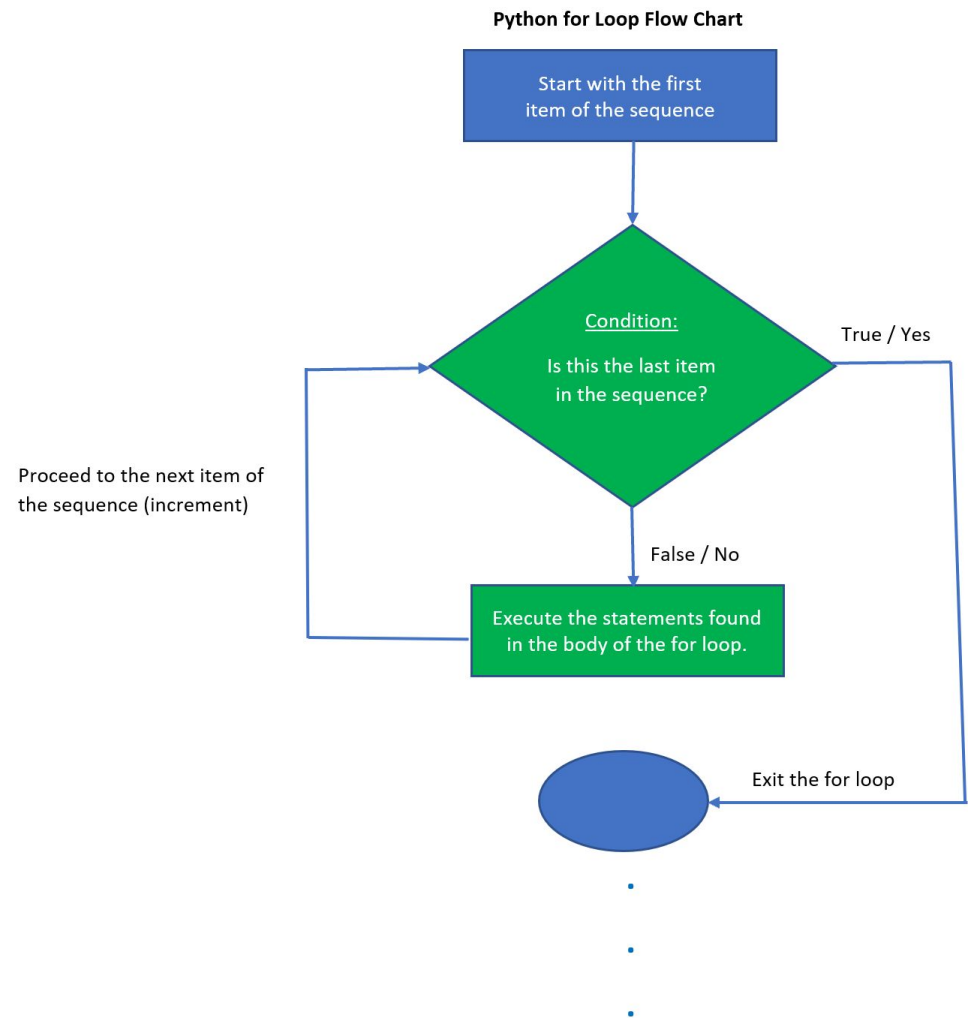
# for LOOP BASICS

- for loops repeat a piece/block of code for each item of a sequence  
... i.e. , for loops “iterate” over sequences
- for loops in Python will iterate over any “iterable” object
  - ... lists
  - ... tuples
  - ... sets
  - ... strings
  - ... dictionaries
- Iteration (iterating) is also known as traversal (traversing)



# HOW for LOOPS WORK

- This is a flow chart that shows how for loops work.
- The important point to understand is that for loops repeat a block of code for every item in the sequence.





# for LOOPS ITERATE OVER ITERABLES AND SEQUENCES

- Question:     What is an iterable?
- Answer:       Iterables are objects that can be iterated over ... i.e., iterables are something that we can loop over
- Examples of iterables:
  - lists
  - strings
  - tuples
  - dictionaries
  - sets
  - ... and others





# “for” LOOP SYNTAX



# for LOOP SYNTAX

Use the keyword **for**  
to start a for loop.

This is a Python sequence or iterable.  
(i.e., something that can be iterated over  
... e.g. a list, tuple, set, etc.)

```
for item in sequence :  
    code block to repeat
```

This code block will only be executed for every iteration of the loop  
... i.e., this code will execute for every item in the sequence.  
The code block is indented 4 spaces.



# CODE BLOCKS OF A for LOOP MUST BE INDENTED

Here, item is a variable that is exclusively defined for the for loop.

Item is just a placeholder that holds the element of the sequence, as we iterate through the sequence.

We can name this variable anything that we want.

```
for item in sequence:  
    code block to repeat
```

This indentation must be present ...

The white space (i.e., the indentation) is syntactically meaningful in Python.

The best practice is to use 4 spaces to indent



# CODE BLOCKS ARE 1 OR MORES LINES LONG

```
for item in sequence :  
    code block to repeat
```

The code block can be 1 line long or “hundreds” of lines long

... i.e., the code block can be as long as we want!!

... but, all of the lines of a code block must be indented properly.



# SIMPLE EXAMPLES: for LOOP SYNTAX

The lists, `your_list` and `my_list` are a type of iterable.

In the first example, we iterate over the elements of the sequence (1, 2, 3, 4).

In the second example, we iterate over the elements of the sequence (apple, banana, grape).

For every element in the respective sequences, we print out the element using a `print()` statement.

In both examples, we also exit the for loop and continue on with the flow of the program.

```
your_list = [1, 2, 3, 4]

for item in your_list:
    print(item)

print('This line being printed indicates that we have exited the for loop.')
```

1  
2  
3  
4  
This line being printed indicates that we have exited the for loop.

```
my_list = ['apple', 'banana', 'grape']

for item in my_list:
    print(item)

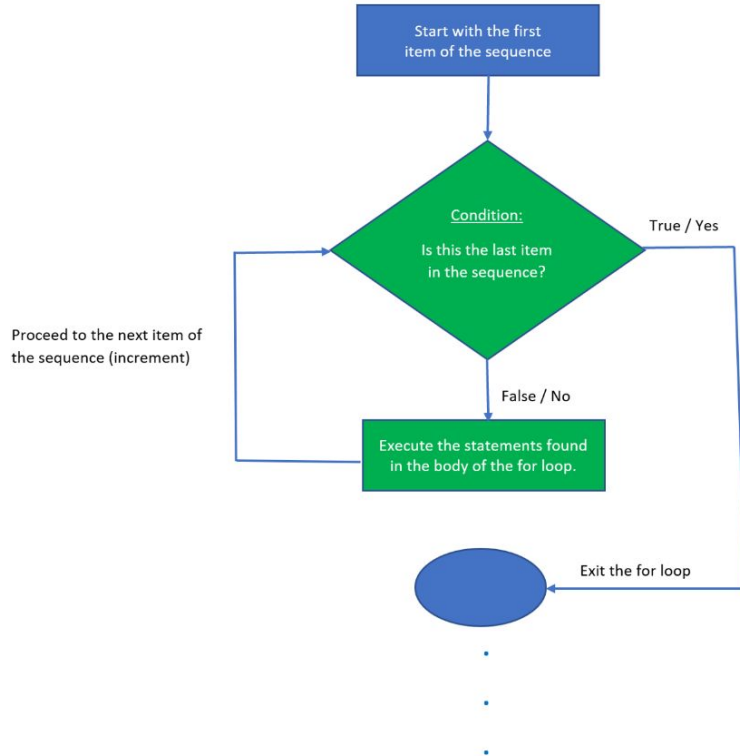
print('This line being printed indicates that we have exited the for loop.')
```

apple  
banana  
grape  
This line being printed indicates that we have exited the for loop.



# THINK OF for LOOP SYNTAX LIKE A FLOW CHART

Python for Loop Flow Chart



```
your_list = [1, 2, 3, 4]

for item in your_list:
    print(item)

print('This line being printed indicates that we have exited the for loop.')
```

```
1
2
3
4
```

This line being printed indicates that we have exited the for loop.

```
my_list = ['apple', 'banana', 'grape']

for item in my_list:
    print(item)

print('This line being printed indicates that we have exited the for loop.')
```

```
apple
banana
grape
```

This line being printed indicates that we have exited the for loop.



# “for” LOOP EXAMPLES



# EXAMPLES OF for LOOPS

- for loops operate in a similar fashion on different sequences
  - lists
  - tuples
  - strings
  - etc.
- The slides that follow will provide concrete examples that demonstrate how the code iterates through the sequence.





## EXAMPLE: LOOP OVER A LIST

```
sports_list = ['football', 'basketball', 'golf', 'tennis']  
  
for sport in sports_list:  
    print(sport)
```

```
football  
basketball  
golf  
tennis
```

- This code iterates over each item of `sports_list` ... when it iterates over a particular item in the list, the code in the body of the loop prints out that sport.
- The variable name “sport” was used as the placeholder -- it just represents the items of `sports_list`.



## FOR EVERY ITEM IN `sports_list`, THE LIST ITEM GETS PRINTED OUT

```
sports_list = ['football', 'basketball', 'golf', 'tennis']  
  
for sport in sports_list:  
    print(sport)
```

```
football  
basketball  
golf  
tennis
```

[Python Tutor -- Visualize Code Execution](#)

```
sports_list = ['football', 'basketball', 'golf', 'tennis']
```

```
for sport in sports_list:  
    print(sport)
```

```
print('This line is being printed to indicate the the program has exited the for loop.')
```



# EXAMPLE: LOOP OVER A STRING

```
our_string = "Python"

for letter in our_string:
    print(letter)
```

P  
y  
t  
h  
o  
n

- This code iterates over each item of `our_string` ... when it iterates over a particular letter in the string, the code in the body of the loop prints out that letter.
- The variable name “letter” was used as the placeholder -- it just represents the items of `our_string`.



## FOR EVERY ITEM IN `our_string`, THE STRING CHARACTER GETS PRINTED OUT

```
our_string = "Python"

for letter in our_string:
    print(letter)
```

P  
y  
t  
h  
o  
n

[Python Tutor -- Visualize Code Execution](#)

```
our_string = "Python"
```

```
for letter in our_string:
    print(letter)
```

```
print('This line is being printed to indicate the the program has exited the for loop.')
```



## FOR EVERY ITEM IN `mixed_tuple`, THE TUPLE ITEM GETS PRINTED OUT

```
mixed_tuple = (15, 'dog', 4.6, [3, 5], ('a', 'b', 'c' ))  
  
for item in mixed_tuple:  
    print(item)
```

```
15  
dog  
4.6  
[3, 5]  
( 'a', 'b', 'c' )
```

- This code iterates over each character of `mixed_tuple` ... when it iterates over a particular item in the tuple, the code in the body of the loop prints out that item.
- The variable name “item” was used as the placeholder -- it just represents the items of `mixed_tuple`.



# EXAMPLE: LOOP OVER A TUPLE

```
mixed_tuple = (15, 'dog', 4.6, [3, 5], ('a', 'b', 'c' ))  
  
for item in mixed_tuple:  
    print(item)
```

```
15  
dog  
4.6  
[3, 5]  
( 'a', 'b', 'c' )
```

[Python Tutor -- Visualize Code Execution](#)

```
mixed_tuple = (15, 'dog', 4.6, [3, 5], ('a', 'b', 'c' ))
```

```
for item in mixed_tuple:  
    print(item)
```

```
print('This line is being printed to indicate the the program has exited the for loop.')
```



# SUMMARY OF THE SIMPLE for LOOP EXAMPLES

*IN EACH OF THESE EXAMPLES, WE ITERATED OVER AN “ITERABLE”.*

- In all three cases, the for loop iterated over a sequence / iterable
  - The list was an iterable
  - The string was an iterable
  - The tuple was an iterable
- For every element of the sequence / iterable the code block gets executed

*Note: The code block in a for loop can be much more complicated than in these simple examples.*



# The range() FUNCTION





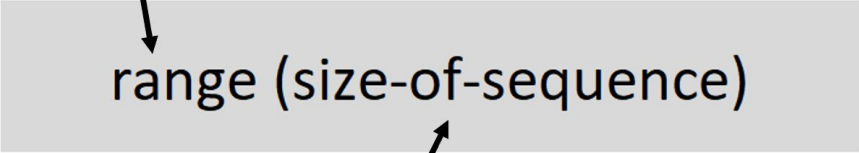
# THE range() FUNCTION

- The range() function generates sequences of numbers
- For example, range(10) generates the sequence 0, 1, 2, 3, 4, 5, 6, 7, 8, 9
- We can use these sequences in our for loops



# range() SYNTAX

Initiate the range() function



range (size-of-sequence)

The diagram shows a light gray rectangular box containing the text 'range (size-of-sequence)'. An arrow points from the text 'Initiate the range() function' above to the 'range' part of the code. Another arrow points from the text 'Inside of range(), we specify the size of the numeric sequence we want to create' below to the '(size-of-sequence)' part of the code.

Inside of range(), we specify the size of the numeric sequence we want to create

For example, range(7) will create a sequence that is 7 numbers long ... 0, 1, 2, 3, 4, 5, 6



# A FEW IMPORTANT DETAILS ABOUT THE `range()` FUNCTION

- The output of `range()` starts at 0 by default.
- The output ends at 1 less than the size of the output ...
  - Example: `range(4)` produces the output 0, 1, 2, 3
  - Note: `range(4)` does not include 4!
- `range()` can be used to generate a wide variety of arithmetic sequences ... the more complicated uses of the `range()` built-in function won't be covered in this presentation.



## EXAMPLE: USING range() IN A for LOOP

The for loop iterates over the items of the numeric sequence and prints them out.

```
for number in range(5):  
    print(number)
```

0  
1  
2  
3  
4

The range() function creates a sequence of 5 numbers that we can iterate over.

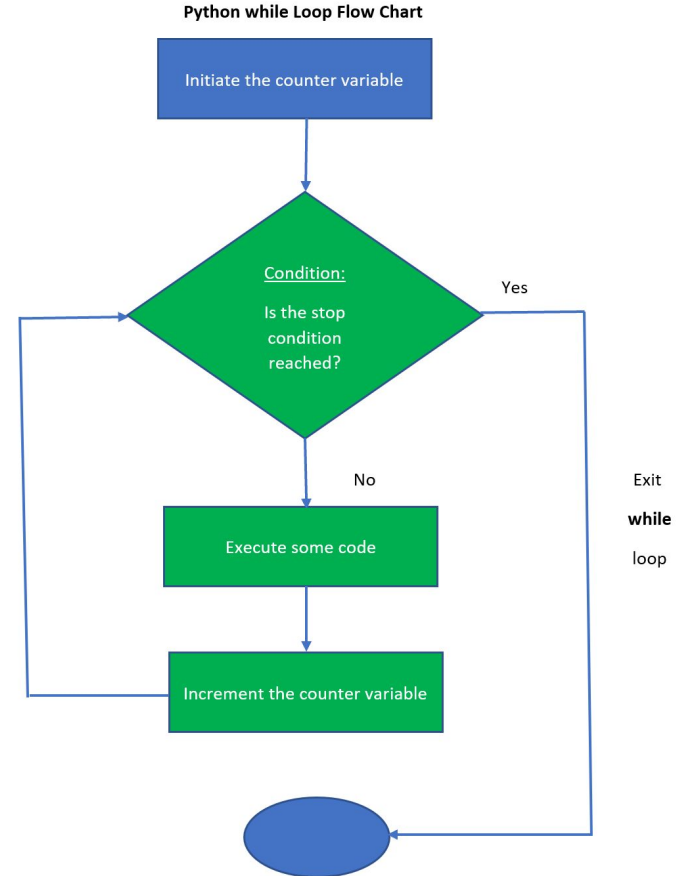


# while LOOPS



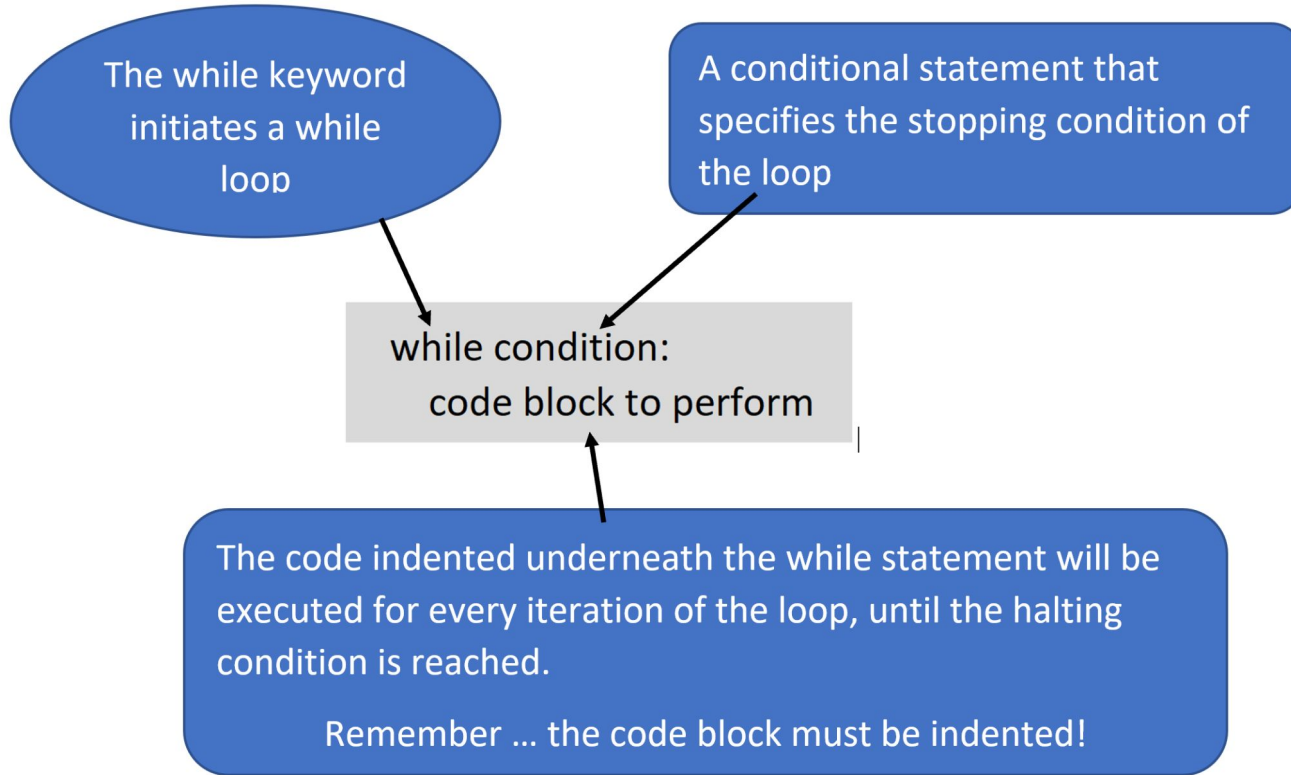
# WHAT IS A while LOOP?

- while loops are similar to for loops
- They “loop” through and repeatedly execute some code until a stopping condition is reached.





# while LOOP SYNTAX





# EXAMPLE: SIMPLE while LOOP

Here, a variable called counter is being created

```
counter = 0
```

The code block will print the square of value of counter and then increment the counter by 1.

```
while counter < 5:  
    print(counter ** 2)  
    counter = counter + 1
```

This stopping condition will cause the loop to run until counter == 5.

```
0  
1  
4  
9  
16
```

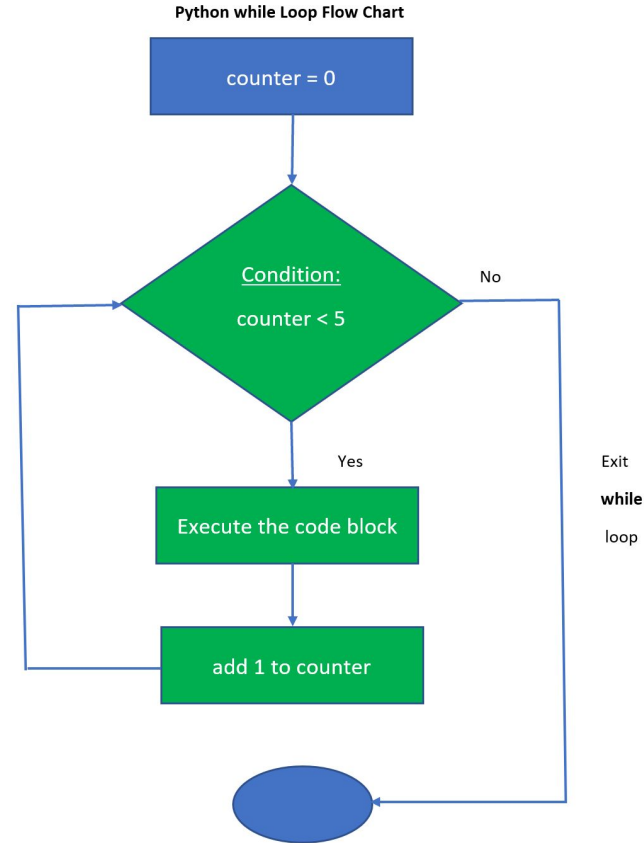




# HOW THIS while LOOP WORKS

Every loop iteration ...

- Check the halt condition
- Execute the code block
- Add 1 to the counter
- Repeat until stopping condition is reached





# NOTES ABOUT while LOOPS

- while loops continue to loop until a halting condition is reached
- The body code should include a variable that we use in the halting condition
  - we need to create a way to halt the loop
  - Without a halting condition, the loop will continue to repeat ... this is called an “infinite loop”



# break STATEMENTS



# break STATEMENTS

- We can use **break** to “break out” of a **for** loop or **while** loop.
- Note about nested for loops:
  - **break** will break out of the “smallest” **for** loop
  - i.e., the nearest **for** enclosing the **break** statement



## EXAMPLE: break STATEMENT

```
break_list = [2, 4, 6, 8, 100, 12]
for number in break_list:
    print(number)
    if number == 100:
        print('We found it!!')
        break
```

```
2
4
6
8
100
We found it!!
```

- If `number == 100`, then we break out of the for loop
  - **break** stops execution
  - the loop won't continue on to 12



For LOOPS

VS

While LOOPS



# WHEN TO USE 'for' VS 'while'

- For is best when ...
  - you have 'iterable' objects
  - You have sequences: lists, tuples, sets, strings, etc.
- While is best when ...
  - You don't have iterable objects or sequences to iterate through
  - You don't have a simple data structure to drive the looping process
  - You have logical conditions that can't be represented by a sequence



# LIST COMPREHENSIONS





# LIST COMPREHENSIONS ARE A CONCISE WAY TO CREATE LISTS

- List comprehensions have a compact syntax to create lists
  - list comprehensions put a **for** loop inside brackets
- The output of a list comprehension is a list.



## SYNTAX: LIST COMPREHENSIONS

This is a for loop that defines how we will repeat the *expression* for every value of *iterable*



```
new_list = [ expression for x in iterable ]
```

Expression is a piece of code that will execute for every iteration of the for loop



## SYNTAX: LIST COMPREHENSIONS

Notice that all of this is enclosed inside of brackets



```
new_list = [ expression for x in iterable ]
```

The code snippet is displayed within a light gray rectangular box. A blue arrow points from the text 'all of this' in the paragraph above to the opening square bracket '[' of the list comprehension. Another blue arrow points from the text 'enclosed inside of brackets' in the same paragraph to the closing square bracket ']' of the list comprehension.

So the output of a list comprehension is a list.



## EXAMPLE: LIST COMPREHENSION

```
odd_list = [(2 * x + 1) for x in range(7)]  
print(odd_list)
```

```
[1, 3, 5, 7, 9, 11, 13]
```



## EXAMPLE: LIST COMPREHENSION

The code `range(7)` generates the sequence of integers 0, 1, 2, 3, 4, 5, 6

```
odd_list = [(2 * x + 1) for x in range(7)]  
print(odd_list)
```

```
[1, 3, 5, 7, 9, 11, 13]
```



## EXAMPLE: LIST COMPREHENSION

For every value  $x$  of `range(7)`, this expression will output 2 times  $x$  plus 1

This for loop will iterate over every value of `range(7)`

```
odd_list = [(2 * x + 1) for x in range(7)]  
print(odd_list)
```

```
[1, 3, 5, 7, 9, 11, 13]
```



## EXAMPLE: LIST COMPREHENSION

```
odd_list = [(2 * x + 1) for x in range(7)]  
print(odd_list)
```

[1, 3, 5, 7, 9, 11, 13]



Notice that the output is 2 times every value of range(7) plus 1

Also, notice that the output is a list

Remember: list comprehensions are just a concise way to create lists.



# SYNTAX COMPARISON: for LOOP VS. LIST COMPREHENSION

Remember ... list comprehensions are like for loops that generate a list

## List Comprehension

```
new_list = [2 * x + 1 for x in range(7)]  
  
print(new_list)
```

[1, 3, 5, 7, 9, 11, 13]

## For Loop

```
new_list = []  
for x in range(7):  
    new_list.append(2 * x + 1)  
  
print(new_list)
```

[1, 3, 5, 7, 9, 11, 13]





# THESE TWO PIECES OF CODE PRODUCE THE SAME OUTPUT

Remember ... list comprehensions are like for loops that generate a list

## List Comprehension

```
new_list = [2 * x + 1 for x in range(7)]  
  
print(new_list)
```

[1, 3, 5, 7, 9, 11, 13]

## For Loop

```
new_list = []  
for x in range(7):  
    new_list.append(2 * x + 1)  
  
print(new_list)
```

[1, 3, 5, 7, 9, 11, 13]



## WHEN TO USE LIST COMPREHENSIONS

- List comprehensions replace **for** loops in some instances
- List comprehensions are good for creating lists that contain sequences ... i.e., regular sequences that can be described mathematically

## BE CAREFUL WITH LIST COMPREHENSIONS!

- List comprehensions are more concise
- But, list comprehensions are harder to debug!! So be careful!!!