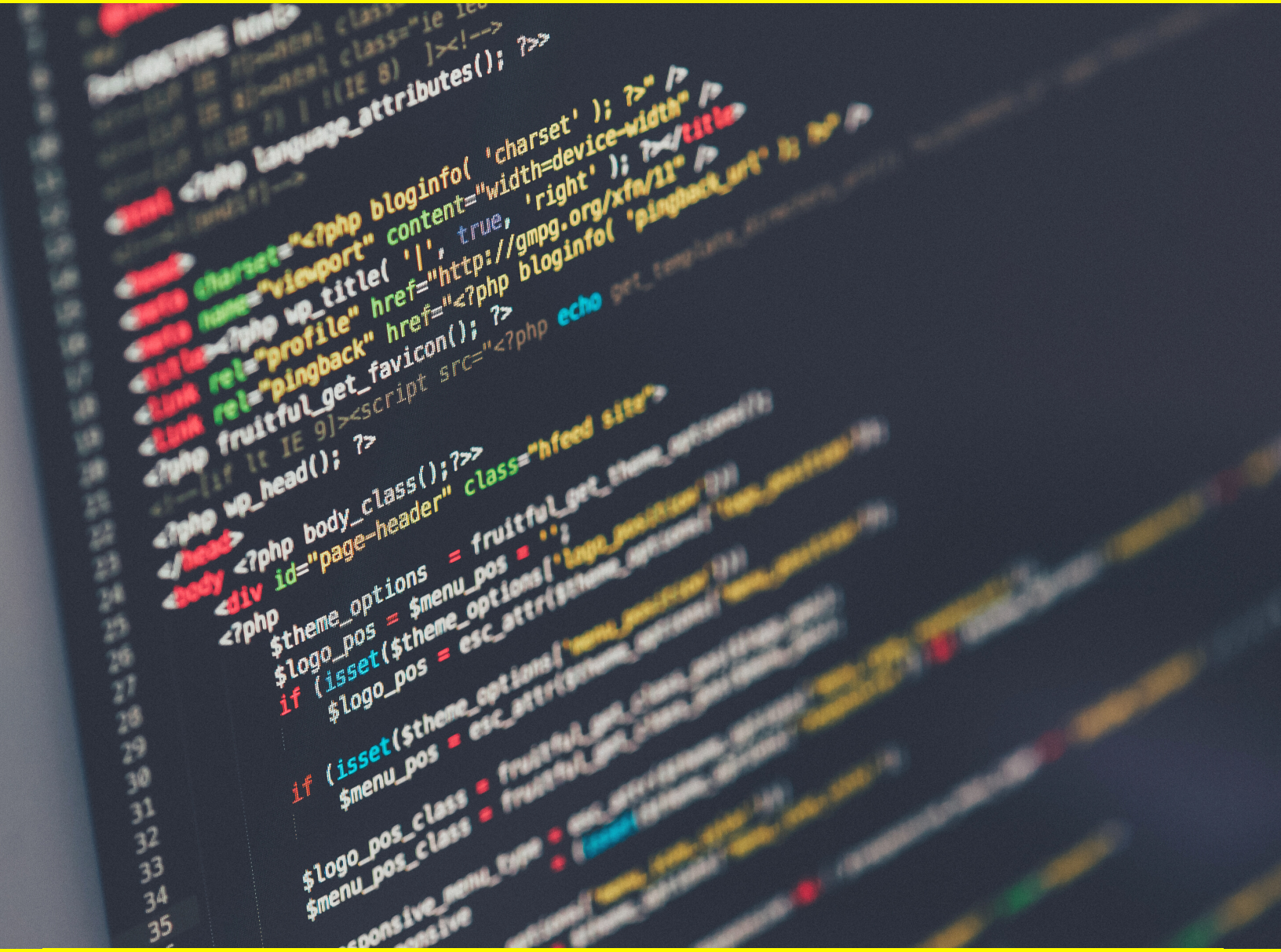
the fast lane to python

PART 4



BY:

Arshadul Shaikh

Ref taken from Coursera + Udemy

***KEYNOTE:**

Congratulations again , welcome to part 4

We still have a long way to go, if you along with me are practicing and learning with the same excitement as you had on the first minute when you started with "the fast lane to python", then consider yourself on the right track and soon achieving what you always wanted.

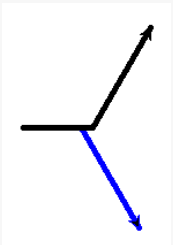
“Learning never exhausts the mind.” Lot said, let's roll:

Turtles and Conditionals

You will have used Whatsapp or using right now, as it has become a part of our daily life, and you await eagerly for turning those 2 ticks to blue when you have sent the message to someone whom you really want to read that message asap. The logic behind is simple it sets the ticks to blue (if you have not turned one the stealth mode) if the message was read by the user or else it would stay grey.

This is done with something called a selection or a conditional statement.In the context of turtle drawings, using this kind of statement will allow us to check conditions and change the behavior of the program accordingly.

```
import turtle
wn = turtle.Screen()

shiv = turtle.Turtle()
shiv_color=input('which color you want to paint shiv -hint blue ')
shiv.pencolor(shiv_color)
shiv.pensize('5')
shiv.forward(50)
if shiv.pencolor() == "blue":
    shiv.right(60)
    shiv.forward(100)
else:
    shiv.left(60)
    shiv.forward(100)

gopi = turtle.Turtle()
gopi_color=input('which color you want to paint shiv -hint black ')
gopi.pencolor(gopi_color)
gopi.pensize('5')

gopi.forward(60)

if gopi.pencolor() == "Pink":
    gopi.right(60)
    gopi.forward(100)
else:
    gopi.left(60)
    gopi.forward(100)

wn.exitonclick()
```



```
which color you want to paint shiv -hint blue blue
which color you want to paint shiv -hint black black
```

- we first set shiv's pen color to be “blue” and then move him forward.
- Next we want one of two actions to happen, either shiv should move right and then forward, or left and then forward. The direction that we want him to go in depends on her pen color. If his pen color is set to blue - which is determined by writing
  shiv.pencolor() == "blue"
  which checks to see if the value returned by shiv.pencolor() is the equivalent to the string “blue” - then we should have shiv move right and forward.
- Else (or otherwise) shiv should move left and forward. Both things can’t happen though. He can’t move right, forward and left, forward.
- We then do the same thing for gopi, though in this case

Boolean Values and Boolean Expressions

The Boolean data type is a data type that has one of two possible values (usually denoted true and false) which is intended to represent the two truth values of logic . It is named after George Boole, who first defined an algebraic system of logic in the mid 19th century.

The only two values that Python accepts and are standard bool value are True and False. **Capitalization is important**, since true and false are not boolean values (remember Python is case sensitive).

```
print(True)
print(type(True)) #True/False without inverted commans and with
print(type(False))  #initial capitals are bool values and will return bool class
print(type('True')) #anything between inverted commas is a string
print(type(true)) #anything other than True/Fales is a name/variable and needs to be defined
                # true here is not bool and we got NameError i.e Runtime Error


```

```
True
<class 'bool'>
<class 'bool'>
<class 'str'>

---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-41-0c72287f4637> in <module>
      3 print(type(False))
      4 print(type('True'))
----> 5 print(type(true))
      6

NameError: name 'true' is not defined
```

*****KEY POINT:**

Boolean values are not strings!

It is extremely important to realize that True and False are not strings. They are not surrounded by quotes. They are the only two values in the data type bool.

```
print(5 == 5)
print(5 == 6)
```

```
True
False
```

a Boolean expression is an expression used in programming languages that produces a Boolean value when evaluated. A Boolean expression may be composed of a combination of the Boolean constants true or false, Boolean-typed variables, Boolean-valued operators, and Boolean-valued functions.

The equality operator, ==, compares two values and produces a boolean value related to whether the two values are equal to one another.

The == operator is one of six common comparison operators; the others are:

```
x != y               # x is not equal to y
x > y                # x is greater than y
x < y                # x is less than y
x >= y               # x is greater than or equal to y
x <= y               # x is less than or equal to y
```

Although these operations are probably familiar to you, the Python symbols are different from the mathematical symbols.

********A common error is to use a single equal sign (=) instead of a double equal sign (==).********

Remember that **=** as we have learned previously in depth, is an **assignment operator** and **==** is a **comparison operator**.

XXXXXX Also, there is no such thing as =< or => **XXXXXX**

- **N**ote too that an equality test is symmetric, but assignment is not. For example, if a == 7 then 7 == a.
- But in Python, the statement a = 7 is legal and 7 = a is not, as we know a is a variable and can have some value but 7=a means you are assigning variables value to a primitive data type which is constant.

3==4 , (3+4)==7 , "False"=="True" , these are all boolean expressions meaning they will return either True or False if you put them inside print

Logical operators

We have already read about **Operators**, which are special symbols in Python that carry out arithmetic or logical computation. And the value that the operator operates on is called the **operand**.

In Python, Logical operators are used on conditional statements (either True or False). They perform **Logical AND, Logical OR and Logical NOT** operations.

Logical Operators		
L.H.S	AND	R.H.S
L.H.S	OR	R.H.S
	NOT	Bool

```
x = 5
print(x>0 and x<10)  #x>0 is L.H.S which is true
                     #x<10 is R.H.S which is also true
                     #as per the below AND table, new bool value will be true

n = 25
print(n%2 == 0 or n%3 == 0)# n%2==0 is L.H.S which is false cause odd_no%2 will always leave 1 remainder
                     #n%3 == 0 is R.H.S which is also false
                     #cause %3 will be 0 only if it is a multiple of 3 and 25 is not
                     #as per below OR table we will fall in false case

```

```
True
False
```

AND	L.H.S True	L.H.S False
R.H.S True	TRUE	FALSE
R.H.S False	FALSE	FALSE

OR	L.H.S True	L.H.S False
R.H.S True	TRUE	TRUE
R.H.S Fals	TRUE	FALSE

Bool	B true	B False
Not	FALSE	TRUE

****** KEY POINT:**

Common Mistake!
There is a very common mistake that occurs when programmers try to write boolean expressions.
For example, what if we have a variable number and we want to check to see if its value is 5, 6, or 7?
In words we might say: “number equal to 5 or 6 or 7”. However, if we translate this into Python, number == 5 or 6 or 7, it will not be correct.
The or operator must join the results of three equality checks. The correct way to write this is number == 5 or number == 6 or number == 7.This may seem like a lot of typing but it is absolutely necessary. You cannot take a shortcut.
Well, actually, you can take a shortcut but not that way. Later you’ll learn about the in operator for strings and sequences: you could
write number in [5, 6, 7].

The in and not in operators

The in operator tests if one string is a substring of another

```
1 print('p' in 'apple')
2 print('i' in 'apple')
3 print('ap' in 'apple')
4 print('pa' in 'apple')
```

```
True
False
True
False
```

```
1 print('' in 'a')
2 print('' in 'apple')
```

```
True
True
```

Note that a string is a substring of itself, and the empty string is a substring of any other string.

The not in operator returns the logical opposite result of in

```
1  print('x' not in 'apple')
```

```
True
```

We can also use the in and not in operators on lists!

```
1  print("a" in ["a", "b", "c", "d"])
2  print(9 in [3, 2, 9, 10, 9.0])
3  print('wow' not in ['gee wiz', 'gosh golly', 'wow', 'amazing'])
```

```
True
True
False
```

However, remember how you were able to check to see if an “a” was in “apple”? Let’s try that again to see if there’s an “a” somewhere in the following list.

```
1  print("a" in ["apple", "absolutely", "application", "nope"])
```

```
False
```

Clearly, we can tell that a is in the word apple, and absolutely, and application.
For some reason though, the Python interpreter returns False.

Why is that?
When we use the in and not in operators on lists, Python checks to see if the item on the left side of the expression is equivalent to an element in the item on the right side of the expression.
In this case, Python is checking whether or not an element of the list is the string “a” - nothing more or less than that

Precedence of Operators

Level	Category	Operators
7(high)	exponent	**
6	multiplication	*,/,//,%
5	addition	+,-
4	relational	==,!=,<=,>=,>,<
3	logical	not
2	logical	and
1(low)	logical	or

summary of all Operator precedence in Python, from lowest to highest precedence .Operators in the same box have the same precedence.

Operator	Description
`:=`	Assignment expression
`lambda`	Lambda expression
`if – else`	Conditional expression
`or`	Boolean OR
`and`	Boolean AND
`not x`	Boolean NOT
`in`, `not in`, `is`, `is not`, `<`, `<=`, `>`, `>=`, `!=`, `==`	Comparisons, including membership tests and identity tests
`|`	Bitwise OR
`^`	Bitwise XOR
`&`	Bitwise AND
`<<`, `>>`	Shifts
`+`, `-`	Addition and subtraction
`*`, `@`, `/`, `//`, `%`	Multiplication, matrix multiplication, division, floor division, remainder [5]
`+x`, `-x`, `~x`	Positive, negative, bitwise NOT
`**`	Exponentiation [6]
`await x`	Await expression
`x[index]`, `x[index:index]`, `x(arguments...)`, `x.attribute`	Subscription, slicing, call, attribute reference
`(expressions...)`, `[expressions...]`, `{key: value...}`, `{expressions...}`	Binding or parenthesized expression, list display, dictionary display, set display

*****KEY POINT:**

Common Mistake!

Students often incorrectly combine the in and or operators. For example, if they want to check that the letter x is inside of either of two variables then they tend to write it the following way:

'x' in y or z

Written this way, the code would not always do what the programmer intended. This is because the in operator is only on the left side of the or statement. It doesn't get implemented on both sides of the or statement. In order to properly check that x is inside of either variable, the in operator must be used on both sides which looks like this:

'x' in y or 'x' in z

Add parentheses to the following expression to make the order of evaluation more clear.

year % 4 == 0 and year % 100 != 0 or year % 400 == 0

Options:

a = (((year % 4) == 0) and ((year % 100) != 0)) or ((year % 400) == 0)
b = (year % 4 == 0 and year % 100 != 0) or (year % 400 == 0)
c = (year % 4 == 0 and year % 100 != 0) or year % 400 == 0
d = year % 4 == 0 and year % 100 != 0 or year % 400 == 0
e = ((year % 4 == 0) and (year % 100 != 0)) or (year % 400 == 0)

Ans:

((year % 4 == 0) and (year % 100 != 0)) or (year % 400 == 0)

Which of the following properly expresses the precedence of operators (using parentheses) in the following expression:
5*3 > 10 and 4+6==11

A. ((5*3) > 10) and ((4+6) == 11)
B. (5*(3 > 10)) and (4 + (6 == 11))
C. ((((5*3) > 10) and 4)+6) == 11
D. ((5*3) > (10 and (4+6))) == 11

Ans:
A. ((5*3) > 10) and ((4+6) == 11)

Conditional Execution: Binary Selection

we almost always need the ability to check conditions and change the behavior of the program accordingly. Selection statements, sometimes also referred to as conditional statements, give us this ability. The simplest form of selection is the if statement. This is sometimes referred to as binary selection since there are two possible paths of execution.

```
1  x = 15
2
3  if x % 2 == 0:
4      print(x, "is even")
5  else:
6      print(x, "is odd")
```

```
15 is odd
```

Let us try to compare the same odd/even number program in other languages :

In C++

```cpp
#include <iostream>
using namespace std;

int main()
{
    int n;

    cout << "Enter an integer: ";
    cin >> n;

    if ( n % 2 == 0)
        cout << n << " is even.";
    else
        cout << n << " is odd.";

    return 0;
}
```

You see, the amount of lines we have saved when we wrote the same program in python compared with all other OOP

jAVA

```java
import java.util.Scanner;

public class EvenOdd {

    public static void main(String[] args) {

        Scanner reader = new Scanner(System.in);

        System.out.print("Enter a number: ");
        int num = reader.nextInt();

        if(num % 2 == 0)
            System.out.println(num + " is even");
        else
            System.out.println(num + " is odd");
    }
}
```

- The **boolean expression** after the if statement is called the condition. If it is true, then the indented statements get executed. If not, then the statements indented under the else clause get executed.
- As with the function definition from the last chapter and other compound statements like for, the if statement consists of a header line and a body. The **header line begins with the keyword if followed** by a **boolean expression** and **ends** with a **colon (:)**
- **The indented statements that follow are called a block**.
- The **first unindented** statement marks the **end** of the **block**.
- Each of the **statements** inside the **first block** of statements is **executed** in order if the boolean expression evaluates to **True**.
- The entire first block of statements is **skipped** if the boolean expression evaluates to **False**, and instead all the statements under the else clause are executed.
- There is **no limit** on the **number of statements** that can appear under the two clauses of an if statement, but there has to be at least one statement in each block.

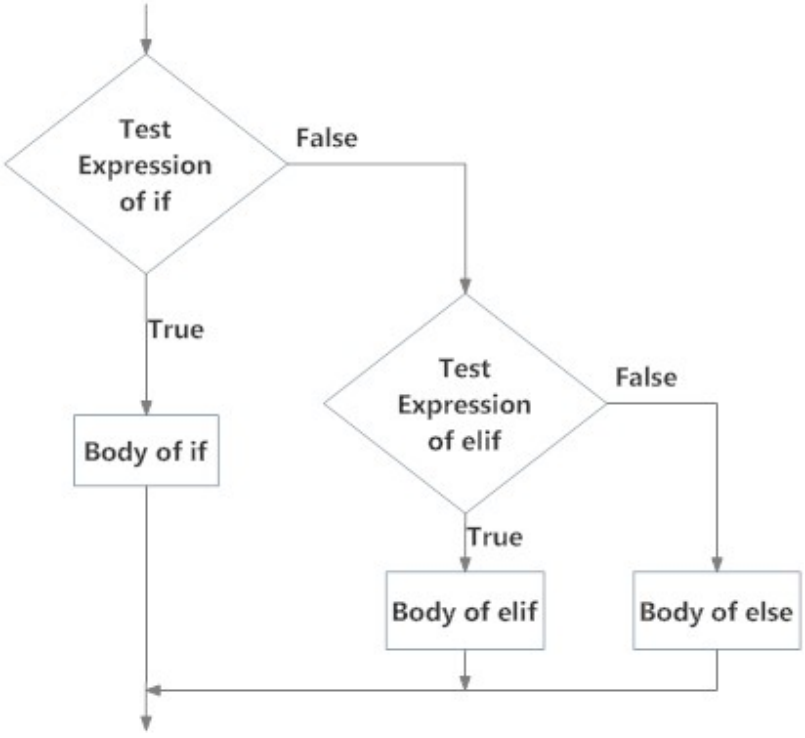


Fig: Operation of if...elif...else statement

Practice Questions:

Question 1: How many lines of code can appear in the indented code block below the if and else lines in a conditional?

A. Just one.
B. Zero or more.
C. One or more.
D. One or more, and each must contain the same number.

Hint: A block must contain at least one statement and can have many statements.

Question 2: What does the following code print?

```
if (4 + 5 == 10):
   print("TRUE")
else:
   print("FALSE")
```

A. TRUE
B. FALSE
C. TRUE on one line and FALSE on the next
D. Nothing will be printed

Hint: Since 4+5==10 evaluates to False, Python will skip over the if block and execute the statement in the else block.

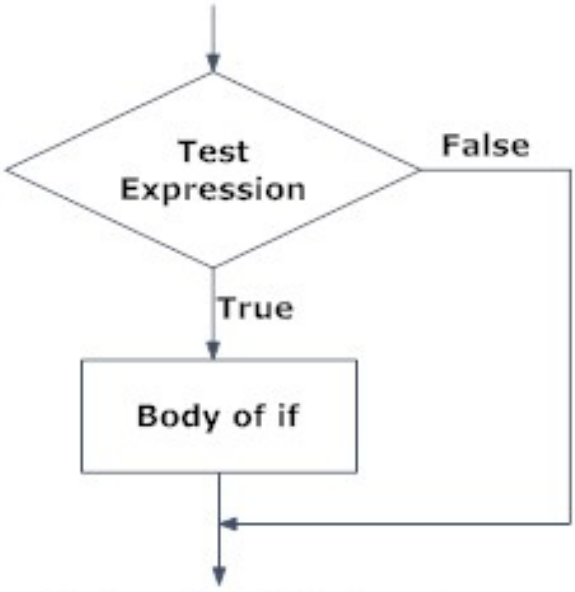
Question 3: What does the following code print?

```
if (4 + 5 == 10):
   print("TRUE")
else:
   print("FALSE")
print("TRUE")
```

Hint: Python will print FALSE from within the else-block (because 5+4 does not equal 10), and then print TRUE after the if-else statement completes.

Omitting the else Clause: Unary Selection

Another form of the if statement is one in which the else clause is omitted entirely. This creates what is sometimes called unary selection. In this case, when the condition evaluates to True, the statements are executed. Otherwise the flow of execution continues to the statement after the body of the if.



Sample example:

```
x = 10   #you can any give value here, by taking input
         #from user by adding input method
if x < 0:  #this condition indirectly means x should be negative
         #below line will get printed only if the x is negative
    print("The negative number ",  x, " is not valid here.")


print("This is always printed") #this will get printed always
                                #irrespective of whatever the value of x is
```

```
This is always printed
```

```
x =-10   #you can any give value here, by taking input
        #from user by adding input method
if x < 0:  #this condition indirectly means x should be negative
        #below line will get printed only if the x is negative
    print("The negative number ",  x, " is not valid here.")


print("This is always printed") #this will get printed always
                                #irrespective of whatever the value of x is
```

```
The negative number  -10  is not valid here.
This is always printed
```

```
x=10
y=5
if x < y:
    print("x is less than y")
else:
    print("x is greater than y")

else:
    print("x and y must be equal")
```

```
  File "<ipython-input-54-c86f2103c467>", line 8
    else:
       ^
SyntaxError: invalid syntax
```

the above code gave an error :O but why?

- because we have 2 else statements and one if, every else needs to have an if condition cause else is equivalent to otherwise and if you just say "otherwise i will do this" and end a statement , first question that the listener will ask is "otherwise, what? why you talking otherwise first tell me the first clause"

- if want to check condition 1 , condition 2 and then end with else part, you would need to use nested conditionals

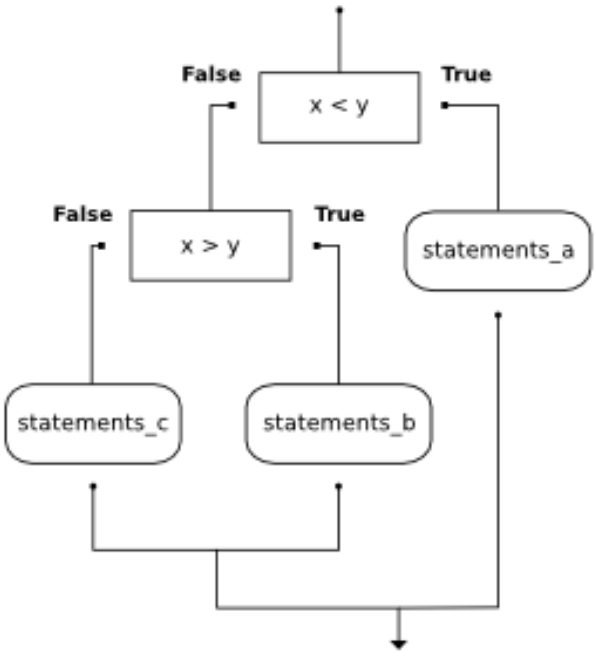
Nested conditional

Assume we have two integer variables, x and y. The following pattern of selection shows how we might decide how they are related to each other.

```
x=10
y=5
if x < y:
    print("x is less than y")  #first conditio check

else:   #we came to the otherwise and here we still think
        #there could be more possibilities so if we add another if-else block

    if x > y:  #this if block will be executed only when the starting if
               #is false and it enters the else part

        print("x is greater than y")
    else:
        print("x and y must be equal")
```

```
x is greater than y
```

The outer conditional contains two branches. The second branch (the else from the outer) contains another if statement, which has two branches of its own. Those two branches could contain conditional statements as well.The flow of control for this example can be seen in this flowchart illustration.



```python
x=10
y=10
if x < y:
    print("x is less than y")  #first conditio check

else:   #we came to the otherwise and here we still think
        #there could be more possibilities so if we add another if-else block

    if x > y:  #this if block will be executed only when the starting if
                #is false and it enters the else part

        print("x is greater than y")
    else:
        print("x and y must be equal")
```

```
x and y must be equal
```

***** KEY POINT:**
some programming languages, matching the if and the else is a problem. However, in Python this is not the case. The indentation pattern tells us exactly which else belongs to which if.
But the indentation might get complex when you have several if-else blocks so remember to meticulously keep an eye on all blocks

Chained conditionals

Sometimes there are more than two possibilities and we need more than two branches. which is where chained conditional structure come into picture.

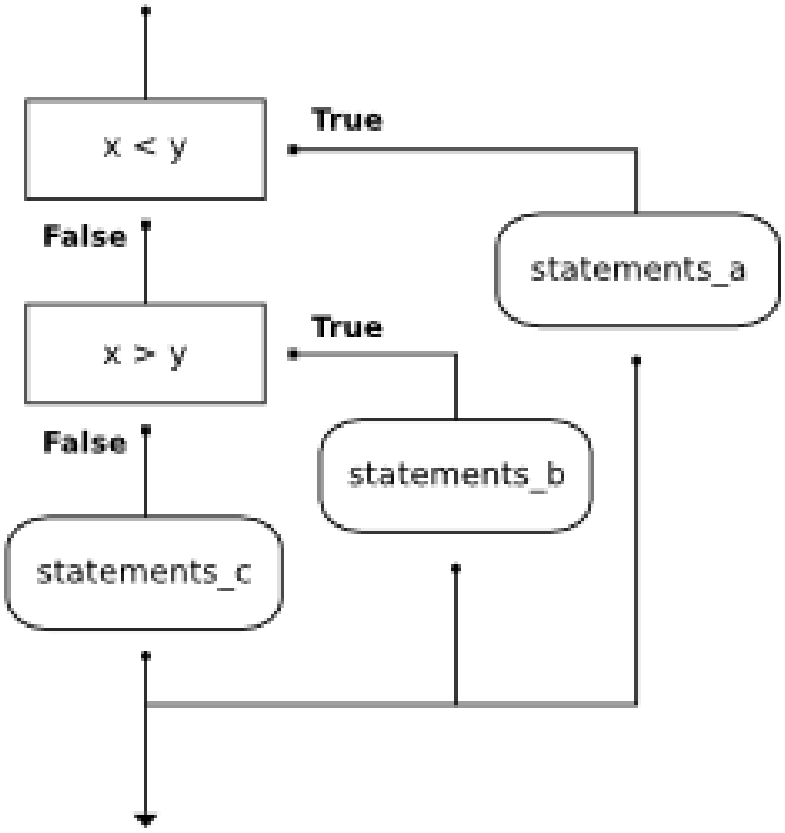
Python provides this alternative way to write nested selection

Let's see that with an example:

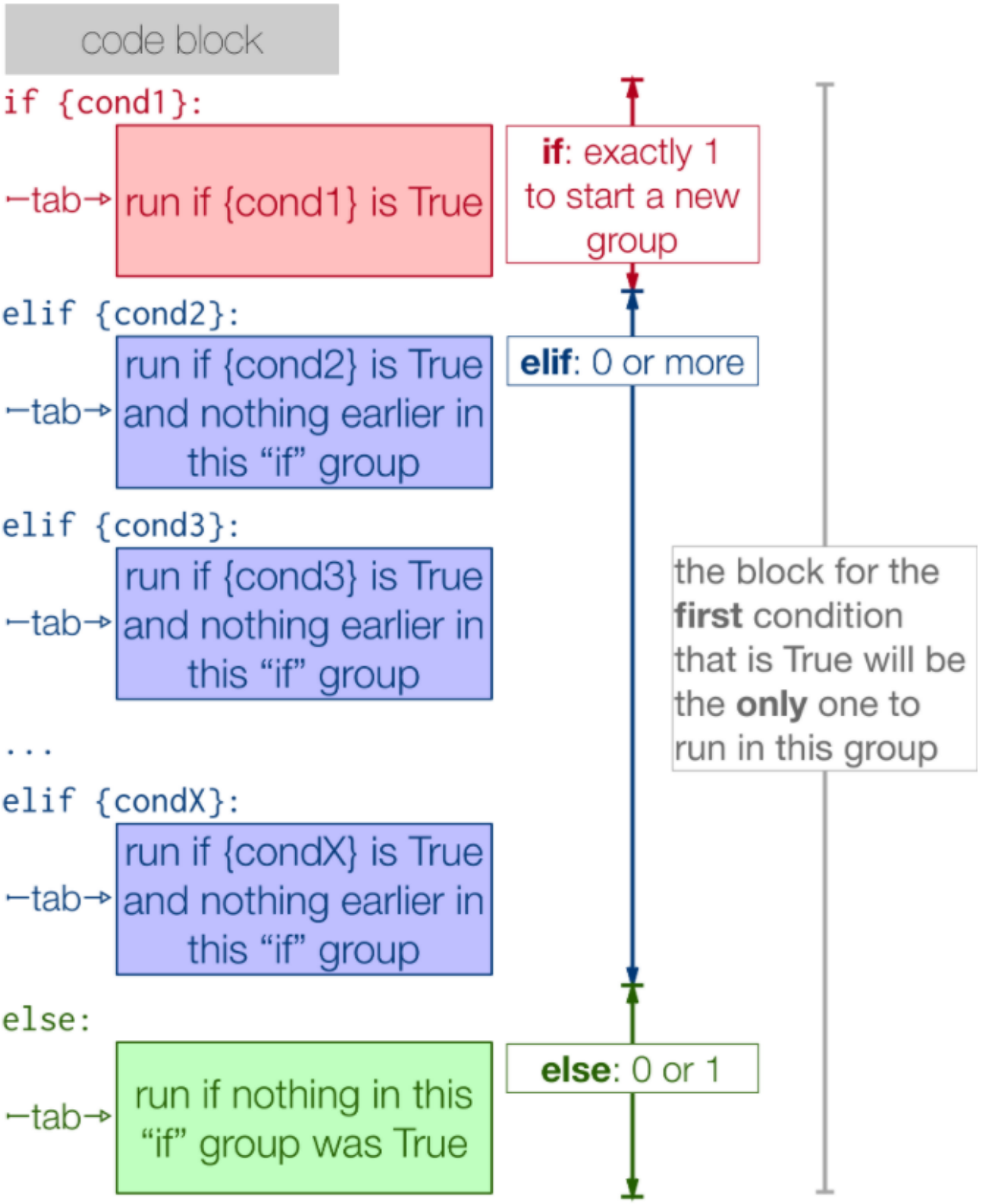
```
x=10
y=10
if x < y:  # this will be the first condition checked
    print("x is less than y")

elif x > y: #this is the interesting part
            #this condition works in 2 ways
            #else part for the previous if block
            #and also if part for the next else block
            #as what we saw in earlier examples
            #every else block needed if block

            #this will be the second condition checked
    print("x is greater than y")

else:       #if both above conditions turn false, this is the last block
            #which will get executed
    print("x and y must be equal")
```

```
x and y must be equal
```

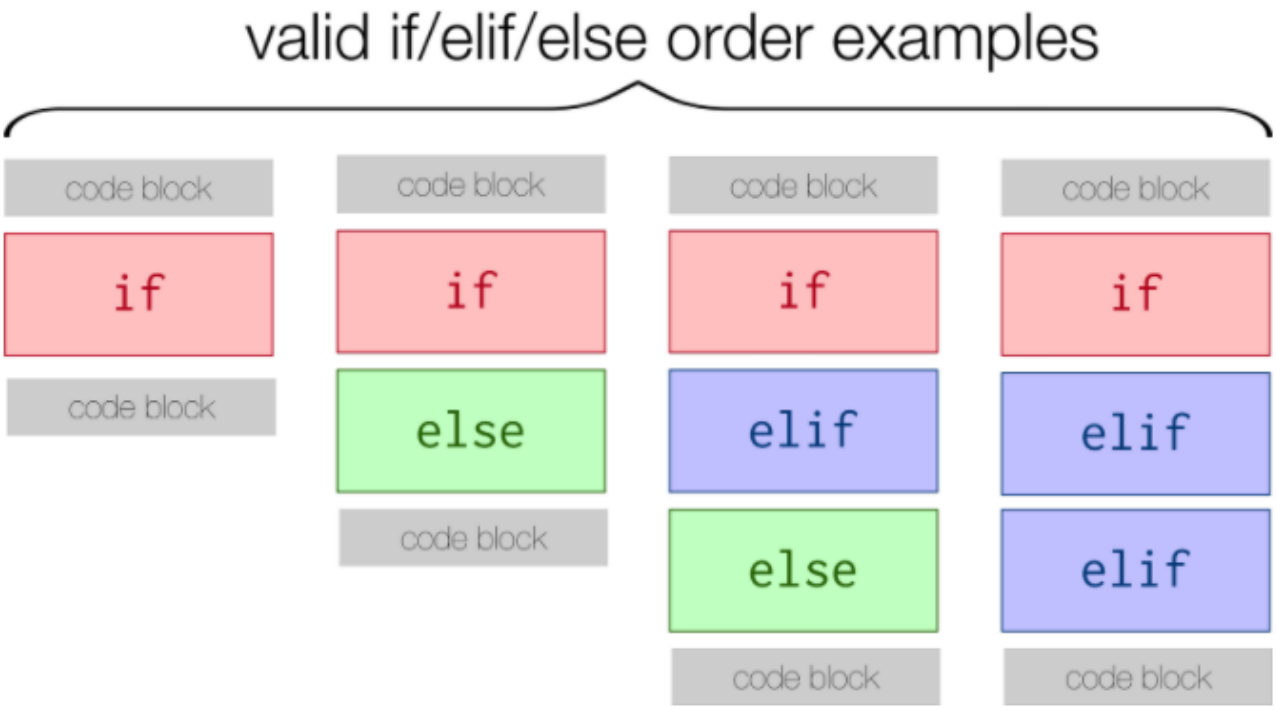
The flow of control can be drawn in a different orientation but the resulting pattern is identical to the one shown above



elif is an abbreviation of else if. Again, exactly one branch will be executed. There is no limit of the number of elif statements but only a single (and optional) final else statement is allowed and it must be the last branch in the statement.



The following image highlights different kinds of valid conditionals that can be used. Though there are other versions of conditionals that Python can understand (imagine an if statement with twenty elif statements), those other versions must follow the same order as seen below.



What will the following code print if x = 3, y = 5, and z = 2?

```
if x < y and x < z:
    print("a")
elif y < x and y < z:
    print("b")
else:
    print("c")
```

A. a

B. b

C. c

Which of I, II, and III below gives the same result as the following nested if?

```
# nested if-else statement
x = -10
if x < 0:
    print("The negative number ",  x, " is not valid here.")
else:
    if x > 0:
        print(x, " is a positive number")
    else:
        print(x, " is 0")
```

I.

```
if x < 0:
    print("The negative number ",  x, " is not valid here.")
else (x > 0):
    print(x, " is a positive number")
else:
    print(x, " is 0")
```

II.

```
if x < 0:
    print("The negative number ",  x, " is not valid here.")
elif (x > 0):
    print(x, " is a positive number")
else:
    print(x, " is 0")
```

III.

```
if x < 0:
    print("The negative number ",  x, " is not valid here.")
if (x > 0):
    print(x, " is a positive number")
else:
    print(x, " is 0")
```

A. I only
B. II only
C. III only
D. II and III
E. I, II, and III

Create an empty list called resps. Using the list percent_rain, for each percent, if it is above 90, add the string ‘Bring an umbrella.’ to resps, otherwise if it is above 80, add the string ‘Good for the flowers?’ to resps, otherwise if it is above 50, add the string ‘Watch out for clouds!’ to resps, otherwise, add the string ‘Nice day!’ to resps. Note: if you’re sure you’ve got the problem right but it doesn’t pass, then check that you’ve matched up the strings exactly.

percent_rain = [94.3, 45, 100, 78, 16, 5.3, 79, 86]

Answer:

```
percent_rain = [94.3, 45, 100, 78, 16, 5.3, 79, 86]
resps=[]
for i in percent_rain:
  if i>90:
    resps.append("Bring an umbrella.")
  elif i>80 and i<=90:
    resps.append("Good for the flowers?")
  elif i>50 and i<=80:
    resps.append("Watch out for clouds!")
  else:
    resps.append('Nice day!')
print(resps)
```

The Accumulator Pattern with Conditionals

Sometimes when we're accumulating, we don't want to add to our accumulator every time we iterate. Consider, for example, the following program which counts the number of letters in a phrase.

```
1 phrase = "Example of accumulator pattern with Conditionals"
2 tot = 0
3 for char in phrase:
4     if char != " ":
5         tot = tot + 1
6 print(tot)
```

```
43
```

- Here, we initialize the accumulator variable to be zero on line two.
- We iterate through the sequence (line 3).
- The update step happens in two parts. First, we check to see if the value of char is not a space. If it is not a space, then we update the value of our accumulator variable tot (on line 5) by adding one to it.
- If that conditional proves to be False, which means that char is a space, then we don't update tot and continue the for loop.
- We could have written **tot = tot + 1 or tot += 1**, either is fine.
- At the end, we have accumulated a the total number of letters in the phrase. **Without using the conditional**, we would have only been able to **count how many characters** there are in the string and **not been able to differentiate between spaces and non-spaces**.

We can use conditionals to also count if particular items are in a string or list. Let's see more examples:

```
vowel = "We will count the number of vowels this sentence has"
x = 0
for i in vowel:
    if i in ['a', 'e', 'i', 'o', 'u']:
        x += 1
print(x)

```

```
15
```

Accumulating the Max Value

We can also use the accumulation pattern with conditionals to find the maximum or minimum value. Instead of continuing to build up the accumulator value like we have when counting or finding a sum, we can reassign the accumulator variable to a different value.

The following example shows how we can get the maximum value from a list of integers.

```
nums = [9, 3, 8, 11, 5, 29, 2]
best_num = 0
for n in nums:
    if n > best_num:
        best_num = n
print(best_num)
```

```
29
```

- Here, we initialize best_num to zero, assuming that there are no negative numbers in the list.
- In the for loop, we check to see if the current value of n is greater than the current value of best_num.

- If it is, then we want to update best_num so that it now is assigned the higher number. Otherwise, we do nothing and continue the for loop.

You may notice that the current structure could be a problem. If the numbers were all negative what would happen to our code?
What if we were looking for the smallest number but we initialized best_num with zero?
To get around this issue, we can initialize the accumulator variable using one of the numbers in the list.

```
1  nums = [9, 3, 8, 11, 5, 29, 2]
2  best_num = nums[0]
3  for n in nums:
4      if n > best_num:
5          best_num = n
6  print(best_num)
```

```
29
```

let's try to see several programs that we can use to find the biggest number in list:

Way 1:
sort()/sorted()
Python lists have a built-in list.sort() method that modifies the list in-place. There is also a sorted() built-in function that builds a new sorted list from an iterable.

A simple ascending sort is very easy: just call the sorted() function. It returns a new sorted list:

```
nums = [9, 3, 8, 11, 5, 29, 2]
nums.sort()
print(nums[-1])
```

29

```
nums = [9, 3, 8, 11, 5, 29, 2]
print(sorted(nums)[-1])
```

29

Way 2 :
max()

Return the largest item in an iterable or the largest of two or more arguments.

```
nums = [9, 3, 8, 11, 5, 29, 2]
print(max(nums))
```

29

Way3:
When the user provides inputs and we have to find the max element in the list:

```python
# creating empty list
list1 = []

# asking number of elements to put in list
num = int(input("Enter number of elements in list: "))

# iterating till num to append elements in list
for i in range(1, num + 1):
    ele = int(input("Enter elements: "))
    list1.append(ele)

# print maximum element
print("Largest element is:", max(list1))
print("Largest element is:", sorted(list1)[-1])

```

```
Enter number of elements in list: 4
Enter elements: 23
Enter elements: 21
Enter elements: 4
Enter elements: 5
Largest element is: 23
Largest element is: 23
```

What is printed by the following statements?

```
list= [5, 2, 1, 4, 9, 10]
min_value = 0
for item in list:
  if item < min_value:
     min_value = item
print(min_value)
```

For each word in words, add ‘d’ to the end of the word if the word ends in “e” to make it past tense. Otherwise, add ‘ed’ to make it past tense. Save these past tense words to a list called past_tense.

words = ["adopt", "bake", "beam", "confide", "grill", "plant", "time", "wave", "wish"]

Answer:

```
words = ["adopt", "bake", "beam", "confide", "grill", "plant", "time", "wave", "wish"]
past_tense=[]
for i in words:
   if i[-1] == 'e':
      i=i+'d'
      past_tense.append(i)
   else:
      i=i+'ed'
      past_tense.append(i)
print(past_tense)
```

For each string in the list words, find the number of characters in the string. If the number of characters in the string is greater than 3, add 1 to the variable num_words so that num_words should end up with the total number of words with more than 3 characters.

words = ["water", "chair", "pen", "basket", "hi", "car"]

Answer:

```
words = ["water", "chair", "pen", "basket", "hi", "car"]
num_words=0
for i in words:
   if len(i)> 3:
      num_words +=1

print(num_words)
```

Mutability

A mutable object is an object whose state can be modified after it is created. It may or may not represent the same value during the execution of the program.

Some Python collection types - strings and lists so far - are able to change and some are not. If a type is able to change, then it is said to be mutable. If the type is not able to change then it is said to be immutable.

Life of Lists being Mutable

Unlike strings, **lists** are **mutable**. This means we can change an item in a list by accessing it directly as part of the assignment statement. Using the indexing operator (square brackets) on the left side of an assignment, we can update one of the list items.

```
1 lang = ["Java", "python", "Javascript"]
2 print(lang)
3
4 lang[0] = "python"
5 lang[-1] = "python"
6 print(lang)
```

```
['Java', 'python', 'Javascript']
['python', 'python', 'python']
```

we see that java/javascript was easily replaced by python in lang list.

An assignment to an element of a list is called **item assignmen**t.
Item assignment does **not** work for **strings**. because that **strings** are **immutable**.

```python
#we can easily replace several items at once in just one line
alist = ['a', 'b', 'c', 'd', 'e', 'f']
alist[1:3] = ['x','y'] #this command is slicing the list from
                       #index [1],[2] and replacing whatever the value was
                       #with x,y
                       #meaning alist[1] will be replaced from b to x
                       #and alist[2] from c to y
print(alist)

alist[1:3] =[] #we are removing all the elements here from index [1] to index[2]
               #so after running these commands element present on
               #these indexes will be free'ed and the list would shrink

print(alist)

alist[1:2]=['g','h'] #this will do 2 things
                     #replace the value at index[1]
                     #and as you see the we have 2 values
                     #so the other would go at index[2]
                     #and now the element on index[2] will shift to index[3]
                     # and so on and this all will happen internally
                     #you don't have to worry about it, thats the beauty of python

print(alist)

```

```
['a', 'x', 'y', 'd', 'e', 'f']
['a', 'd', 'e', 'f']
['a', 'g', 'h', 'e', 'f']
```

Being Immutable -Strings and Tuples

Strings:

One final thing that makes strings different from some other Python collection types is that you are not allowed to modify the individual characters in the collection.

After going through all the great features of mutable list, It is tempting to use the [] operator on the left side of an assignment, with the intention of changing a character in a string.

For example, in the following code, we would like to change the first letter of greeting.

```
greeting = "Hello, world!"
greeting[0] = 'J'             # ERROR!
print(greeting)

```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-29-635c1b9842b9> in <module>
      1 greeting = "Hello, world!"
----> 2 greeting[0] = 'J'             # ERROR!
      3 print(greeting)

TypeError: 'str' object does not support item assignment
```

Instead of producing the output Jello, world!, this code produces the runtime error TypeError: 'str' object does not support item assignment.

Strings are immutable, which means **you cannot change** an existing string.

But this isn't correct, you might end up in situations where you would want to change the value of string, how would you do it?

Python has got the answer to it too, but unlike lists, you wouldn't like the solution that much.

```
greeting = "Hello, world!"
newGreeting = 'J' + greeting[1:]
print(newGreeting)
print(greeting)             # same as it was

```

```
Jello, world!
Hello, world!
```

The solution here is to concatenate a new first letter onto a slice of greeting. **This operation has no effect on the original string.** We have to create a new string using the old one to get desired o/p

Tuples:

As with strings, if we try to use item assignment to modify one of the elements of a tuple, we get an error. In fact, that's the key difference between lists and tuples: tuples are like immutable lists. None of the operations on lists that mutate them are available for tuples. Once a tuple is created, it can't be changed.

```
1 tuple_lang= ('java','cpp','c','python','ruby','unix')
2 tuple_lang[1]
3 tuple_lang[1]='c#'
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-14-7fd4de934105> in <module>
      1 tuple_lang= ('java','cpp','c','python','ruby','unix')
      2 tuple_lang[1]
----> 3 tuple_lang[1]='c#'

TypeError: 'tuple' object does not support item assignment
```

List Element Deletion

Using slices to delete list elements can be awkward and therefore error-prone. Python provides an alternative that is more readable.
The del statement removes an element from a list by using its position.

```
1 lang = ['java', 'python', 'cpp']
2 del lang[0]
3 print(lang)
4
5 alist = ['a', 'b', 'c', 'd', 'e', 'f']
6 del alist[1:5]
7 print(alist)
8
```

```
['python', 'cpp']
['a', 'f']
```

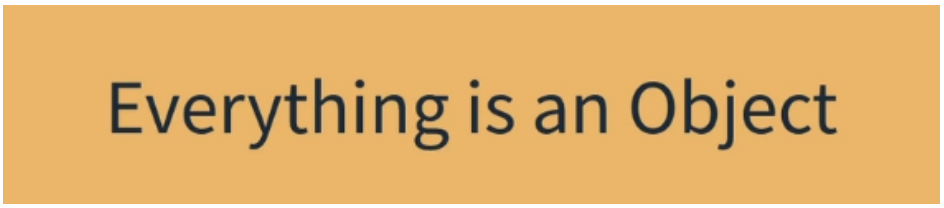
- As you might expect, del handles negative indices and causes a runtime error if the index is out of range.
- In addition, you can use a slice as an index for del. As usual, slices select all the elements up to, but not including, the second index.

```
lang = ['java', 'python', 'cpp']
del lang[-1]  #we are expecting to remove the last element
              #that is cpp from the list
print(lang)


```

```
['java', 'python']
```

Objects and References

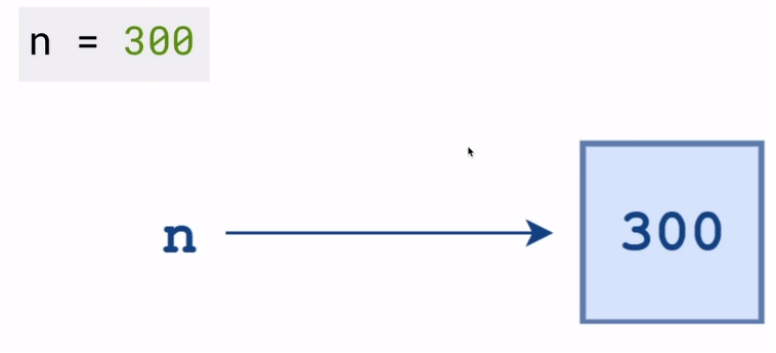
We need to feed this in our memory, consider the ROM memory of our brain. That in python

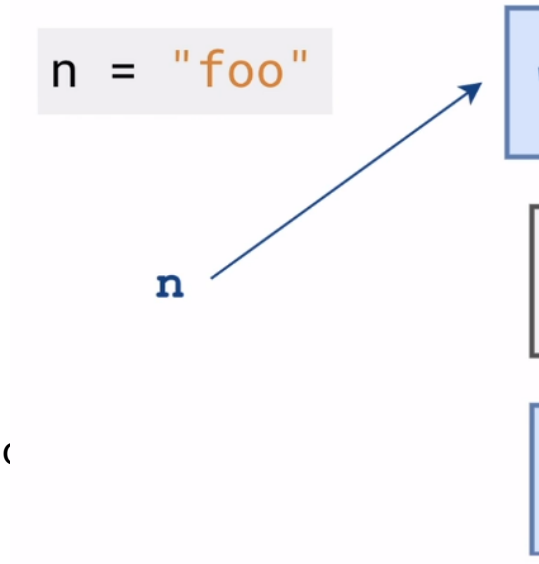


Integers, floats, strings, lists, tuples even functions are objects while variables are the references or if you have learned C++ are pointers.

Let's try to see with an example:

- Consider we assign a variable n with 300.



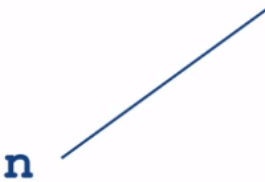


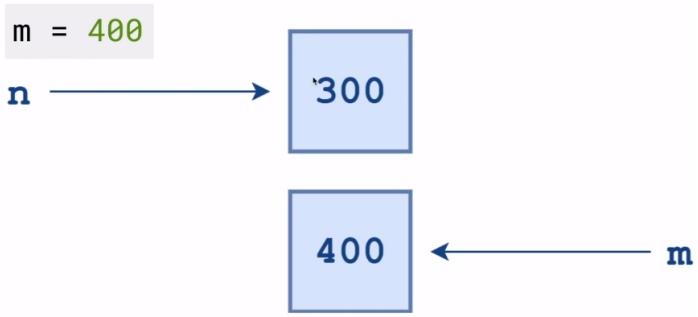
- Now the variable n is referring to a block of memo
  an integer object with value 300
- We now create another variable m
  >>> m = n
  - you cannot do n = m cause this would mean m is
    m but does not have any value , n has , and we n
    the value of n to m so if you try to do n=m , you v
    generating and error)

Garbage Colle

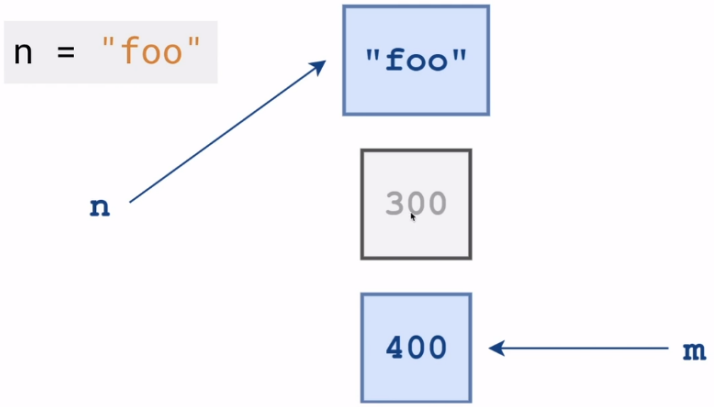
with above expression we now the variable m is also pointing to the same object of integer in the memory block.

- what if we change the value of m? will it change the value of n
  well? No, it won't cause we know variables are just references
  change the value of m it will start pointing to some other obje
  located at some other memory.



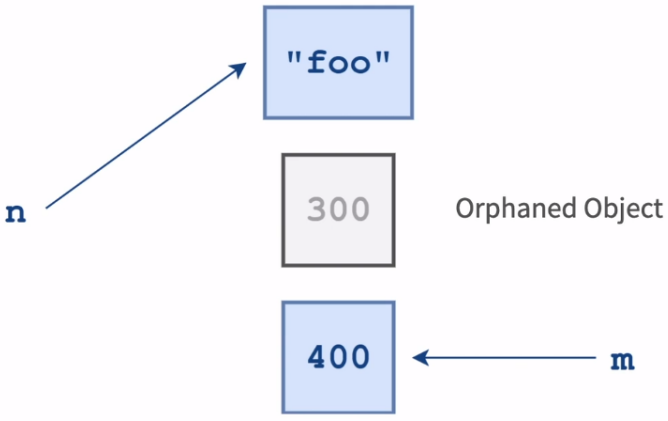


- Now the both variables are pointing to 2 different objects in the memory. let's try to assign variable n with some other value
-



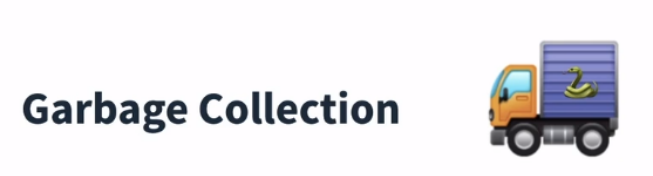
Garbage Collection

- Now the object with value 300 is being pointed by no variable meaning it has no source of way by which it can be retrieved or used hence the memory that it is occupying is wasting the space of Heap memory.



- It needs to be removed/cleaned so that the occupied memory get's free to be used by other objects, which is where concept of Garbage

collection comes into picture.



- Python immediately removes the orphaned/dead objects from the memory.

Understanding Variables/references in depth:

For people coming from other language backgrounds and mostly C/C++ , for them Python variables are fundamentally different

In fact, Python doesn't even have variables(**alarming line for you**). Python has **names, not variables**.This might seem pedantic, and for the most part, it is.
Most of the time, it's perfectly acceptable to think about **Python names as variables**, but **understanding** the **difference** is **important**. This is especially true when you're navigating the tricky subject of pointers in Python.

To help drive home the difference, you can take a look at how variables work in C, what they represent, and then contrast that with how names work in Python.

Variables in C
Let's say you had the following code that defines the variable x

```
int x = 2337;
```

This one line of code has several, distinct steps when executed:

- Allocate enough memory for an integer.
- Assign the value 2337 to that memory location
- Indicate that x points to that value

Shown in a simplified view of memory, it might look like this:

X	
Location	0x7f1
Value	2337

Here, you can see that the variable x has a fake memory location of 0x7f1 and the value 2337. If, later in the program, you want to change the value of x, you can do the following:

>>x = 2338;

The above code assigns a new value (2338) to the variable x, thereby overwriting the previous value. This means that the variable x is mutable. The updated memory layout shows the new value:

X	
Location	0x7f1
Value	2338

Notice that the location of x didn't change, just the value itself. This is a significant point. It means that x is the memory location, not just a name for it.

******* *Another way to think of this concept is in terms of ownership. In one sense, x owns the memory location. x is, at first, an empty box that can fit exactly one integer in which integer values can be stored.*

When you assign a value to x, you're placing a value in the box that x owns. If you wanted to introduce a new variable (y), you could add this line of code:

>> int y = x;

X	
Location	0x7f1
Value	2338

Y	
Location	0x7f5
Value	2338

Notice the new location 0x7f5 of y. Even though the value of x was copied to y, the variable y owns some new address in memory. Therefore, you could overwrite the value of y without affecting x:

>>y = 2339;

X	
Location	0x7f1
Value	2338

Y	
Location	0x7f5
Value	2339

Again, you have modified the value at y, but not its location. In addition, you have not affected the original x variable at all. This is in stark contrast with how Python names work.

*** *Another way to think of this concept is in terms of ownership. In one sense, x owns the memory location. x is, at first, an empty box that can fit exactly one integer in which integer values can be stored.*

When you assign a value to x, you're placing a value in the box that x owns. If you wanted to introduce a new variable (y), you could add this line of code:

>> int y = x;

X	
Location	0x7f1
Value	2338

Y	
Location	0x7f5
Value	2338

Notice the new location 0x7f5 of y. Even though the value of x was copied to y, the variable y owns some new address in memory. Therefore, you could overwrite the value of y without affecting x:

>>y = 2339;

X	
Location	0x7f1
Value	2338

Y	
Location	0x7f5
Value	2339

Again, you have modified the value at y, but not its location. In addition, you have not affected the original x variable at all. This is in stark contrast with how Python names work.

Variables/Names in Python:

Python does not have variables. It has names. Yes, this is a pedantic point, and you can certainly use the term variables as much as you like. It is important to know that there is a difference between variables and names.

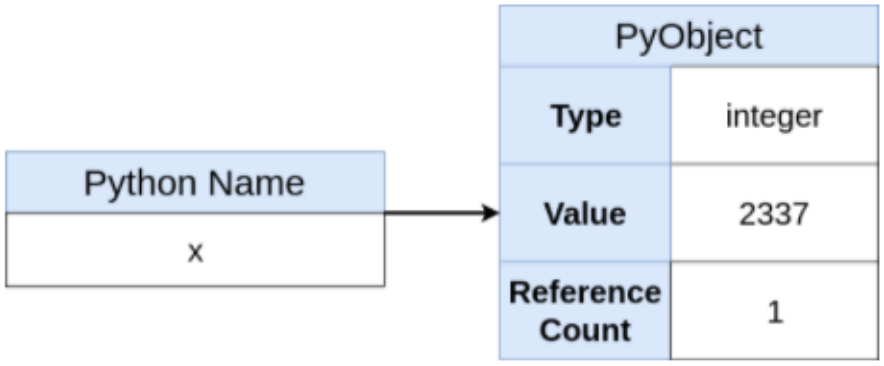
Let's take the equivalent code from the above C example and write it in Python:
>>> x = 2337

Much like in C, the above code is broken down into several distinct steps during execution:

- Create a PyObject
- Set the typecode to integer for the PyObject
- Set the value to 2337 for the PyObject
- Create a name called xPoint x to the new PyObject
- Increase the refcount of the PyObject by 1 (this object will be deleted as soon as this refcount gets 0 which will become when the current x will start pointing to some other value)

*****KEY POINT:**
The PyObject is not the same as Python's object. It's specific to CPython and represents the base structure for all Python objects.PyObject is defined as a C struct, so if you're wondering why you can't call typecode or refcount directly, its because you don't have access to the structures directly. Method calls like sys.getrefcount() can help get some internals.

In memory, it might looks something like this:



You can see that the memory layout is vastly different than the C layout from before. Instead of x owning the block of memory where the value 2337 resides, the newly created Python object owns the memory where 2337 lives. The Python name x doesn't directly own any memory address in the way the C variable x owned a static slot in memory.

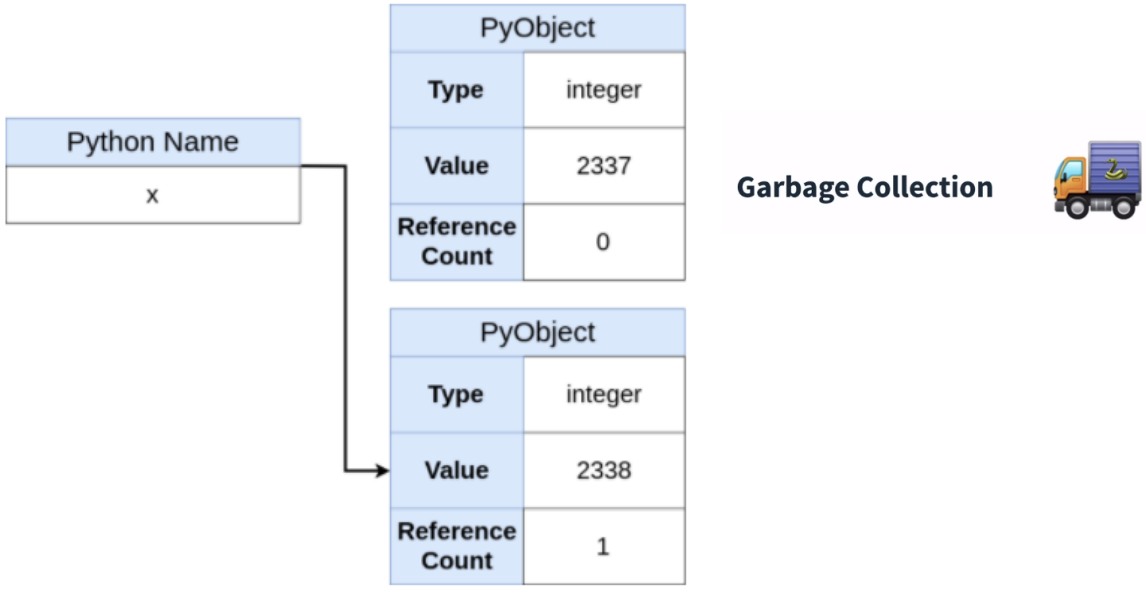
If you were to try to assign a new value to x, you could try the following:

>>> x=2338

What's happening here is different than the C equivalent, but not too different from the original bind in Python.This code:

- Creates a new PyObject
- Sets the typecode to integer for the PyObject
- Sets the value to 2338 for the PyObject
- Points x to the new PyObject
- Increases the refcount of the new PyObject by 1
- Decreases the refcount of the old PyObject by 1 (Garbage collector will be active and flush the memory of the old int object with value 2337)

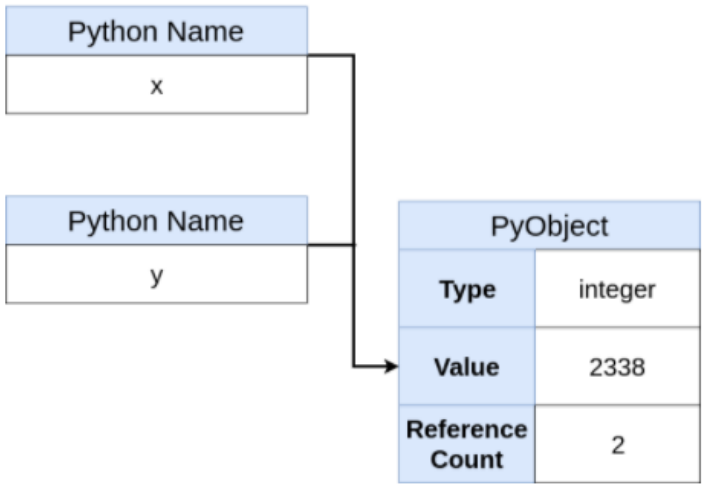
Now in memory, it would look something like this:



This diagram helps illustrate that x points to a reference to an object and doesn't own the memory space as before. It also shows that the x = 2338 command is not an assignment, but rather binding the name x to a reference.In addition, the previous object (which held the 2337 value) is now sitting in memory with a ref count of 0 and will get cleaned up by the garbage collector.

You could introduce a new name, y, to the mix as in the C example:
>>> y=x (you should remember why not x=y)

In memory, you would have a new name, but not necessarily a new object:

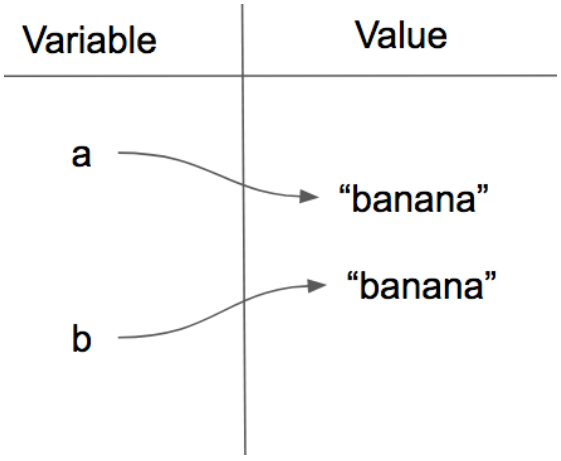


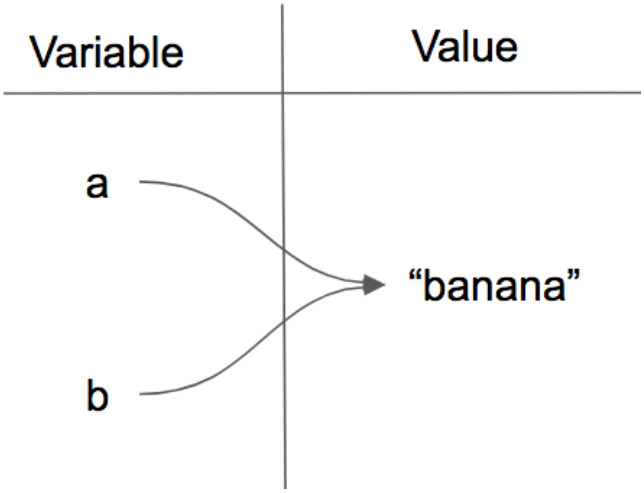
Let's see another example but without considering memory management:

```
a="banana"
b="banana"
```

we know that a and b will refer to a string with the letters "banana".
But we don’t know yet whether they point to the same string.

There are two possible ways the Python interpreter could arrange its internal states:





- In one case, a and b refer to two different string objects that have the same value.
- In the second case, they refer to the same object.
- We clearly now know,that a variable refers to an object.

We can **test** whether **two names refer** to the **same object** using the **is** operator. The **is operator** will **return** true **if the two references** are to the **same** object.
In other words, the references are the same.

```
1  a="banana"
2  b="banana"
3
4  print(a is b)
```

```
True
```

- The answer is True.
- This tells us that both **a and b refer** to the **same object**, and that it is the second of the two reference diagrams that describes the relationship.
- Python assigns every object a unique id and when we ask a is b what python is really doing is checking to see **if id(a) == id(b).**

```
1  a="banana"
2  b="banana"
3
4  print(a is b)
5  print(id(a))
6  print(id(b))
```

```
True
2538723654096
2538723654096
```

- Since strings are immutable, the **Python interpreter** often **optimizes resources** by making **two names** that **refer** to the **same string value** refer to the **same object**.

- You shouldn't count on this (that is, use == to compare strings, not is), but don't be surprised if you find that two variables,each bound to the string "banana", have the same id.

- This is **not the case with lists**, which **never share** an **id** just **because** they have the **same contents**.
- Consider the following example. Here, a and b refer to two different lists, each of which happens to have the same element values. They need to have different ids so that mutations of list a do not affect list b.

```
a=["banana","apple","orange"]
b=["banana","apple","orange"]

print(a is b)
print(a == b)
print(id(a))
print(id(b))
```

```
False
True
2538724356360
2538724356232
```

Aliasing

Since variables refer to objects, if we assign one variable to another, both variables refer to the same object.

Aliasing + Mutable objects creates a lot of confusion sometimes.

Why?
let's see few examples

```
a=["banana","apple","orange"]
b=a
print(a)
print(b)
print (a is b)
print(a == b)

```

```
['banana', 'apple', 'orange']
['banana', 'apple', 'orange']
True
True
```

The same list has two different names after b =a , this means that those 2 variables are aliased.

- Changes made with one alias affect the other. So you are writing a big 1000 lines of code and you are using a alias variable but you forget that it was alias of another variable and you try to modify the values of the object to which that reference variable, it will create problems for the other variable which is being used at some other place in the code.

- This will happen mostly with the objects which are mutable which is most likely to happen with List objects. Let's see an example:

```
a=["banana","apple","orange"]
b=["banana","apple","orange"]

print (a is b)
print(a == b)

b=a
print (a is b)
print(a == b)
print(id(a))
print(id(b))

b[1]="grapes"
print(a)

```

```
False
True
True
True
2538722700552
2538722700552
['banana', 'grapes', 'orange']
```

- Suppose your mother makes 2 similar lists of fruits, one for you and one for your other brother/sister which needs to be bought and sent to you both as you are staying away from home at some college hostel.

- After sometime she realizes that the fruit list of kid b i.e your sibling should not have apples but grapes cause he/she likes grapes but on the contrary, you have an allergy with grapes but that is known only by your mother as she has been taking care of all the fruits.

- As both lists are alias which your mother mistakenly forgot and she made changes in list of kid b but the same got reflected in list a.

- Now banana, grapes and oranges were delivered to you both, you both ravishly ate all the fruits but the difference was your sibling was contented eating those fruits and you ended up lying on bed with medications for your allergy.

So, when you do aliasing, try to remember that any changes in reference variable will changes in the object being pointed by some other variable too.

Cloning lists

But we cannot keep on writing a new similar list pointed by some another variable, fearing that if we create another object using alias it would mess up your code.

```
grocery_list=["banana","apple","orange","pineapple","pears","lentils","sugar","salt","Milk"]
```

I will not write this list again if i need to make a copy of it. So what can we do?

We can use slicing operator to close. lets see how

```
grocery_list=["banana","apple","orange","pineapple","pears","lentils","sugar","salt","Milk"]

fruits_others = grocery_list[:]

print (grocery_list is fruits_others)
print(grocery_list == fruits_others)

print(id(grocery_list))
print(id(fruits_others))

fruits_others[1]="grapes"
print(grocery_list)

```

```
False
True
2538724359944
2538723613128
['banana', 'apple', 'orange', 'pineapple', 'pears', 'lentils', 'sugar', 'salt', 'Milk']
```

Now we are free to make changes to fruits_others without worrying about grocery_list.

Mutating Methods

We have seen some of the methods like count and index already while going through previous topics and understanding them with examples.

Methods are either mutating or non-mutating.
Mutating methods are ones that change the object after the method has been used.
Non-mutating methods do not change the object after the method has been used.

count and index these 2 methods we have already seen earlier and these are non-mutating as they are not changing anything in the object.

- Count returns the number of occurrences of the argument given but does not do anything to the original string or list.
- Similarly, index returns the leftmost occurrence of the argument but does not change the original string or list.

List Methods

The dot operator can also be used to access built-in methods of list objects. append is a list method which adds the argument passed to it to the end of the list.

```
mylist = []          #creating an empty list
mylist.append(5)     #we append, appends the list from right end
                     #but this is an empty list so 5 would be appended to the list
mylist.append(27)    #27 will be appended at list[-1]

mylist.append(3)     #3 will be same, now the list is ['5','27','3']
mylist.append(12)    #12 will be same, now the list is ['5','27','3',12]
print(mylist)

mylist.insert(1, 12) #syntax =insert(pos,value)
                      #unlike assigning a value at index, insert method
                      #creates a new blank entry at the defined position
                      #which is 1 here and shift value of list[1] to list[2]
                     #and so on for all the elements
                     #so after running insert the list would come out as
                     #  ['5','12','27','3',12]

print(mylist)
print(mylist.count(12)) #we already konw this method will give 2 as o/p

print(mylist.index(3))  #this will find '3' in the list and return the index
                        #of its first occurence
```

```
[5, 27, 3, 12]
[5, 12, 27, 3, 12]
2
3
```

```
mylist.reverse()  #as the name suggest , it will reverse the list

print(mylist)

mylist.sort()    #as the name of method itself suggests, it will return the list
                #arranged in default ascending order
print(mylist)


#The sort() method accepts a reverse parameter as an optional argument.
mylist.sort(reverse=True)  ##Setting reverse = True sorts the list in the descending order
print(mylist)

mylist.remove(5) #remove as the name suggests will remove the first occurence of
                 #given argument which is 5
print(mylist)

lastitem = mylist.pop() #pop will pop the last index value i.e list[-1]
                        # and the list will be reduced in size
print(lastitem)
print(mylist)

```

```
[12, 3, 27, 12, 5]
[3, 5, 12, 12, 27]
[27, 12, 12, 5, 3]
[27, 12, 12, 3]
3
[27, 12, 12]
```

> **Pop Method:** There are two ways to use the pop method. The first, with no parameter which we used in above example, will remove and return the last item of the list. If you provide a parameter for the position, pop will remove and return the item at that position. Either way the list is changed.

The below image provides a summary of the list methods. The column labeled result gives an explanation as to what the return value is as it relates to the new value of the list.

The **word mutator** means that the list is changed by the method but nothing is returned (actually None is returned).

A hybrid method is one that not only changes the list but also returns a value as its result.

Finally, if the result is simply a return, then the list is unchanged by the method.Be sure to experiment with these methods to gain a better understanding of what they do.
(https://docs.python.org/3/library/stdtypes.html#sequence-types-str-bytes-bytearray-list-tuple-range)

Method	Parameters	Result	Description
append	item	mutator	Adds a new item to the end of a list
insert	position, item	mutator	Inserts a new item at the position given
pop	none	hybrid	Removes and returns the last item
pop	position	hybrid	Removes and returns the item at position
sort	none	mutator	Modifies a list to be sorted
reverse	none	mutator	Modifies a list to be in reverse order
index	item	return idx	Returns the position of first occurrence of item
count	item	return ct	Returns the number of occurrences of item
remove	item	mutator	Removes the first occurrence of item

***KEY POINT:**
It is important to remember that methods like **append, sort, and reverse** all **return** None. They change the list; they don't produce a new list.
So, while we did reassignment to increment a number, as in x = x + 1, doing the analogous thing with these operations will lose the entire list contents

```
sort_eg=[23,134,11,531]
print(sort_eg.sort()) #will return none as we know sort will change the list
                      #it will not return anything, if you want to see
                     #the changes, you need to run print(list_name)
print(sort_eg)
new_sort=sort_eg.sort(reverse=True)
print(new_sort)  #this value has None, because we know sort returns None
                #so do not try to make this mistake of assigning a list
                #to a variable while running the sort/append/reverse methods
                #it will be a big blunder
print(sort_eg)
```

```
None
[11, 23, 134, 531]
None
[531, 134, 23, 11]
```

Append vs Concatenate

We have used **append** and **concatenate** a lot by now. append method adds a new item to the end of a list while we have seen concatenate (+) operator while using strings, but we have also seen in examples how concatenate could be too be use elements to the list.

So you mean append and concatenate both are doing the same?

Well yes and no, yes because concatenate does add the new element at the end of the list like append but no because concatenate makes entirely a new list, so the old list if no other variable is pointing will become an orphaned object and cleaned by the garbage collection of Python. Let's see it with examples:

```
1 origlist = [45,32,88]
2 print(id(origlist))
3 origlist.append("append a new string")
4 print(origlist)
5 print(id(origlist))
```

```
2538729726728
[45, 32, 88, 'append a new string']
2538729726728
```

- Observations from the above example shows that we appended a new string to origlist but still the id did not change
- Which means the list object stayed the same, changes were done on the main list object

Let's see how concatenate behaves:

```
1 origlist = [45,32,88]
2 print(id(origlist))
3 origlist= origlist + ["append a new string"]
4 print(origlist)
5 print(id(origlist))
```

```
2538729707592
[45, 32, 88, 'append a new string']
2538727845640
```

- The ID's i.e identifiers was not the same after we concatenated the new string, which means concatenation is creating a new object all together and the old object to which origlist was pointing to will be removed

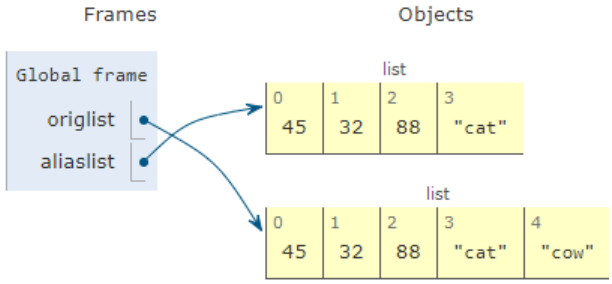
This might be difficult to understand since the o/p of both lists be it append or concatenate appears same.

But now with the id function which takes a single parameter, the object that you are interested to know details about should have made it

clear that they are not the same.

*****KEY POINT:**
We have previously described x += 1 as a shorthand for x = x + 1. With lists**, += is actually a little different.** In particular, origlist += [“append a new string”] appends “append a new string” to the end of the original list object. If there is **another alias for origlist**, this can make a **difference**

```
1  origlist = [45,32,88]
2  aliaslist = origlist
3  origlist += ["cat"]
4  origlist = origlist + ["cow"]
```



Non-mutating Methods on Strings

```
1  best_line= "Hello, World"
2  print(best_line.upper())
3  print(best_line.lower())
4  print(best_line)
5
```

```
HELLO, WORLD
hello, world
Hello, World
```

- In above example, you would see upper method was invoked by our string object and it created a new string in which all the characters were in uppercase.
- lower works in a similar fashion changing all characters in the string to lowercase. (The original string ss remains unchanged)

Method	Parameters	Description
upper	none	Returns a string in all uppercase
lower	none	Returns a string in all lowercase
count	item	Returns the number of occurrences of item
index	item	Returns the leftmost index where the substring item is found and causes a runtime error if item is not found
strip	none	Returns a string with the leading and trailing whitespace removed
replace	old, new	Replaces all occurrences of old substring with new
format	substitutions	Involved! See String Format Method, below

```
1  best_line = "    Hello, World    "  #this line has white spaces before and after
2                                      #Hello, World , will see why
3
4  l_count = best_line.count("l")   #count method works the same way as it worked with
5                                   #lists, it will search the given argument in the
6                                   #entire string and give you the number of its occurence
7  print(l_count)
8
9  print(best_line.strip())  #strip method helps you remove all the white spaces,
10                          #it works same as the The LTRIM() and RTRIM function
11                         #of SQL removing leading spaces from a string.
12
13 print("$$$" + best_line.strip() + "$$$")
14
15
16 greet = best_line.replace("o", "***") #will replace o with ***
17 print(greet)
18
```

```
3
Hello, World
$$$Hello, World$$$
    Hell***, W***rld
```

You should experiment with these methods so that you understand what they do. **Note once again that the methods that return strings do not change the original**.

You can also consult the Python documentation for strings. https://docs.python.org/3/library/stdtypes.html#string-methods

You should experiment with these methods so that you understand what they do. **Note once again that the methods that return strings do not change the original**.

You can also consult the Python documentation for strings.
https://docs.python.org/3/library/stdtypes.html#string-methods

```
1  s = "python rocks"
2  print(s[1])
3  print(s.index("n"))
4  print(s[1]*s.index("n"))
5
```

```
y
5
yyyyy
```

Try to understand in your brain what would have happened at interpreter level for this problem.

String Format Method

If you are coming from C/C++ background, below program would look quite fresh to you when you would have started learning it:

```
1  #include <iostream>
2  using namespace std;
3  int main()
4  {
5      float var = 37.66666;
6
7      // Directly print the number with .2f precision
8      printf("%.2f", var);
9      return 0;
10 }
```

returns : 37.66

```cpp
#include <iostream>
#include <string>

int main() {
    std::string str = "123.4567";

    // convert string to float
    int num_int = std::stof(str);

    // convert string to double
    float num_float = std::stod(str);

   std:: cout<< "num_int = " << num_int << " and num_float =" << num_float << std::endl;

    return 0;
}
```

Output

```
num_int = 123 and num_float =123.457
```

std:: cout<< "num_int = " << num_int << " and num_float =" << num_float << std::endl;

So much to write for getting an output like:
>>num_int = 123 and num_float =123.457

But wait, let's try to get the same output in python:

```python
str = "123.4567"
num_float =float(str)
num_int=int(num_float)

print("num_int= ",num_int,"num_float =",num_float )

```

```
num_int=  123 num_float = 123.4567
```

Comparatively better than C/C++ but what if we want to assign the output to a string?

lets see:

```
input_str = "123.4567"
num_float =float(input_str)
num_int=int(num_float)


output= "num_inst =" + str(num_int) + " num_float =" + str(num_float)
print(output)
```

```
num_inst =123 num_float =123.4567
```

- this looks a little extra now, we have to concatenate and then convert the input to strings and then finally refer it via a variable

Let's see another example:

```
ranks = [("Python", 1), ("Java", 2), ("C#", 3)]
for lang in ranks:
    name = lang[0]
    score = lang[1]
    print("the rank of " + name + " as per current stats is " + str(score))
```

```
the rank of Python as per current stats is 1
the rank of Java as per current stats is 2
the rank of C# as per current stats is 3
```

we have created strings with variable content using the + operator to concatenate partial strings together. That works, but the problem here is that for some people it might be very hard to read or debug a code line that includes variable names and strings and complex expressions

What could we do to make these same programs more like readable and easy to understand?

```
ranks = [("Python", 1), ("Java", 2), ("C#", 3)]
for lang in ranks:
    name = lang[0]
    score = lang[1]
    print("the rank of {} as per current stats is {}".format(name,score))
```

```
the rank of Python as per current stats is 1
the rank of Java as per current stats is 2
the rank of C# as per current stats is 3
```

- Now the code looks much better
- I also know what is actually going to print

In grade school quizzes a common convention is to use fill-in-the blanks. For instance,
Hello _____!
and you can fill in the name of the person greeted, and combine given text with a chosen insertion.
We use this as an analogy: Python has a similar construction, better called fill-in-the-braces.
The string method format, makes substitutions into places in a string enclosed in braces

```
person = input('Your name: ')
greeting = 'Hello {}!'.format(person)
print(greeting)

```

```
Your name: python
Hello python!
```

There are several **new ideas here!**

The **string** for the **format method** has a **special form**, with **braces embedded**. Such a string is called **a format string**.

Places where braces are embedded are replaced by the value of an expression taken from the parameter list for the format method.

There are many variations on the syntax between the braces.
In this case we use the syntax where the first (and only) location in the string with braces has a substitution made from the first (and only) parameter.

In the code above, this new string is assigned to the identifier greeting, and then the string is printed.The identifier greeting was introduced to break the operations into a clearer sequence of steps.

However, since the value of greeting is only referenced once, it can be eliminated with the more concise version:

```
1 person = input('Enter your name: ')
2 print('Hello {}!'.format(person))
```

```
Enter your name: Python
Hello Python!
```

There can be multiple substitutions, with data of any type. Next we use floats.

```
1 origPrice = float(input('Enter the original price: $'))
2 discount = float(input('Enter discount percentage: '))
3 newPrice = (1 - discount/100)*origPrice
4 calculation = '${} discounted by {}% is ${}.'.format(origPrice, discount, newPrice)
5 print(calculation)
```

```
Enter the original price: $99.99
Enter discount percentage: 9.99
$99.99 discounted by 9.99% is $90.000999.
```

- It is **important** to **pass arguments** to the **format method** in the **correct order**, because they are **matched positionally** into the **{} places** for interpolation where there is more than one.

- If you see the output , result is not satisfying.
- Prices should appear with exactly two places beyond the decimal point, but that is not the default way to display floats.
- Format strings can give further information inside the braces showing how to specially format data.
- In particular floats can be shown with a specific number of decimal places. For two decimal places, put :.2f

We saw below program of C/C++ at the very start of this format string method section

```c
#include <iostream>
using namespace std;
int main()
{
    float var = 37.66666;

    // Directly print the number with .2f precision
    printf("%.2f", var);
    return 0;
}
```

returns : 37.66

Let's see the same precision with Python in a better way:

```python
origPrice = float(input('Enter the original price: $'))
discount = float(input('Enter discount percentage: '))
newPrice = (1 - discount/100)*origPrice
calculation = '${:.2f} discounted by {}% is ${:.2f}.'.format(origPrice, discount, newPrice)
print(calculation)
```

```
Enter the original price: $99.99
Enter discount percentage: 9.99
$99.99 discounted by 9.99% is $90.00.
```

- The 2 in the format modifier can be replaced by another integer to round to that specified number of digits.

- This kind of format string depends directly on the order of the parameters to the format method.

It is also **important** that you **give format** the **same amount of arguments** as there are **{} waiting for interpolation in the string**.

If you have **a {} in a string** that you **do not pass** arguments for, you will get **get an error** if you are using the **latest python version** but in **older versions** you may **not get an error**, but you will see a **weird undefined value** you probably did not intend suddenly inserted into your string.

Lets see it in an example:

```
1  name = "Python"
2  greeting = "You are the best"
3  s = "Hello, {}. {}."
4
5  print(s.format(name,greeting)) # will print Hello, Python. you are the best.
6
7  print(s.format(greeting,name)) # will print Hello, you are the best. Python.
8
9  print(s.format(name)) # 2 {}s, only one interpolation item! Not ideal.
10
```

```
Hello, Python. You are the best.
Hello, You are the best. Python.

---------------------------------------------------------------------------
IndexError                                Traceback (most recent call last)
<ipython-input-9-ca0eb8d9ba35> in <module>
      7 print(s.format(greeting,name)) # will print Hello, you are the best. Python.
      8
----> 9 print(s.format(name)) # 2 {}s, only one interpolation item! Not ideal.

IndexError: tuple index out of range
```

*****KEY POINT:**

Since braces have special meaning in a format string, there must be a special rule if you want braces to actually be included in the final formatted string.
The rule is to double the braces: {{ and }}.

Accumulator Pattern + Lists

We can accumulate values into a list rather than accumulating a single numeric value.

```
1 nums = [3, 5, 8]
2 accum = []
3 for w in nums:
4     x = w**2
5     accum.append(x)
6 print(accum)
```

```
[9, 25, 64]
```

- we initialize the accumulator variable to be an empty list, on line 2.
- We iterate through the sequence (line 3).
- On each iteration we transform the item by squaring it (line 4).
- The update step appends the new item to the list which is stored in the accumulator variable (line 5).
- The update happens using the .append(), which **mutates** the **list rather than using a reassignment.**
- **accum = accum + [x], or accum += [x]** would have given us the same output.
- In either case, we'd need to concatenate a list containing x, not just x itself.
- At the end, we have accumulated a new list of the same length as the original, but with each item transformed into a new item.
- **This is called a mapping operation,** will see it later

Try solving below problems:

Question 1:
Get an [8,5,14,9,6,12] from below list and assign it to a new variable:
lst= [3,0,9,4,1,7]

Question 2:
For each word in the list verbs, add an -ing ending. Save this new list in a new list, ing.

verbs = ["kayak", "cry", "walk", "eat", "drink", "fly"]

Question 3:
Given the list of numbers, numbs, create a new list of those same numbers increased by 5. Save this new list to the variable newlist.

numbs = [5, 10, 15, 20, 25]

Accumulator Pattern + Strings

```
1  s = input("Enter some text")
2  ac = ""
3  for c in s:
4      ac = ac + c + "-" + c + "-"
5
6  print(ac)
7
```

```
Enter some textPython
P-P-y-y-t-t-h-h-o-o-n-n-
```

- Look carefully at line 4 in the above program **(ac = ac + c + "-" + c + "-").**

- In words, it says that the **new value of ac will be the old value of ac concatenated with** the **current character**, a dash, then the current character and a dash again.

- We are building the result string character by character.Take a close look also at the initialization of ac. We start with an empty string and then begin adding new characters to the end.

- Also note that I have given it a different name this time, ac instead of accum. There's nothing magical about these names. You could use any valid variable and it would work the same (try substituting x for ac everywhere in the above code).

Question:

Take input from the user and print the input in CAPITAL letters and reverse order.

Answer:

```
s = input("your input)"
r = ""
for item in s:
  r = item.upper() + r
print(r)
```

Question:
Assign an empty string to the variable output. Using the range function, write code to make it so that the variable output has 35 a s inside it (like "aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa").
Hint: use the accumulation pattern!

Question:
For each character in the string already saved in the variable str1, add each character to a list called chars.

str1 = "I love python"
Note : assign it in both ways

- using accum list variable
- using different methods we learned above

THINGS LEARNED :

how to iterate through a list:

```
colors = ["Red", "Orange", "Yellow", "Green", "Blue", "Indigo", "Violet"]

for color in colors:
    print(color)
```

```
Red
Orange
Yellow
Green
Blue
Indigo
Violet
```

Accumulate a list by appending or deleting items!

```
colors = ["Red", "Orange", "Yellow", "Green", "Blue", "Indigo", "Violet"]
initials = []

for color in colors:
    initials.append(color[0])

print(initials)

```

```
['R', 'O', 'Y', 'G', 'B', 'I', 'V']
```

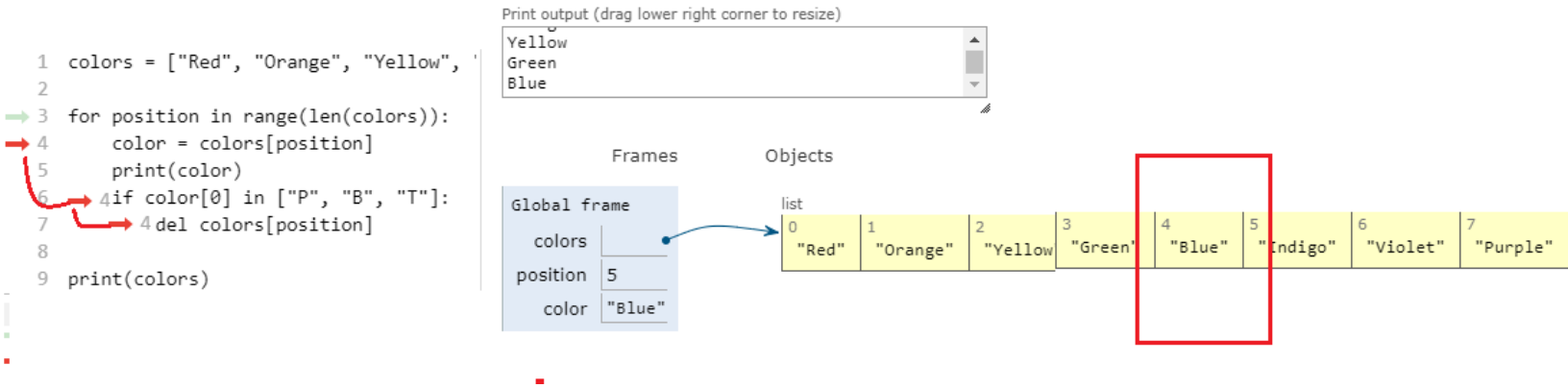
You may be tempted now to iterate through a list and accumulate some data into it or delete data from it, however that often becomes very confusing.

Let's try to understand this from an example:

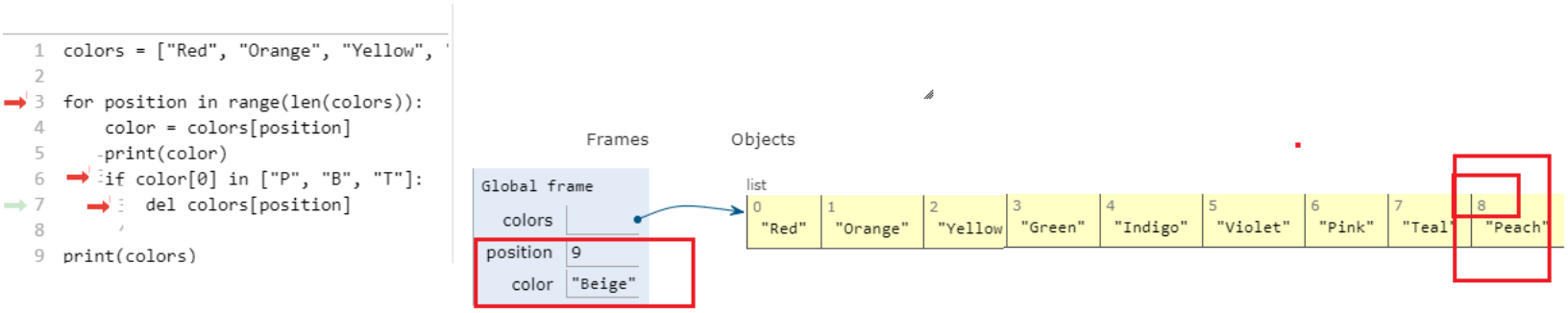
```
1  colors = ["Red", "Orange", "Yellow", "Green", "Blue", "Indigo", "Violet", "Purple", "Pink", "Brown
2
3  for position in range(len(colors)):
4      color = colors[position]
5      print(color)
6      if color[0] in ["P", "B", "T"]:
7          del colors[position]
8
9  print(colors)
10
```

```
Red
Orange
Yellow
Green
Blue
Violet
Purple
Brown
Turquois
Beige
---------------------------------------------------------------------------
IndexError                                Traceback (most recent call last)
<ipython-input-43-3406d8ece12e> in <module>
      2
      3 for position in range(len(colors)):
----> 4     color = colors[position]
      5     print(color)
      6     if color[0] in ["P", "B", "T"]:

IndexError: list index out of range
```

- In the code above, we iterated through **range(len(colors))** because it made it easier to locate the position of the item in the list and delete it.
- However, we run into a problem because as we delete content from the list, the list becomes shorter.
- Not only do we have an issue indexing on line 4 after a certain point, but we also skip over some strings because they've been moved around.



- As we reach point where any of the condition is satisfied, we will enter the if block and execute commands.
- We have del method which will delete the element Blue, which would mean the size of list will get reduced on the fly
- So might reach to a point as below :



If you simulate on your book with every iteration , you will find yourself ending in above situation with index out of range.

Glossary

for loop traversal (for)
Traversing a string or a list means accessing each character in the string or item in the list, one at a time.
For example, the following for loop:

```
for ix in 'Example':
    ...
```

executes the body of the loop 7 times with different values of ix each time.

range
A function that produces a list of numbers. For example, range(5), produces a list of five numbers, starting with 0, [0, 1, 2, 3, 4].

pattern
A sequence of statements, or a style of coding something that has general applicability in a number of different situations. Part of becoming a mature programmer is to learn and establish the patterns and algorithms that form your toolkit.

index
A variable or value used to select a member of an ordered collection, such as a character from a string, or an element from a list.

accumulator pattern
A pattern where the program initializes an accumulator variable and then changes it during each iteration, accumulating a final result.

Question:

What will be the value of a after the following code has executed?

```
a = ["holiday", "celebrate!"]
quiet = a
quiet.append("company")
```

Majority of you know the answer but if i ask you to type, many might make a very basic mistake.

a) ['holiday', 'celebrate!','company']
b) ['holiday', 'celebrate!', 'company']

option a would be correct at first look but the difference between a and b is the space before company.

Always remember there is a space after ','

PART 5