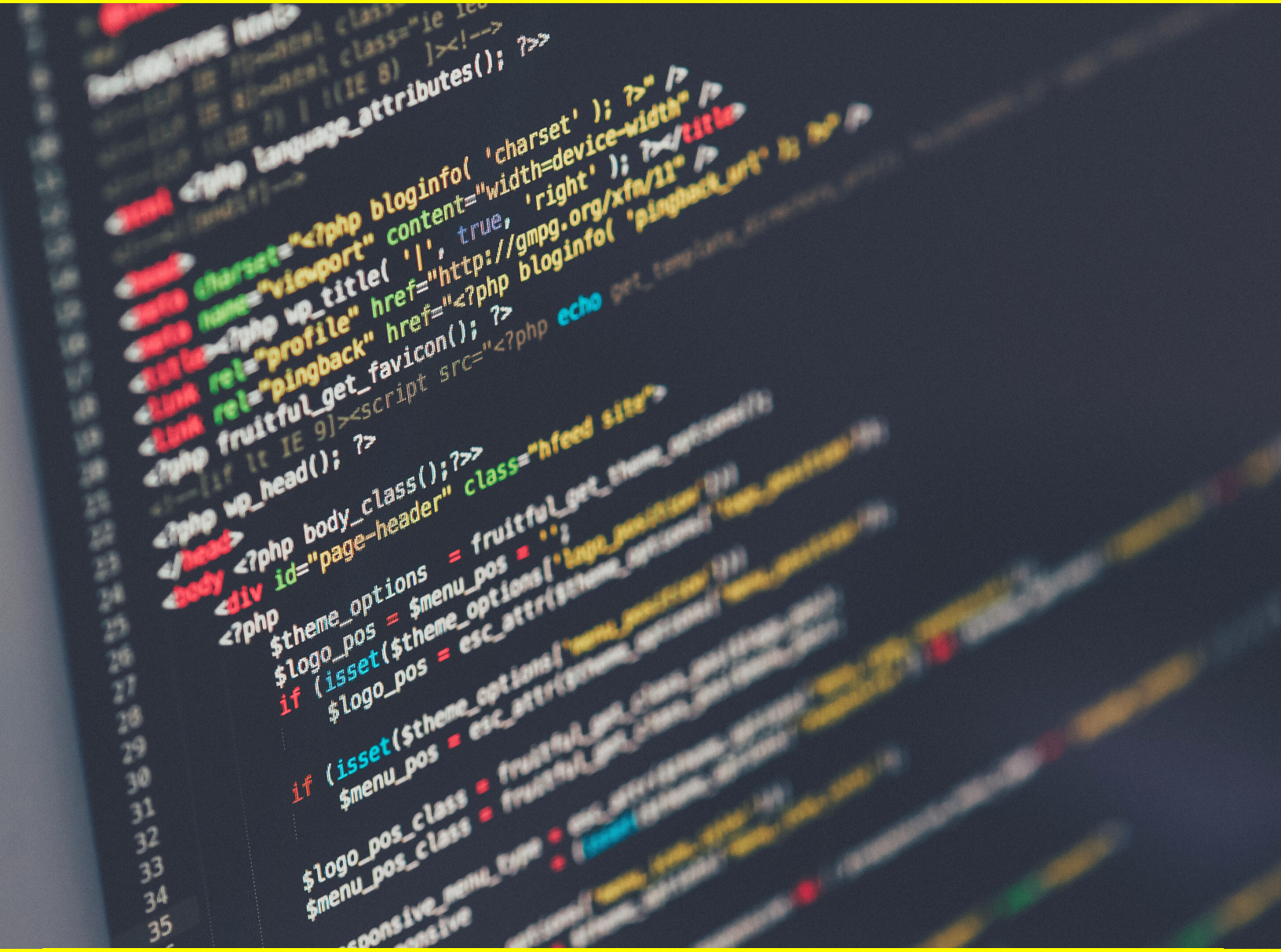
the fast lane to python

PART 3



BY:

Arshadul Shaikh

Ref taken from Coursera + Udemy

*****KEYNOTE:**
Congratulations that you have made it till 3rd part, which means you are dedicated to become one "**in demand python developers"** companies are searching for.
If you have directly started with part 3, i would say try to go through part 1 and turtle(part 2) as well, won't take much of your time to get you on board from where we are going to start. Lot said, let's roll:

Introduction to Sequences:

A **sequence** is a **succession of values** bound together by a container that reflects their type. Almost every stream that you put in Python is a sequence. Some of the sequences that Python supports are strings, lists, tuples and Xrange objects. Python has a bevy of methods and formatting operations that it can perform on each of these.

TOo tough to understand, right? Let's break it down in easy terms

In the real world most of the data we care about doesn't exist on its own. Usually data is in the form of some kind of collection or sequence.

- For example, a grocery list helps us keep track of the individual food items we need to buy, and our todo list organizes the things we need to do each day.
- Notice that both the grocery list and the todo list are not even concerned with numbers as much as they are concerned with words.
- This is true of much of our daily life, and so Python provides us with many features to work with lists of all kinds of objects (numbers, words, etc.) as well as special kind of sequence, the character string, which you can think of as a sequence of individual letters.

We all are very well aware about the **built-in** data types **int, float , double**, and **str**.
int, float , double and such data types are considered to be **primitive** or **atomic** data types because their values are **not composed** of any **smaller parts**. They cannot be broken down

But when you talk about **strings**, they can be broken down to further **smaller pieces** as string is a **collection of characters**. Same is for list, they are collection of some data types, which when you try to break down, it would be collection of data types.

Types that are comprised of smaller data types are called as Collection data types. Depending on what we are doing, we may want to treat a collection data type as a single entity or we may want to access its parts.

This is a feature of ambiguity which is comes handy during coding and which also makes programming ease to write.

Strings

We have known by now that anything that is inside single or double quotes is a string and everything we wanted to print some words/phrases we have used strings.
And as we talk about sequence we know one feature of string that it is a collection of characters and so, string is not a primitive data type but a collection data type.

What would that mean?
It means that the **individual characters** that make up a **string** are in a **particular order** from left to right.

Strings are immutable objects i.e it cannot be modified/cannot be changed. It will represent the same value until the variable pointing to it does not start pointing to some other string/data type .

- For example—The string object ‘h’ will always represent the python value h. No matter what happens!

```
x='hello'
print(x[1])
x[1]='o'
```

```
e
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-31-8a1f251c8b5f> in <module>
      1 x='hello'
      2 print(x[1])
----> 3 x[1]='o'
      4

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

- In the example there is a string object ‘hello’ and you want to change the character at the second position that is ‘e’ to ‘o’ to make the string object ‘hollo’. There is no way, you can do this in python because of their immutability as you see in above image.

```
x = 'hello'

print(id(x))      #This will print some id like 8879437848.

x = 'python'

print(id(x))      #This will print some id like 4387438948 different
                  # from the previous one.
```

```
1667344197088
1667302229248
```

Since the string object ‘hello’ is different from the string object ‘python’. The function id returns different unique ids. This implies that the memory location where these objects are stored is different and these are two different objects.

let's look at few ways that we can do with strings and it's predefined functions:

```
##remember strings is a collection of characters
##strings are immutable, meaning everything that we did below
#will not change anything in the main string

book_name='the fast lane to python'
writer="    arshadul shaikh"
empty_string=""
print('the book= ' +"book_name" + " by " + writer) #we can easily concatenate
                                                   #existing strings using +

line="abcabcabc"
print(line[2]) #strings as are collection of characters, you can access any letter
               #using the existing indexes
               #always remember the indexing starts at 0

print(len(line)) # we jave already used len method a lot of times in earlier examples

print(book_name[2:5]) #this is slicing feature which strings provide
                      #the output of this would give characters placed
                      #at 2 3 4

print(book_name.count('a')) #another method which would give the count of how many
                            #character a is present in the string book_name

print(book_name.index('a')) #this method will give you the output of first occurence
                            #of character placed inside round parentheses

print(writer.upper()) #it would return output in all upper case
print(writer.strip()) #this method returns the string removing all extra space
                      # blank lines

words= book_name.split() #splits the entire string into list of words
print(words[1])
print(words)

```

```
the book= book_name by     arshadul shaikh
c
9
e f
2
5
    ARSHADUL SHAIKH
arshadul shaikh
fast
['the', 'fast', 'lane', 'to', 'python']
```

A string that contains no characters, often referred to as the **empty string,** is still considered to be a string. cause it is simply a sequence of zero characters , represented by ‘’ or “” (two single or two double quotes with nothing in between).

Lists

While talking about collection data types we already know collection and list go hand in hand.

A **list** is a **sequential collection of Python data values**, where each value is identified by an index. The values that make up a list are called its elements.
Lists are similar to strings, which are ordered collections of characters, except that the elements of a list can have any type and for any one list, the items can be of different types.

If you could recall the definition of **Array**, it said a **homogeneous collections of data types**, which sounded good at first until you end in a situation where you no more wanted collection to be of same data type but rather should be of any data type.

There are several ways to create a new list. The simplest is to enclose the elements in square brackets ([and]).
[10, 20, 30, 40]
["I will", "be the best", "in python"]

The first example is a list of four integers. The second is a list of three strings.

Let's look after some basic examples:

```
##list is a sequential collection of objects,
##each item in the list is identified by it's index/position value
## unlike strings, list is mutable meaning you can change the values

tools=[] #this creates a list
tools=['nail', 'screw']
tools.append('nut')
hardwares=['screwdriver','plas','hammer']

hardware_tools=tools + hardwares
print (hardware_tools)

#we all know that indexing starts from 0
#similarly here as if you want to get nut from the list
#you will have to count from 0,1 ,2 and so LIST[2]
#will give you the correct output

print(hardware_tools[2])

#we are now also aware about slicing
#if we can pick a sub_sequence from the sequence of list
#if i want to have from screw to screwdriver
#i will count 0,1-screw, 2,3-screwdriver
#so LIST[1:4], the end_index i.e 4 is excluded
#this is slicing

hardware_tools[1:4]
hardware_tools[4]='driverScrew' #as lists are mutable you can change
hardware_tools[4]               #the index value on the fly

```

```
['nail', 'screw', 'nut', 'screwdriver', 'plas', 'hammer']
nut

'driverScrew'
```

But hey, those are still of same data types examples!!! Yes, and you know by now the elements of a list don't have to be the same type. The following list contains a string, a float, an integer, and another list.

["I will be the best in python", 2.0, 5, [10, 20]]

*****KEY POINT:**

WAIT WAIT WAIT, don't Mix Types!
You'll likely might see some examples with odd combinations, but when you create lists you should generally not mix types together. A list of just strings or just integers or just floats is generally easier to deal with.

Tuples

A **tuple**, like a list, is a **sequence of items of any type**. The printed representation of a tuple is a **comma-separated** sequence of values, enclosed in parentheses. In other words, the representation is just like lists, except with parentheses () instead of square brackets [].

difference in parentheses wouldn't be only reason , right? we are not here to learn 2 similar collection data types, but then what is the difference?

One way to create a tuple is to write an expression, enclosed in parentheses, that consists of multiple other expressions, separated by commas.

julia = ("I", "will", "be the best", "in python", 2020)

What is the difference?

The **key difference** between **lists** and **tuples** is that a **tuple is immutable**, meaning that its contents can't be changed after the tuple is created.

You will understand mutable and immutable concepts in depth, need to stay consistent .

***KEY POINT:**
To create a tuple with a single element (but you're probably not likely to do that too often), we have to include the final comma, because without the final comma, Python treats the (5) below as an integer in parentheses

```
1  t = (5,)
2  print(type(t))
3
4  x = (5)
5  print(type(x))
```

```
<class 'tuple'>
<class 'int'>
```

let's see some examples:

```
##tuple is a sequential collection of objects,
##each item in the tuple is identified by it's index/position value
##same as strings,tuple is also immutable

tools=() #this creates a tuple
tools=('nail', 'screw')
hardwares=('screwdriver','plas','hammer')

hardware_tools=tools + hardwares
print (type(hardware_tools))

#we all know that indexing starts from 0
#similarly here as if you want to get nut from the tuple
#you will have to count from 0,1 ,2 and so tuple[2]
#will give you the correct output

print(hardware_tools[2])

#we are now also aware about slicing
#we can pick a sub_sequence from the sequence of tuple
#if i want to have from screw to screwdriver
#i will count 0,1-screw, 2,3-screwdriver
#so tuple[1:4], the end_index i.e 4 is excluded
#this is slicing

print(hardware_tools[1:4])
#hardware_tools[4]='driverScrew' #as lists are mutable you can change
hardware_tools[4]                #the index value on the fly

```

```
<class 'tuple'>
screwdriver
('screw', 'screwdriver', 'plas')

'hammer'
```

tuples are immutable:

```
1 hardwares=('screwdriver','plas','hammer')
2 print(hardwares[1])
3 hardwares[1]='nut'
```

```
plas

---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-59-dfb9519efc86> in <module>
      1 hardwares=('screwdriver','plas','hammer')
      2 print(hardwares[1])
----> 3 hardwares[1]='nut'

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

Why no append method??

because tuple is an immutable object. Hence any operation that tries to modify it (like append) is not allowed. However, you can convert tuple to list by built-in function list().

You can always append item to list object. Then use another built-in function tuple() to convert this list object back to tuple.

```
1 hardwares=('screwdriver','plas','hammer')
2 hardwares=list(hardwares)
3 hardwares.append('screw')
4 hardwares=tuple(hardwares)
5 print(hardwares)
6
```

```
('screwdriver', 'plas', 'hammer', 'screw')
```

Working in Depth with Index Operator(Strings):

Like other programming languages, indexing operator(gives access to a sequence's element) selects a single character from a string.

The characters are accessed by their positions/index value. And python uses square brackets to enclose the index.

Index of the item you want to access, must be an integer.

Since all sequences are ordered and indexed arrays of objects, each object stored in a sequence has its associated index number

- positive one, zero indexed and starting from left
- the negative one starting at -1 from the right.

```
>>> +---+---+---+---+
>>> |-4 |-3 |-2 |-1 |  <= negative indexes
>>> +---+---+---+---+
>>> | A | B | C | D |  <= sequence elements
>>> +---+---+---+---+
>>> | 0 | 1 | 2 | 3 |  <= positive indexes
>>> +---+---+---+---+
```

```
1  book='the fast lane to python'
2  print(len(book))
3  print(book[11])
4  print(book[-12])
```

```
23
n
n
```

- We used expression **len(book)** so as to get the total length and then we can decide which would be the center number which we can check from both positive and negative indexes

- The expression **book[11]** selects the character at **index 11** from **book string**, and prints this one character on screen.

- The **letter at index zero** of "the fast lane to python" is **t**. So at position [11] we have the letter n. If you want the zero-eth letter of a string, you just put 0, or any expression with the value 0, in the brackets. The expression as we know in brackets is called an **index.**

***KEY POINT:**
Note that indexing returns a string — Python has no special type for a single character. It is just a string of length 1.

```
1 # this example show how to retrieve elements of a sequence
2 #without assigning the string to any variable
3 print("PYTHON -THE LEGEN"[5])
4
```

```
N
```

```
1 #Trying to access an element out of range throws an IndexError.
2
3 print("PYTHON -THE LEGEN"[21])
4
```

```
---------------------------------------------------------------------------
IndexError                                Traceback (most recent call last)
<ipython-input-76-729ed760f446> in <module>
      1 #Trying to access an element out of range throws an IndexError.
      2
----> 3 print("PYTHON -THE LEGEN"[21])

IndexError: string index out of range
```

```
# using negative indexes to get the last element
print("PYTHON -THE LEGEND"[-1])

```

```
D
```

Working in Depth with Index Operator(List or Tuple):

The syntax for accessing the elements of a list or tuple is the same as the syntax for accessing the characters of a string. We use the index operator ([] – not to be confused with an empty list).

The expression inside the brackets specifies the index. Remember that the indices start at 0. Any integer expression can be used as an index and as with strings, negative index values will locate items from the right instead of from the left.

When we say the first, third or nth character of a sequence, we generally mean counting the usual way, starting with 1. The nth character and the character AT INDEX n are different then: The nth character is at index n-1.

Make sure you are clear on what you mean!

Try to predict what will be printed out by the following code, and then run it to check your prediction. (Actually, it's a good idea to always do that with the code examples. You will learn much more if you force yourself to make a prediction before you see the output.)

```
#exampples of positive & negative index
numbers = [17, 123, 87, 34, 66, 8398, 44]
print(numbers[2])
print(numbers[9-8]) #you can also write another expression inside []
                    #to place the exact value of index
print(numbers[-2])
```

```
87
123
8398
```

```python
#exampples of positive & negative index for tuples
prices = (1.99, 2.00, 5.50, 20.95, 100.98)
print(prices[0])
print(prices[-1])
print(prices[3-5])
```

```
1.99
100.98
20.95
```

```python
# index lookups can be chained to access nested containers
([0, 1], [2, 3])[1][1]
#if you break down
#list[0]=[0,1]
#list[1]=[2,3]
#list[0][0] == [0,1][0] which is 0
#list[0][1] == [0,1][1] which is 1
#list[1][0] == [2,3][0] which is 2
#list[1][1] == [2,3][1] which is 3

```

```
3
```

```python
# since lists are mutable indexes can be used
#for item assignment or deletion
list_Eg = [0, 1, 2, 3]
print(list_Eg)
list_Eg[0] = "ABCD"
print(list_Eg)

```

```
[0, 1, 2, 3]
['ABCD', 1, 2, 3]
```

```python
#we can also perfrom manipulations on string indexes
s = "python rocks"
print(s[2] + s[-4])
```

```
to
```

```
#example of how len method counts the length
#of the entire string/list/tuple
#and will always be +1 from the last index
#cause indexing starts from 0
#while len method counts from 1

alist = [3, 67, "cat", [56, 57, "dog"], [ ], 3.14, False]
print(alist[len(alist)])

```

```
---------------------------------------------------------------------------
IndexError                                Traceback (most recent call last)
<ipython-input-89-cf5473d2b1ae> in <module>
      6
      7 alist = [3, 67, "cat", [56, 57, "dog"], [ ], 3.14, False]
----> 8 print(alist[len(alist)])

IndexError: list index out of range
```

- We got an error because, we are trying to access the index with value of the length of the list, but the maximum index in all sequences would be len() -1

```
alist = [3, 67, "cat", [56, 57, "dog"], [ ], 3.14, False]
print(alist[len(alist) -1])

```

```
False
```

*****KEY POINT:**

you should/would have a question in your by now why does counting start at 0 going from left to right, but at -1 going from right to left?

Well, indexing starting at 0 has a long history in computer science having to do with some low-level implementation details that we won't go into. For indexing from right to left, it might seem natural to do the analogous thing and start at -0. Unfortunately, -0 is the same as 0, so s[-0] can't be the last item.

Disambiguating []: creation vs indexing

Square brackets [] are used in quite a few ways in python. When you're first learning how to use them it may be confusing, but with practice and repetition they'll be easy to incorporate!

You have currently encountered **two instances** where we have used square brackets. The first is **creating lists** and the second is **indexing**. At first glance, creating and indexing are difficult to distinguish.
However, indexing requires referencing an already created list while simply creating a list does not.

```
1 #empty list example
2 new_lst = []
3 print(new_lst)
4 print(new_lst[0])
```

```
[]

---------------------------------------------------------------------------
IndexError                                Traceback (most recent call last)
<ipython-input-93-1e6d39d0e085> in <module>
      2 new_lst = []
      3 print(new_lst)
----> 4 print(new_lst[0])

IndexError: list index out of range
```

In the code above we got an error of index out of range because a new list is created using the empty brackets. Since there's nothing in it though, we can't index into it so new_lst[x] where x is anything will always give you an error of index out of range.

```
1 new_lst = ["NFLX", "AMZN", "GOOGL", "DIS", "XOM"]
2 part_of_new_lst = new_lst[0]
3 print(part_of_new_lst)
```

```
NFLX
```

- In the code above, you'll see how, now that we have elements inside of new_lst, we can index into it. In order to extract an element of the list, we do use [], but we first have to specify which list we are indexing.
- Imagine if there was another list in the active code. How would python know which list we want to index into if we don't tell it?
- Additionally, we have to specify what element we want to extract. This belongs inside of the brackets.Though it may be easier to distinguish in this above active code, below may be a bit more difficult.

```
1 lst = [0]
2 n_lst = lst[0]
3
4 print(lst)
5 print(n_lst)
```

```
[0]
0
```

- Here, we see a list called lst being assigned to a list with one element, zero.
- Then, we see how n_lst is assigned the value associated with the first element of lst.
- Despite the variable names, only one of the above variables is assigned to a list.
- Note that in this example, what sets creating apart from indexing is the reference to the list to let python know that you are extracting an element from another list.

Let's play with some examples:

```
1 fruit = "Banana"
2 sz = len(fruit)
3 lastch = fruit[sz-1]
4 print(lastch)
```

```
a
```

Typically, a Python programmer would combine lines 2 and 3 from the above example (as we did in earlier 1 example) into a single line:
lastch = fruit[len(fruit)-1]

Though, we know that negative index fruit[-1] would be a more appropriate way to access the last index in a list.

- We have seen before in one example how you can use the len function to access other predictable indices, like the middle character of a string.

```
1  language = "python"
2  midchar = language[len(language)//2]
3  print(midchar)
4  #here H is not the midchar as the number of
5  #characters is even which means we will have 2 midchars
6  #so might need a little more additional expression
7  #to make a general solution for both type of strings
8  #one with even numbers and other with odd
9  #but you get the idea how you can use the len method
10
```

```
h
```

```
#nested lists
nested_list =  ["python", 3.0, 'part 3']
print(len(nested_list))
print(len(nested_list[0])) #this tells you the length of
                           #value of data present at list[0]
                           #so 6 is the length of python
```

```
3
6
```

Slice Operator (in detail):

Substring of a string is a **slice.** It gives access to a specified range of sequence elements.
slice[start:stop]

- start
  Optional. Starting index of the slice. Defaults to 0.

- stop
  Optional. The **last index of the slice** or the number of items to get. Defaults to len(sequence).

The slice operator **slice[start:stop] returns** the part of the **string** starting with the character at **index start** and go up to but not including the character at **index stop**.

Or with normal counting from 1, this is the (n+1)st character up to and including the mth character.

If you omit the first index (before the colon), the slice starts at the beginning of the string. If you omit the second index, the slice goes to the end of the string.

String Slices:

```python
lang = "Python, Java, C++"

print(lang[0:6]) #if you count from 0th index to get python
                 #you have
                 #0=P
                 #1=y
                 #2=t
                 #3=h
                 #4=o
                 #5=n
                 #6=,
                 #so the slicing should have been lang[0:5]
                 #but we have already seen that last value in
                 #slicing is always not excluded so
                 #lang[0:6] will indirectly fecth index values
                 #from 0 to 5
print(lang[8:12])
print(lang[14:17])
```

```
Python
Java
C++
```

```python
lang = "python"
print(lang[:3]) #takes 0 as the first index
print(lang[3:]) #takes len -1 as the last index
print(lang[:]) #takes 0 as the first and len -1 as the last
```

```
pyt
hon
python
```

List Slices:

The slice operation we saw with strings works the same on lists. Remember that the first index is the starting point for the slice and the second number is one index past the end of the slice (up to but not including that element).

Recall also that if you omit the first index (before the colon), the slice starts at the beginning of the sequence. If you omit the second index, the slice goes to the end of the sequence.

```
lang = ['python', 'java', 'c++', 'Javascript', 'flutter', 'Unix']
print(lang[1:3]) #
print(lang[:4])
print(lang[3:])
print(lang[:])
```

```
['java', 'c++']
['python', 'java', 'c++', 'Javascript']
['Javascript', 'flutter', 'Unix']
['python', 'java', 'c++', 'Javascript', 'flutter', 'Unix']
```

Tuple Slices:

We can’t modify the elements of a tuple as they are immutable, but we can make a variable reference a new tuple holding different information. Thankfully we can also use the slice operation on tuples as well as strings and lists.

To construct the new tuple, we can slice parts of the old tuple and join up the bits to make the new tuple.

List Slices:

The slice operation we saw with strings works the same on lists. Remember that the first index is the starting point for the slice and the second number is one index past the end of the slice (up to but not including that element).

Recall also that if you omit the first index (before the colon), the slice starts at the beginning of the sequence. If you omit the second index, the slice goes to the end of the sequence.

```
lang = ['python', 'java', 'c++', 'Javascript', 'flutter', 'Unix']
print(lang[1:3]) #
print(lang[:4])
print(lang[3:])
print(lang[:])
```

```
['java', 'c++']
['python', 'java', 'c++', 'Javascript']
['Javascript', 'flutter', 'Unix']
['python', 'java', 'c++', 'Javascript', 'flutter', 'Unix']
```

Tuple Slices:

We can’t modify the elements of a tuple as they are immutable, but we can make a variable reference a new tuple holding different information. Thankfully we can also use the slice operation on tuples as well as strings and lists.

To construct the new tuple, we can slice parts of the old tuple and join up the bits to make the new tuple.

```python
info = ("python", "was invented by", "Guido van Rossum","in" ,1990)
print(info[2])
print(info[2:5])

print(len(info))

info = info[:3] + ("last stable python version is", 3.8)
print(info)
```

```
Guido van Rossum
('Guido van Rossum', 'in', 1990)
5
('python', 'was invented by', 'Guido van Rossum', 'last stable python version i
s', 3.8)
```

1 Question that might arise in some of yours mind,

Aren't tuples immutable?

Then why does
info[:3] + ("last stable python version is", 3.8)
line work?

- First **info** pointed to tuple with with several collection of primitive data types and strings.
- Then we changed the same variable info to point at a new tuple info[:3] + ("last stable python version is", 3.8)
- We didn't actually mutate the tuple ("python", "was invented by", "Guido van Rossum","in" ,1990) .

Tuple are immutable, variables can point at whatever they want.
Info is just pointing now to a new tuple in memory

```python
info = ("python", "was invented by", "Guido van Rossum","in" ,1990)
print(id(info))
info = info[:3] + ("last stable python version is", 3.8)
print(id(info))
```

```
1667339383472
1667344611288
```

1 Question that might arise in some of yours mind,

Aren't tuples immutable?

Then why does
info[:3] + ("last stable python version is", 3.8)
line work?

- First **info** pointed to tuple with several collection of primitive data types and strings.
- Then we changed the same variable info to point at a new tuple info[:3] + ("last stable python version is", 3.8)
- We didn't actually mutate the tuple ("python", "was invented by", "Guido van Rossum","in" ,1990) .

Tuple are immutable but variables can point at whatever they want.
Info is just pointing now to a new tuple in memory

Concatenation

We have seen so many examples with "+" the concatenation operator.
Similar to strings, the + operator concatenates lists too.
It is important to see that + operator create a new lists from the elements of the operand lists.
If you concatenate a list with 2 items and a list with 4 items, you will get a new list with 6 items (not a list with two sublists).

```
1 fruits = ["apple","orange","banana","cherry"]
2 print([1,2] + [3,4])
3 print(fruit+[6,7,8,9])
4
5 print([0] * 4)
```

```
[1, 2, 3, 4]
['apple', 'orange', 'banana', 'cherry', 6, 7, 8, 9]
[0, 0, 0, 0]
```

Repetition

Similar to + the **"*"** operator repeats the items in a list a given number of times.

```
fruits = ["apple","orange","banana","cherry"]
print(fruit * 2)
```

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

As you see, fruit was a list and when repeated 2 times , created a new list with the same values repeated again

*****KEY POINT:**
Beware when adding different types together, as Python doesn't understand how to concatenate different types together.

Thus, if you try to add a string to a list with ['first'] + "second" then the interpreter will return an error.
To do this you'll need to make the two objects the same type. In this case, it means putting the string into its own list and then adding the two together like so:
['first'] + ["second"].

This process will look different for other types though. Remember that

```
A = [[None] * 2] * 3
A[0][0] = 5
print(A)
```

```
[[5, None], [5, None], [5, None]]
```

Exception:

```
A = [[None] * 2] * 3
A[0][0] = 5
print(A)
```

```
[[5, None], [5, None], [5, None]]
```

You would have expected to have the value of 5 only on first index sub_list first index.

- **T**he reason is that replicating a list with * doesn't create copies, it only creates references to the existing objects.
- The *3 creates a list containing 3 references to the same list of length two.
- Changes to one row will show in all rows, which is almost certainly not what you want.
- The suggested approach is to create a list of the desired length first and then fill in each element with a newly created list:

```
A = [None] * 3
for i in range(3):
    A[i] = [None] * 2
print(A)
A[0][0]=5
print(A)
```

```
[[None, None], [None, None], [None, None]]
[[5, None], [None, None], [None, None]]
```

Count method

Count similar to other methods requires one argument, which is what you would like to count.
The method then returns the number of times that the argument occurred in the string/list the method was used on. There are some differences between count for strings and count for lists.
When you use count on a string, the argument can only be a string. You can't count how many times the integer 2 appears in a string, though you can count how many times the string "2" appears in a string.
For lists, the argument is not restricted to just strings.

Eg. for strings :

```
count_method = "this is a string for checking count method!"
print(count_method.count("i")) #it will check the first occurence of
                               #i in the string from left
                                #and give you the index
print(count_method.count("is")) # check for first occurence of "is"
```

```
4
2
```

When you run the code in below image , you'll see how count with a list works. Notice how "4" has a count of zero but 4 has a count of three?
This is because the list z only contains the integer 4. There are never any strings that are 4.

Additionally, when we check the count of "a", we see that the program returns zero. Though some of the words in the list contain the letter "a", the program is looking for items in the list that are just the letter "a".

```
z = ['atoms', 4, 'neutron', 6, 'proton', 4, 'electron', 4,
         'electron', 'atoms']
print(z.count("4"))  #will try to search for 4 which is string and not int
                     #so if you are expecting to return 3 as an o/p
                     #then you are probably need to remember the basics
                     #anything between "" this is a string
                     #do not get mistaken, stay strong with your basics

print(z.count(4))    #this will return what your expectation was above

print(z.count("a")) #this again will search for an exact "a" string
                     #in the list
                     #if you are thinking, index of a from atoms
                     #would be returned then you are mistaken
                     #z[0] is entire string "atoms" the values of a will be
                     #z[0][0] and z.count("") method will search on entire
                     #list and not on sub_list
                     #so it will return 0

print(z.count("electron")) #this will return the first occurence of
                           #electron which is 2
```

```
0
3
0
2
```

Index Method

The other method that will be helpful for both strings and lists is the index method.
The index method requires one argument, and, like the count method, it takes only strings when index is used on strings, and any type when it is used on lists.

For both strings and lists, index returns the leftmost index where the argument is found. If it is unable to find the argument in the string or list, then an error will occur.

```python
info = "Python is the best of the best programming language"
old_lang = ["java", "java", "Javascript", [], "cpp", "c", 1960,
            "java"]

print(info.index("b"))  #will search for the first index of char "b"
                        #in the string info

print(info.index("best")) #will search again for the first occurence
                        #of string "best" but will return the
                        #starting index of "b"

print(old_lang.index("java")) #search for the first occurence of java

print(old_lang.index([]))  #old_lang list has an empty sub_list
                           #so it will search for an empty sub_list
                           #in the list
print(old_lang.index(1960))
```

```
14
14
0
3
6
```

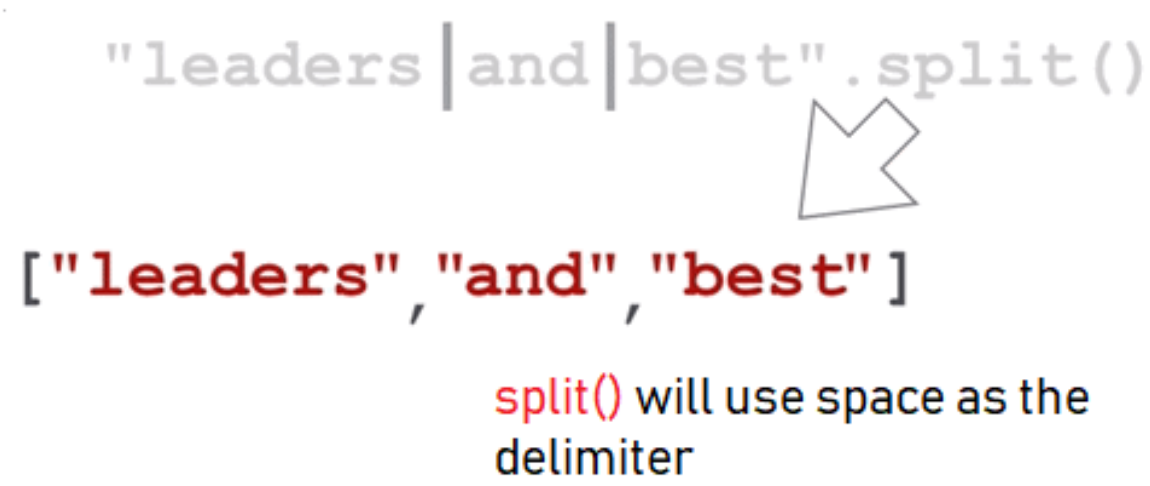
```python
seasons = ["winter", "spring", "summer", "fall"]

print(seasons.index("autumn"))  #Error!
```

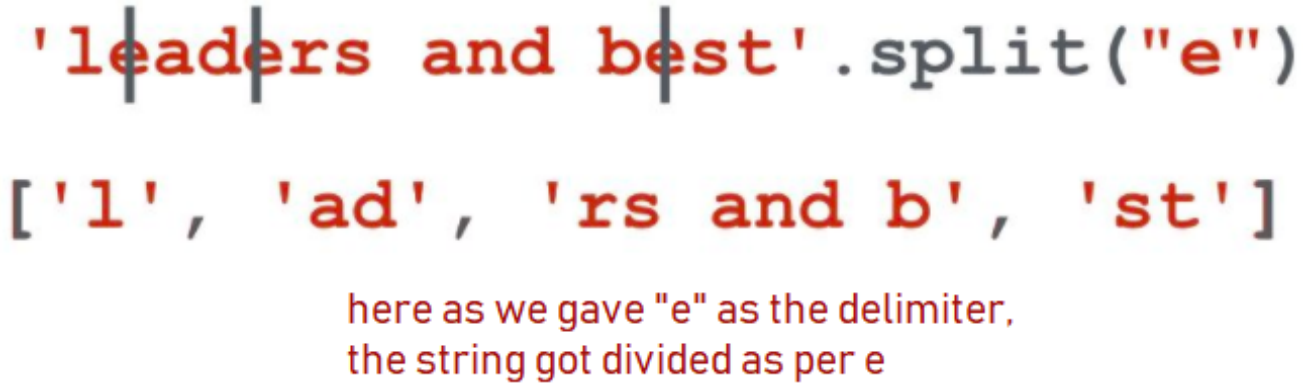
```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-129-81aaff4a97b6> in <module>
      1 seasons = ["winter", "spring", "summer", "fall"]
      2
----> 3 print(seasons.index("autumn"))  #Error!

ValueError: 'autumn' is not in list
```

we're trying to see where "autumn" is in the list seasons. However, there is no string called autumn (though there is string called "fall" which is likely what the program is looking for)**. Remember that an error occurs if the argument is not in the string or list.**

Splitting & Joining

Two of the most useful methods on strings involve lists of strings. The split method breaks a string into a list of words. By default, any number of whitespace characters is considered a word boundary.



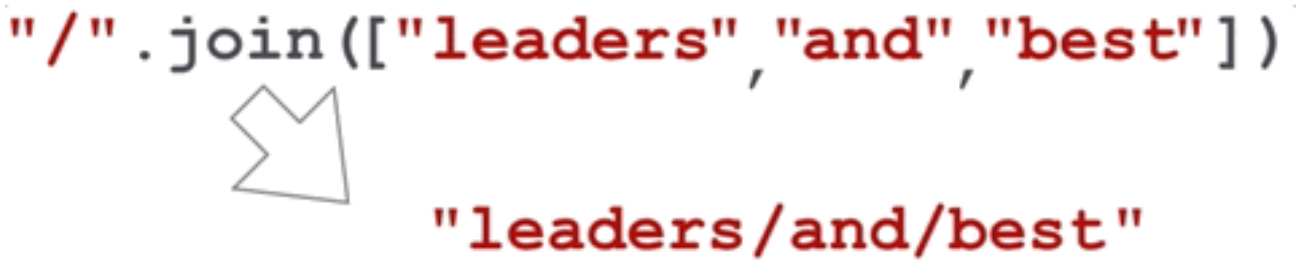


```
song = "We will, we will rock you"
song_list = song.split() #as we did not give any delimiter
                #the default space would be considered as the
                #delimiter
print(song_list)

```

```
['We', 'will,', 'we', 'will', 'rock', 'you']
```

Method - Join()

The inverse of the split method is join. You choose a desired separator string, (often called the glue) and join the list with the glue between each of the elements.




```
colors = ["red", "blue", "green"]
delimiter = ' :) '
color_string = delimiter.join(colors)
print(colors)
print(color_string)

print("***".join(colors))
print("".join(colors))
```

```
['red', 'blue', 'green']
red :) blue :) green
red***blue***green
redbluegreen
```

Iteration:

Computers are often used to automate repetitive tasks. Repeating identical or similar tasks without making errors is something that computers do well and people do poorly.Repeated execution of a set of statements is called iteration. Because iteration is so common, Python provides several language features to make it easier.

Iterators are everywhere in Python. They are elegantly implemented within for loops, comprehensions, generators etc. but are hidden in plain sight.

Iterator in Python is simply an object that can be iterated upon. An object which will return data, one element at a time.

Technically speaking, a **Python iterator object** must **implement** two special **methods**, **__iter__() and __next__(),** collectively **called** the **iterator protocol**.

An object is called iterable if we can get an iterator from it. Most built-in containers in Python like: **list, tuple, string** etc. are **iterables**.
The **iter()** function (which in turn calls the __iter__() method) returns an iterator from them.

The for Loop

When it comes to loops, be it scripting language or a programming language, **for** has been our all time favourite. Because of its syntax and the how easy we all find to analyze how the loop would look like and return results.

Back when we drew the images with turtle (in 2nd part of this series of book Turtle- Drawing the python way) it could have been quite tedious for you, while trying to draw a square as you would have to move then turn, move then turn ... four times , sounds repetitive and writing the same steps again and again is not at all efficient.

And what if someone approached this writing same steps again and again for drawing a hexagon, or an octagon, or a polygon with 42 sides, it would have been a nightmare to duplicate all that code for him/her.

A basic building block of all programs is to be able to repeat some code over and over again. **We refer to this repetitive idea as iteration**.

In Python, the for statement allows us to write programs that implement iteration.

As a simple example, let's say we have a party on Python being selected as the most loved and efficient programming language , and we'd like to ask all other languages to join the celebration party, how can send everyone a message in the most efficient way?

```
4  print("Hi Java Please come to the celebration party on Saturday!")
5  print("Hi cpp Please come to the celebration party on Saturday!")
6  print("Hi JavaScript Please come to the celebration party on Saturday!")
7  print("Hi PHP Please come to the celebration party on Saturday!")
8  print("Hi Ruby Please come to the celebration party on Saturday!")
9  print("Hi Swift Please come to the celebration party on Saturday!")
10 print("Hi C# Please come to the celebration party on Saturday!")
11
```

```
Hi Java Please come to the celebration party on Saturday!
Hi cpp Please come to the celebration party on Saturday!
Hi JavaScript Please come to the celebration party on Saturday!
Hi PHP Please come to the celebration party on Saturday!
Hi Ruby Please come to the celebration party on Saturday!
Hi Swift Please come to the celebration party on Saturday!
Hi C# Please come to the celebration party on Saturday!
```

- I had to write the almost the same print lines 7 times replacing the name of programming language all the time.
- this was still possible but what python had to invite every single language ??? Python won't be writing same lines again and again for let's suppose 1000 languages. it is possible but the worst possible way to send an invite.

Let's see another way of doing this:

```
lang='java'
print("Hi",lang,"Please come to the celebration party on Saturday!")
lang='cpp'
print("Hi",lang,"Please come to the celebration party on Saturday!")
lang='JavaScript'
print("Hi",lang,"Please come to the celebration party on Saturday!")
lang='PHP'
print("Hi",lang,"Please come to the celebration party on Saturday!")
lang='Ruby'
print("Hi",lang,"Please come to the celebration party on Saturday!")
lang='Swift'
print("Hi",lang,"Please come to the celebration party on Saturday!")
lang='C#'
print("Hi",lang,"Please come to the celebration party on Saturday!")

```

```
Hi java Please come to the celebration party on Saturday!
Hi cpp Please come to the celebration party on Saturday!
Hi JavaScript Please come to the celebration party on Saturday!
Hi PHP Please come to the celebration party on Saturday!
Hi Ruby Please come to the celebration party on Saturday!
Hi Swift Please come to the celebration party on Saturday!
Hi C# Please come to the celebration party on Saturday!
```

- This method is even worse then the earlier way of printing an invite for all languages.
- We are assigning all string names to a variable and then using that variable name to print in the print statement
- But hey!!!! we now know that we can print all the invites if all these names can be assigned to one variable and we sequentially print all names of languages one after the other

Let's see how finally how we can loop in all those names :

```
1 for lang in ["java", "cpp", "JavaScript", "PHP", "Ruby", "Swift", "C#"]:
2     print("Hi", lang, "Please come to my party on Saturday!")
```

```
Hi java Please come to my party on Saturday!
Hi cpp Please come to my party on Saturday!
Hi JavaScript Please come to my party on Saturday!
Hi PHP Please come to my party on Saturday!
Hi Ruby Please come to my party on Saturday!
Hi Swift Please come to my party on Saturday!
Hi C# Please come to my party on Saturday!
```

WOW!!! All invites sent in just 2 lines, that is amazing

Here's how it works:

- **lang** in this for statement is called the **loop variable** or, alternatively, the **iterator variable**.

- The **list** of languages **in** the **square brackets** is the **sequence** over which we will **iterate**.

- Line 2 is the **loop body**. *"The loop body is always indented"*. The indentation determines exactly what statements are "in the loop".

The **loop body** is **performed one time** for **each name in the list**.

- On **each iteration** or pass of the loop, first a **check** is done to see if there are still **more items** to be **processed**. If there are none left (this is called the **terminating condition** of the loop), the loop has finished.

- Program execution continues at the next statement after the loop body.If there are items still to be processed, the loop variable is updated to refer to the next item in the list. This means, in this case, that the loop body is executed here 7 times, and each time lang will refer to a different language from the list.

- At the end of each execution of the body of the loop, Python returns to the for statement, to see if there are more items to be handled.The overall syntax is
  for <loop_var_name> in <sequence>:
  Between the words **for and in**, there **must** be a **variable name** for the **loop variable.** *"You can't put a whole expression there"*.

- A colon is required at the end of the line

- **After** the word **in and before** the **colon** is an **expression** that must evaluate to a sequence (e.g, a string or a list or a tuple). It could be a literal, or a variable name, or a more complex expression.

Flow of Execution of the for Loop

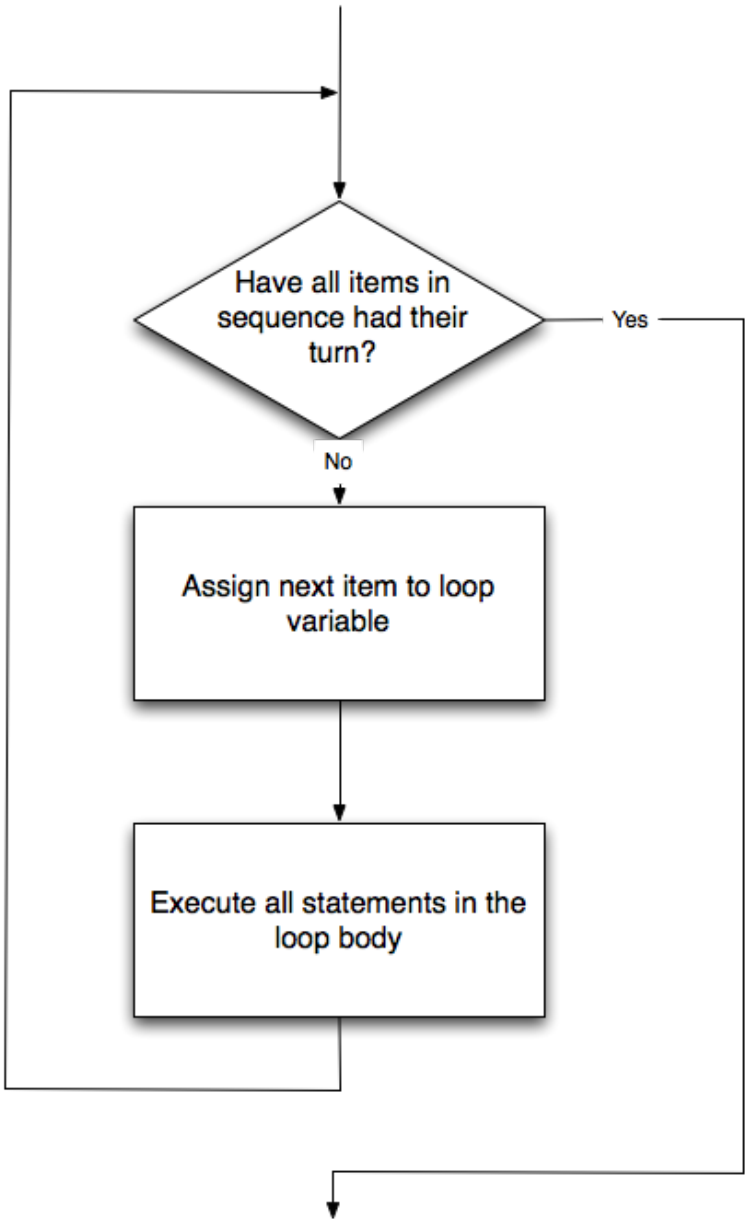
As a program executes, the interpreter always keeps track of which statement is about to be executed. We call this the control flow, or the **flow of execution of the program**.

When you execute a program, you would often use your finger to point to each statement in turn. So you could think of control flow as **“Python’s moving finger”.**

Control flow until now has been **strictly top to bottom**, **one statement at a time**. We call this type of **control sequential**.

Sequential flow of control is always assumed to be the default behavior for a computer program. The **for statement changes this**.Flow of control is often easy to visualize and understand if we draw a flowchart.

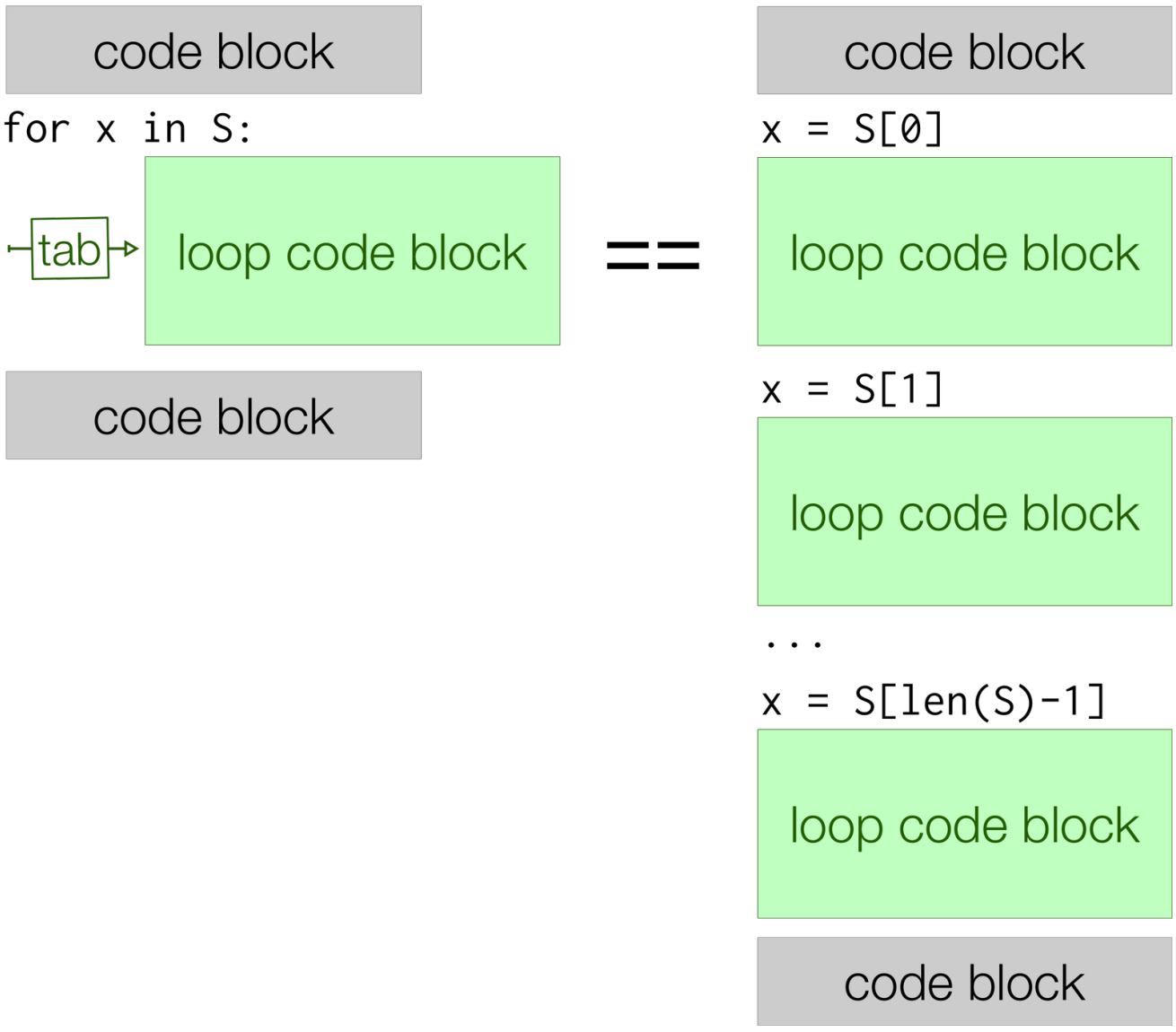
This flowchart shows the exact steps and logic of how the for statement executes.



While loops may not seem to be necessary when you're iterating over a few items, it is extremely helpful when iterating over lots of items.

Imagine if you needed to change what happened in the code block.

- On the left, when you use iteration, this is easy.
- On the right, when you have hard coded the process, this is more difficult.



Strings and for loops

Since a string is simply a sequence of characters, the for loop iterates over each character automatically. (As always, try to predict what the output will be from this code before your run it.)

```
for achar in "vertical python":
    print(achar)
```

```
v
e
r
t
i
c
a
l

p
y
t
h
o
n
```

The loop variable achar is automatically assigned each character in the string “vertical python”. **We will refer to this type of sequence iteration as iteration by item.**

Note that the for loop processes the characters in a string or items in a sequence one at a time from left to right.

Lists and for loops

It is also possible to perform **list traversal** using iteration by item. A list is a sequence of items, so the for loop iterates over each item in the list automatically.

```python
lang = ["python", "java", "cpp", "Javascript"]

for languages in lang:      # by item
    print(languages)
```

```
python
java
cpp
Javascript
```

It almost reads like natural language: For (every) fruit in (the list of) fruits, print (the name of the) fruit.

Range in for loop

We already have seen range while practicing turtle examples. A range() function returns a sequence of numbers, starting from 0 by default, and increments by 1 (by default), and stops before a specified number.

Depending on how many arguments user is passing to the function, user can decide where that series of numbers will begin and end as well as how big the difference will be between one number and the next.range() takes mainly three arguments.

- **start:**
  integer starting from which the sequence of integers is to be returned

- **stop:**
  integer before which the sequence of integers is to be returned.
  The range of integers end at stop – 1.

- **step:**
  integer value which determines the increment between each integer in the sequence

```python
# printing a number
for i in range(10):
    print(i, end =" ")
print()

# using range for iteration
l = [10, 20, 30, 40]
for i in range(len(l)):
    print(l[i], end =" ")
print()

# performing sum of natural
# number
sum = 0
for i in range(1, 11):
    sum = sum + i
print("Sum of first 10 natural number :", sum)
```

```
0 1 2 3 4 5 6 7 8 9
10 20 30 40
Sum of first 10 natural number : 55
```

There are three ways you can call range() :

- range(stop) takes one argument.

```
for i in range(10):
    print(i, end =" ")
```

- range(start, stop) takes two arguments.

```
for i in range(1, 10):
    print(i, end =" ")
```

- range(start, stop, step) takes three arguments.

```
for i in range(0, 30, 3):
    print(i, end = " ")
```

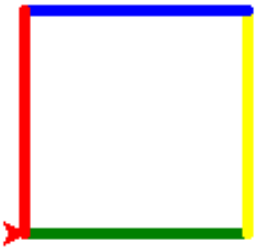
iteration simplifies our Turtle Program

Remember the turtle drawings we learned in 2nd part of this series turtle?

Let's look again at how we can use for loops there!To draw a square we'd like to do the same thing four times — move the turtle forward some distance and turn 90 degrees. We previously used 8 lines of Python code to have alex draw the four sides of a square.

This next program does exactly the same thing but, with the help of the for statement, uses just three lines (not including the setup code). Remember that the for statement will repeat the forward and left four times, one time for each value in the list.

```
1  import turtle
2  wn=turtle.Screen()
3  new_turt=turtle.Turtle()
4  new_turt.pensize('5')
5
6  for i in ['green','yellow','blue','red']:
7      new_turt.color(i)
8      new_turt.forward(100)
9      new_turt.left(90)
10
11 wn.exitonclick()
```



The Accumulator Pattern

What is Accumulator pattern?

It works by initializing a variable that keeps track of how much we have counted so far. Then we can write a for loop to go through a list of elements in a sequence and add each element's value to the accumulator.

One common programming “pattern” is to traverse(travel across) a sequence, accumulating a value as we go, such as the sum-so-far or the maximum-so-far. That way, at the end of the traversal we have accumulated a single value, such as the sum total of all the items or the largest item.

The anatomy of the accumulation pattern includes:

- **initializing** an “accumulator” variable to an initial value (such as 0 if accumulating a sum)
- **iterating** (e.g., traversing the items in a sequence)
- **updating** the accumulator variable on each iteration (i.e., when processing each item in the sequence)

```
1 nums = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10] #list of 10numbers
2 accum = 0  #initializing the accumulator variable
3 for i in nums:    #iteration
4     accum = accum + i    #updating the accumulator variable
5 print(accum)
```

```
55
```

Let's try to understand the small program in depth, what exactly happened :

- Notice that the variable **accum** starts out with a value of 0.
- Next, the iteration is performed 10 times. Inside the for loop, the update occurs.
- i has the value of current item (1 the first time, then 2, then 3, etc.). accum is reassigned a new value which is the old value plus the current value of i.
- This pattern of iterating the updating of a variable is commonly referred to as **the accumulator pattern**.
- We **refer** to the **variable** as the **accumulator**. This pattern will come up over and over again. Remember that the key to making it work successfully is to be sure to initialize the variable before you start the iteration.
- Once **inside** the **iteration**, it is required that you **update** the **accumulator**.

```
1  nums = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10] #list of 10numbers
2  accum = 0  #initializing the accumulator variable
3  for i in nums:    #iteration
4      accum = accum + i    #updating the accumulator variable
5      print(accum)
```

```
1
3
6
10
15
21
28
36
45
55
```

We indented the print accum statement into for loop and we got the result of everything that is getting add up during every iteration.

Range with Accumulator Pattern

Let us see how can we utilize the **range function** in accumulator pattern.. Previously, you've seen it being used when we wanted to draw in turtle in part 2 series of these books. Also we saw a little more detail about it in the section **"Range with For loop"**

The range function takes at least one input - which should be an integer - and returns a list as long as your input. While you also know by now that you can provide two inputs with start and stop option.

```
print("range(5): ")
for i in range(5): #With one input, range will start at zero and go up to - but not include - the input
    print(i)

print("range(0,5): ")
for i in range(0, 5): #With 2 inputs, range will start at your first input
                      #and go up to the second input -1
    print(i)

# Notice the casting of `range` to the `list`
print(list(range(5)))
print(list(range(0,5)))
```

```
range(5):
0
1
2
3
4
range(0,5):
0
1
2
3
4
[0, 1, 2, 3, 4]
[0, 1, 2, 3, 4]
```

```
accum = 0
for i in range(11):
    accum = accum + i
print(accum)

# or, if you use two inputs for the range function

sec_accum = 0
for i in range(1,11):
    sec_accum = sec_accum + i
print(sec_accum)

```

```
55
55
```

```
nums = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
count = 0
for i in nums:
    count = count + 1
print(count)
```

```
10
```

In this example we don't make use of i even though the iterator variable (loop variable) is a necessary part of constructing a for loop.
Instead of adding the value of i to count we add a 1 to it, because we're incrementing the value of count when we iterate each time through the loop.
Though in this scenario we could have used the len function, there are other cases later on where len won't be useful but we will still need to count.

Traversal and the for Loop by Index

With a for loop, the loop variable is bound, on each iteration, to the next item in a sequence. Sometimes, it is natural to think about iterating through the positions, or indexes of a sequence, rather than through the items themselves.

For example, consider the list ['apple', 'pear', 'apricot', 'cherry', 'peach']. 'apple' is at position 0, 'pear' at position 1, and 'peach' at position 4.Thus, we can iterate through the indexes by generating a sequence of them, using the range function.

```
fruits = ['apple', 'pear', 'apricot', 'cherry', 'peach']
for n in range(5):
    print(n, fruits[n])
```

```
0 apple
1 pear
2 apricot
3 cherry
4 peach
```

Glossary:

Control Flow
Also known as the **flow of execution**, the order in which a program executes. By default, the control flow is ***sequential***.

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.

Iterator
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.

Loop body
The loop body contains the statements of the program that will be iterate through upon each loop. The loop body is always indented.

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.

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].

sequential flow
The execution of a program from top to bottom, one statement at a time

terminating condition
A condition which stops an iteration from continuing

traverse
To iterate through the elements of a collection, performing a similar operation on each.

Practice questions:

Question 1 : First day of school, Jack, Kack, Lack, Mack, Nack, Ouack, Pack, and Quack were the students of class. There is a automated call of their names in order. but the coder of the autobot made a mistake and now Ouack and Quack are getting misspelled. try to fix it

Program:

```
prefixes = "JKLMNOPQ"
suffix = "ack"

for p in prefixes:
  print('Welcome to the class',p + suffix)
```

Question 2: Get the user to enter some text and print it out in reverse order. Do not use any function or split function or stack or recursion approach.

Question 3: Assume you have a list of numbers (12, 10, 32, 3, 66, 17, 42, 99, 20)

Write a loop that prints each of the numbers on a new line.
Write a loop that prints each number and its square on a new line.

Question 4: Write a program that asks the user for the number of sides, the length of the side, the color, and the fill color of a regular polygon. The program should draw the polygon and then fill it in.

Question 5: take an input string from the user and write down one for loop to print out each character of the string

Question 6: take input from user and then write code to count the number of characters in given string using the accumulation pattern and assign the answer to a variable num_chars.
Do NOT use the len function to solve the problem

Question 7: week_temps_f is a string with a list of fahrenheit temperatures separated by the , sign. Write code that uses the accumulation pattern to compute the average (sum divided by number of items) and assigns it to avg_temp. Do not hard code your answer (i.e., make your code compute both the sum or the number of items in week_temps_f) (You should use the .split(",") function to split by "," and float() to cast to a float)

week_temps_f = "75.1,77.7,83.2,82.5,81.0,79.5,85.7"

PART 4