Om Namah Shivay Jai Bajarang Bali Jai BhairavNath Ki

Lesson 1 INTEGERS, FLOATS & BOOLEANS

```
#!pip install nbmerge

#!nbmerge Integers,Floats&Booleans.ipynb List_Comprehension.ipynb  > merge.ipynb
```

Different data types we will encounter in Python

- Numeric - Numeric variables take values which are numbers like 9, 3.14, 0, Inf
- String - String variables are used to store textual information
- Boolean - Boolean variables have two modes either True or False. A definite judge of statements!
- Datetime - These variables are used to store date and time values such as 2020-08-01 12:23:54

Integers and Floats

Basic Arithmetic

```
# Addition
2+1
```

```
3
```

```
# Subtraction
2-5
```

```
-3
```

```
# Multiplication
2*2
```

```
4
```

```
# Division
3/2
```

```
1.5
```

```
# Floor Division
7//2
```

```
3
```

```
# Exponentiation
2**5
```

```
32
```

```
# Modulus
5%6
```

```
5
```

```
# Order of Operations follow
ed in Python
2 + 10 * 10 + 3
```

```
---------------------------------------------------------------------------
-----
NameError                                 Traceback (most recent call
last)
<ipython-input-10-c65d57169454> in <cell line: 2>()
      1 # Order of Operations follow
----> 2 ed in Python
      3 2 + 10 * 10 + 3

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

```
2+ 10* (10+3)
```

```
# Scientific Notation for representing large numbers
4E6
```

Let's talk about numbers!

- A lot many different types of numbers are supported in Python like integers (int type), real numbers (float type), complex numbers. We will mostly use integer and floating point numbers.
- Integers are just whole numbers, positive or negative. For example: 2 and -2 are examples of integers.
- Floating point numbers in Python are notable because they have a decimal point in them, or use an exponential (E) to define the number. For example 2.0 and -2.1 are examples of floating point numbers. 4E2 (4 times 10 to the power of 2) is also an example of a floating point number in Python.
- In computing, floating-point arithmetic is arithmetic using formulaic representation of real numbers as an approximation to support a trade-off between range and

precision. You can always control the number of digits coming after the decimal, hence they are called floating-point numbers

The table below summarises the two numeric data types, Integers and Floats:

What is a Variable?

VARIABLES are entities which help us store information and retrieve it later.

- A variable with a fixed name can store information of nature like numeric, textual, boolean etc.
- A Python variable is a reserved memory location to store values. In other words, a variable in a python program gives data to the computer for processing.
- The type of data contained in a variable can be changed at user's will.

```
# You can store numbers in variables.
# The standard rule is you write the variable name followed by = sign
and the value it will take

x=5

x

y=6.4
y

print(y)
```

Basic Arithmetic operations we can do on x and y. Later we will be doing operations on thousands of such numbers in one go!

```
# Addition
z = x+y

print(z)

# Printing the memory address the variable z occupies
print(hex(id(z)))
```

- A variable can be assigned different values and data types and it will store the last value assigned

```
# Subtraction
z = x-y
```

```
# Use the in-built print function to print the variable
print(z)

# Printing the memory address the variable z occupies
print(hex(id(z)))

 # Find out the data type of variable z
type(y)

# Multiplication
z = x*y

print(z)  # Print the variable z
type(z)   # Get the data type of variable z

# Division
z = x/y

print(z) # Print the variable z
type(z)  # Get the data type of variable z

# Floor division
z= x//y  #  Remember x=5, y=6.4
print(z)
```

Waittt! Shouldn't it be 0.75??

- The reason we get this result is because we are using "*floor*" division. The // operator (two forward slashes) is the mathematical equivalent of doing [0.75] which returns the greatest integer less than or equal to 0.75

```
# Modulo operator
y=5
x=3

z = y%x       # Modulus is denoted by % sign
print(z)

# Using powers and exponents

z = x**y    # We did not even need to store it in another variable nor
use print command
print(z)

# BODMAS nostalgia
some_random_operation =(x+y)/y + (y-x)*x

print(some_random_operation)
type(some_random_operation)

# Storing large integer numbers
avogadro = 6.22E23
```

```
print(avogadro)
```

Rules for naming a variable in Python

- Variables names must start with a letter or an underscore like _ product , product_
- The remainder of your variable name may consist of letters, numbers and underscores
- spacy1, pyThon,machine_learning are some valid variable names
- Names are case sensitive.
- case_sensitive, CASE_SENSITIVE, and Case_Sensitive are each a different variable.

```
loNone = 4

onone_abc_1 = 5

list = 5

list
```

- Names cannot begin with a number. Python will throw an error when you try to do so
- Names can not contain spaces, use _ instead
- Names can not contain any of these symbols:

```
:'",<>/?|\!@#%^&*~-+
```

- It is considered best practice that names are lowercase with underscores
- Avoid using Python built-in keywords like `list`, `str` , `def` etc. We will talk more about such conventions later on

Boolean Variables

- A Boolean variable only takes two values either True or False. It is used for comparisons

Comparison Operators

- These operators will allow us to compare variables and output a Boolean value (True or False).
- If you have any sort of background in Math, these operators should be very straight forward.

- First we'll present a table of the comparison operators and then work through some examples:
- In the table below, a=3 and b=4.
- Python comes with Booleans (with predefined True and False displays that are basically just the integers 1 and 0). It also has a placeholder object called None. Let's walk through a few quick examples of Booleans (we will dive deeper into them later in this course).

```
# Set object to be a boolean
boolean_variable = False
type(boolean_variable)

#Show
boolean_variable
```

Equal

```
2 == 3

2==0
```

- Note that == is a comparison operator, while = is an assignment operator.

Not equal

```
2!=0

2!=2
```

Greater than

```
a=3
b=2
a> b

a == 3

b > 4
```

Less than

```
10 < 45

True

4 < 2

False
```

Greater than or equal to

```
3 >=2
```

```
True
```

```
4 >= 4
```

```
True
```

Less than or equal to

```
3 <= 0
```

```
False
```

```
1 <= 2
```

```
True
```

Introduction to Strings

- Strings are used in Python to record text information, such as names. It could either be a word, a phrase, a sentence, a paragraph or an entire encyclopedia. Strings in Python are actually a *sequence*, which basically means Python keeps track of every element in the string as a sequence. For example, Python understands the string "joker' to be a sequence of letters in a specific order. This means we will be able to use indexing to grab particular letters (like the first letter, or the last letter).
- This idea of a sequence is an important one in Python and we will touch upon it later on in the future.

Creating a String

- To create a string in Python you need to use either single quotes or double quotes. For example:

```
# Single word
my_first_string= 'algebra'
my_first_string
```

```
{"type":"string"}
```

```
# Entire phrase
phrase = 'Statistics sits at the heart of machine learning'
print(phrase)
```

```
Statistics sits at the heart of machine learning
```

```
# Statement to get the type of the variable
type(phrase)
```

```
str
```

```
# We can also use double quote
my_string = "String built with double quotes"
print(my_string)   # Use the print command
```

```
String built with double quotes
```

```
# Be careful with quotes!
sentence= 'I'm using single quotes, but this will create an error'
print(sentence)
```

```
  File "<ipython-input-21-5e7b6d7da870>", line 2
    sentence= 'I'm using single quotes, but this will create an error'
                                                                     ^
SyntaxError: unterminated string literal (detected at line 2)
```

- The reason for the error above is because the single quote in I'm stopped the string. You can use combinations of double and single quotes to get the complete statement.

```
sentence= "I'm using single quotes, but this will create an error"
print(sentence)

hashtag = "#"
print(hashtag)

type(hashtag)
```

How to print strings

- Using Jupyter notebook with just a string in a cell will automatically output strings, but the correct way to display strings in your output is by using a print function.

```
# We can simply declare a string
'Deep Learning'

# Note that we can't output multiple strings this way
'Linear Algebra'
'Calculus'
```

We can use the print() statement to print a string.

```
# print('Linear Algebra')
# print('Calculus')
print('Use \n to print a new line')
print('\n')
print('See what \n I mean?')
```

Playing with strings

- We can also use a function called len() to check the length of a string!

```
algo = 'regres sion '
len(algo)
```

Python's built-in len() function counts all of the characters in the string, including spaces and punctuation.

String Indexing

- We know strings are a sequence, which means Python can use indexes to call parts of the sequence.
- A string index refers to the location of an element present in a string.
- The indexing begins from 0 in Python.
- The first element is assigned an index 0, the second element is assigned an index of 1 and so on and so forth.
- In Python, we use brackets [] after an object to call its index.

```
# Assign string as a string
string  = 'Principal Component Analysis!'

# Print the object
print(string)
```

- Let's start indexing!

```
# Show first element (in this case a letter)
print(string[5])

print(string[15])

len(string)

# Grab the element at the index -1, which is the LAST element
print(string[28])

print(string[-2])
```

String Slicing

- We can use a : to perform *slicing* which grabs everything up to a designated point.
- The starting index is specified on the left of the : and the ending index is specified on the right of the :.
- Remember the element located at the right index is not included.

```
# Grab everything past the first term all the way to the length of s
which is len(s)
```

```
print(string)
print(string[:13])

string[13]

string[12]

# Grab everything starting from index 10 till index 18
print(string[10:])
```

- If you do not specify the ending index, then all elements are extracted which comes after the starting index including the element at that starting index. The operation knows only to stop when it has run through the entire string.

```
print(string[3:5])
```

- If you do not specify the starting index, then all elements are extracted which comes befores the ending index excluding the element at the specified ending index. The operation knows only to stop when it has extracted all elements before the element at the ending index.

```
print(string[2:4])
```

- If you do not specify the starting and the ending index, it will extract all elements of the string.

```
#Everything
print(string[:])
print(string)
```

- We can also use negative indexing to go backwards.

```
# Last letter (one index behind 0 so it loops back around)
string[-29]
```

- We can also extract the last four elements. Remember we can use the index -4 to extract the FOURTH LAST element

```
string[-1:-4]

string[-4:]
```

- We can also use index and slice notation to grab elements of a sequence by a specified step size (the default is 1). For instance we can use two colons in a row and then a number specifying the frequency to grab elements. For example:

```
# Grab everything, but go in steps size of 1
print(string[::1])

print(string[::3])
```

```
# Grab everything, but go in step sizes of 5
print(string[5:15:5])
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-22-54ab99977e51> in <cell line: 2>()
      1 # Grab everything, but go in step sizes of 5
----> 2 print(string[5:15:5])

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

```
string[::1]
```

```
# We can use this to print a string backwards
string[::-1]
```

```
# We can use this to print a string backwards with steps
string[2:4:-1]
```

```
string[4:2:-1]
```

```
s = 'foobar'
s[0::-3]
```

String Properties

- It's important to note that strings have an important property known as *immutability*.
- This means that once a string is created, the elements within it can not be changed or replaced via item assignment. We will see how we can do such operation using string methods

```
# Can we change our string 'Hello' to 'Cello'? Lets try replacing the first letter H with C
string='Hello'
string[0] = 'C'
```

- Notice how the error tells us directly what we can't do, that is we can't change the item assignment!
- Something we *can* do is concatenate strings!

```
# Concatenate strings!

string1='abc'
string2='def'
print(string1 + ' ' + string2 )
```

```
print(string1 + 4 + string2)
```

- To convert an integer into a string, you can use the str() function or you can simply write the number in quotes

```
# Concatenate strings!

string1='abc'
string2='def'
num = 4
print(string1 + str(4) + string2)

str(num)

# Concatenate strings!
string1='abc'
string2='def'
string1 + '4'+ string2

print(string)

# We can reassign string completely though!
string = string + ' concatenate me!'
print(string)

letters = 'wubba'
letters*3
```

String functions and methods

```
algorithm = 'Neural Networks'

print(algorithm)
```

len()

- len() function returns the length of the string

```
# Print the length of the string
len(algorithm)
```

lower()

- lower() method converts the string to lowercase

```
# Conver the string to lowercase
algorithm.lower()

---------------------------------------------------------------------------
NameError                                 Traceback (most recent call
last)
```

```
<ipython-input-23-d259a585a69e> in <cell line: 2>()
      1 # Conver the string to lowercase
----> 2 algorithm.lower()

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

```
print(algorithm)
```

- What about lower(algorithm)? Is it not similar to the len()function?
- An important point to note here is len() is a function and lower() is a method for a string object. It will be pretty clear about the exact differences between a function and a method as we move ahead.

```
# Lets try that out
lower(algorithm)
```

upper()

- upper() method converts the string to uppercase

```
# Convert the string to uppercase
algorithm.upper()

print(algorithm)
```

count()

- count() method returns the count of a string in the given string. Unlike lower() and upper() method, the count() method takes a string as an argument

```
print(algorithm)

algorithm.count('Ne')

algorithm.count('eu')

algorithm.count(' ')

algorithm.count('Neural')

algorithm.count('Neurla')
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call
last)
<ipython-input-24-8bdc9d14fe82> in <cell line: 1>()
----> 1 algorithm.count('Neurla')

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

find()

- find() method returns the index of the first occurrence of a string present in a given string. Similar to the count() method, the find() method takes a string as an argument

```
print(algorithm)

algorithm.find('e')

algorithm.find('Neural')

algorithm.find('Box')
```

- An important point to note here is that if the string which you are looking for, is not contained in the original string, then the find method will return a value of -1

replace()

- replace() method takes two arguments - (i) the string to replace and (ii) the string to replace with, and returns a modified string after the operation

```
print(algorithm)

algorithm.replace(' ','-')

algorithm.replace('N','L')

print(algorithm)
```

- Another important point worth noting here is applying any method on a string does not actually change the original string. For example, when you print out the algorithm string, it still contains the original value 'Neural Networks'

- We need to store the modified string in another variable

```
# Storing the modified string
algorithm_revised  = algorithm.replace('Neural','Artificial Neural')
print(algorithm_revised)
print(algorithm)
```

Printing strings a bit differently

```
first_name = 'Rahul'
last_name = 'Modi'

full_name = f'Left plus right makes {first_name}  {last_name}'  # Use
{} to print the variable you want to
print(full_name)
```

```
Left plus right makes Rahul  Modi
```

```
print(first_name + ' '+last_name)
```

```
Rahul Modi
```

```
first_name = 'Vikash'
middle_name = ''
last_name = 'Srivastava'

full_name = f'I am none other than {first_name} {middle_name}
{last_name}. I am a Data Scientist'
print(full_name)
```

```
I am none other than Vikash Srivastava. I am a Data Scientist
```

```
print(f'I am none other than {first_name} {middle_name}{last_name}. I
am a Data Scientist')
```

```
I am none other than Vikash Srivastava. I am a Data Scientist
```

Check if a string contains a particular word or character

```
my_string = 'Albert Einstein'

'Albert' in my_string
```

```
True
```

```
'Alberta' in my_string
```

```
False
```

Earlier when discussing strings we introduced the concept of a sequence in Python.

- Lists can be thought of the most general version of a *sequence* in Python.
- Unlike strings, they are mutable, meaning the elements inside a list can be changed!
- Lists are constructed with brackets [] and commas separating every element in the list.

```
# Assign a list to an variable named my_list
my_list = [1,2,3,4]

print(my_list)
```

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

```
type(my_list)
```

```
list
```

- We just created a list of integers, but lists can actually hold different object types. For example:

```
my_list = ['A string',23,100.232,'o',True]

my_list
```

```
['A string', 23, 100.232, 'o', True]
```

- Just like strings, the len() function will tell you how many items are in the sequence of the list.

```
len(my_list)
```

```
5
```

List Indexing

- Indexing work just like in strings.
- A list index refers to the location of an element in a list.
- Remember the indexing begins from 0 in Python.
- The first element is assigned an index 0, the second element is assigned an index of 1 and so on and so forth.

```
# Consider a list of strings
loss_functions = ['Mean Absolute Error','Mean Squared Error','Huber
Loss','Log Loss','Hinge Loss']
print(loss_functions)
```

```
['Mean Absolute Error', 'Mean Squared Error', 'Huber Loss', 'Log
Loss', 'Hinge Loss']
```

```
# Grab the element at index 0, which is the FIRST element
print(loss_functions[0])
```

```
Mean Absolute Error
```

```
len(loss_functions)
```

```
5
```

```
# Grab the element at index 3, which is the FOURTH element
print(loss_functions[3])
```

```
Log Loss
```

```
print(loss_functions[6])
```

```
---------------------------------------------------------------------------
IndexError                                Traceback (most recent call last)
<ipython-input-42-36fa04b37479> in <cell line: 1>()
----> 1 print(loss_functions[6])

IndexError: list index out of range
```

```
# Grab the element at the index -1, which is the LAST element
print(loss_functions[-1])

# Grab the element at the index -3, which is the THIRD LAST element
print(loss_functions[-3])
```

List Slicing

- We can use a : to perform *slicing* which grabs everything up to a designated point.
- The starting index is specified on the left of the : and the ending index is specified on the right of the :.
- Remember the element located at the right index is not included.

```
# Print our list
print(loss_functions)

# Grab the elements starting from index 1 and everything past it
loss_functions[1:4]
```

- If you do not specify the ending index, then all elements are extracted which comes after the starting index including the element at that starting index. The operation knows only to stop when it has run through the entire list.

```
# Grab everything starting from index 2
loss_functions[2:]
```

- If you do not specify the starting index, then all elements are extracted which comes befores the ending index excluding the element at the specified ending index. The operation knows only to stop when it has extracted all elements before the element at the ending index.

```
# Grab everything before the index 4
loss_functions[:4]
```

- If you do not specify the starting and the ending index, it will extract all elements of the list.

```
# Grab everything
print(loss_functions[:])
```

- We can also extract the last four elements. Remember we can use the index -4 to extract the FOURTH LAST element

```
# Grab the LAST FOUR elements of the list
loss_functions[-4:]

loss_functions[-10]
```

- It should also be noted that list indexing will return an error if there is no element at that index.

```
len(loss_functions)

loss_functions[8]
```

List Operations

- Remember we said that lists are mutable as opposed to strings. Lets see how can we change the elements of a list
- We can also use + to concatenate lists, just like we did for strings.

```
print(loss_functions)

loss_functions + ['Kullback-Leibler']
```

- Note: This doesn't actually change the original list!

```
print(loss_functions)
```

- You would have to reassign the list to make the change permanent.

```
# Reassign
loss_functions = loss_functions + ['Kullback-Leibler']

loss_functions
```

- We can also use the * for a duplication method similar to strings:

```
# Make the list double
len(loss_functions * 6)
```

List Functions

len()

- len() function returns the length of the list

```
print(loss_functions)

len(loss_functions)
```

min()

- min() function returns the minimum element of the list
- min() function only works with lists of similar data types

```
new_list = [6, 9, 1, 3, 5.5]
min(new_list)
my_new_list = ['a','b', 'z' ,'y' , 'm']
min(my_new_list)
```

max()

- max() function returns the maximum element of the list
- max() function only works with lists of similar data types

```
new_list = ['Argue','Burglar','Parent','Linear','shape']
max(new_list)
```

sum()

- sum() function returns the sum of the elements of the list
- sum() function only works with lists of numeric data types

```
new_list = [6, 9, 1, 3, 5.5]
sum(new_list)
```

sorted()

- sorted() function returns the sorted list
- sorted() function takes reverse boolean as an argument
- sorted() function only works on a list with similar data types

```
new_list
sorted(new_list)
print(new_list)
new_list = ['Argue','Burglar','Parent','Linear','shape']
sorted(new_list)
print(new_list)
```

```
sorted(new_list,reverse=True)

sorted(new_list)
```

- sorted() function does not change our list

```
new_list
```

List Methods

- If you are familiar with another programming language, you might start to draw parallels between arrays in another language and lists in Python. Lists in Python however, tend to be more flexible than arrays in other languages for a two good reasons: they have no fixed size (meaning we don't have to specify how big a list will be), and they have no fixed type constraint (like we've seen above).
- Let's go ahead and explore some more special methods for lists:

```
# Create a new list
my_list = [1,2,3,1,1,1,3,10,5,8]
```

append()

- Use the append() method to permanently add an item to the end of a list.
- append() method takes the element which you want to add to the list as an argument

```
# Print the list
my_list

len(my_list)

# Append to the end of the list
my_list.append('Append me!')
```

- Ah darn it! I was expecting some output. Lets see what happened to my_list

```
print(my_list)

len(my_list)
```

- Woah! Calling the append() method changed my list? Yes, the append() method changes your original list!

```
# Show
my_list.append(2.73)

print(my_list)
```

- We can in fact add a list object to our my_list object

```
my_list.append([1,2,3])

my_list

len(my_list)

my_list.append([10,[19,20],30])

len(my_list)
```

extend()

- Use the extend() method to merge a list to an existing list
- extend() method takes a list or any iterable(don't worry about it now) as an argument.
- Quite helpful when you have two or more lists and you want to merge them together

```
# Print the list
print(my_list)

my_list.extend(['Wubba','Lubba','Dub Dub'])

print(my_list)

len(my_list)
```

pop()

- Use pop() to "pop off" an item from the list.
- By default pop() takes off the element at the last index, but you can also specify which index to pop off.
- pop() takes the index as an argument and returns the elenent which was popped off.

```
# Print the list
print(my_list)

# Pop off the 0 indexed item
my_list.pop()
```

- pop() method changes the list by popping off the element at the specified index

```
print(my_list)

# Assign the popped element, remember default popped index is -1
my_list.pop(-1)

len(my_list)
```

```
print(my_list)
```

remove()

- Use remove() to remove an item/element from the list.
- By default remove() removes the specified element from the list.
- remove() takes the element as an argument.

```
# Print the list
print(my_list)

# Remove the element which you want to
my_list.remove([10, [19, 20], 30])

# Show
print(my_list)

my_list.remove([1,2,3])

print(my_list)

len(my_list)

my_list.remove(1)

print(my_list)

len(my_list)

my_list.clear
```

count()

- The count() method returns the total occurrence of a specified element in the list

```
print(my_list)

# Count the number of times element 1 occurs in my_list
my_list.count('Wubba')

my_list.count(1)
```

index()

- The index() method returns the index of a specified element.

```
print(my_list)

my_list.index('Wubba')

my_list.index(3)
```

sort()

- Use sort() to sort the list in either ascending/descending order
- The sorting is done in ascending order by default
- sort() method takes the reverse boolean as an argument
- sort() method only works on a list with elements of same data type

```
new_list = [6, 9, 1, 3, 5]

# Use sort to sort the list (this is permanent!)
new_list.sort()

print(new_list)

# Use the reverse boolean to set the ascending or descending order
new_list.sort(reverse=True)
print(new_list)

boolean_list = [True, False]

boolean_list.sort(reverse=True)

print(boolean_list)
```

reverse()

- reverse() method reverses the list

```
my_list = [1, 1, 1, 1, 1.43, 2, 3, 3, 5, 8, 10, 'Lubba', 'Dub Dub']

print(my_list)

my_list.reverse()

print(my_list)
```

Nested Lists

- A great feature of of Python data structures is that they support *nesting*. This means we can have data structures within data structures. For example: A list inside a list.

```
# Let's make three lists
lst_1=[1,2,3]
lst_2=['b','a','d']
lst_3=[7,8,9]

# Make a list of lists to form a matrix
list_of_lists = [lst_1,lst_2,lst_3]

print(list_of_lists)
```

```
# Show
type(list_of_lists)

# Grab first item in matrix object
list_of_lists[1]

# Grab first item of the first item in the matrix object
list_of_lists[1]

list_of_lists[1][1][][][]
```

Tuples

- In Python tuples are very similar to lists, however, unlike lists they are *immutable* meaning they can not be changed.
- You would use tuples to present things that shouldn't be changed, such as days of the week, or dates on a calendar.
- You'll have an intuition of how to use tuples based on what you've learned about lists. We can treat them very similarly with the major distinction being that tuples are immutable.

Constructing Tuples

- The construction of a tuples use () with elements separated by commas.

```
# Create a tuple
my_tuple = (1,'a',3)

print(my_tuple)

type(my_tuple)

# Can also mix object types
another_tuple = ('one',2, 4.53, 'asbc')

# Show
another_tuple

my_list = [1]

type(my_list)

my_tuple = (1,)

type(my_tuple)

my_tuple = 1,2,3
```

```
type(my_tuple)

my_tuple = (1,2,3,4)

len(my_tuple)
```

Tuple Indexing

- Indexing work just like in lists.
- A tuple index refers to the location of an element in a tuple.
- Remember the indexing begins from 0 in Python.
- The first element is assigned an index 0, the second element is assigned an index of 1 and so on and so forth.

```
print(another_tuple)

# Grab the element at index 0, which is the FIRST element
another_tuple[0]

# Grab the element at index 3, which is the FOURTH element
another_tuple[3]

# Grab the element at the index -1, which is the LAST element
another_tuple[-1]

# Grab the element at the index -3, which is the THIRD LAST element
another_tuple[-3]
```

Tuple Slicing

- We can use a : to perform *slicing* which grabs everything up to a designated point.
- The starting index is specified on the left of the : and the ending index is specified on the right of the :.
- Remember the element located at the right index is not included.

```
# Print our list
print(another_tuple)

# Grab the elements starting from index 1 and everything past it
another_tuple[1:3]
```

- If you do not specify the ending index, then all elements are extracted which comes after the starting index including the element at that starting index. The operation knows only to stop when it has run through the entire tuple.

```
# Grab everything starting from index 2
another_tuple[2:]
```

- If you do not specify the starting index, then all elements are extracted which comes befores the ending index excluding the element at the specified ending index. The operation knows only to stop when it has extracted all elements before the element at the ending index.

```
# Grab everything before the index 4
another_tuple[:6]
```

- If you do not specify the starting and the ending index, it will extract all elements of the tuple.

```
# Grab everything
another_tuple
```

- We can also extract the last four elements. Remember we can use the index -4 to extract the FOURTH LAST element

```
# Grab the LAST FOUR elements of the list
another_tuple[-4:]
```

- It should also be noted that tuple indexing will return an error if there is no element at that index.

```
another_tuple[5]

# Check len just like a list
len(another_tuple)

random_tuple = (1,5,6,2)

sorted(random_tuple)
```

Tuple Methods

- Tuples have built-in methods, but not as many as lists do.

index()

- The index() method returns the index of a specified element.

```
my_tuple =(1,2,3,4,5,6,1,1,2)

# Use .index to enter a value and return the index
my_tuple.index(2)
```

count()

- The count() method returns the total occurrence of a specified element in a tuple

```
# Use .count to count the number of times a value appears
my_tuple.count(2)

my_tuple.count(1)

my_tuple.

my_list = [1,2,3,4]

my_list.
```

Immutability

- It can't be stressed enough that tuples are immutable.

```
print(my_tuple)

my_tuple[0]

my_tuple[0] = 'change'
```

- Because of this immutability, tuples can't grow. Once a tuple is made we can not add to it.
- Let us consider a list and lets see if we can do this operation on them

```
# Create a list
my_list = [5,7,9,3,2]

# Replace the FIRST element with the value 1
my_list[0] = 1

# Our modified list
my_list
```

- Tuple does not support methods such as append(), extend(), remove(), pop()

```
my_tuple.append('Great!')
```

zip()

- zip() function takes multiple lists as arguments and zips them together
- This function returns a list of n-paired tuples where n is the number of lists being zipped

```
city_list = ['Delhi','Patna','Cuttack','Guwahati']
river_list = ['Yamuna','Ganga','Mahanadi','Brahmaputra']

zip(city_list, river_list)

city_and_their_rivers = list(zip(city_list,river_list))
```

```
city_and_their_rivers

city_list = ['Delhi','Patna','Cuttack','Guwahati']
river_list = ['Yamuna','Ganga','Mahanadi','Brahamputra','Thames']

list(zip(city_list,river_list))
```

When to use Tuples

- You may be wondering, "Why bother using tuples when they have fewer available methods?" To be honest, tuples are not used as often as lists in programming, but are used when immutability is necessary. If in your program you are passing around an object and need to make sure it does not get changed, then a tuple becomes your solution. It provides a convenient source of data integrity.
- You will find them often in functions when you are returning some values
- You should now be able to create and use tuples in your programming as well as have an understanding of their immutability.

Sets

- Sets are an unordered collection of *unique* elements. We can construct them by using the set() function.
- Sets cannot have duplicates.
- Sets are mutable just like lists.
- You can create a non-empty set with curly braces by specifying elements separated by a comma.

```
# Create an empty set
empty_set = set()

type(empty_set)

# Create a non-empty set within curly braces
non_empty_set = {1,6,4,'abc'}

type(non_empty_set)
```

A Time for Caution

- An empty set cannot be represented as {}, which is reserved for an empty dictionary which we will get to know in a short while

```
my_object = {}

type(my_object)
```

```
my_set = set()

type(my_set)
```

- We can cast a list with multiple repeat elements to a set to get the unique elements.

```
# Create a list with repeats
my_list = [1,1,2,2,3,4,5,6,1,1]

# Cast as set to get unique values
my_set = set(my_list)

my_set

# A set cannot have mutable items
my_set = {1, 2, (3, 4)}
```

- We cannot create a set whose any of the elements is a list

```
# But we can have tuples as set elements, they are immutable
my_set = {1, 2, (2,3)}

my_set

my_set = {1,2, {3,5}}

my_set
```

add()

- add() method adds an element to a set
- This method takes the element to be added as an argument

```
# We add to sets with the add() method
my_set = set()
my_set.add('a')

#Show
my_set

# Add a different element
my_set.add(2)

#Show
print(my_set)

# Lets add another element 1
x.add(1)

#Show
x
```

```
# Try to add the same element 1 again
x.add(1)

x
```

- Notice how it won't place another 1 there. That's because a set is only concerned with unique elements!

update()

- update() method helps to add multiple elements to a set

```
my_set = {5,7,9,3}

# add multiple elements
my_set.update([2, 3, 4])
print(my_set)
```

remove()

- Use remove() to remove an item/element from the set.
- By default remove() removes the specified element from the set.
- remove() takes the element as an argument.

```
non_empty_set = {1,5,6,73,2}

non_empty_set.remove(5)

non_empty_set

non_empty_set.remove(45)
```

- remove() throws an error when we try to remove an element which is not present in the set

union()

- union() method returns the union of two sets
- Also denoted by the operator |

```
# Initialize sets A and B
A = {1, 2, 3, 4, 5}
B = {4, 5, 6, 7, 8}

A.union(B)

A

# Also denoted by the operator |
A | B
```

intersection()

- intersection() method returns the intersection of two sets
- Also denoted by the operator &

```
A = {1, 2, 3, 4, 5}
B = {4, 5, 6, 7, 8}

A.intersection(B)

# Also denoted by the operator &
A & B
```

difference()

- difference() method returns the difference of two sets
- Difference of the set B from set A i.e, (A - B) is a set of elements that are only in A but not in B
- Also denoted by the operator -

```
A = {1, 2, 3, 4, 5}
B = {4, 5, 6, 7, 8}

A.difference(B)

# Also denoted by the operator -
A - B

B.difference(A)

B - A
```

symmetric_difference()

- symmetric_difference() method returns the set of elements in A and B but not in both (excluding the intersection)
- Also denoted by the operator ^

```
A = {1, 2, 3, 4, 5}
B = {4, 5, 6, 7, 8}

A.symmetric_difference(B)

A^B
```

Dictionaries

- We've been learning about *sequences* in Python but now we're going to switch gears and learn about *mappings* in Python.
- If you're familiar with other languages you can think of these Dictionaries as hash tables.
- So what are mappings? Mappings are a collection of objects that are stored by a *key*, unlike a sequence that stored objects by their relative position. This is an important distinction, since mappings won't retain order since they have objects defined by a key.
- A Python dictionary consists of a key and then an associated value. That value can be almost any Python object. So a dictionary object always has elements as key-value pairs

Constructing a Dictionary

- A dictionary object is constructed using curly braces {key1:value1,key2:value2,key3:value3}

```
# Make a dictionary with {} and : to signify a key and a value
marvel_dict = {'Name':'Thor','Place':'Asgard','Weapon' : 'Hammer',
1:2, 3 : 'power', 'alibies' : ['Ironman','Captain America'], 'abc' :
{1:2, 4:5}}

# Call values by their key
marvel_dict['Place']

type(marvel_dict)

marvel_dict['Name']

marvel_dict['Random']

marvel_dict['Weapon']

marvel_dict['alibies']

type(marvel_dict['abc'])
```

Dictionary Methods

keys()

- keys() method returns the list of keys in the dictionary object

```
marvel_dict.keys()

list(marvel_dict.keys())
```

values()

- values() method returns the list of values in the dictionary object

```
print(marvel_dict)

list(marvel_dict.values())
```

items()

- items() method returns the list of the keys and values

```
# Get the keys and their corresponding values
list(marvel_dict.items())
```

- We can also use the get() method to extract a particular value of key-value pair.

```
marvel_dict.get('Place')
```

get()

- get() method takes the key as an argument and returns None if the key is not found in the dictionary.
- We can also set the value to return if a key is not found. This will be passed as the second argument in get()

```
marvel_dict.get('Place')

marvel_dict.get('Random')


marvel_dict.get('Random','Not Found')

marvel_dict['friend']
```

Its important to note that dictionaries are very flexible in the data types they can hold. For example:

```
employee_dict = {'Name':'Sanket','Skills':['Python','Machine
Learning','Deep Learning'],'Band':6.0,'Promotion Year':
[2016,2018,2020]}

len(employee_dict.keys())

# Let's call items from the dictionary
employee_dict['Skills']

# Can call an index on that value
employee_dict['Skills'][0]
```

```
# Can then even call methods on that value
employee_dict['Skills'][0].upper()

# Add a new key

employee_dict['Designation'] ='Senior Data Scientist'

employee_dict
```

update()

- You can add an element which is a key-value pair using the update() method
- This method takes a dictionary as an argument

```
employee_dict.update({'Salary':'2,000,000'})

employee_dict
```

- We can also use the update() method to update the existing values for a key

```
employee_dict.update({'Name' : 'Varun Saini'})

employee_dict
```

- We can affect the values of a key as well without the update() method

```
employee_dict['Name']

# Subtract 123 from the value
employee_dict['Name'] = employee_dict['Name'] + ' ' + 'Raj'

#Check
employee_dict
```

dict()

- We can also create dictionary objects from sequence of items which are pairs. This is done using the dict()method
- dict() function takes the list of paired elements as argument

```
country_list = ['India','Australia','United States','England']
city_list = ['New Delhi', 'Canberra' , 'Washington DC','London']

country_city_list = list(zip(country_list,city_list))

country_city_list
```

```
[('India', 'New Delhi'),
 ('Australia', 'Canberra'),
```

```
 ('United States', 'Washington DC'),
 ('England', 'London')]

dict(country_city_list)

{'India': 'New Delhi',
 'Australia': 'Canberra',
 'United States': 'Washington DC',
 'England': 'London'}

# Let us create a list of paired tuples
country_city_tuples = [('India','New Delhi'),('Australia','Canberra'),
('United States','Washington DC'),('England','London')]

country_city_dict = dict(country_city_tuples)

country_city_dict

{'India': 'New Delhi',
 'Australia': 'Canberra',
 'United States': 'Washington DC',
 'England': 'London'}
```

pop()

- pop() method removes and returns an element from a dictionary having the given key.
- This method takes two arguments/parameters (i) key - key which is to be searched for removal, (ii) default - value which is to be returned when the key is not in the dictionary

```
country_city_dict.pop('England')

{"type":"string"}

country_city_dict

{'India': 'New Delhi',
 'Australia': 'Canberra',
 'United States': 'Washington DC'}

element_to_pop = country_city_dict.pop('England')

---------------------------------------------------------------------------
KeyError                                  Traceback (most recent call
last)
<ipython-input-161-f98023b73e7b> in <cell line: 1>()
----> 1 element_to_pop = country_city_dict.pop('England')

KeyError: 'England'
```

```
element_to_pop

country_city_dict
```

We can use the zip() and dict() methods to create a dictionary object from two lists

```
name = ["Manjeet", "Nikhil", "Shambhavi"]
marks = [40, 50, 60]

mapped = zip(name, marks)

mapped

print(dict(mapped))

name = ["Manjeet", "Nikhil", "Shambhavi"]
marks = [40, 50, 60,80]

mapped = zip(name, marks)

print(dict(mapped))
```

Understanding the difference between Statements and Expressions

- A Statement is an instruction that the Python interpreter can execute. We have only seen the assignment statement so far. Some other kinds of statements that we'll see shortly are while statements, for statements, if statements, and import statements. (There are other kinds too!)
- An Expression is a combination of values, variables, operators, and calls to functions. Expressions need to be evaluated. If you ask Python to print an expression, the interpreter evaluates the expression and displays the result.

```
# Assignment statement
x = 3

print(1 + 1)
print(len("Hello Team AlmaBetter"))
```

- In this example len() is a built-in Python function that returns the number of characters in a string. We've previously seen the print() and the type() functions, so this is an example of another function!
- The evaluation of an expression produces a value, which is why expressions can appear on the right hand side of assignment statements. A value all by itself is a

simple expression, and so is a variable. Evaluating a variable gives the value that the variable refers to.

```
y = 3.14

y

a = 'Mighty'
# b = a == True

a == True
```

- Note that when we enter the assignment statement, y = 3.14 , only the prompt is returned. There is no value. This is due to the fact that statements, such as the assignment statement, do not return a value. They are simply executed.

```
# Import statement
import pandas as pd

print(y)

y
```

- On the other hand, the result of executing the assignment statement is the creation of a reference from a variable, y, to a value, 3.14. When we execute the print() function working on y, we see the value that y is referring to. In fact, evaluating y by itself results in the same response.
- Instructions that a Python interpreter can execute are called statements. For example, a = 1 is an assignment statement. if statement, for statement, while statement, etc. are other kinds of statements which will be discussed later.

Multi-line Statements

- In Python, the end of a statement is marked by a newline character. But we can make a statement extend over multiple lines with the line continuation character ().

```
x = 1
y = 2
+ 5
z = 3

y

a = 1 + 2 + 3 + 4 +
5 + 6+ 7+ 8+
9 + 10

a
```

- This is an explicit line continuation. In Python, line continuation is implied inside parentheses ( ), brackets [ ], and braces { }. For instance, we can implement the above multi-line statement as:

```
my_list = [1]
my_tuple = (1,)

my_list, my_tuple

a = (1 + 2 + 3 + 4 +

     5 + 6 + 7 +
     8 + 9 + 10

a

type(a)
```

- Here, the surrounding parentheses ( ) do the line continuation implicitly. Same is the case with [ ] and { }. For example:

```
a = 1
b = 2

colors = 'Set theory is the mathematical theory of well-determined
collections, called sets, of objects that are called members, or
elements, of the set. Pure set theory deals exclusively with sets, so
the only sets under consideration are those whose members are also
sets. The theory of the hereditarily-finite sets, namely those finite
sets whose elements are also finite sets, the elements of which are
also finite, and so on, is formally equivalent to arithmetic. So, the
essence of set theory is the study of infinite sets, and therefore it
can be defined as the mathematical theory of the actual—as opposed to
potential—infinite.'

colors = 'Set theory is the mathematical theory of well-determined
collections, called sets, \
of objects that are called members, or elements, of the set. Pure set
theory deals exclusively \
with sets, so the only sets under consideration are those whose
members are also sets.\
The theory of the hereditarily-finite sets, namely those finite sets
whose elements are also finite sets,\
 the elements of which are also finite, and so on, is formally
equivalent to arithmetic. \
 So, the essence of set theory is the study of infinite sets, and \
 therefore it can be defined as the mathematical theory of the actual—
as opposed to potential—infinite.'

colors
```

- We can also put multiple statements in a single line using semicolons, as follows:

```
a = 1
b = 2
c = 3

a = 1 ; b = 2 ; c = 3

c

a = 1
b = 2
c = 3

a, b, c = [1, 'a', True], (45, 67), {'Name' : 'Vikash' , 'Age' : 27}

a

b

c

a, b = b, a

a

b
```

Python Comments

- Comments are very important while writing a program. They describe what is going on inside a program, so that a person looking at the source code does not have a hard time figuring it out.
- You might forget the key details of the program you just wrote in a month's time. So taking the time to explain these concepts in the form of comments is always fruitful.
- In Python, we use the hash (#) symbol to start writing a comment.
- It extends up to the newline character. Comments are for programmers to better understand a program. Python Interpreter ignores comments.

```
# This is a comment
# print('Hello')
print('Hello')
```

- We can have comments that extend up to multiple lines. One way is to use the hash(#) symbol at the beginning of each line. For example:

```
# This is a long comment
# and it extends
# to multiple lines
```

- Another way of doing this is to use triple quotes, either ''' or """.
- These triple quotes are generally used for multi-line strings. But they can be used as a multi-line comment as well. Unless they are not docstrings, they do not generate any extra code.

```
'''This is also a
perfect example of
ahjcdjh
sdjbdkjas
mnbscjsab
bkjabfa
multi-line comments'''

a = 3
print(a)
```

Indentation in Python

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

- Let's create a simple statement that says: "If a is greater than b, assign 2 to a and 4 to b"
- Take a look at how to write a simple if statement

```
 a=2
 b=1

# if b > a:
#   a = 5
#   b = 5

 a

b

a, b = 1, 2

if a < b :
```

- You'll notice that Python is less cluttered and much more readable than other languages like C or Java. How does Python manage this?
- The statement is ended with a colon, and whitespace(tab) is used (indentation) to describe what takes place in case of the statement.

- Another major difference is the lack of semicolons in Python.
- Semicolons are used to denote statement endings in many other languages, but in Python, **the end of a line is the same as the end of a statement.**
- Lastly, to end this brief overview of differences, let's take a closer look at indentation syntax in Python vs other languages:

Conditional Statements

- if Statements in Python allows us to tell the computer to perform alternative actions based on a certain set of results.
- Verbally, we can imagine we are telling the computer:"Hey if this case happens, perform some action"
- We can then expand the idea further with elif and else statements, which allow us to tell the computer: "Hey if this case happens, perform some action. Else, if another case happens, perform some other action. Else, if *none* of the above cases happened, perform this action."
- Let's go ahead and look at the syntax format for if statements to get a better idea of this:

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

```
if case1 :
  # do some action
  elif case 5 :
    # another action
elif case2 :
  # do some another action
else:
  # do some yet another action

x=20   # Statement 1 - Assignment statement

if x == 30 :
  print('x is greater than 10!')
  print(f'x is {x}')
```

- if statement evaluates the True condition

```
x == 20

if True:
  print('It was true!')
```

```
if False:
  print('It was true!')
```

Let's add in some else logic:

```
x = False
y = True

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

- Let's get a fuller picture of how far if, elif, and else can take us!
- We write this out in a nested structure. Take note of how the if, elif, and else line up in the code. This can help you see what if is related to what elif or else statements.

```
city = 'Pune'

if city == 'New Delhi':
  print('Welcome to the Indian Capital')
elif city == 'Mumbai':
  print('Welcome to the Financial Capital of India!')
else:
  print('Where are you?')
```

- Note how the nested if statements are each checked until a True boolean causes the nested code below it to run. You should also note that you can put in as many elif statements as you want before you close off with an else.
- Let's create two more simple examples for the if, elif, and else statements:

```
person = 'Johnny'

if person == 'John':
    print(f'Welcome {person}!')
else:
    print("Welcome, what's your name?")

number1 = -2
number2 = 9

if number1 > 0:
  if number2 < 10 :
    print(number2 + number1)
  else:
    print("Should be greater than 10")
```

```
elif number1 < 0:
    print("Ah, that's a negative number")
else:
    print("The number is zero!")
```

Truthy & Falsy

A "truthy" value will satisfy the check performed by if or while statements. We use "truthy" and "falsy" to differentiate from the boolean values True and False.

All values are considered "truthy" except for the following, which are "falsy":

- None
- False
- 0
- 0.0
- [] - an empty list
- {} - an empty dict
- () - an empty tuple
- '' - an empty str

```
# Consider the following statement

if False:
  print("What do you say?")
else:
  print("I have nothing to say")

if [None]:
  print("Well I had a really bad day!")
else:
  print("Come on you aced your test.")

if {1:2,3:4}:
  print("This is True")
else:
  print("This is False")
```

Loops are important in Python or in any other programming language as they help you to execute a block of code repeatedly. You will often come face to face with situations where you would need to use a piece of code over and over but you don't want to write the same line of code multiple times.

In Python we have mainly two different types of loops :

- for loop : In the context of most data science work, Python for loops are used to loop through an iterable object (like a list, tuple, set, etc.) and perform the same action for each entry. For example, a for loop would allow us to iterate through a list, performing the same action on each item in the list.
- while loop : The while loop is somewhat similar to an if statement, it executes the code inside, if the condition is True. However, as opposed to the if statement, the while loop continues to execute the code repeatedly as long as the condition is True.

for loops

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

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

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

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

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

Example 1

Let us print each element of our list of strings using a for loop statement

```
# Consider a list of strings
```

```
got_houses = ['Stark', 'Arryn', 'Baratheon', 'Tully', 'Greyjoy',
'Lannister', 'Tyrell', 'Martell', 'Targaryen']

# A simple for loop to print the houses of GOT universe
for house in got_houses[::-1]:
  print(f"House {house}")
```

Another interesting way to loop through the elements of a list is to use the enumerate() function. Using enumerate requires us two iterators index and element

```
got_houses

# Using enumerate function to loop through the elements of a list
for number,house in enumerate(got_houses):
  print(f"The house no of house {house} is {number + 1}")

x = 1,2

x

list(enumerate(got_houses))
```

Example 2

Suppose you are given a list of numbers. You need to find the corresponding squares of these numbers and zip them together in a dictionary

```
# The list of numbers
list_of_numbers = [1, 2, 4, 6, 11, 14, 17, 20]

# Let us first print the squares
for number in list_of_numbers:
  squared_number = number**2
  print(f"The square of {number} is {squared_number}")
```

So this was a pretty straight forward way to print out these squares.

Example 3

Imagine a scenario where we not only needed to print these numbers for each iteration but also we need to store these elements somewhere else

```
list_of_numbers

# Let us first initialize a list where we will be appending the
squares in each iteration

squared_numbers = []
```

```
for number in list_of_numbers:
  square = number**2
  # Use the append method to add the numbers one by one to our list
  squared_numbers.append(square)
  # print(squared_numbers)

print(f"The list of squared numbers is {squared_numbers}")

print(list_of_numbers)
print(squared_numbers)

# Let us zip the original numbers and squares together and get the
dictionary
zipped_dict = dict(zip(list_of_numbers,squared_numbers))

# Let us print the dictionary where the key is the number and the
value is the square of that number
print(zipped_dict)
```

Example 4

Now suppose we only want to print the squares of those numbers which are even. Let us see how we can do this

```
print(list_of_numbers)

# Let us first print the squares
for number in list_of_numbers:
  if number%2 == 0:
    squared_number = number**2
    print(f"The square of {number} is {squared_number}")
  else:
    print(f"I am an odd number {number}.My master prohibits me from
printing the squares of odd numbers. Strange but okay!")

# print("\n")
# print("I am finished with my iteration")
```

range() function is a pretty useful function to get a sequence of numbers. It takes three arguments : start, stop, step

```
list(range(10))

# Let us first print the squares
for number in range(0,10,2):
  squared_number = number**2
  print(f"The square of {number} is {squared_number}")

for i in range(2,21,2):
  print(i)
```

Example 5

Next suppose we wanted to find the sum of the squares of our numbers in the list

```
# Print our list
print(list_of_numbers)

# Let us the initialize the sum of the squares with 0. This makes
sense right!
sum_squares = 0

for number in range(1,11):
  sum_squares = sum_squares + number
  print(sum_squares)

# Now we have added the squares of all the numbers in our list
print(f"The sum of the squares of our list of numbers is
{sum_squares}")
```

Up till now, we have only implemented for loops for list objects. From the last week, we know that there are other objects in Python which are sequence of elements such as strings, tuples etc. Let us try and apply for loop to iterate over their elements.

Example 6

We are given a sentence : "I am the one who knocks!"

```
# Let us first store the sentence in a string variable
heisenberg_quote =  "I am the one who knocks!"
```

The next step is to print all the characters which are separated by whitespace

```
for char in heisenberg_quote:
  print(f" The character is {char}")

# Lets apply a for loop to do the above task

for index,char  in enumerate(heisenberg_quote):
  print(f" The index is {index} and the character is {char}")
```

We saw how we can iterate through each element of our string. What if you wanted to iterate through each word of the sentence and not each element. There is an important method available with strings known as split() method.

This method returns the list of words separated by the character we want to separate them by.

Example 7

```
heisenberg_quote = "It ceases to exist without me. No, you clearly
don't know who you're talking to, so let me clue you in. I am not in
danger, Skyler. I am the danger."

words_by_walter = heisenberg_quote.split(' ')
print(words_by_walter)

# Now we can print each word by iterating through this list
for word in words_by_walter:
  if word not in ['I','you']:
    print(f"The word is {word}")
```

We can also iterate through each element of tuple as well.

Example 8

```
# Suppose we have a tuple of days
days =
('Monday' ,'Tuesday' ,'Wednesday' ,'Thursday' ,'Friday','Saturday','Su
nday')

for day in days:
  print(f"Today is {day}")
```

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

Remember from earlier we had a list of tuples of country-city

Example 9

```
country_city_river_list = [('India','New Delhi','Ganga'),
('Australia','Canberra','Rovers'),('United States','Washington
DC','Missouri'),('England','London','Thames')]
```

```
# Let us iterate through each tuple element of this list and unpack
each item

for country,city,river in country_city_river_list:
  # print(country_city)
  print(f"The capital of the country {country} is {city} and it also
has the river {river}.")
```

Finally we can iterate through the items of a dictionary as well.

Example 10

```
# Lets convert the list of tuples to a dictionary
country_city_dict = dict(country_city_list)
print(country_city_dict)

for item in country_city_dict:
  print(item)

country_city_dict.items()
```

This just prints the keys of the dictionary

Next we can iterate through the key-value pairs using the items() method which you should remember from earlier

Since the items() method supports iteration, we can perform *dictionary unpacking* to separate keys and values

```
for country,city in country_city_dict.items():
  print(f"The capital of {country} is {city}")

for elem in {1,2,3}:
  print(elem)
```

And of course you can iterate through the list of keys and list of values independently depending upon the scenario

While Loops

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

The general format of a while loop is:

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

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

```
x = 0

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

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

```
x = 0

while x < 5:
  print('x is currently: ',x)
  print(' x is still less than 10, adding 1 to x')
  x+=1

print("All done")
print("I am done with the iterations")
```

break, continue, pass

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

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

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

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

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

```
x = 0

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

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

```
x = 0

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

```
  x+=1
  break

else:
  print('continuing...')

x = 0

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

else:
  print('continuing...')
```

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

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

```
for elem in list_of_numbers:
  while elem > 3:
    pass
  else:

l = []
for elem in l :
  print(elem)
```

What are list comprehensions?

List comprehensions are a tool for transforming one list (any iterable actually) into another list. During this transformation, elements can be conditionally included in the new list and each element can be transformed as needed.

List comprehensions in Python are great, but mastering them can be tricky because they don't solve a new problem: they just provide a new syntax to solve an existing problem.

From loops to comprehensions

Every list comprehension can be rewritten as a for loop but not every for loop can be rewritten as a list comprehension .The key to understanding when to use list comprehensions is to practice identifying problems that smell like list comprehensions. If you can rewrite your code to look just like this for loop, you can also rewrite it as a list comprehension:

Let us take an example of getting the squares of the numbers in a list

```
# The list of numbers
list_of_numbers = [1, 2, 4, 6, 11, 14, 17, 20]
```

The usual way to do this using for loop is described below:

```
# Let us first initialize a list where we will be appending the
squares in each iteration

squared_numbers = []

for number in list_of_numbers:
  # Use the append method to add the numbers one by one to our list
  squared_numbers.append(number**2)
  # print(squared_numbers)
```

```
print(f"The list of squared numbers is {squared_numbers}")
```

We can do the same task using a list comprehension

```
# Getting the list of squares using list comprehension
squared_numbers = [number**2 for number in list_of_numbers]
print(squared_numbers)
```

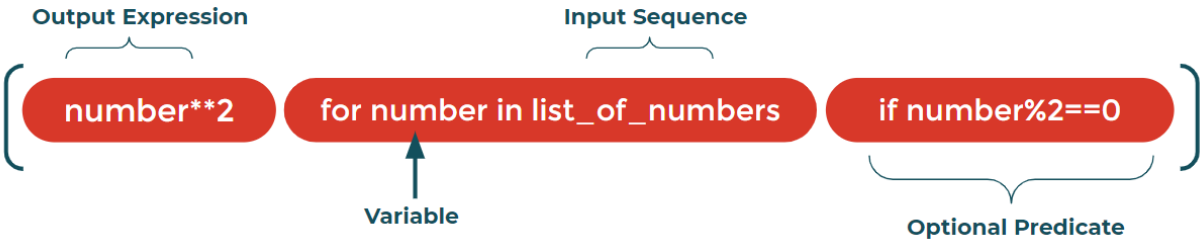
Further we can go ahead and use some conditionals.

Suppose we wanted to get the squares of only those numbers which are even

```
print(list_of_numbers)

# Writing list comprehension using conditionals
even_squared_numbers = [number**2 for number in list_of_numbers if
number%2==0]
print(even_squared_numbers)

# Writing list comprehension using conditionals
odd_squared_numbers = [number**2 for number in list_of_numbers if
number%2!=0]
print(odd_squared_numbers)

new_numbers = [number**2 for number in list_of_numbers if number%2!=0
else number**3]
print(new_numbers)

# Writing list comprehension using conditionals
odd_squared_numbers = [number**2 if number%2!=0 else number**3 for
number in list_of_numbers ]
print(odd_squared_numbers)
```

A list comprehension consists of the following parts:

- An Input Sequence.
- A Variable representing members of the input sequence.
- An Optional Predicate expression.
- An Output Expression producing elements of the output list from members of the Input Sequence that satisfy the predicate.



- The iterator part iterates through each member number of the input sequence list_of_numbers.
- The predicate checks if the member is even.
- If the member is even then it is passed to the output expression, squared, to become a member of the output list.

Using for loop

```
even_squared_numbers = []

for number in list_of_numbers:
  if number%2 == 0:
    even_squared_numbers.append(number**2)
```

Using list comprehension

```
even_squared_numbers = [number**2 for number in list_of_numbers if
number%2==0]
```

So how did we convert our for loop into a list comprehension?

- Copying the variable assignment for our new empty list
- Copying the expression number**2 that we've been appending into this new list
- Copying the for loop line, excluding the final :
- Copying the if statement line, also without the :

Writing list comprehensions for nested loops

Let's go ahead and write a nested for loop

```
list(range(4))
```

```
list_of_numbers = []

for num1 in range(4):
  for num2 in range(3):
    # print(num1,num2)
    list_of_numbers.append(num1)

print(list_of_numbers)

# We can write it in a slightly different way
list_of_numbers = [num1 for num1 in range(4) for num2 in range(3)]
print(list_of_numbers)
```

Note: My brain wants to write this list comprehension as:

```
list_of_numbers = [num2**2 for num2 in range(3) for num1 in range(4)]
print(list_of_numbers)
```

But that's not right! I've mistakenly flipped the for loops here. The correct version is the one above. When working with nested loops in list comprehensions remember that the for clauses remain in the same order as in our original for loops.

Now let us see how we can create list of lists using list comprehension

```
sample = [[]]
len(sample)

sample[0]

list_of_lists = []

for num1 in range(4):
  # Append an empty sublist inside the list
  list_of_lists.append([])  # [[0]] , list_of_lists[1]
  # print(list_of_lists)

  for num2 in range(3):
    # Append the elements within the sublist
    list_of_lists[num1].append(num2)  #list_of_lists[0]

print(list_of_lists)

# Using list comprehension we can write the same thing
list_of_lists = [[num2 for num2 in range(3)] for num1 in range(4)]
print(list_of_lists)
```

```
# Using list comprehension we can write the same thing
list_of_lists = [[num1 for num2 in range(3)] for num1 in range(4)]
print(list_of_lists)
```

Up till now, we have only implemented list comprehension for list objects. We know that there are other iterables in Python which are sequence of elements such as strings, tuples etc. Let us try and apply list comprehension to iterate over their elements.

```
heisenberg_quote = "It ceases to exist without me. No, you clearly
don't know who you're talking to, so let me clue you in. I am not in
danger, Skyler. I am the danger."

words_by_walter = heisenberg_quote.split(' ')
print(words_by_walter)

heisenberg_quote.split('.')

# Let u store the first letter of each word in another list

first_letters = []

for word in words_by_walter:
  first_letters.append(word[0])

# Print the first_letters list
print(first_letters)

# The same task but now with list comprehension
first_letters = [word[0] for word in words_by_walter]
print(first_letters)

heisenberg_quote.split(',')

my_string = 'Alphabet Inc.'

list_of_characters = [char for char in my_string]
print(list_of_characters)
```

We can also use list comprehension to get a tuple as well.

```
# Suppose we have a tuple of days
days =
('Monday' ,'Tuesday' ,'Wednesday' ,'Thursday' ,'Friday','Saturday','Su
nday')

tuple(day for day in days)

# Let us create a tuple where we want to remove the word 'day' from
each tuple element
```

```
days_without_day = tuple([day[:-3] for day in days])
print(days_without_day)
```

Note we cannot just specify parentheses and get this to work. It would create a generator object so we need to call the tuple function

Set Comprehension

```
# Using a for loop
first_letters = set()
for word in words_by_walter:
    first_letters.add(word[0])

print(first_letters)

# Using set comprehension
first_letters = {word[0] for word in words_by_walter}
print(first_letters)
```

Dictionary Comprehensions

```
country_city_list = [('India','New Delhi'),('Australia','Canberra'),
('United States','Washington DC'),('England','London')]

# Lets convert the list of tuples to a dictionary
country_city_dict = dict(country_city_list)
print(country_city_dict)
```

Let us try to swap the keys and values for this dictionary

```
# The conventional for loop approach
flipped = {}
for country, city in country_city_dict.items():
    flipped[city] = country

print(flipped)

# The new dictionary comprehension
flipped = {city:country for country, city in
country_city_dict.items()}
print(flipped)

# We can further add conditionals
flipped = { city : country for country, city in
```

```
country_city_dict.items() if country[0] not in ['E','A']}
print(flipped)

# The new dictionary comprehension
flipped = {country[:2] : city[-2:] for country, city in
country_city_dict.items()}
print(flipped)

print(country_city_dict)
```

Dream Premier League (DPL)

A match of DPL is going on. Team Aravali has a made score of **195** for **7** in **20** overs and the opponent Team Shivalik is doing some analytics to find out what they need to do to win. You are the Data Scientist of the Team Shivalik and you have been asked to help them.

Problem 1

###Team Shivalik has stored the runs made by Team Aravali players in the following dictionary.

```
aravali ={ "Dhoni":25, "Virat":31, "Pollard":11, "Rohit": 0,
"Maxwell":12, "Sachin":59, "Sehwag":12 }
```

Get the list of all players in Team Aravali

```
# Get the list of players in Team Aravali
```

Problem 2

By mistake, the runs made by Rohit was recorded as 0. Your next task is to figure out how many runs were made by Rohit and update the dictionary

```
# We know the total runs




# Print runs scored by Rohit
Rohit_runs
```

```
# Update the dictionary with correct value of runs
```

Problem 4

Your next task is to find out who scored the second highest runs in Team Aravali

```
# Your code below
```

Problem 4

Just out of curiosity, you want to find out the unique runs made by Team Aravali players.

```
# Your code below
run_set =

# Print the unique runs
run_set
```

Problem 5

Team Shivalik has 6 fixed players and 5 slots for players who are playing good currently. Create two collections using appropriate data structure to write this 6 fixed and 5 mutable players. You can choose any player you want.

Available Players in the squad :

['Vijay', 'Lasith', 'Dravid', 'Smith', 'Ambati', 'Hardik', 'Sushant', 'Mandeep', 'Harbhajan', 'Yuvraj', 'Jadeja','Rajeev','Amrit']

```
# Your code below
fixed =
mutable =

# Print them on same line using comma to seperate them
```

Problem 6

Try changing fixed player and mutable player

```
# Change fixed player

# Change mutable player
```

Problem 7

Find out the runrate required for Team Shivalik to win (for 20 overs)

Hint: Runrate is runs required per over to win the match

```
# Your code here
runrate =

# Print the run rate
```

Problem 8

You have just received a secret message form your informant stating that some players of the other team are into match fixing. You have to decode a message and inform authorities about it.

You received a string **"skdlfjnvuerhw qefnnaosfu qrhviudhfv wuirhv adknlkxjcier vafuvhkajn iuvhsf vasuif KJSHFKJ aeuihvasf akjfhiufe"** and index of "i" are going to be "no balls".

Find the first and last no ball from the string.

```
#First no ball
message = "skdlfjnvuerhw qefnnaosfu qrhviudhfv wuirhv adknlkxjcier
vafuvhkajn iuvhsf vasuif KJSHFKJ aeuihvasf akjfhiufe"



# Last no ball

# Print the first and last ball numbers
```

Problem 9

You have given the information about fixing to the authorities and they are going to verify it during the match. But still you have to work on your strategy.

It is in your hands to automate the decision on who goes on 4th position for batting depending on following criteria:

- if runs made by Team Shivalik is less than 50, Smith will play
- if runs are between 51 to 100 then Sir Jadeja will go
- if runs are above 100 then Hardik will play

```
# Your code below
B_runs = int(input())
```

Hurray! Turns out you won the match and the guilty players are punished as well. You should be proud of yourself!

```

```

Functions

Functions will be one of our main building blocks when we construct larger and larger amounts of code to solve problems.

So what is a function?

Formally, a function is a useful device that groups together a set of statements so they can be run more than once. They can also let us specify parameters that can serve as inputs to the functions.

On a more fundamental level, functions allow us to not have to repeatedly write the same code again and again. If you remember back to the lessons on strings and lists, remember that we used a function len() to get the length of a string. Since checking the length of a sequence is a common task you would want to write a function that can do this repeatedly at command.

Functions will be one of the most basic levels of **reusing code in Python**.

def Statements

Let's see how to build out a function's syntax in Python. It has the following form:

```
def example_function(arg1, arg2):
  '''
  This is where the function's Document String (docstring) goes
  '''
  # Do stuff here
  return desired_result
```

- We begin with def then a space followed by the name of the function. Try to keep names relevant, for example len() is a good name for a length() function. Also be careful with names, you wouldn't want to call a function the same name as a built-in function in Python (such as len).
- Next come a pair of parentheses with a number of arguments separated by a comma. These arguments are the inputs for your function. You'll be able to use these inputs in your function and reference them. After this you put a colon.
- Now here is the important step, you must indent to begin the code inside your function correctly. Python makes use of *whitespace* to organize code. Lots of other programing languages do not do this, so keep that in mind.
- Next you'll see the docstring, this is where you write a basic description of the function. Using iPython and iPython Notebooks, you'll be able to read these docstrings by pressing Shift+Tab after a function name. Docstrings are not necessary for simple functions, but it's good practice to put them in so you or other people can easily understand the code you write.

After all this you begin writing the code you wish to execute.

- The best way to learn functions is by going through examples. So let's try to go through examples that relate back to the various objects and data structures we learned about before.
- return allows a function to *return* a result that can then be stored as a variable, or used in whatever manner a user wants.

Example 1: Extract a list of first letters from a sentence

```
heisenberg_quote = "It ceases to exist without me. No, you clearly
don't know who you're talking to, so let me clue you in. I am not in
danger, Skyler. I am the danger."

# Using the list comprehension, we can do the above task in this way
words_by_walter = heisenberg_quote.split(' ')

first_letters = [word[0] for word in words_by_walter]
print(first_letters)
```

Now let us write a function which does the same task for any given sentence

```
def extract_first_letters(sentence):
  '''
  This function takes a sentence as an input and returns
  the list of first letters of each word
  '''
```

```
  # Step 1 : Get the list of words in this sentence
  words_in_sentence = sentence.split(' ')

  # Step 2: Write a list comprehension to extract the first letters
  first_letters = [word[0] for word in words_in_sentence]

  # Step 3: Return the output list first_letters
  return first_letters

print(extract_first_letters(heisenberg_quote))

print(extract_first_letters(starwars_quote))

# Store the result of the function in another variable
first_letters_heisenberg = extract_first_letters(heisenberg_quote)
print(first_letters_heisenberg)
```

Let us test this function on another sentence

```
starwars_quote = "May the Force be with you."
first_letters_starwars = extract_first_letters(starwars_quote)
print(first_letters_starwars)
```

Example 2 : Let us now write a function which returns the factorial of a number.

In mathematics, the factorial of a positive integer n, denoted by $n!$, is the product of all positive integers less than or equal to n

```
def factorial(num):
    '''
    Function for calculating factorial of a number
    '''

    # Step 1 : Write an if  statement to print a message if the input
number is negative
    if num < 0:
      print("Factorial is not defined for negative numbers.")

    # Step 2: Write the elif statement to calculate the factorial
    elif num > 0:
      result = 1
      for i in range(1, num+1):
        result = result * i
      # print(f"The factorial of {num} is {result}.")

    # Step 3: Write the else statement to take care of the input 0
    else:
      print("The factorial of 0 is 1.")
```

```
    # Step 4 : Return the output variable result

    return result

fact_seven = factorial(10)
print(fact_seven)

# Now let us define a list of numbers for which we want to calculate
the factorials
list_of_numbers = [3, 5, 8, 12, 4, 6]

# We can write a list comprehension for the same task
factorial_list = [factorial(number) for number in list_of_numbers if
number%2!=0]
print(factorial_list)
```

Example 3: Let us write a function which takes a date as a string and prints the quarter as output.

Remember it is not required that functions will return a value. This is an example of such functions

```
def assignQuarter(user_date):
  '''
  This function prints the quarter for a given date string
  '''

  # Step 1 : Extract the year and month from the input date string
  date_year = user_date[:4]
  date_month = user_date[5:7]

  # Step 2 : Store the quarter value in a string variable named
quarter
  if (date_month >= '01') & (date_month <='03'):
    quarter = date_year + '-Q1'
  elif (date_month >= '04') & (date_month <='06'):
    quarter = date_year + '-Q2'
  elif (date_month >= '07') & (date_month <='09'):
    quarter = date_year + '-Q3'
  else:
    quarter = date_year + '-Q4'

  # Step 3 : Print the message you want to display to the user. Here
we are not returning anything
  print(f'The corresponding quarter for the date {user_date} is
{quarter}')
  # return quarter
```

```
sample_date = '2020-07-01'
date_quarter = assignQuarter(sample_date)
print(date_quarter)

my_quarter = assignQuarter(sample_date)

print(my_quarter)

assignQuarter
```

Example 4 : Let us write a function with multiple return statements

```
def check_even_or_odd(num):
  '''
  This function checks whether a number is odd or even and then
returns the corresponding string
  '''
  print(f'Given number is {num}')

  # Step 1 : Check if the number is even. If it is return 'even'
  if num%2 == 0:
    return 'even'

  # Step 2 : Else just return 'odd'

  return 'odd'

is_even_odd = check_even_or_odd(10)
print(is_even_odd)
```

Example 5 : Next let us write a function which returns multiple variables at once.

In this specific example, the input is a list of numbers. And we have to return the mean and median of this list of numbers

```
heights =  [172,175,170,168,170,200]

def calculate_averages(my_list):
  '''
  This function calculates the mean and median of a given list of
numbers.
  '''
  # Step 1 : Calculate the mean
  mean = sum(my_list)/len(my_list)

  # Step 2 : Calculate the median
  sorted_list = sorted(my_list)
  list_length = len(sorted_list)
```

```
  if list_length%2 == 0:
    median = (sorted_list[int(list_length/2)-1] +
sorted_list[int(list_length/2)])/2
  else :
    median = sorted_list[int((list_length+1)/2) - 1]

  return  mean, median

calculate_averages(heights)

mean_height, median_height = calculate_averages(heights)

mean_height

median_height
```

Example 6: Now let us write a function which adds an element to a list. This functions takes the list as an argument and the element to be added

```
def list_append(my_list, elem):
  '''
  This function appends an element to the list
  '''
  # Step 1 : Add the element to the list
  new_list = my_list + [elem]

  # Step 2 : Return the new_list
  return new_list

print(heights)

list_append(heights,190)

print(heights)

list_append(heights, elem = 40)

print(heights)

heights.append(60)

print(heights)
```

Let us find out what are the differences between functions and methods

- Function and method both look similar as they perform in an almost similar way, but the key difference is the concept of 'Class and its Object'. Functions can be called

only by its name, as it is defined independently. But methods cannot be called by its name only we need to invoke the class by reference of that class in which it is defined, that is, the method is defined within a class and hence they are dependent on that class.

- A method is called by its name but it is associated with an object (dependent). It is implicitly passed to an object on which it is invoked. It may or may not return any data. A method can operate the data (instance variables) that is contained by the corresponding class.

Methods

We've already seen a few example of methods when learning about lists in Python. Methods are essentially functions built into objects.

Methods perform specific actions on an object and can also take arguments, just like a function.

Methods are of the form:

```
object.method(arg1,arg2,etc...)
```

```
def insert_value_to_list(list_obj, index_to_place, value_to_place):
  '''
  This function inserts an element to a list at a specified index
  '''

  # Step 1 : Get the first part
  first_part = list_obj[0:index_to_place]

  # Step 2 : Append the element at the end of the first part
  first_part.append(value_to_place)

  # Step 3 : Get the last part
  last_part = list_obj[index_to_place:]

  # Step 4 : Use the extend method to add the last part to the first
part
  first_part.extend(last_part)

  # Step 5 : Return the first part
  return first_part

new_heights = insert_value_to_list(heights, 2, 180)

new_heights

heights.insert(2,180)

heights
```

Scope

Now that we have gone over writing our own functions, it's important to understand how Python deals with the variable names you assign. When you create a variable name in Python the name is stored in a *name-space*. Variable names also have a *scope*, the scope determines the visibility of that variable name to other parts of your code.

Let's start with a quick thought experiment; imagine the following code:

```
def experiment():
  global x
  x = 50

  return x

x = 25
```

What do you imagine the output of experiment() is? 25 or 50? What is the output of print x? 25 or 50?

```
experiment()

experiment()

x
```

How do we make this function take the global variable x and change it?

```
def new_experiment():

  x = 50

  return x

x = 40

x

new_experiment()
```

What is Production Environment?

Production environment is a term used mostly by developers to describe the setting where software and other products are actually put into operation for their intended uses by end users.

Concepts that will be useful in writing production grade code:

- Object Oriented Programming
- Handling Errors and Exceptions

1. Object Oriented Programming

Object Oriented Programming (OOP) tends to be one of the most interesting concepts to learn in Python.

For this lesson we will construct our knowledge of OOP in Python by building on the following topics:

- Objects
- Using the *class* keyword
- Creating class attributes
- Creating methods in a class
- Learning about Polymorphism

Lets start the lesson by remembering about the Basic Python Objects. For example:

```
empty_list = []

empty_list

another_empty_list = list()

another_empty_list
```

Remember how we could call methods on a list?

```
another_empty_list.append(2)

another_empty_list

another_empty_list.append(1)
another_empty_list.append(3)

another_empty_list

another_empty_list.index(3)
```

What we will basically be doing in this lecture is exploring how we could create an Object type like a list. We've already learned about how to create functions. So let's explore Objects in general:

Objects

In Python, *everything is an object*. Remember from previous lectures we can use type() to check the type of object something is:

```
the_sixth_sense = 6
print(type(the_sixth_sense))

two_and_a_half_men = 2.5
print(type(two_and_a_half_men))

schindlers_list = list()
print(type(schindlers_list))

another_list = list()

tharoor = {'farrago':'a confused mixture'}
print(type(tharoor))
```

So we know all these things are objects, so how can we create our own Object types? That is where the class keyword comes in.

class

User defined objects are created using the class keyword. The class is a blueprint that defines the nature of a future object. From classes we can construct instances. An instance is a specific object created from a particular class. For example, above we created the object another_empty_list which was an instance of a list object.

Let see how we can use class:

```
# Create a new object type called FirstClass
class FirstClass:
    pass

# Instance of FirstClass
x = FirstClass()
print(type(x))

y = FirstClass()
print(type(y))
```

By convention we give classes a name that starts with a capital letter. Note how x is now the reference to our new instance of a FirstClass class. In other words, we **instantiate** the FirstClass class.

Inside of the class we currently just have pass. But we can define class attributes and methods.

An **attribute** is a characteristic of an object. A **method** is an operation we can perform with the object.

For example, we can create a class called Dog. An attribute of a dog may be its breed or its name, while a method of a dog may be defined by a .bark() method which returns a sound.

Let's get a better understanding of attributes through an example.

Attributes

The syntax for creating an attribute is:

```
self.attribute = something
```

There is a special method called:

```
__init__()
```

This method is used to initialize the attributes of an object. For example:

```
class Dog:

    def __init__(self, breed, name):
        self.breed_attribute = breed
        self.name_attribute = name

sam_object = Dog(breed='Lab', name = 'Sam')
frank_object = Dog(breed='Huskie', name = 'Frank')

sam_object.breed_attribute
```

Lets break down what we have above.The special method

```
__init__()
```

is called automatically right after the object has been created:

```
def __init__(self, breed):
```

Each attribute in a class definition begins with a reference to the instance object. It is by convention named self. The breed is the argument. The value is passed during the class instantiation.

```
 self.breed = breed
```

Now we have created two instances of the Dog class. With two breed types, we can then access these attributes like this:

```
sam_object.name_attribute

frank_object.name_attribute
```

Note how we don't have any parentheses after breed; this is because it is an attribute and doesn't take any arguments.

Methods

Methods are functions defined inside the body of a class. They are used to perform operations with the attributes of our objects. Methods are a key concept of the OOP paradigm. They are essential to dividing responsibilities in programming, especially in large applications.

You can basically think of methods as functions acting on an Object that take the Object itself into account through its *self* argument.

Let's go through an example of creating a Circle class:

```
class Circle:

  def __init__(self, radius=1):
    self.radius = radius
    self.area = 3.14 * radius * radius

  def setRadius(self, new_radius):
    self.radius = new_radius
    self.area = 3.14 * new_radius * new_radius

  def getCircumference(self):
    return 2 * 3.14 * self.radius

c = Circle()

c.area

c.getCircumference()

print('Radius is: ', c.radius)
print('Area is: ', c.area)
print('Circumference is: ', c.getCircumference())
```

Now let's change the radius and see how that affects our Circle object:

```
c.setRadius(4)

print('Radius is: ', c.radius)
print('Area is: ', c.area)
print('Circumference is: ', c.getCircumference())
```

Great! Notice how we used self. notation to reference attributes of the class within the method calls. Review how the code above works and try creating your own method.

Polymorphism

We've learned that while functions can take in different arguments, methods belong to the objects they act on. In Python, *polymorphism* refers to the way in which different object classes can share the same method name, and those methods can be called from the same place even

though a variety of different objects might be passed in. The best way to explain this is by example:

```
class HouseStark:
    def __init__(self, sigil):
        self.sigil = sigil

    def motto(self):
        return "House Stark with sigil " + self.sigil + " has the
motto 'Winter is coming'"

class HouseLannister:
    def __init__(self, sigil):
        self.sigil = sigil

    def motto(self):
        return "House Lannister with sigil " + self.sigil + " has the
motto 'Hear me roar'"

arya = HouseStark('direwolf')
tyrion = HouseLannister('golden lion')

print(arya.motto())

print(tyrion.motto())
```

Here we have a HouseStark class and a HouseLannister class, and each has a `.motto()` method. When called, each object's `.motto()` method returns a result unique to the object.

There a few different ways to demonstrate polymorphism. First, with a for loop:

```
for warrior in [arya, tyrion]:
    print(warrior.motto())
```

Another is with functions:

```
def get_motto(warrior):
    print(warrior.motto())

get_motto(arya)
get_motto(tyrion)
```

In both cases we were able to pass in different object types, and we obtained object-specific results from the same mechanism.

Great! By now you should have a basic understanding of how to create your own objects with class in Python.

2. Errors and Exception Handling

Now we will learn about Errors and Exception Handling in Python. You've definitely already encountered errors by this point in the course. For example:

```
print('XGBoost)
```

Note how we get a SyntaxError, with the further description that it was an EOL (End of Line Error) while scanning the string literal. This is specific enough for us to see that we forgot a single quote at the end of the line. Understanding these various error types will help you debug your code much faster.

This type of error and description is known as an Exception. Even if a statement or expression is syntactically correct, it may cause an error when an attempt is made to execute it. Errors detected during execution are called exceptions and are not unconditionally fatal.

You can check out the full list of built-in exceptions here. Now let's learn how to handle errors and exceptions in our own code.

try and except

The basic terminology and syntax used to handle errors in Python are the try and except statements. The code which can cause an exception to occur is put in the try block and the handling of the exception is then implemented in the except block of code. The syntax follows:

```
try:
   You do your operations here...
   ...
except:
   If there is an exception, then execute this block.
else:
   If there is no exception then execute this block.
```

To get a better understanding of this let's check out an example:

```
try:
    print("Good to go!")
    print('a', fwdfdwafwf)
    print('b')
except:
    # This will check for any exception and then execute this print
statement
    print("Oops!")
    print('c')
    print('d')
else:
    print("No errors encountered!")
```

```
    print('e')
    print('f')
```

Great! Now we don't actually need to memorize that list of exception types! Now what if we kept wanting to run code after the exception occurred? This is where finally comes in.

finally

The finally: block of code will always be run regardless if there was an exception in the try code block. The syntax is:

```
try:
   Code block here
   ...
   Due to any exception, this code may be skipped!
finally:
   This code block would always be executed.
```

For example:

```
try:
  print("Execute try statements")
finally:
  print("Always execute finally code blocks")
```

We can use this in conjunction with except. Let's see a new example that will take into account a user providing the wrong input:

```
a = 1
b = 0.2

try:
    print(afdwfwf, type(aefwffewf))
except:
    print(b, type(b))
finally:
    print('Type printed')

try:
    a = 1
    print(adefewdwd)
    try:
      print(type(afggdgdsgg))
    except:
      print('Error occured')
except Exception as ex:
    print('The error is : ', ex)
    print(b, type(b))
```

Great! Now you know how to handle errors and exceptions in Python with the try, except, else, and finally notation!

Real World Use Case

Uber's simplified pricing model

When you request an Uber, you enter your pick-up location and the destination. Based on the distance, peak hours, willingness to pay and many other factors, Uber uses a machine learning algorithm to compute what prices will be shown to you.

Let's consider a simplistic version of the Uber Pricing Model where you compute the price based on the distance between the pick-up and the drop location and the time of the booking.

User Inputs:

- Pick-up location (pick_up_latitude, pick_up_longitude)
- Drop location (drop_latitude, drop_longitude)
- Time of booking

Output:

- Final Price

Development Code

```
# Calculate the distance between the pick-up location and the drop location

import geopy.distance

def get_distance(location_1, location_2):

    distance = geopy.distance.distance(location_1, location_2).km

    return distance

def get_price_per_km(hour):

    if (hour > 8) & (hour < 11):
        price_per_km = 20
    elif (hour > 18) & (hour < 21):
        price_per_km = 15
    else:
        price_per_km = 10

    return price_per_km

def get_final_price(pick_up_location, drop_location, booking_hour):
```

```
    total_distance = get_distance(pick_up_location, drop_location)
    actual_price_per_km = get_price_per_km(booking_hour)

    final_price = round(total_distance * actual_price_per_km, 2)

    return final_price

# Inputs

pick_up_location = (24, 70)
drop_location = (24.1, 70.1)
booking_time = 19

# Output

get_final_price(pick_up_location, drop_location, booking_time)
```

Production Grade Code

```
# Calculate the distance between the pick-up location and the drop
location

import geopy.distance
import math

class Maps:

    def __init__(self):
        pass

    def get_distance(self, location_1, location_2):

        try:
            distance = geopy.distance.distance(location_1,
location_2).km
        except:
            distance = math.sqrt((location_1[0]-location_2[0])^2 +
(location_1[1]-location_2[1])^2)

        return distance

class SurgePricing:

    def __init__(self):
        pass

    def get_price_per_km(self, hour):

        try:

            if (hour > 8) & (hour < 11):
```

```
                price_per_km = 20
            elif (hour > 18) & (hour < 21):
                price_per_km = 15
            else:
                price_per_km = 10

        except:

            price_per_km = 10

        return price_per_km

def get_final_price(pick_up_location, drop_location, booking_hour):

    maps = Maps()
    surge = SurgePricing()

    total_distance = maps.get_distance(pick_up_location,
drop_location)
    actual_price_per_km = surge.get_price_per_km(booking_hour)

    final_price = round(total_distance * actual_price_per_km, 2)

    return final_price

# Output

get_final_price(pick_up_location, drop_location, booking_time)
```

Time : 1 hour

Problem Statement

A student is taking a cryptography class and has found anagrams to be very useful. Two strings are anagrams of each other if the first string's letters can be rearranged to form the second string. In other words, both strings must contain the same exact letters in the same exact frequency. For example, bacdc and dcbac are anagrams, but bacdc and dcbad are not.

The student decides on an encryption scheme that involves two large strings. The encryption is dependent on the minimum number of character deletions required to make the two strings anagrams. You need to determine this number.

Given two strings, a and b, that may or may not be of the same length, determine the minimum number of character deletions required to make a and b anagrams. Any characters can be deleted from either of the strings. The strings a and b consist of lowercase English alphabets.

Example

a = 'cde'

b = 'dcf'

Delete e from a and f from b so that the remaining strings are $c\,d$ and $d\,c$ which are anagrams. This takes 2 character deletions.

Function Description

Create a makeAnagram function below.

Inputs:

string a: a string

string b: another string

Output:

int: the minimum total characters that must be deleted

Solution

```

```

Given a list of lists, write a function which returns the intersection of all the lists.

Example 1:

```
input = [[5, 5, 10],
      [5, 5, 7, 8],
      [5, 5, 8, 9, 11]]

output = [5, 5]
```

Example 2:

```
input = [[5, 6],
      [5, 5, 7, 8],
      [5, 5, 8, 9]]

output = [5]
```

Method 1

```
lsts = [[5, 5],
      [5, 5, 7, 8],
      [5, 5, 8, 9]]
```

```
def get_intersection_1(lsts):

  min_len = len(lsts[0])
  for l in lsts:
      if len(l)<=min_len:
          min_len = len(l)
          min_len_list = l

  d = {}
  for elem in min_len_list:
      if elem in d.keys():
          d[elem] = d[elem]+1
      else:
          d[elem] = 1

  for key in d.keys():
    for l in lsts:
      count = 0
      for item in l:
        if key==item:
          count = count + 1

      if d[key] > count:
        d[key] = count

  lst = []
  for key, value in d.items():
      for v in range(value):
          lst.append(key)

  return lst

get_intersection_1(lsts)
```

- Time Complexity: O(m^2 * n) where n -> no. of lists and m -> no. of elements in each list
- Auxiliary Space Complexity: O(m)

Method 2

```
lsts = [[5, 5],
      [5, 5, 7, 8],
      [5, 5, 8, 9]]

def get_intersection_2(lsts):

  global_map = {}

  for l in lsts:
    temp_map = {}
    for elem in l:
```

```
      temp_map[elem] = temp_map.get(elem, 0) + 1

    for key in temp_map:
      if key not in global_map.keys():
        global_map[key] = [1, temp_map[key]]
      else:
        value = global_map[key]
        global_map[key] = [value[0] + 1, min(temp_map[key], value[1])]

  answer_list = []
  for key, value in global_map.items():
    if value[0]==len(lsts):
      answer_list += [key] * value[1]

  return answer_list

get_intersection_2(lsts)
```

- Time Complexity: O(m * n) where n -> no. of lists and m -> no. of elements in each list
- Auxiliary Space Complexity: O(m * n)

Method 3

```
lsts = [[5, 5],
      [5, 5, 7, 8],
      [5, 5, 8, 9]]

def get_frequency_map(l):
  map_ = {}
  for elem in l:
    map_[elem] = map_.get(elem, 0) + 1
  return map_

def get_intersection_3(lsts):

  global_map = {}

  for l in lsts:
    temp_map = get_frequency_map(l)

    for key in temp_map.keys():
      if key not in global_map.keys():
        global_map[key] = [1, temp_map[key]]
      else:
        value = global_map[key]
        global_map[key] = [value[0] + 1, min(temp_map[key], value[1])]

  answer_list = []
  for key, value in global_map.items():
    if value[0]==len(lsts):
```

```
      answer_list += [key] * value[1]

  return answer_list

get_intersection_3(lsts)
```

- Time Complexity: O(m * n) where n -> no. of lists and m -> no. of elements in each list
- Auxiliary Space Complexity: O(m * n)

How to compare the performance of different algorithms?

Sometimes, there are more than one way to solve a problem. We need to learn how to compare the performance different algorithms and choose the best one to solve a particular problem. While analyzing an algorithm, we mostly consider two parameters:

1. **Time complexity** : Time complexity of an algorithm quantifies the amount of time taken by an algorithm to run as a function of the length of the input.
2. **Space complexity** : Space complexity of an algorithm quantifies the amount of space or memory taken by an algorithm to run as a function of the length of the input.

Time and space complexity depends on lots of things like hardware, operating system, processors, etc. However, we don't consider any of these factors while analyzing the algorithm. We will only consider the time taken and space comsumed by an algorithm.

Time Complexity

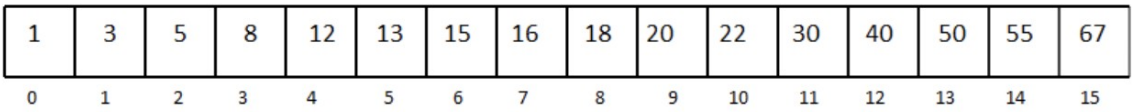
Lets start with a simple example. Suppose you are given a sorted list and an integer and you have to find if it exists in the list.

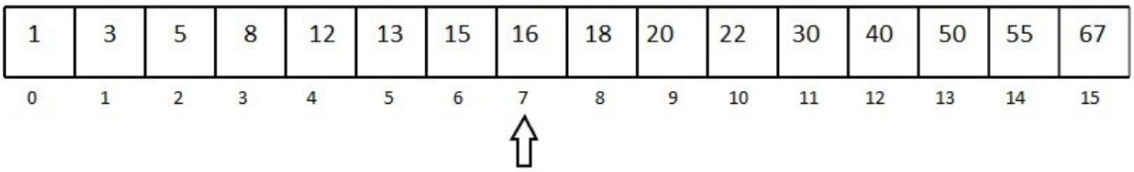
Simple solution to this problem is to traverse the whole list and check if any element is equal to the integer that is given.

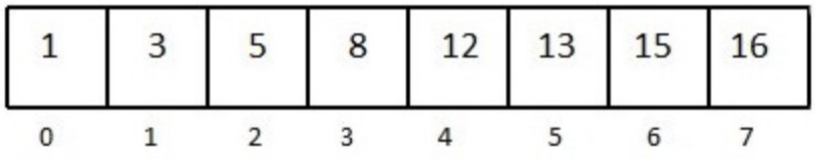
```
integer_to_match = 13
list_to_search = [1, 3, 5, 8, 12, 13, 15, 16, 18, 20, 22, 30, 40, 50,
55, 67]

for i in range(len(list_to_search)):
  if integer_to_match==list_to_search[i]:
    print("Match found at index: ", i)

list_to_search.index(integer_to_match)
```

The time complexity of the above algorithm is **O(n)**.

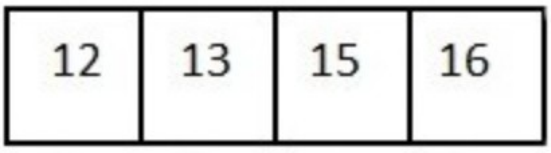
Another solution can be:



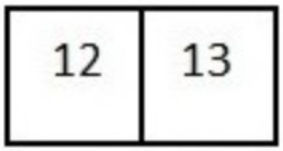




$$16 * \frac{1}{2} = 8$$



$$8 * \frac{1}{2} = 4$$



$$4 * \frac{1}{2} = 2$$

$$2 * \frac{1}{2} = 1$$

$$n = 2^k$$

$$\log_2 n = k$$

The time complexity of the above algorithm is **O(log n)**.

```
def binary_search(arr, low, high, x):

    # Check base case
    if high >= low:

        mid = (high + low) // 2

        # If element is present at the middle itself
        if arr[mid] == x:
            return mid

        # If element is smaller than mid, then it can only
        # be present in left subarray
        elif arr[mid] > x:
            return binary_search(arr, low, mid - 1, x)

        # Else the element can only be present in right subarray
        else:
            return binary_search(arr, mid + 1, high, x)

    else:
```

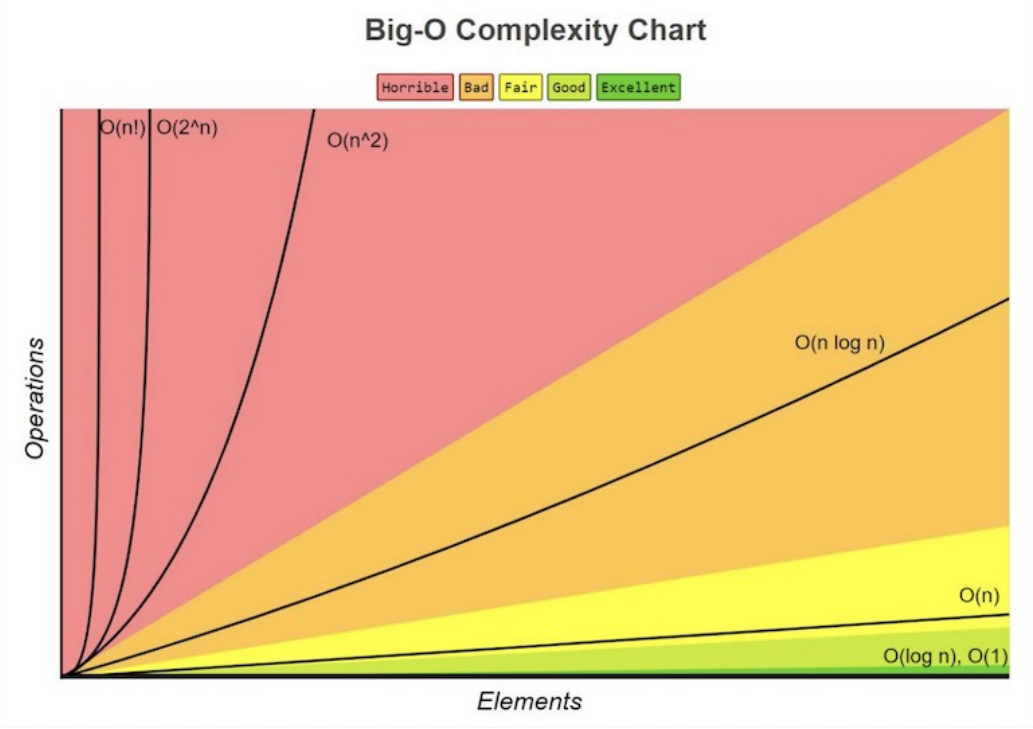
```
        # Element is not present in the array
        return -1

# Test array
integer_to_match = 13
list_to_search = [1, 3, 5, 8, 12, 13, 15, 16, 18, 20, 22, 30, 40, 50,
55, 67]

# Function call
result = binary_search(list_to_search, 0, len(list_to_search)-1,
integer_to_match)

print("Match found at index", str(result))
```

What if you knew the position of the number. How many operations would be required to find the number?

```
print(list_to_search[5]==integer_to_match)
```

The time complexity of the above algorithm is **O(1)**.

Mathematical Definition of Big-O: f(n) = O(g(n)) if there exists a positive integer n0 and a positive constant c, such that f(n) ≤ c.g(n) ∀ n≥n0.



Space Complexity

The term Space Complexity is misused for Auxiliary Space at many places. Following are the correct definitions of Auxiliary Space and Space Complexity.

Auxiliary Space is the extra space or temporary space used by an algorithm.

Space Complexity of an algorithm is total space taken by the algorithm with respect to the input size. Space complexity includes both Auxiliary space and space used by input.

Lets take a simple example. You want to calculate the sum of the numbers in a list.

```
list_to_sum = [1, 2, 3, 4, 5, 6, 7]

sum = 0
for i in range(len(list_to_sum)):
  sum = sum + list_to_sum[i]

sum
```

For the approach above:

- Auxiliary Space: O(1)
- Space Complexity: O(n)

```
sum = []
for i in range(len(list_to_sum)):
  if i==0:
    sum.append(list_to_sum[i])
  else:
    sum.append(sum[-1] + list_to_sum[i])
  print(sum)

sum

sum[-1]
```

For the approach above:

- Auxiliary Space: O(n)
- Space Complexity: O(n)

Example

You are given two lists A and B. A and B both contain 10 positive integers. You have to find the rectangle with the maximum area that can be formed using two integers (one as length and one as breadth) each one coming from one of these two lists.

```
# Approach 1

A = [1,11,4,9,16,2,23,45,77,15]
B = [51,29,82,32,67,5,2,22,78,98]
areas = []
for a in A:
  for b in B:
    areas.append(a * b)
```

```
max_area = 0
for area in areas:
  if area>max_area:
    max_area=area

len(areas)

max_area

max(areas)
```

What is the time and space complexity?

```
# Approach 2

A = [1,11,4,9,16,2,23,45,77,15]
B = [51,29,82,32,67,5,2,22,78,98]

max_a = 0
for a in A:
  if a>max_a:
    max_a=a

max_b = 0
for b in B:
  if b>max_b:
    max_b=b

max_a * max_b
```

What is the time and space complexity?