In [38]:

```python
from IPython.display import Image
Image(filename='logo.png', height=340, width=900)
```

Out[38]:

02 - DATA STRUCTURES IN PYTHON

WHAT IS A DATA STRUCTURE?

`Data structures` are basically just that - they are structures which can hold some data together. In other words, they are used to store a collection of related data.

There are four built-in data structures in Python:

- **`list`**
- **`tuple`**
- **`dictionary`**
- **`set`**

DATA STRUCTURE: LISTS

A list is a mutable, ordered collection of objects. "Mutable" means a list can be altered after it is created. You can, for example, add new items to a list or remove existing items. Lists are heterogeneous, meaning they can hold objects of different types.

A list is always within square brackets and different items are separated by comma

CREATION OF LISTS

In [39]:

```python
# Construct a list with a comma separated sequence of objects within square brackets:
my_list = ["Lesson", 5, "Is Fun?", True, 5, "Lesson"]
print(my_list)
```

```
['Lesson', 5, 'Is Fun?', True, 5, 'Lesson']
```

In [40]:

```python
mylist=["Neha", 29, False, "No download"]
print(mylist)
print(mylist)
print(type(mylist))
```

```
['Neha', 29, False, 'No download']
<class 'list'>
```

In [41]:

```
# Alternatively, you can construct a list by passing some other iterable into the list() function.
# An iterable describes an object you can look through one item at a time, such as lists, tuples,
strings and other sequences.

second_list = list("Life is Study")      # Create a list from a string
print(second_list)
```

```
['L', 'i', 'f', 'e', ' ', 'i', 's', ' ', 'S', 't', 'u', 'd', 'y']
```

In [42]:

```
# A list with no contents is known as the empty list:
empty_list = []
print( empty_list)
```

```
[]
```

ADDING AN ITEM TO A LIST

- **APPEND**
- **INSERT**
- **EXTEND**

APPEND()

You can add an item to an existing list with the `list.append()` method:

In [43]:

```
empty_list.append("I'm no longer empty!")
print(empty_list)
```

```
["I'm no longer empty!"]
```

In [44]:

```
empty_list.append(5)
print(empty_list)
```

```
["I'm no longer empty!", 5]
```

INSERT()

To add an item at the specified index, use the `.insert()` method:

In [45]:

```
fruits=["apple", "banana", "cherry"]
fruits.insert(2,"orange")                    #Adding an item at INDEX 2
fruits
```

Out[45]:

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

In [46]:

```
fruits.insert(2,"Apricot")
print(fruits)
```

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

EXTEND

You can also add a sequence to the end of an existing list with the `list.extend()` method:

In [47]:

```
third_list=["Atsla Vista", "Hello", 10]
mylist.extend(third_list)
print(mylist)

third_list=["Atsla Vista", "Hello", 10]
my_list.extend(third_list)
print(my_list)
```

```
['Neha', 29, False, 'No download', 'Atsla Vista', 'Hello', 10]
['Lesson', 5, 'Is Fun?', True, 5, 'Lesson', 'Atsla Vista', 'Hello', 10]
```

REMOVING AN ITEM FROM A LIST

- **REMOVE**
- **POP**
- **DEL**
- **CLEAR**

REMOVE()

Remove a matching item from a list with `list.remove()`:

In [48]:

```
my_list = ["Lesson", 5, "Is Fun?", True]
my_list.remove(5)
print(my_list)
```

```
['Lesson', 'Is Fun?', True]
```

`Note`: Remove deletes the first matching item only.

POP()

You can also remove items from a list using the `list.pop()` method. It removes the specified index, (or the last item if index is not specified)

In [49]:

```
anotherlist=["Hello", "my", "bestest", "friend"]
print(anotherlist)
anotherlist.pop()    #NO INDEX SPECIFIED
print(anotherlist)
anotherlist.pop(1)   #ITEM at INDEX 1 removed
print(anotherlist)
```

```
['Hello', 'my', 'bestest', 'friend']
['Hello', 'my', 'bestest']
```

```
['Hello', 'bestest']
```

DEL()

You can use indexing to change the values within a list or delete items in a list:

In [50]:

```
anotherlist=["Hello", "my", "bestest", "friend"]
anotherlist[2]="new"       #Set the value at index 1 = "new"
print(anotherlist)
del anotherlist[2]
print(anotherlist)
```

```
['Hello', 'my', 'new', 'friend']
['Hello', 'my', 'friend']
```

In [51]:

```
vowels=['a','e','i','o','u']
vowels.pop(2)
print(vowels)
```

```
['a', 'e', 'o', 'u']
```

In [52]:

```
print(vowels)
del vowels[2]
```

```
['a', 'e', 'o', 'u']
```

In [53]:

```
print(vowels)
```

```
['a', 'e', 'u']
```

The del keyword can also delete the list completely:

In [54]:

```
del fruits
```

In [55]:

```
fruits
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-55-5842b46f59fb> in <module>()
----> 1 fruits

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

CLEAR()

The `.clear()` method empties the list:

In [56]:

```
vowels.clear()
print(vowels)
```

```
[]
```

LEN, MAX, MIN, SUM

We can also check the length, maximum, minimum and sum of a list with the `.len()`, `.max()`, `.min()` and `.sum()` functions, respectively

In [57]:

```
num_list=[1,3,5,7,9]
print(len(num_list))                # Check the length
print(max(num_list))                # Check the max
print(min(num_list))                # Check the min
print(sum(num_list))                # Check the sum
print(sum(num_list)/len(num_list))  # Check the average
```

```
5
9
1
25
5.0
```

`Note`: Python does not have a built-in function to calculate the mean, but the numpy library we will introduce in upcoming lessons does.

MEMBERSHIP OPERATORS

You can check whether a list contains a certain object with the "in" keyword:

In [58]:

```
1 in num_list
```

Out[58]:

```
True
```

Add the keyword "not" to test whether a list does not contain an object:

In [59]:

```
1 not in num_list
```

Out[59]:

```
False
```

COUNT

Count the occurrences of an object within a list using the `list.count()` method:

In [60]:

```
num_list.append(3)
num_list.append(3)
num_list.append(3)
```

In [61]:

```
num_list
```

```
Out[61]:

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

```
In [62]:

print(num_list.count(1))
print(num_list.count(3))
```

```
1
4
```

SORT & REVERSE

Other common list methods include `list.sort()` and `list.reverse()`:

```
In [63]:

new_list=[1,5,4,2,3,6]                                   # Make new list
new_list.reverse()
print("Reversed List:", new_list)                         # Reverse the list

new_list.sort()
print("Ascending Sorted List:", new_list)

new_list.sort(reverse=True)
print("Descending Sorted List:", new_list)                         # Sort the List
```

```
Reversed List: [6, 3, 2, 4, 5, 1]
Ascending Sorted List: [1, 2, 3, 4, 5, 6]
Descending Sorted List: [6, 5, 4, 3, 2, 1]
```

SORTING IN DESCENDING ORDER

Use the tab to within the `()` and use the `REVERSE` PARAMETER to sort in descending order:

```
In [64]:

# Vowels List
vowels = ['a','e','i', 'o', 'u']

#sort the vowels
vowels.sort()

#print vowels
print("Sorted Vowels:", vowels)

#print vowels in reverse order
vowels.sort(reverse=True)
print("Reverse Sorted List", vowels)
```

```
Sorted Vowels: ['a', 'e', 'i', 'o', 'u']
Reverse Sorted List ['u', 'o', 'i', 'e', 'a']
```

DATA STRUCTURE: TUPLES

A tuple is a collection which is ordered and unchangeable. A tuple can also include duplicate elements. In Python tuples are written with round bracket.

They are an immutable sequence data type that are commonly used to hold short collections of related data. For instance, if you wanted to store latitude and longitude coordinates for cities, tuples might be a good choice, because the values are related and not likely to change. Like lists, tuples can store objects of different types.

CREATING TUPLES

Construct a tuple with a comma separated sequence of objects within parentheses:

In [65]:

```
mytuple=(1,3,5)
print(mytuple)
print(type(mytuple))
```

```
(1, 3, 5)
<class 'tuple'>
```

Alternatively, you can construct a tuple by passing an iterable into the **`tuple()`** function:

In [66]:

```
mylist=[2,3,1,4]
anothertuple=tuple(mylist)
anothertuple
```

Out[66]:

```
(2, 3, 1, 4)
```

`NOTE`: Tuples generally support the same indexing and slicing operations as lists and they also support some of the same functions, with the caveat that tuples cannot be changed after they are created. *This means WE CAN DO things like find the length, max or min of a tuple, but WE CAN'T APPEND new values to them OR REMOVE VALUES from them:*

DATA STRUCTURE: DICTONARIES

SIGNIFICANCE OF DICTIONARIES Sequence data types like lists are ordered. Ordering can be useful in some cases, such as if your data is sorted or has some other natural sense of ordering, but it comes at a price. When you search through sequences like lists, your computer has to go through each element one at a time to find an object you're looking for. Consider the following code:

my_list = [1,2,3,4,5,6,7,8,9,10]

0 in my_list

When running the code above, Python has to search through the entire list, one item at a time before it returns that 0 is not in the list. This sequential searching isn't much of a concern with small lists like this one, but if you're working with data that contains thousands or millions of values, it can add up quickly.

Dictionaries let you check whether they contain objects without having to search through each element one at a time, at the cost of have no order and using more system memory.

A dictionary is a collection which is unordered, changeable and indexed.

In other words, it is an object that maps a set of named indexes called keys to a set of corresponding values. Dictionaries are mutable, so you can add and remove keys and their associated values. A dictionary's keys must be immutable objects, such as int, string or tuple, but the values can be anything.

In Python dictionaries are written with curly brackets, and they have keys and values.

CREATE A DICTONARY

Create a dictionary with a comma-separated list of key: value pairs within curly braces:

In [67]:

```
mydict={'name':'Joe', "age": 10, "city": "Paris"}
print(mydict)
print(type(mydict))
```

```
{'name': 'Joe', 'age': 10, 'city': 'Paris'}
<class 'dict'>
```

```
<class 'dict'>
```

Notice that in the printed dictionary, the items don't appear in the same order as when we defined it, since dictionaries are unordered.

ACCESS A DICTONARY

We can access the items of a dictionary by referring to its key name, inside square brackets:

In [68]:

```
mylist[2]
```

Out[68]:

```
1
```

In [69]:

```
mytuple[1]
```

Out[69]:

```
3
```

In [70]:

```
mydict['age']
```

Out[70]:

```
10
```

We can also use the get() method to get the same result:

In [71]:

```
mydict.get('age')
```

Out[71]:

```
10
```

CHANGING VALUES

We can change the value of a specific item by referring to its key name:

In [72]:

```
mydict={'name':'Joe', "age": 10, "city": "Paris"}
```

In [73]:

```
mydict['age']=15
print(mydict)
```

```
{'name': 'Joe', 'age': 15, 'city': 'Paris'}
```

ADDING NEW ITEMS

Add new items to an existing dictionary with the following syntax:

In [74]:

```
mydict['Gender']='Male'
print(mydict)

mydict['Background']='Management'
print(mydict)
```

```
{'name': 'Joe', 'age': 15, 'city': 'Paris', 'Gender': 'Male'}
{'name': 'Joe', 'age': 15, 'city': 'Paris', 'Gender': 'Male', 'Background': 'Management'}
```

REMOVING ITEMS

POP()

The `.pop()` method removes the item with the specified key name

In [75]:

```
mydict.pop('city')
print(mydict)
```

```
{'name': 'Joe', 'age': 15, 'Gender': 'Male', 'Background': 'Management'}
```

DEL()

The `del` keyword removes the item with the specified key name. The del keyword can also delete the dictionary completely.

In [76]:

```
del mydict
print(mydict)
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-76-2a0d72ed7222> in <module>()
      1 del mydict
----> 2 print(mydict)

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

In [77]:

```
#Delete existing key: value pairs with del:

mydict={'name':'Joe', "age": 10, "city": "Paris", "Gender": 'Male'}
print(mydict)
del mydict['Gender']
print(mydict)
```

```
{'name': 'Joe', 'age': 10, 'city': 'Paris', 'Gender': 'Male'}
{'name': 'Joe', 'age': 10, 'city': 'Paris'}
```

In [78]:

```
mydict['Gender']=["Male", "Female"]
print(mydict)
```

```
{'name': 'Joe', 'age': 10, 'city': 'Paris', 'Gender': ['Male', 'Female']}
```

LENGTH OF A DICTONARY

Check the number of ***items(KEYS)**** in a dict with `len()`:

Check the number of "items(KEYS)" in a dict with **len()**:

In [79]:

```
len(mydict)
```

Out[79]:

```
4
```

CHECKING EXISTENCE

Check whether a certain key exists with "in":

In [80]:

```
"name" in mydict
```

Out[80]:

```
True
```

In [81]:

```
"Gender" in mydict
```

Out[81]:

```
True
```

In [82]:

```
"gender" not in mydict
```

Out[82]:

```
True
```

KEYS(), VALUE() AND ITEMS() FUNCTIONS

You can access all the keys, all the values or all the key: value pairs of a dictionary with the **`.keys()`, `.value()` and `.items()`** methods respectively:

In [83]:

```
mydict.keys()
```

Out[83]:

```
dict_keys(['name', 'age', 'city', 'Gender'])
```

In [84]:

```
mydict.values()
```

Out[84]:

```
dict_values(['Joe', 10, 'Paris', ['Male', 'Female']])
```

In [85]:

```
mydict.items()
```

Out[85]:

```
dict_items([('name', 'Joe'), ('age', 10), ('city', 'Paris'), ('Gender', ['Male', 'Female'])])
```

Real world data often comes in the form tables of rows and columns, where each column specifies a different data feature like name or age and each row represents an individual record. We can encode this sort of tabular data in a dictionary by assigning each column label a key and then storing the column values as a list.

Name	Age	City
Joe	10	PARIS
Bob	15	NEW YORK
Harry	20	TOKYO

We can store this data in a dictionary like so:

In [86]:

```
mytabledict={'name':["Joe", "Bob", "Harry"], 'age':[10,15,20], 'city':['Paris', 'New York', 'Tokyo'
]}
print(mytabledict)
```

```
{'name': ['Joe', 'Bob', 'Harry'], 'age': [10, 15, 20], 'city': ['Paris', 'New York', 'Tokyo']}
```

DATA STRUCTURE: SETS

Sets are unordered, mutable collections of objects that CANNOT CONTAIN DUPLICATES. Sets are useful for storing and performing operations on data where each value is unique. In Python sets are written with curly brackets.

CREATING SETS

Create a set with a comma separated sequence of values within curly braces:

In [87]:

```
myset={1,2,3,4,5,6,7,8,9}
print(myset)
```

```
{1, 2, 3, 4, 5, 6, 7, 8, 9}
```

In [88]:

```
set_list=[1,2,3,4,5,6,6,6,6,6,'Hello','Hello']
mynewset=set(set_list)
print(mynewset)
```

```
{1, 2, 3, 4, 5, 6, 'Hello'}
```

ADD()

- **To add one item to a set use the `.add()` method**
- **To add more than one item to a set use the `.update()` method**

In [89]:

```
myset.add(10)
print(myset)

myset.update([11,12,13,14,15])
print(myset)
```

```
{1, 2, 3, 4, 5, 6, 7, 8, 9, 10}
{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15}
```

REMOVE()

An item can be removed from the set using the `.remove()` method

In [90]:

```
myset.remove(15)
print(myset)
```

```
{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14}
```

Alternatively, use **`.discard()`** to remove the item from the set.

In [91]:

```
myset.discard(12)
print(myset)
```

```
{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 13, 14}
```

In [92]:

```
myset.discard(12)
print(myset)
```

```
{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 13, 14}
```

`NOTE`: Discard does not throw an error if an item is not in the set but remove throws an error in case the item to be removed is not in the set.

Like in Dictonary,

- **`.pop()` will remove the last item in the set**
- **`.clear()` will clear the set**
- **`del()` will delete the set completely**

SET OPERATIONS

The main purpose of sets is to perform set operations that compare or combine different sets. Python sets support many common mathematical set operations like:

- union,
- intersection,
- difference and
- checking whether one set is a subset of another:

UNION SETS

In [93]:

```
set1={1,3,5,6}
set2={1,2,3,4}
set1.union(set2)                        #Get the Union of two sets
```

Out[93]:

```
{1, 2, 3, 4, 5, 6}
```

In [94]:

```
set1.intersection(set2)                 # Get the intersection of two sets
```

```
set1.intersection(set2)                 # Get the intersection of two sets
```

Out[94]:

```
{1, 3}
```

In [95]:

```
print(set1.difference(set2))           # Get the difference between two sets
print(set2.difference(set1))
```

```
{5, 6}
{2, 4}
```

In [96]:

```
set1={1,3,5,6}
set3={1,3,6}
print(set1.issubset(set3))                      # Check whether set1 is a subset of set3
print(set3.issubset(set1))                      # Check whether set3 is a subset of set1
```

```
False
True
```

NOTE: Read syntax as: whether set3 is a subset of set1; whether set1 is a subset of set3

LISTS INTO SETS

We can convert lists into set using the set() function. Converting a list to a set drops any duplicate elements in the list. This can be a useful way to strip unwanted duplicate items or count the number of unique elements in a list:

In [97]:

```
list11=[1,2,2,2,3,3,3,4,4,4,4,4,5,5,6,6,6,6,6,7]
set(list11)
```

Out[97]:

```
{1, 2, 3, 4, 5, 6, 7}
```

In [98]:

```
len(set(list11))
```

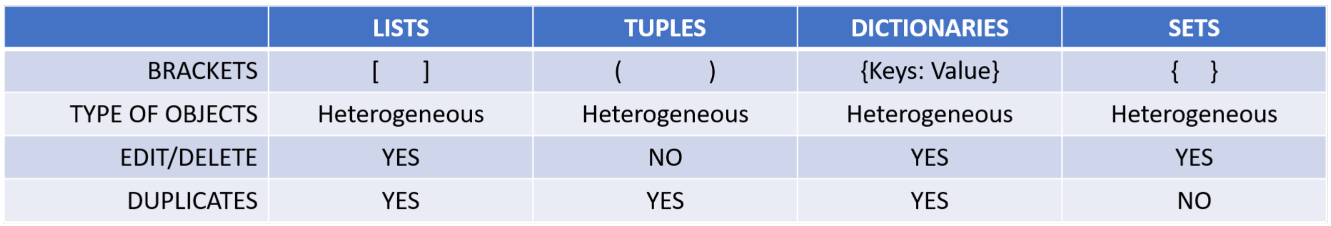
Out[98]:

```
7
```

In [101]:

```
Image(filename='DataStructures.png', height=600, width=900)
```

Out[101]:



	LISTS	TUPLES	DICTIONARIES	SETS
BRACKETS	[ ]	( )	{Keys: Value}	{ }
TYPE OF OBJECTS	Heterogeneous	Heterogeneous	Heterogeneous	Heterogeneous
EDIT/DELETE	YES	NO	YES	YES
DUPLICATES	YES	YES	YES	NO