Unit III: Data Structures in Python

Lists - Operations, Slicing, Methods; Tuples: Creating, Printing, properties of tuples, Sets, Dictionaries, Sequences and their properties. Defining Functions, Calling Functions, Passing Arguments, Keyword Arguments, (Function Returning Values), Scope of the Variables in a Function - Global and Local Variables

What is a List?

In other programming languages, list objects are declared **similarly to arrays**. Lists don't have to **be homogeneous all the time, so they can simultaneously store items of different data types**. Lists are helpful when we need **to iterate over some elements** and keep hold of the items.

What is a Tuple?

A tuple is another data structure to store the collection of items of many data types**, but unlike mutable lists, tuples are immutable.** Because of its static structure, the tuple is more efficient than the list.

Differences between Lists and Tuples

List and Tuple Syntax Differences

The syntax of a list differs from that of a tuple. Items of a **tuple are enclosed by parentheses or curved brackets (), whereas items of a list are enclosed by square brackets [].**

In [56]:

```
1 list1 = [4, 5, 7, 1, 7]
2 tuple1 = (4, 1, 8, 3, 9)
3
4 print("List is: ", list1)
5 print("Tuple is: ", tuple1)
6 print(type(list1))
7 print(type(tuple1))
```

```
List is:  [4, 5, 7, 1, 7]
Tuple is:  (4, 1, 8, 3, 9)
<class 'list'>
<class 'tuple'>
```

Mutable List vs. Immutable Tuple

An important difference between a list and a tuple is that lists are mutable, whereas tuples are immutable. **It means a list's items can be changed or modified, whereas a tuple's items cannot be changed or modified.**

In [57]:

```
# creating a list and a tuple
list1= ["Python", "Lists", "Tuples", "Differences"]
tuple1 = ("Python", "Lists", "Tuples", "Differences")

list1[3] = "Mutable"
print( list1 )
try:
    tuple1[3] = "Immutable"
    print( tuple1 )
except TypeError:
    print( "Tuples cannot be modified because they are immutable" )
```

```
['Python', 'Lists', 'Tuples', 'Mutable']
Tuples cannot be modified because they are immutable
```

Python Slicing

Both lists and tuples allow you to extract a subset of elements using slicing

In [1]:

```
1 my_list = [1, 2, 3, 4, 5]
2 my_tuple = (6, 7, 8, 9, 10)
3
4 print(my_list[1:3])
5 print(my_tuple[:3])
6
```

```
[2, 3]
(6, 7, 8)
```

Tuples are Memory Efficient

As tuples are stored in a single memory block therefore they don't require extra space for new objects whereas the lists are allocated in two blocks, first the fixed one with all the Python object information and second a variable-sized block for the data.

In [26]:

```
1 import sys
2 a_list = []
3 a_tuple = ()
4 a_list1 = ["A","B"]
5 a_tuple1 = ("A","B")
6 print(sys.getsizeof(a_list))
7 print(sys.getsizeof(a_tuple))
8 print(sys.getsizeof(a_list1))
9 print(sys.getsizeof(a_tuple1))
```

```
56
40
72
56
```

Python Indexing

Both lists and tuples allow you to access individual elements using their index, starting from 0.

In [2]:

```
1 my_list = [1, 2, 3]
2 my_tuple = (4, 5, 6)
3
4 print(my_list[0])
5 print(my_tuple[1])
```

```
1
5
```

Python Slicing

Both lists and tuples allow you to extract a subset of elements using slicing.

In [4]:

```
1 my_list = [1, 2, 3, 4, 5]
2 my_tuple = (6, 7, 8, 9, 10)
3 print(my_list[1:3])
4 print(my_tuple[:3])
```

```
[2, 3]
(6, 7, 8)
```

Python Concatenation

Both lists and tuples can be concatenated using the “+” operator.

In [1]:

```
1 list1 = [1, 2, 3]
2 list2 = [4, 5, 6]
3 tuple1 = (7, 8, 9)
4 tuple2 = (10, 11, 12)
5 print(list1 + list2)
6 print(tuple1 + tuple2)
```

```
[1, 2, 3, 4, 5, 6]
(7, 8, 9, 10, 11, 12)
```

Python Append
Lists can be appended with new elements using the append() method.

Python Extend
Lists can also be extended with another list using the extend() method.

Python Remove
Lists can have elements removed using the remove() method.

In [2]:

```
#Append
print("Append")
my_list = [1, 2, 3]
my_list.append(4)
print(my_list)
print("Extend")
#Extend
list1 = [1, 2, 3]
list2 = [4, 5, 6]
list1.extend(list2)
print(list1)
print("Remove")
#Remove
my_list = [1, 2, 3, 4]
my_list.remove(2)
print(my_list)
```

```
Append
[1, 2, 3, 4]
Extend
[1, 2, 3, 4, 5, 6]
Remove
[1, 3, 4]
```

Differences between List and Tuple in Python

Sno	LIST	TUPLE
1	Lists are mutable	Tuples are immutable
2	The implication of iterations is Time-consuming	The implication of iterations is comparatively Faster
3	The list is better for performing operations, such as insertion and deletion.	A Tuple data type is appropriate for accessing the elements
4	Lists consume more memory	Tuple consumes less memory as compared to the list
5	Lists have several built-in methods	Tuple does not have many built-in methods.
6	Unexpected changes and errors are more likely to occur	In a tuple, it is hard to take place.

Tuples and Lists: Key Similarities

- They both hold collections of items and **are heterogeneous data types, meaning they can contain multiple data types** simultaneously.
- They're both ordered, which implies the items or **objects are maintained in the same order** as they were placed until changed manually.
- Because they're both **sequential data structures, we can iterate through the objects they hold;** hence, they are iterables.
- An integer index, **enclosed in square brackets [index],** can be used to access objects of both data types.

When to Use Tuples Over Lists?

In Python, tuples and lists are both used to store collections of data, but they have some important differences. Here are some situations where you might want to use tuples instead of lists –

Immutable Data – Tuples are immutable, thus once they are generated, their contents cannot be changed. This makes **tuples a suitable option** for storing information that shouldn't change, **such as setup settings, constant values, or other information that should stay the same** while your program is running.

Performance – Tuples are more **lightweight than lists and might be quicker to generate, access, and iterate** through since they are immutable. **Using a tuple can be more effective** than using a list if you have a **huge collection of data that you need to store, retrieve, and use regularly and that data does not need to be altered**.

Data integrity – By ensuring that **the data's structure and contents stay consistent**, **tuples can be utilized to ensure data integrity.** To make sure the caller is aware of how much data to expect, for instance, **if a function returns a set amount of values, you might want to return t**hem as a tuple rather than a list.

Python - Access List Items

1. Access Items

List items are indexed and you can access them by referring to the index number.

2. Negative Indexing

Negative indexing means start from the end
-1 refers to the last item, -2 refers to the second last item etc.

3. Range of Indexes

You can specify a range of indexes by specifying where to start and where to end the range.
When specifying a range, the return value will be a new list with the specified items.

```
#1.Accessing items in a list
print("1")
thislist = ["apple", "banana", "cherry"]
print(thislist[1])


#2.Negative Indexing
print("2")
thislist = ["apple", "banana", "cherry"]
print(thislist[-1])


#3.Range of Indexes
print("3")
thislist = ["apple", "banana", "cherry", "orange", "kiwi", "melon", "mango"]
print(thislist[2:5])

thislist = ["apple", "banana", "cherry", "orange", "kiwi", "melon", "mango"]
print(thislist[:4])

thislist = ["apple", "banana", "cherry", "orange", "kiwi", "melon", "mango"]
print(thislist[2:])
```

```python
#4.Range of Negative Indexes
print("4")
thislist = ["apple", "banana", "cherry", "orange", "kiwi", "melon", "mango"]
print(thislist[-4:-1])



#5.Check if Item Exists
print("5")
thislist = ["apple", "banana", "cherry"]
if "apple" in thislist:
  print("Yes, 'apple' is in the fruits list")
```

```
1
banana
2
cherry
3
['cherry', 'orange', 'kiwi']
['apple', 'banana', 'cherry', 'orange']
['cherry', 'orange', 'kiwi', 'melon', 'mango']
4
['orange', 'kiwi', 'melon']
5
Yes, 'apple' is in the fruits list
```

Python - Change List Items

1. Change Item Value

To change the value of a specific item, refer to the index number.

2. Change a Range of Item Values

To change the value of items within a specific range, define a list with the new values, and refer to the range of index numbers where you want to insert the new values.

3. Insert Items

To insert a new list item, without replacing any of the existing values, we can use the insert() method. The insert() method inserts an item at the specified index.

```
#Change Item Value
print("1")
thislist = ["apple", "banana", "cherry"]
thislist[1] = "blackcurrant"
print(thislist)


#Change a Range of Item Values
print("2")
thislist = ["apple", "banana", "cherry", "orange", "kiwi", "mango"]
thislist[1:3] = ["blackcurrant", "watermelon"]
print(thislist)


thislist = ["apple", "banana", "cherry"]
thislist[1:2] = ["blackcurrant", "watermelon"]
print(thislist)


thislist = ["apple", "banana", "cherry"]
thislist[1:3] = ["watermelon"]
print(thislist)
```

```
#Insert Items
print("3")
thislist = ["apple", "banana", "cherry"]
thislist.insert(2, "watermelon")
print(thislist)
```

```
1
['apple', 'blackcurrant', 'cherry']
2
['apple', 'blackcurrant', 'watermelon', 'orange', 'kiwi', 'mango']
['apple', 'blackcurrant', 'watermelon', 'cherry']
['apple', 'watermelon']
3
['apple', 'banana', 'watermelon', 'cherry']
```

Python - Add List Items

1. Append Items

To add an item to the end of the list, use the append() method.

2. Insert Items

To insert a list item at a specified index, use the insert() method.
The insert() method inserts an item at the specified index.

3. Extend List

To append elements from another list to the current list, use the extend() method.

4. Add Any Iterable

The extend() method does not have to append lists, you can add any iterable object (tuples, sets, dictionaries etc.).

```
#Append Items
print("1")
thislist = ["apple", "banana", "cherry"]
thislist.append("orange")
print(thislist)


#Insert Items
print("2")
thislist = ["apple", "banana", "cherry"]
thislist.insert(1, "orange")
print(thislist)


#Extend List
print("3")
thislist = ["apple", "banana", "cherry"]
tropical = ["mango", "pineapple", "papaya"]
thislist.extend(tropical)
print(thislist)


#Add Any Iterable
print("4")
thislist = ["apple", "banana", "cherry"]
thistuple = ("kiwi", "orange")
thislist.extend(thistuple)
print(thislist)
```

```
1
['apple', 'banana', 'cherry', 'orange']
2
['apple', 'orange', 'banana', 'cherry']
3
['apple', 'banana', 'cherry', 'mango', 'pineapple', 'papaya']
4
['apple', 'banana', 'cherry', 'kiwi', 'orange']
```

Python - Remove List Items

1. Remove Specified Item

The remove() method removes the specified item.

2. Remove Specified Index

The pop() method removes the specified index.

3. Clear the List

The clear() method empties the list.
The list still remains, but it has no content.

```
#Remove Specified Item
print("1")
thislist = ["apple", "banana", "cherry"]
thislist.remove("banana")
print(thislist)

thislist = ["apple", "banana", "cherry", "banana", "kiwi"]
thislist.remove("banana")
print(thislist)

#Remove Specified Index
print("2")
thislist = ["apple", "banana", "cherry"]
thislist.pop(1)
print(thislist)


thislist = ["apple", "banana", "cherry"]
thislist.pop()
print(thislist)


thislist = ["apple", "banana", "cherry"]
del thislist[0]
print(thislist)
```

```python
thislist = ["apple", "banana", "cherry"]
del thislist

#Clear the List
print("3")
thislist = ["apple", "banana", "cherry"]
thislist.clear()
print(thislist)
```

```
1
['apple', 'cherry']
['apple', 'cherry', 'banana', 'kiwi']
2
['apple', 'cherry']
['apple', 'banana']
['banana', 'cherry']
3
[]
```

Python - List Comprehension

List Comprehension

List comprehension offers a shorter syntax when you want to create a new list based on the values of an existing list.

Example:

Based on a list of fruits, you want a new list, containing only the fruits with the letter "a" in the name.

Without list comprehension you will have to write a for statement with a conditional test inside:

```
#1st way List Comprehension
fruits = ["apple", "banana", "cherry", "kiwi", "mango"]
newlist = []
for x in fruits:
  if "a" in x:
    newlist.append(x)
print(newlist)


#2nd way List Comprehension
fruits = ["apple", "banana", "cherry", "kiwi", "mango"]
newlist = [x for x in fruits if "a" in x]
print(newlist)
```

```
['apple', 'banana', 'mango']
['apple', 'banana', 'mango']
```

Python - Loop Lists

1. Loop Through a List

You can loop through the list items by using a for loop

2. Loop Through the Index Numbers

You can also loop through the list items by referring to their index number.

Use the range() and len() functions to create a suitable iterable.

3. Using a While Loop

You can loop through the list items by using a while loop. Use the len() function to determine the length of the list, then start at 0 and loop your way through the list items by referring to their indexes.

Remember to increase the index by 1 after each iteration.

4. Looping Using List Comprehension

List Comprehension offers the shortest syntax for looping through lists

```
#Loop Through a List
print("1")
thislist = ["apple", "banana", "cherry"]
for x in thislist:
  print(x)

#Loop Through the Index Numbers
print("2")
thislist = ["apple", "banana", "cherry"]
for i in range(len(thislist)):
  print(thislist[i])


#Using a While Loop
print("3")
thislist = ["apple", "banana", "cherry"]
i = 0
while i < len(thislist):
  print(thislist[i])
  i = i + 1

#Looping Using List Comprehension
print("4")
thislist = ["apple", "banana", "cherry"]
[print(x) for x in thislist]
```

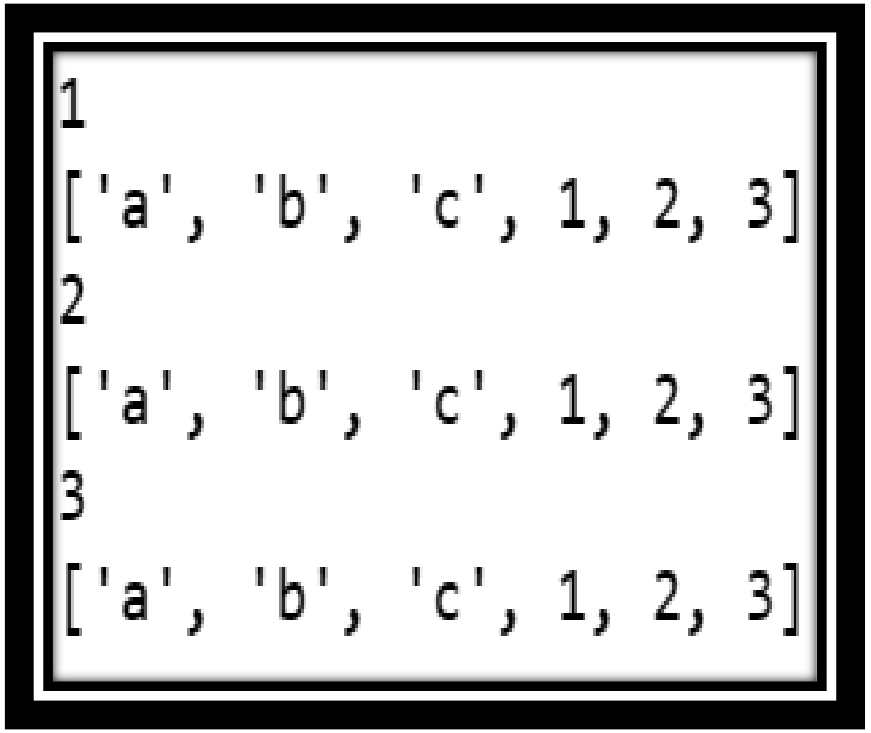
```
1
apple
banana
cherry
2
apple
banana
cherry
3
apple
banana
cherry
4
apple
banana
cherry
```

Python - Join Lists

Join Two Lists

There are several ways to join, or concatenate, two or more lists in Python.

1. One of the easiest ways are by using the + operator.
2. Another way to join two lists is by appending all the items from list2 into list1, one by one.
3. Or you can use the extend() method, where the purpose is to add elements from one list to another list




```
#'+' operator
print("1")
list1 = ["a", "b", "c"]
list2 = [1, 2, 3]
list3 = list1 + list2
print(list3)

# append() method
print("2")
list1 = ["a", "b" , "c"]
list2 = [1, 2, 3]
for x in list2:
  list1.append(x)
print(list1)

#extend() method
print("3")
list1 = ["a", "b" , "c"]
list2 = [1, 2, 3]
list1.extend(list2)
print(list1)
```

Copy a List

You cannot copy a list simply by typing list2 = list1, because: list2 will only be a reference to list1, and changes made in list1 will automatically also be made in list2.

There are ways to make a copy, one way is to use the built-in List method copy().

```
#Copy a List
print("1")
thislist = ["apple", "banana", "cherry"]
mylist = thislist.copy()
print(mylist)


thislist = ["apple", "banana", "cherry"]
mylist = list(thislist)
print(mylist)
```

```
1
['apple', 'banana', 'cherry']
['apple', 'banana', 'cherry']
```

Python - Sort Lists

1. Sort List Alphanumerically

List objects have a sort() method that will sort the list alphanumerically, ascending, by default.

2. Sort Descending

To sort descending, use the keyword argument reverse = True.

3. Customize Sort Function

Using the keyword argument key = function.
The function will return a number that will be used to sort the list (the lowest number first).

```
#Sort List Alphanumerically
print("1")
thislist = ["orange", "mango", "kiwi", "pineapple", "banana"]
thislist.sort()
print(thislist)

thislist = [100, 50, 65, 82, 23]
thislist.sort()
print(thislist)

#Sort Descending
print("2")
thislist = ["orange", "mango", "kiwi", "pineapple", "banana"]
thislist.sort(reverse = True)
print(thislist)



thislist = [100, 50, 65, 82, 23]
thislist.sort(reverse = True)
print(thislist)

#Customize Sort Function
print("3")
def myfunc(n):
  return abs(n)
thislist = [100, 50, 65, 82, 23]
thislist.sort(key = myfunc)
print(thislist)
```

4. Case Insensitive Sort

By default the sort() method is case sensitive, resulting in all capital letters being sorted before lower case letters.

5. Reverse Order

The reverse() method reverses the current sorting order of the elements.

```
1
['banana', 'kiwi', 'mango', 'orange', 'pineapple']
[23, 50, 65, 82, 100]
2
['pineapple', 'orange', 'mango', 'kiwi', 'banana']
[100, 82, 65, 50, 23]
3
[23, 50, 65, 82, 100]
4
['Kiwi', 'Orange', 'banana', 'cherry']
['banana', 'cherry', 'Kiwi', 'Orange']
5
['cherry', 'Kiwi', 'Orange', 'banana']
```

```python
#Case Insensitive Sort
print("4")
thislist = ["banana", "Orange", "Kiwi", "cherry"]
thislist.sort()
print(thislist)



thislist = ["banana", "Orange", "Kiwi", "cherry"]
thislist.sort(key = str.lower)
print(thislist)

#Reverse Order
print("5")
thislist = ["banana", "Orange", "Kiwi", "cherry"]
thislist.reverse()
print(thislist)
```

Python List count() Method

The count() method returns the number of elements with the specified value.

Python List index() Method

The index() method returns the position at the first occurrence of the specified value.

```
1
count= 1
count= 2
2
index= 2
index= 3
```

```
#count() method
print("1")
fruits = ['apple', 'banana', 'cherry']
x = fruits.count("cherry")
print('count=',x)


points = [1, 4, 2, 9, 7, 8, 9, 3, 1]
x = points.count(9)
print('count=',x)

#index() Method
print("2")
fruits = ['apple', 'banana', 'cherry']
x = fruits.index("cherry")
print('index=',x)

fruits = [4, 55, 64, 32, 16, 32]
x = fruits.index(32)
print('index=',x)
```

Python Tuples

```
mytuple = ("apple", "banana", "cherry")
```

Tuple

Tuples are used to store multiple items in a single variable.

A tuple is a collection which is ordered and unchangeable.

Tuples are written with round brackets.

1. Create a Tuple

Tuple items are ordered, unchangeable, and allow duplicate values.

Tuple items are indexed, the first item has index [0], the second item has index [1] etc.

Ordered

When we say that tuples are ordered, it means that the items have a defined order, and that order will not change.

Unchangeable

Tuples are unchangeable, meaning that we cannot change, add or remove items after the tuple has been created.

Allow Duplicates

Since tuples are indexed, they can have items with the same value:

Tuple Length

To determine how many items a tuple has, use the len() function:

Create Tuple With One Item

To create a tuple with only one item, you have to add a comma after the item, otherwise Python will not recognize it as a tuple.

```
#NOT a tuple
thistuple = ("apple")
print(type(thistuple))
```

Tuple Items - Data Types

Tuple items can be of any data type:

Example

String, int and boolean data types:

tuple1 = ("apple", "banana", "cherry")
tuple2 = (1, 5, 7, 9, 3)
tuple3 = (True, False, False)
tuple4 = ("abc", 34, True, 40, "male")

The tuple() Constructor

It is also possible to use the tuple() constructor to make a tuple.

```
Create a Tuple
('apple', 'banana', 'cherry')
Allow Duplicates
('apple', 'banana', 'cherry', 'apple', 'cherry')
Tuple length
3
Tuple With One Item
<class 'tuple'>
The tuple() Constructor
('apple', 'banana', 'cherry')
```

```
print("Create a Tuple")
thistuple = ("apple", "banana", "cherry")
print(thistuple)

print("Allow Duplicates")
thistuple = ("apple", "banana", "cherry", "apple",
print(thistuple)

print("Tuple length")
thistuple = ("apple", "banana", "cherry")
print(len(thistuple))

print("Tuple With One Item")
thistuple = ("apple",)
print(type(thistuple))

print("The tuple() Constructor")
thistuple = tuple(("apple", "banana", "cherry"))
print(thistuple)
```

Python - Update Tuples

Accessing tuples is same as that of lists

1. Change Tuple Values

Once a tuple is created, you cannot change its values. Tuples are unchangeable, or immutable as it also is called.

2. Add Items

Since tuples are immutable, they do not have a built-in append() method, but there are other ways to add items to a tuple.

1. Convert into a list: you can convert it into a list, add your item(s), and convert it back into a tuple.
2. Add tuple to a tuple: if you want to add one item, (or many), create a new tuple with the item(s), and add it to the existing tuple:

```
#Change Tuple Values
print("1")
x = ("apple", "banana", "cherry")
y = list(x)
y[1] = "kiwi"
x = tuple(y)
print(x)

#Add Items
print("2")
thistuple = ("apple", "banana", "cherry")
y = list(thistuple)
y.append("orange")
thistuple = tuple(y)
print(thistuple)

thistuple = ("apple", "banana", "cherry")
y = ("orange",)
thistuple += y
print(thistuple)
```

3. Remove Items

Tuples are **unchangeable**, so you cannot remove items from it, but you can use the same workaround as we used for changing and adding tuple items:

```
#Remove Items
print("3")
thistuple = ("apple", "banana", "cherry")
y = list(thistuple)
y.remove("apple")
thistuple = tuple(y)
print(thistuple)
```

```
1
('apple', 'kiwi', 'cherry')
2
('apple', 'banana', 'cherry', 'orange')
('apple', 'banana', 'cherry', 'orange')
3
('banana', 'cherry')
```

```
thistuple = ("apple", "banana", "cherry")
del thistuple
print(thistuple)
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
Input In [18], in <cell line: 3>()
      1 thistuple = ("apple", "banana", "cherry")
      2 del thistuple
----> 3 print(thistuple)

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

Python - Unpack Tuples

1. Unpacking a Tuple

When we create a tuple, we normally assign values to it. This is called "packing" a tuple.
But, in Python, we are also allowed to extract the values back into variables. This is called "unpacking".

2. Using Asterisk*

The number of variables must match the number of values in the tuple, if not, you must add an * to the variable name and the values will be assigned to the variable as a list.
If the asterisk is added to another variable name than the last, Python will assign values to the variable until the number of values left matches the number of variables left.

```
print("Unpacking a tuple")
fruits = ("apple", "banana", "cherry")
(green, yellow, red) = fruits
print(green)
print(yellow)
print(red)

print("\nUsing Asterisk*")
fruits = ("apple", "banana", "cherry", "strawberry"
(green, yellow, *red) = fruits
print(green)
print(yellow)
print(red)
print()

fruits = ("apple", "mango", "papaya", "pineapple",
(green, *tropic, red) = fruits
print(green)
print(tropic)
print(red)
```

```
Unpacking a tuple
apple
banana
cherry

Using Asterisk*
apple
banana
['cherry', 'strawberry', 'raspberry']

apple
['mango', 'papaya', 'pineapple']
cherry
```

Python - Loop Tuples

1. Loop Through a Tuple
You can loop through the tuple items by using a for loop.

2. Loop Through the Index Numbers
You can also loop through the tuple items by referring to their index number.
Use the range() and len() functions to create a suitable iterable.

3. Using a While Loop
You can loop through the tuple items by using a while loop.
Remember to increase the index by 1 after each iteration.

```
print("Loop Through a Tuple")
thistuple = ("apple", "banana", "cherry")
for x in thistuple:
  print(x)

print("\nLoop Through the Index Numbers")
thistuple = ("apple", "banana", "cherry")
for i in range(len(thistuple)):
  print(thistuple[i])

print("\nUsing a While Loop")
thistuple = ("apple", "banana", "cherry")
i = 0
while i < len(thistuple):
  print(thistuple[i])
  i = i + 1
```

```
Loop Through a Tuple
apple
banana
cherry

Loop Through the Index Numbers
apple
banana
cherry

Using a While Loop
apple
banana
cherry
```

Python - Join Tuples

1. Join Two Tuples

To join two or more tuples you can use the + operator.

2. Multiply Tuples

If you want to multiply the content of a tuple a given numbe of times, you can use the * operator.

```
print("Join two tuples")
tuple1 = ("a", "b" , "c")
tuple2 = (1, 2, 3)
tuple3 = tuple1 + tuple2
print(tuple3)

print("\nMultiply the fruits tuple by 2")
fruits = ("apple", "banana", "cherry")
mytuple = fruits * 2
print(mytuple)
```

```
Join two tuples
('a', 'b', 'c', 1, 2, 3)

Multiply the fruits tuple by 2
('apple', 'banana', 'cherry', 'apple', 'banana', 'cherry')
```

Tuple Methods

Python has two built-in methods that you can use on tuples.

Method	Description
count()	Returns the number of times a specified value occurs in a tuple
index()	Searches the tuple for a specified value and returns the position of where it was found

Python Sets

- Sets are used to store multiple items in a single variable.

- Set is one of 4 built-in data types in Python used to store collections of data, the other 3 are List, Tuple, and Dictionary.

- A set is a collection which is unordered, unchangeable*, and unindexed.

- * Note: Set items are unchangeable, but you can remove items and add new items.

- Note: Sets are unordered, so you cannot be sure in which order the items will appear.
- Set Items

- Set items are unordered, unchangeable, and do not allow duplicate values.

- There are four collection data types in the Python programming language:

  - ❖ List is a collection which is ordered and changeable. Allows duplicate members.
  - ❖ Tuple is a collection which is ordered and unchangeable. Allows duplicate members.
  - ❖ Set is a collection which is unordered, unchangeable*, and unindexed. No duplicate members.
  - ❖ Dictionary is a collection which is ordered** and changeable. No duplicate members.

- *Set items are unchangeable, but you can remove items and add new items.

- As of Python version 3.7, dictionaries are ordered. In Python 3.6 and earlier, dictionaries are unordered.

Syntax:

```
thisset = {"apple", "banana", "cherry"}
print(thisset)
```

1. Duplicates Not Allowed
Sets cannot have two items with the same value. The values True and 1 are considered the same value in sets, and are treated as duplicates.

2. Get the Length of a Set
To determine how many items a set has, use the len() function.

3. Set Items - Data Types
Set items can be of any data type

4. type()
From Python's perspective, sets are defined as objects with the data type 'set'

5. The set() Constructor
It is also possible to use the set() constructor to make a set.

```
#Duplicates Not Allowed
print("1")
thisset = {"apple", "banana", "cherry", "apple"}
print(thisset)


thisset = {"apple", "banana", "cherry", True, 1, 2}
print(thisset)

#Get the Length of a Set
print("2")
thisset = {"apple", "banana", "cherry"}
print(len(thisset))

#type()
print("3")
myset = {"apple", "banana", "cherry"}
print(type(myset))

#The set() Constructor
print("4")
thisset = set(("apple", "banana", "cherry"))
print(thisset)
```

```
1
{'cherry', 'banana', 'apple'}
{True, 2, 'banana', 'cherry', 'apple'}
2
3
3
<class 'set'>
4
{'cherry', 'banana', 'apple'}
```

Python - Access Set Items

1. Access Items

You cannot access items in a set by referring to an index or a key. But you can loop through the set items using a for loop, or ask if a specified value is present in a set, by using the in keyword.

Python - Add Set Items

1. Add Items

To add one item to a set use the add() method.

2. Add Sets

To add items from another set into the current set, use the update() method.

3. Add Any Iterable

The object in the update() method does not have to be a set, it can be any iterable object (tuples, lists, dictionaries etc.).

```
#Access Items
print("1")
thisset = {"apple", "banana", "cherry"}
for x in thisset:
  print(x)


thisset = {"apple", "banana", "cherry"}
print("banana" in thisset)

#Add Items
print("2")
thisset = {"apple", "banana", "cherry"}
thisset.add("orange")
print(thisset)

#Add sets
print("3")
thisset = {"apple", "banana", "cherry"}
tropical = {"pineapple", "mango", "papaya"}
thisset.update(tropical)
print(thisset)

#Add any iterable
print("4")
thisset = {"apple", "banana", "cherry"}
mylist = ["kiwi", "orange"]
thisset.update(mylist)
print(thisset)
```

```
1
cherry
banana
apple
True
2
{'cherry', 'banana', 'apple', 'orange'}
3
{'pineapple', 'banana', 'mango', 'cherry', 'papaya', 'apple'}
4
{'banana', 'kiwi', 'apple', 'orange', 'cherry'}
```

Change Items

Once a set is created, you cannot change its items, but you can add new items.

Python - Remove Set Items

Remove Item

To remove an item in a set, use the remove(), or the discard() method.
Remove a random item by using the pop() method.
The clear() method empties the set.
The del keyword will delete the set completely.

```
thisset = {"apple", "banana", "cherry"}
thisset.remove("banana")
print(thisset)

thisset = {"apple", "banana", "cherry"}
thisset.discard("banana")
print(thisset)

thisset = {"apple", "banana", "cherry"}
x = thisset.pop()
print(x)
print(thisset)

thisset = {"apple", "banana", "cherry"}
thisset.clear()
print(thisset)

thisset = {"apple", "banana", "cherry"}
del thisset
print(thisset)
```

```
{'cherry', 'apple'}
{'cherry', 'apple'}
cherry
{'banana', 'apple'}
set()

---------------------------------------------------------------------
----------------------
NameError                                                 Trace
ost recent call last)
Input In [2], in <cell line: 20>()
      18 thisset = {"apple", "banana", "cherry"}
      19 del thisset
---> 20 print(thisset)

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

Python - Loop Sets

Loop Items

You can loop through the set items by using a for loop:

Example
Loop through the set, and print the values:

```
thisset = {"apple", "banana", "cherry"}
for x in thisset:
  print(x)
```

Python - Join Sets

1. Join Two Sets

There are several ways to join two or more sets in Python.
You can use the union() method that returns a new set containing all items from both sets, or the update() method that inserts all the items from one set into another:
Both union() and update() will exclude any duplicate items.

2. Keep ONLY the Duplicates

The intersection_update() method will keep only the items that are present in both sets.

The intersection() method will return a new set, that only contains the items that are present in both sets.

```
print("union()")
set1 = {"a", "b" , "c"}
set2 = {1, 2, 3}
set3 = set1.union(set2)
print(set3)

print("\nupdate()")
set1 = {"a", "b" , "c"}
set2 = {1, 2, 3}
set1.update(set2)
print(set1)

print("\nintersection_update()")
x = {"apple", "banana", "cherry"}
y = {"google", "microsoft", "apple"}
x.intersection_update(y)
print(x)

print("\nintersection()")
x = {"apple", "banana", "cherry"}
y = {"google", "microsoft", "apple"}
z =x.intersection(y)
print(z)
```

```
union()
{1, 2, 3, 'c', 'a', 'b'}

update()
{1, 2, 3, 'c', 'a', 'b'}

intersection_update()
{'apple'}

intersection()
{'apple'}
```

3. Keep All, But NOT the Duplicates

The symmetric_difference_update() method will keep only the elements that are NOT present in both sets.

The symmetric_difference() method will return a new set, that contains only the elements that are NOT present in both sets.
The values True and 1 are considered the same value in sets, and are treated as duplicates.

```
symmetric_difference_update()
{'microsoft', 'google', 'cherry', 'banana'}

{'microsoft', 'google', 'cherry', 'banana'}

{2, 'cherry', 'google', 'banana'}
```

```
print("\nsymmetric_difference_update()")
x = {"apple", "banana", "cherry"}
y = {"google", "microsoft", "apple"}
x.symmetric_difference_update(y)
print(x)
print()

x = {"apple", "banana", "cherry"}
y = {"google", "microsoft", "apple"}
z = x.symmetric_difference(y)
print(z)
print()

x = {"apple", "banana", "cherry", True}
y = {"google", 1, "apple", 2}
z = x.symmetric_difference(y)
print(z)
```

Python - Set Methods

1. Python Set copy() Method

Example
Copy the fruits set:

```
fruits = {"apple", "banana", "cherry"}
x = fruits.copy()
print(x)
```

2. Python Set isdisjoint() Method

Example
Return True if no items in set x is present in set y:

```
x = {"apple", "banana", "cherry"}
y = {"google", "microsoft", "facebook"}
z = x.isdisjoint(y)
print(z)
```

3. Python Set issubset() Method

Example

Return True if all items in set x are present in set y:

```
x = {"a", "b", "c"}
y = {"f", "e", "d", "c", "b", "a"}
z = x.issubset(y)
print(z)
```

4. Python Set issuperset() Method

Example

Return True if all items set y are present in set x:

```
x = {"f", "e", "d", "c", "b", "a"}
y = {"a", "b", "c"}
z = x.issuperset(y)
print(z)
```

Python Dictionaries

```
thisdict = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
```

Dictionary

Dictionaries are used to store data values in key:value pairs.

A dictionary is a collection which is ordered*, changeable and do not allow duplicates.

As of Python version 3.7, dictionaries are ordered. In Python 3.6 and earlier, dictionaries are unordered.

Dictionaries are written with curly brackets, and have keys and values:

Ordered or Unordered?

As of Python version 3.7, dictionaries are ordered. In Python 3.6 and earlier, dictionaries are unordered.

Changeable

Dictionaries are changeable, meaning that we can change, add or remove items after the dictionary has been created.

Duplicates Not Allowed

Dictionaries cannot have two items with the same key.

```python
print("Create and print a dictionary")
thisdict ={
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
print(thisdict)
print()
print("Print the \"brand\" value of the dictionary")
thisdict ={
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
print(thisdict["brand"])
print()
print("Duplicate values will overwrite existing values")
thisdict ={
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964,
  "year": 2020
}
print(thisdict)
print()
```

```
Create and print a dictionary
{'brand': 'Ford', 'model': 'Mustang', 'year': 1964}

Print the "brand" value of the dictionary
Ford

Duplicate values will overwrite existing values
{'brand': 'Ford', 'model': 'Mustang', 'year': 2020}
```

Dictionary Length

To determine how many items a dictionary has, use the len() function.

Example

Print the number of items in the dictionary:

```
print(len(thisdict))
```

Dictionary Items - Data Types

The values in dictionary items can be of any data type:

Example

String, int, boolean, and list data types:

```
thisdict =	{
  "brand": "Ford",
  "electric": False,
  "year": 1964,
  "colors": ["red", "white", "blue"]
}
```

type()

From Python's perspective, dictionaries are defined as objects with the data type 'dict': <class 'dict'>

Example

Print the data type of a dictionary:

```
thisdict =	{
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
print(type(thisdict))
```

The dict() Constructor

It is also possible to use the dict() constructor to make a dictionary.

Example

Using the dict() method to make a dictionary:

```
thisdict = dict(name = "John", age = 36,
country = "Norway")
print(thisdict)
```

Python - Access Dictionary Items

Accessing Items

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

Get Keys

The keys() method will return a list of all the keys in the dictionary.

Get Values

The values() method will return a list of all the values in the dictionary.
The list of the values is a view of the dictionary, meaning that any changes done to the dictionary will be reflected in the values list.

```
print("Get the value of the \"model\" key")
thisdict ={
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
x = thisdict["model"]
print(x)
print()

print("get() that will give you the same result")
thisdict ={
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
x = thisdict.get("model")
print(x)
print()
```

```
Get the value of the "model" key
Mustang

get() that will give you the same result
Mustang
```

```python
print("Get a list of the keys")
car = {
"brand": "Ford",
"model": "Mustang",
"year": 1964
}
x = car.keys()
print(x) #before the change
car["color"] = "white"
print(x) #after the change
print()

print("Get a list of the values")
car = {
"brand": "Ford",
"model": "Mustang",
"year": 1964
}
x = car.values()
print(x) #before the change
car["year"] = 2020
print(x) #after the change
print()
```

```
Get a list of the keys
dict_keys(['brand', 'model', 'year'])
dict_keys(['brand', 'model', 'year', 'color'])

Get a list of the values
dict_values(['Ford', 'Mustang', 1964])
dict_values(['Ford', 'Mustang', 2020])
```

Get Items

The items() method will return each item in a dictionary, as tuples in a list.

Check if Key Exists

To determine if a specified key is present in a dictionary use the in keyword

```
print("Get a list of the key:value pairs")
car = {
"brand": "Ford",
"model": "Mustang",
"year": 1964
}
x = car.items()
print(x)
print()

print("Check if \"model\" is present in the dictionary")
thisdict ={
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
if "model" in thisdict:
  print("Yes, 'model' is one of the keys in the thisdict dictionary")
```

```
Get a list of the key:value pairs
dict_items([('brand', 'Ford'), ('model', 'Mustang'), ('year', 1964)])

Check if "model" is present in the dictionary
Yes, 'model' is one of the keys in the thisdict dictionary
```

Python - Change Dictionary Items

You can change the value of a specific item by referring to its key name.

```
print("Change the \"year\" to 2018")
thisdict ={
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
thisdict["year"] = 2018
print(thisdict)
```

```
Change the "year" to 2018
{'brand': 'Ford', 'model': 'Mustang', 'year': 2018}
```

Python - Add Dictionary Items

Adding an item to the dictionary is done by using a new index key and assigning a value to it.

Update Dictionary

The update() method will update the dictionary with the items from a given argument. If the item does not exist, the item will be added.

The argument must be a dictionary, or an iterable object with key:value pairs.

```
print("Add Dictionary Items")
thisdict ={
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
thisdict["color"] = "red"
print(thisdict)
print()

print("update() method")
thisdict ={
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
thisdict.update({"brand": "Audi"})
thisdict.update({"price": "1000USD"})
print(thisdict)
```

```
Add Dictionary Items
{'brand': 'Ford', 'model': 'Mustang', 'year': 1964, 'color': 'red'}

update() method
{'brand': 'Audi', 'model': 'Mustang', 'year': 1964, 'price': '1000USD'}
```

Python - Remove Dictionary Items

popitem() method removes the last inserted item (in versions before 3.7, a random item is removed instead)

```
pop() method removes the item with the specified key name
{'brand': 'Ford', 'year': 1964}

popitem() method removes the last inserted item
{'brand': 'Ford', 'model': 'Mustang'}
```

```
print("pop() method removes the item with the specified key name")
thisdict ={
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
thisdict.pop("model")
print(thisdict)
print()

print("popitem() method removes the last inserted item")
thisdict ={
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
thisdict.popitem()
print(thisdict)
print()
```

```
print("del keyword removes the item with the specified key name")
thisdict ={
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
del thisdict["model"]
print(thisdict)
print()

print("clear() method empties the dictionary")
thisdict ={
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
thisdict.clear()
print(thisdict)
print()

print("del keyword can also delete the dictionary completely")
thisdict ={
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
del thisdict
print(thisdict)
```

```
del keyword removes the item with the specified key name
{'brand': 'Ford', 'year': 1964}

clear() method empties the dictionary
{}

del keyword can also delete the dictionary completely

---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
Input In [26], in <cell line: 48>()
     42 thisdict ={
     43   "brand": "Ford",
     44   "model": "Mustang",
     45   "year": 1964
     46 }
     47 del thisdict
---> 48 print(thisdict)

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

```
print("Print all key names in the dictionary")
for x in thisdict:
  print(x)
print()

print("Print all values in the dictionary, one by one")
for x in thisdict:
  print(thisdict[x])
print()

print("values() method to return values of a dictionary")
for x in thisdict.values():
  print(x)
print()

print("keys() method to return the keys of a dictionary")
for x in thisdict.keys():
  print(x)
print()

print("keys and values, by using the items() method")
for x, y in thisdict.items():
  print(x, y)
```

Python - Loop Dictionaries

You can loop through a dictionary by using a for loop.

```
Print all key names in the dictionary
brand
model
year
color

Print all values in the dictionary, one by one
Ford
Mustang
1964
Red

values() method to return values of a dictionary
Ford
Mustang
1964
Red

keys() method to return the keys of a dictionary
brand
model
year
color

keys and values, by using the items() method
brand Ford
model Mustang
year 1964
color Red
```

Python - Copy Dictionaries

Copy a Dictionary

```
print("Copy a dictionary with the copy() method")
thisdict ={
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
mydict = thisdict.copy()
print(mydict)
print()

print("Make a copy of a dictionary with the dict()")
thisdict ={
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
mydict = dict(thisdict)
thisdict["color"]="Red"
print(mydict)
```

```
Copy a dictionary with the copy() method
{'brand': 'Ford', 'model': 'Mustang', 'year': 1964}

Make a copy of a dictionary with the dict()
{'brand': 'Ford', 'model': 'Mustang', 'year': 1964}
```

Python - Nested Dictionaries

Nested Dictionaries

A dictionary can contain dictionaries, this is called nested dictionaries.

Access Items in Nested Dictionaries

To access items from a nested dictionary, you use the name of the dictionaries, starting with the outer dictionary.

```
print("Create a dictionary that contain three dictionaries")
myfamily = {
  "child1" : {
    "name" : "Emil",
    "year" : 2004
  },
  "child2" : {
    "name" : "Tobias",
    "year" : 2007
  },
  "child3" : {
    "name" : "Linus",
    "year" : 2011
  }
}
print(myfamily)
print()
```

```
Create a dictionary that contain three dictionaries
{'child1': {'name': 'Emil', 'year': 2004}, 'child2':
{'name': 'Tobias', 'year': 2007}, 'child3': {'name':
'Linus', 'year': 2011}}
```

```python
print("Create one dictionarythat will contain the other three dictionaries")
child1 = {
  "name" : "Emil",
  "year" : 2004
}
child2 = {
  "name" : "Tobias",
  "year" : 2007
}
child3 = {
  "name" : "Linus",
  "year" : 2011
}

myfamily = {
  "child1" : child1,
  "child2" : child2,
  "child3" : child3
}
print(myfamily)
print()

print("Print the name of child 2")
print(myfamily["child2"]["name"])
```

```
Create one dictionarythat will contain the other three
dictionaries
{'child1': {'name': 'Emil', 'year': 2004}, 'child2':
{'name': 'Tobias', 'year': 2007}, 'child3': {'name':
'Linus', 'year': 2011}}

Print the name of child 2
Tobias
```

Python Functions

A function is a block of code which only runs when it is called. You can pass data, known as parameters, into a function. A function can return data as a result.

1. Creating a Function

In Python a function is defined using the def keyword.

2. Calling a Function

To call a function, use the function name followed by parenthesis.

3. Arguments

Information can be passed into functions as arguments. Arguments are specified after the function name, inside the parentheses. You can add as many arguments as you want, just separate them with a comma.

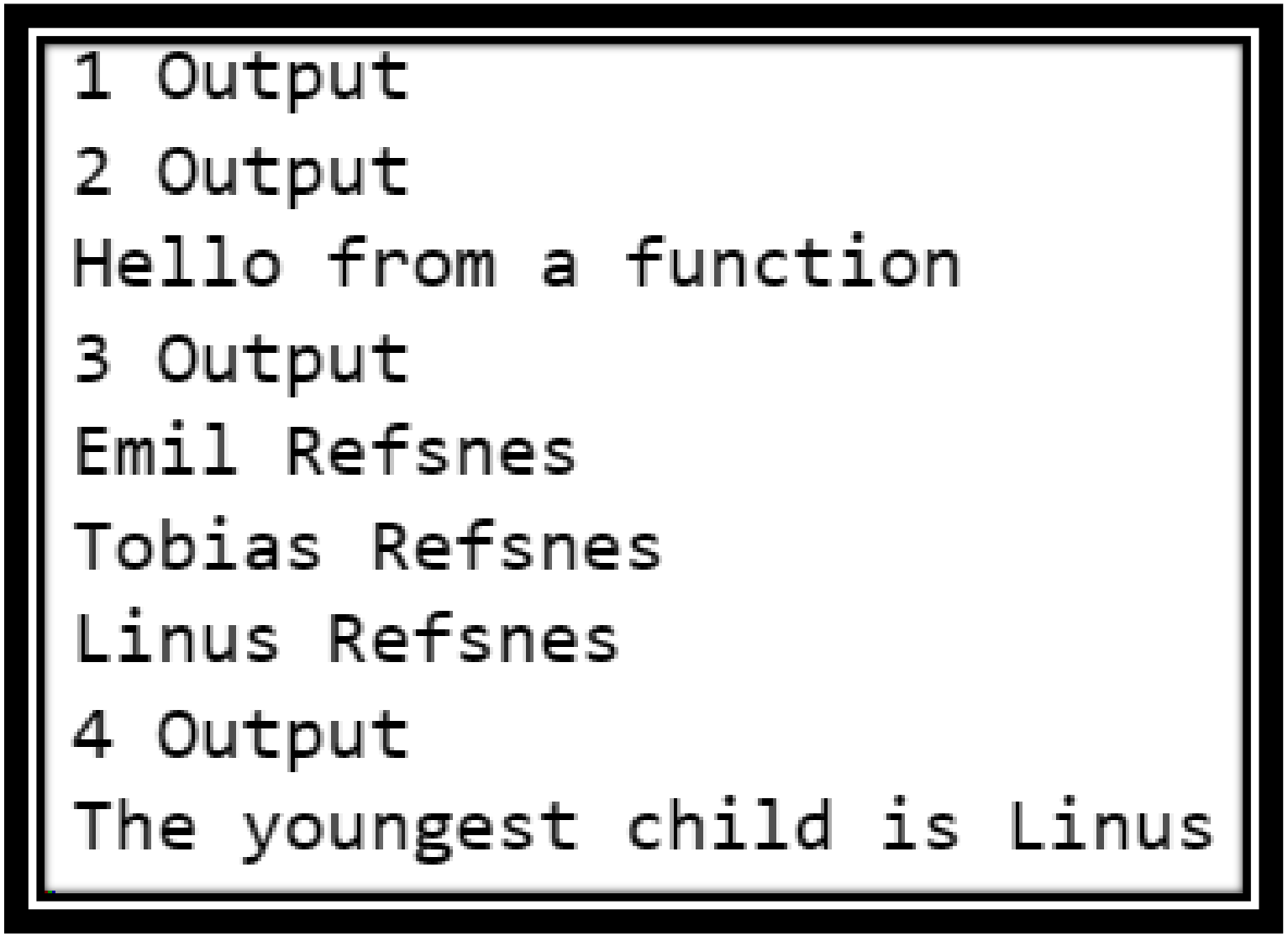
```
#Creating a Function
print("1 Output")
def my_function():
  print("Hello from a function")

#Calling a Function
print("2 Output")
def my_function():
  print("Hello from a function")
my_function()

#Arguments
print("3 Output")
def my_function(fname):
  print(fname + " Refsnes")
my_function("Emil")
my_function("Tobias")
my_function("Linus")

#Arbitrary Arguments, *args
print("4 Output")
def my_function(*kids):
  print("The youngest child is " + kids[2])
my_function("Emil", "Tobias", "Linus")
```

From a function's perspective:
A parameter is the variable listed inside the parentheses in the function definition.
An argument is the value that is sent to the function when it is called.



Number of Arguments

By default, a function must be called with the correct number of arguments. Meaning that if your function expects 2 arguments, you have to call the function with 2 arguments, not more, and not less.

Arbitrary Arguments, *args

If you do not know how many arguments that will be passed into your function, add a * before the parameter name in the function definition.

This way the function will receive a tuple of arguments, and can access the items accordingly.

Keyword Arguments

You can also send arguments with the key = value syntax.

This way the order of the arguments does not matter.

```
#Keyword Arguments
print("5 Output")
def my_function(child3, child2, child1):
  print("The youngest child is " + child3)
my_function(child1 = "Emil", child2 = "Tobias", child3 = "Linus")


#Arbitrary Keyword Arguments, **kwargs
print("6 Output")
def my_function(**kid):
  print("His last name is " + kid["lname"])
my_function(fname = "Tobias", lname = "Refsnes")

#Default Parameter Value
print("7 Output")
def my_function(country = "Norway"):
  print("I am from " + country)
my_function("Sweden")
my_function("India")
my_function()
my_function("Brazil")

#Passing a List as an Argument
print("8 Output")
def my_function(food):
  for x in food:
    print(x)
fruits = ["apple", "banana", "cherry"]
my_function(fruits)
```

```
4 Output
The youngest child is Linus
5 Output
The youngest child is Linus
6 Output
His last name is Refsnes
7 Output
I am from Sweden
I am from India
I am from Norway
I am from Brazil
8 Output
apple
banana
cherry
```

Arbitrary Keyword Arguments, **kwargs

If you do not know how many keyword arguments that will be passed into your function, add two asterisk: ** before the parameter name in the function definition.

This way the function will receive a dictionary of arguments, and can access the items accordingly.
If the number of keyword arguments is unknown, add a double ** before the parameter name.

Default Parameter Value

The following example shows how to use a default parameter value.
If we call the function without argument, it uses the default value.

Passing a List as an Argument

You can send any data types of argument to a function (string, number, list, dictionary etc.), and it will be treated as the same data type inside the function.
E.g. if you send a List as an argument, it will still be a List when it reaches the function.

Return Values

To let a function return a value, use the return statement.

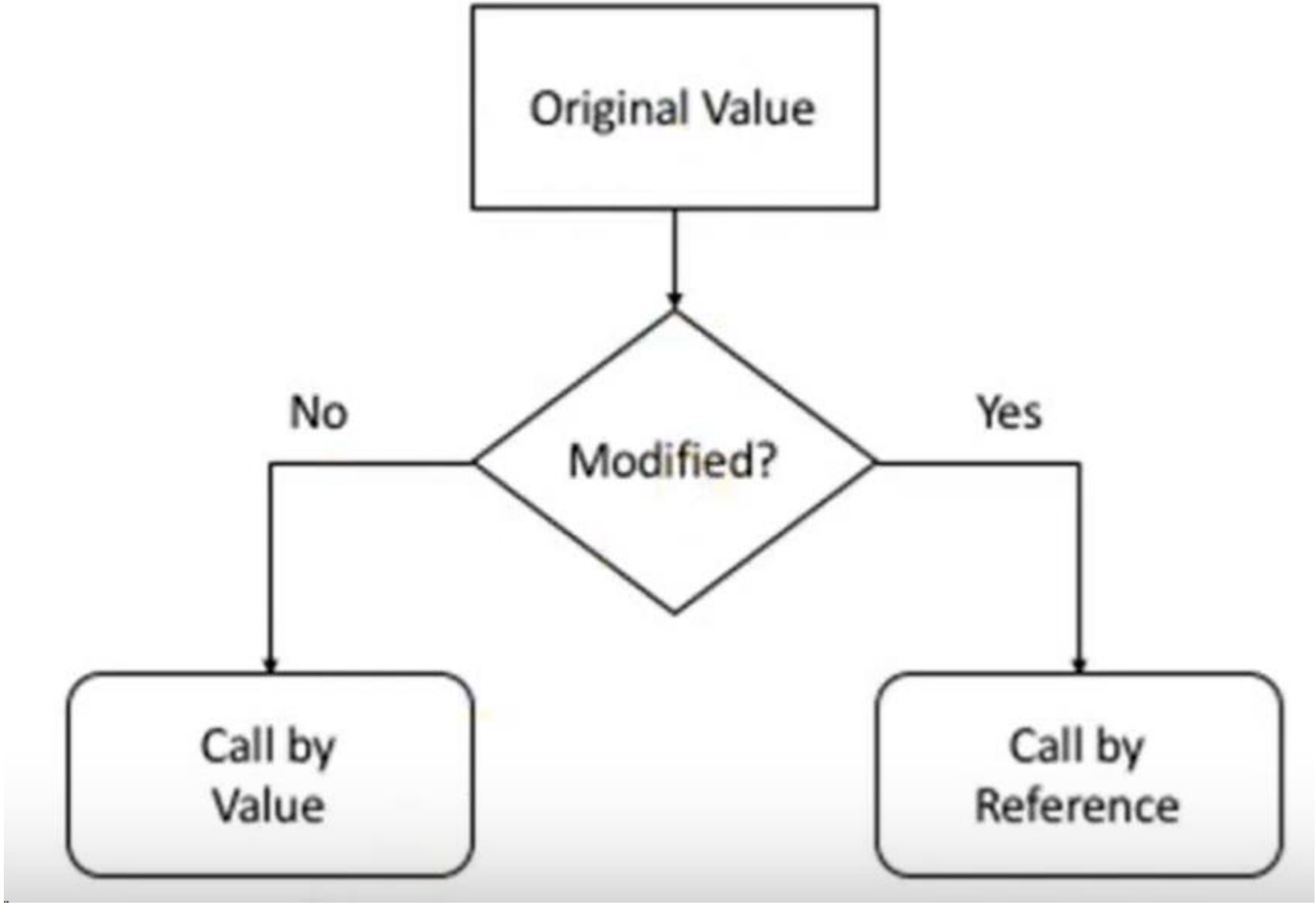
```
9 Output
15
25
45
10 Output
Emil Refsnes
```

```
#Return Values
print("9 Output")
def my_function(x):
  return 5 * x
print(my_function(3))
print(my_function(5))
print(my_function(9))

#Number of Arguments
print("10 Output")
def my_function(fname, lname):
  print(fname + " " + lname)
my_function("Emil", "Refsnes")


def my_function(fname, lname):
  print(fname + " " + lname)
my_function("Emil")
```

Call by value and Call by reference



```
list1=[12,13,14,15]
def test_value(list1):
    list1=[22,23,24,25]
    print("Inside Function:", list1)
test_value(list1)
print("Outside Function:", list1)

def test_reference1(list1):
    list1.append(26)
    list1=[22,23,24,25]
    print("Inside Function:", list1)
test_reference1(list1)
print("Outside Function:", list1)

def test_reference2(list1):
    list1[0]=99
    print("Inside Function:", list1)
test_reference2(list1)
print("Outside Function:", list1)
```

```
Inside Function: [22, 23, 24, 25]
Outside Function: [12, 13, 14, 15]
Inside Function: [22, 23, 24, 25]
Outside Function: [12, 13, 14, 15, 26]
Inside Function: [99, 13, 14, 15, 26]
Outside Function: [99, 13, 14, 15, 26]
```

Global and Local Variables

```
def python_2():
    print("Inside Function:",set3)
set3={40,40,40,50,50,60}
python_2()
print('outside function:',set3)

def python_1():
    set2={22,22,23,23}
    print("Inside Function:",set2)
python_1()
set2={30,31,31}
print('outside function:',set2)

def python():
    set1={11,12,12,13,14,14}
    print("Inside Function:",set1)
python()
print('outside function:',set1)
```

```
Inside Function: {40, 50, 60}
outside function: {40, 50, 60}
Inside Function: {22, 23}
outside function: {30, 31}
Inside Function: {11, 12, 13, 14}

---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
Input In [16], in <cell line: 21>()
     18     print("Inside Function:",set1)
     20 python()
---> 21 print('outside function:',set1)

NameError: name 'set1' is not defined
```