Python Data Structures/Collections

Week 2

Data Structures

LIST []

TUPLE ()

SET {}

DICTIONARY {}

Python Data Types

Text Type: str

Numeric Types: int, float, complex

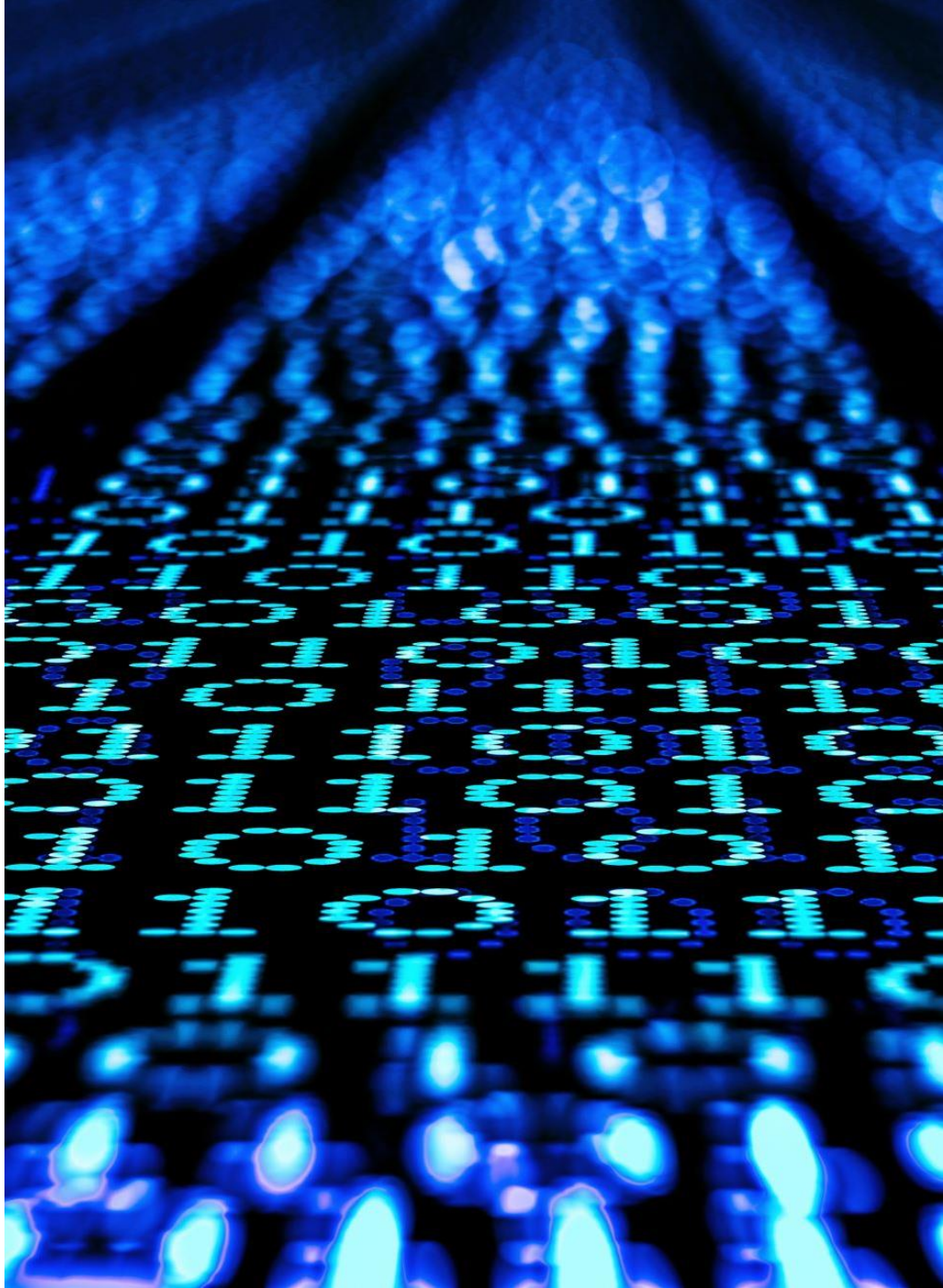
Sequence Types: list, tuple, range

Mapping Type: dict

Set Types: set, frozenset

Boolean Type: bool

Binary Types: bytes, bytearray, memoryview

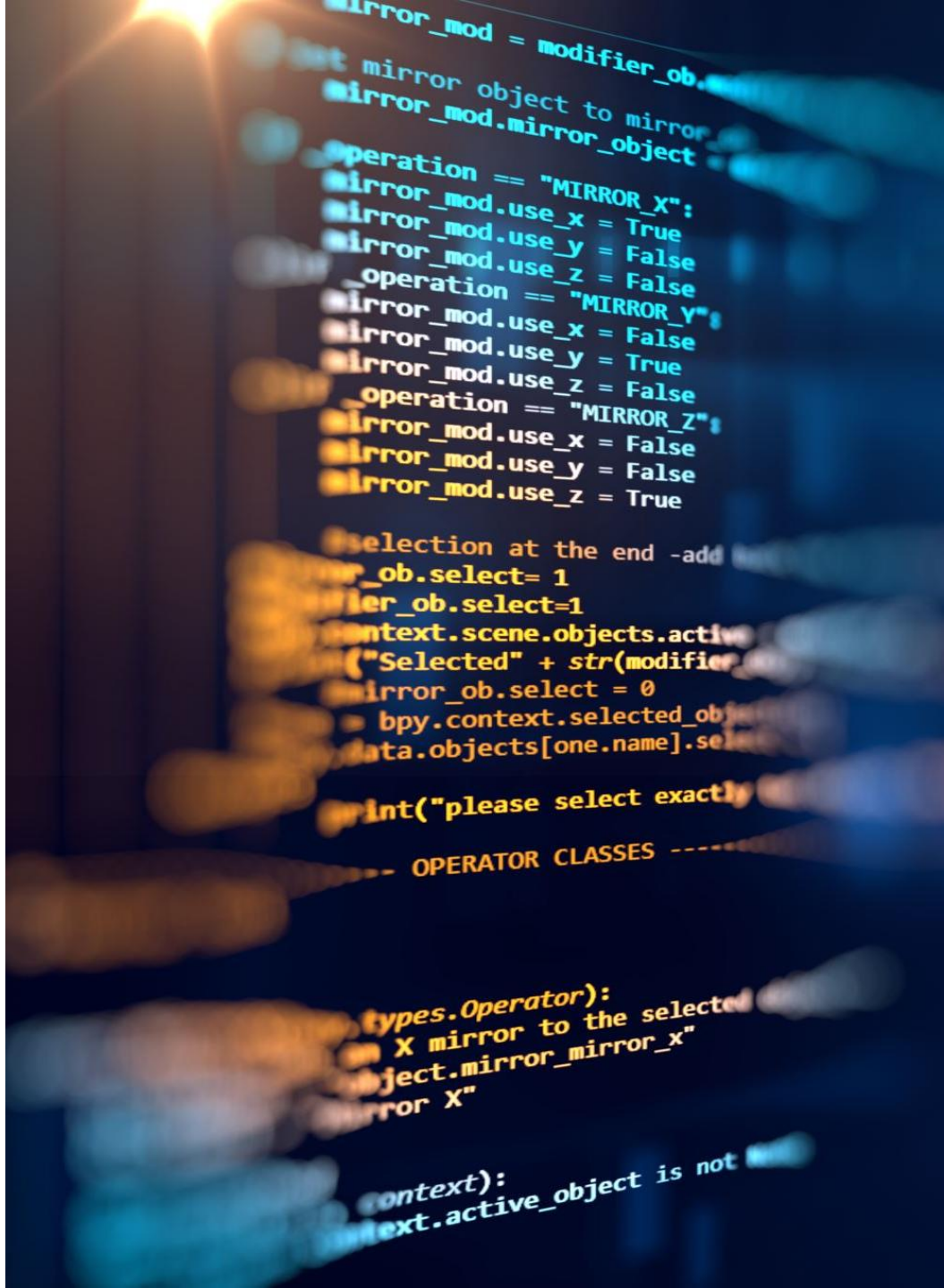


Getting the Data type of a variable

You can get the data type of any object by using the type() function:

```
x = 5

print(type(x))
```



Setting Data Types

Example	Data Type
x = "Hello World"	Str
x = 20	Int
x = 20.5	Float
x = 1j	Complex
x = ["apple", "banana", "cherry"]	List
x = ("apple", "banana", "cherry")	Tuple
x = range(6)	Range
x = {"name" : "John", "age" : 36}	Dict
x = {"apple", "banana", "cherry"}	Set
x = frozenset({"apple", "banana", "cherry"})	Frozenset
x = True	Bool
x = b"Hello"	Bytes
x = bytearray(5)	Bytearray
x = memoryview(bytes(5))	memoryview

Setting the Specific Data Type

Example	Data Type
x = str("Hello World")	Str
x = int(20)	Int
x = float(20.5)	Float
x = complex(1j)	Complex
x = list(("apple", "banana", "cherry"))	List
x = tuple(("apple", "banana", "cherry"))	Tuple
x = range(6)	Range
x = dict(name="John", age=36)	Dict
x = set(("apple", "banana", "cherry"))	Set
x = frozenset(("apple", "banana", "cherry"))	Frozenset
x = bool(5)	Bool
x = bytes(5)	Bytes
x = bytearray(5)	Bytearray
x = memoryview(bytes(5))	memoryview

Python List

Python List

Used to store multiple items in a single variable.

One of 4 built-in data types in Python used to store collections of data, the other 3 are Tuple, Set, and Dictionary, all with different qualities and usage.

Created using square brackets **[]**

Python List Example

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

List Items

Ordered, **changeable**, and **allow duplicate** values.

Indexed, the first item has index [0], the second item has index [1] etc.

Its an Ordered List

The items **have a defined order**, and that **order will not change**.

If you add new items to a list, the **new items will be placed at the end of the list**.

It’s a changeable list

- Meaning that **we can change, add, and remove items in a list** after it has been created.

List can have Duplicates

- Since lists are indexed, **lists can have items with the same value.**

Getting the List Length

- To determine how many items a list has, use the **len()** function:

```
thislist = [“Rogie", “Wids", “Taints"]
print(len(thislist))
```

List Items in different Data Types

list1 = [“A", “B", “C"] *#String*

list2 = [1, 5, 7, 95, 420] *#Integeter*

list3 = [True, False, False] *#Boolean*

List can include different data types

- list = [“AbC", 34, True, 4055, “MALE"]

- “AbC” and “MALE” are **strings**
- 34 and 4055 are an **integers**
- True is **Boolean**

Data Type of a List

- From Python's perspective, lists are defined as objects with the data type 'list’:

type() function can be used to identify its data type or class.

```
myCharacters = [“Mar", “Lui", “Goo"]
print(type(myCharacters))
```

Output: <class 'list'>

The **list()** Constructor

- It is also possible to use the **list()** constructor when creating a new list.

```
scientists = list((“Einstein", “Schrodinger", “LeCun"))
#note it has double round-brackets because it's using a function that is a constructor.
print(scientists)
```

Python List Reminder

REMEMBER:

- A list is **ordered**, **changeable**, and **allows duplicate** members.
- The list is defined with [] square brackets.

Python Tuples

Python Tuples

- Used to store multiple items in a single variable.
- Used to store collections of data, the other 3 are List, Set, and Dictionary, all with different qualities and usage.
- A collection which **is ordered and unchangeable**.
- Written with **round brackets ()**.

Python Tuple Example

```
thistuple = ("apple", "banana", "cherry") #Round brackets are used.
print(thistuple)
```

Tuple Items

- **Ordered, unchangeable, but allow duplicate values**.
- Indexed, the first item has index [0], the second item has index [1] etc.

Tuple Length

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

```
divas = (“Nikki", “Sasha", “Trish")
print(len(divas))
```

Remember **print()** is a function in Python ☺

Create a Tuple With One Item

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

```
thistuple = ("apple",)
print(type(thistuple))
```

Output: <class 'tuple'>

```
#NOT a tuple
thistuple = ("apple") #no comma " , " found.
print(type(thistuple))
```

Output: <class 'str'>

As you can see, the output is NOT A TUPLE. Rather, it's a STRING!
BE CAREFUL!!

Tuple Items - Data Types

- Tuple items can be of any data type:

```
tuple1 = ("apple", “B", “CCC") #string
tuple2 = (1, 5, 7, 9, 3) #integer
tuple3 = (True, False, False) #boolean
```

Multiple Data Types in a Tuple

- A tuple can also contain different data types:

- tuple1 = **(**"abc", 34, True, 40, "male"**)** *#remember* ***TUPLES*** *uses* ***rounded brackets*** **()**

- **What are the existing data types in tuple1?**
- **What are the data types of each item?**

Data Type of a Tuple

- From Python's perspective, tuples are defined as objects with the data type 'tuple’:

type() function can be used to identify its data type or class.

```
thistuple = ("apple", "banana", "cherry“)
print(type(thistuple))
```

Output: <class tuple'>

The **tuple()** Constructor

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

```
thistuple = tuple(("apple", "banana", "cherry")) #note the double round-brackets are used for TUPLES.

print(thistuple)
```

Python Tuple Reminder

REMEMBER:

- A tuple is **ordered**, **unchangeable/immutable**, but **allows duplicate** members.
- The tuple is defined with () parentheses or round brackets.

Python Sets

Python Set

- Used to store multiple items in a single variable.
- Used to store collections of data, the other 3 are List, Tuple, and Dictionary, all with different qualities and usage.
- A collection which is **unordered, changeable*, and unindexed.**
- Sets are written with **curly brackets {}**.
- **Duplicates are Not Allowed in Sets**

- * Note: **Set items are unchangeable**, but **you can remove items and add new items**.
- Once a set is created, you cannot change its items, but you can remove items and add new items.

Creating a Python Set

```
stuff_set = {“Feet", “Keys", “Quasars"}

print(stuff_set)
```

Set Length

my_set = {“Shoes", “Road", “Electric"}

print(len(my_set))

What function was used to print the length of the set?

What is the length of the set, “my_set”?

Set Items - Data Types

- set1 = {"apple", "banana", "cherry"}
- set2 = {1, 5, 7, 9, 3}
- set3 = {True, False, False}

- A set can contain different data types

- mixedset = {"abc", 34, True, 40, "male"}

Set’s Data Type

From Python's perspective, sets are defined as objects with the data type 'set':

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

Output: <class 'set'>

The **set()** Constructor

- It is also possible to use the **set()** constructor to make a set.

```
thisset = set(("apple", "banana", "cherry"))
print(thisset)
```

Python Set Reminder

REMEMBER:

- A set is **unordered**, **changeable**, and **does not allow duplicate** members.
- The list is defined with {} curly braces.

Python Dictionaries

Dictionary

- Used to store data values in key:value pairs.
- A collection which is ordered *(see next slide)*, changeable and do not allow duplicates.
- Written with curly brackets **{}** and have **keys** and **values**.
- This is the same as the JSON format.

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

Are Dictionaries Ordered or Unordered?

- **As of Python version 3.7, dictionaries are ordered. In Python 3.6 and earlier, dictionaries are unordered.**
- If dictionaries are ordered, it means that the items have a defined order, and that order will not change.
- If Unordered, it means that the items does not have a defined order, you cannot refer to an item by using an index.

Dictionary Example

- Print the "brand" value of the dictionary
- An item in the dictionary can be selected by having the dictionary with square brackets **[]** with a specific item inside quotations **“item”.**

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

print(thisdict["brand"])
```

Dictionary Length

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

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

Dictionary Items - Data Types

- The values in dictionary items can be of any data type.

Example of a dictionary with String, int, boolean, and list data types:

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

Dictionary’s Data Type

- From Python's perspective, dictionaries are defined as objects with the data type 'dict’:

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

print(type(thisdict))
```

Output: <class 'dict’>

Reminder **type()** is a function that returns the data type ☺

Python Dictionary Reminder

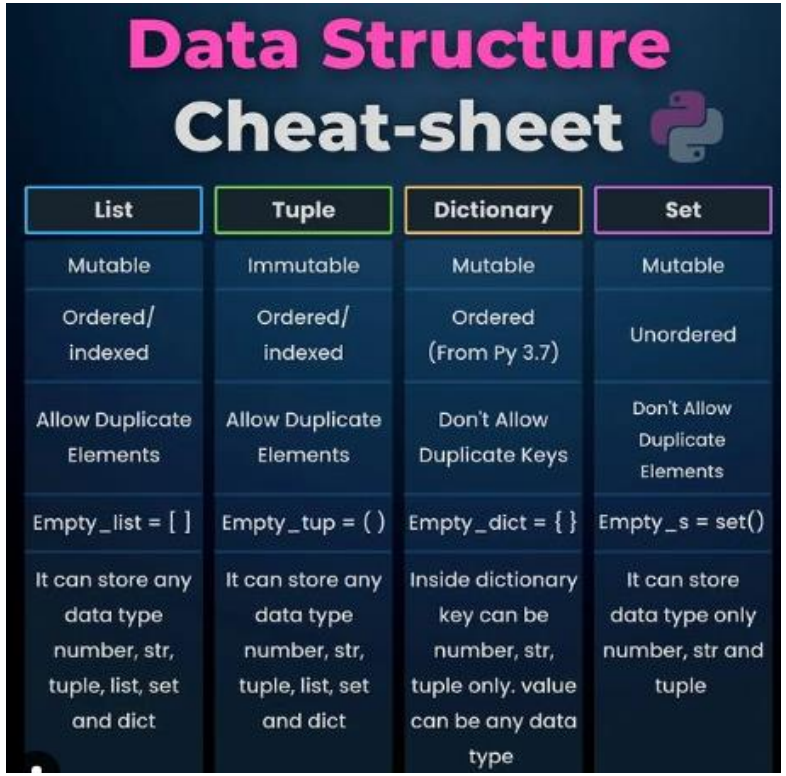
REMEMBER:

- A set is **ordered (since ≥Python 3.7)**, **changeable/mutable**, and **does not allow duplicate** members.
- The list is defined with {} curly braces but requires key and value pairs.

Data Types Cheat Sheet

Cheat Sheets

Data Structure	Ordered	Mutable	Constructor	Example
List	Yes	Yes	`[ ]` or `list()`	`[5.7, 4, 'yes', 5.7]`
Tuple	Yes	No	`( )` or `tuple()`	`(5.7, 4, 'yes', 5.7)`
Set	No	Yes	`{}`* or `set()`	`{5.7, 4, 'yes'}`
Dictionary	No	Yes**	`{ }` or `dict()`	`{'Jun': 75, 'Jul': 89}`



Cheatography

Python3 data structures Cheat Sheet
by desmovalvo via cheatography.com/56139/cs/14893/

Lists and Tuples

What are lists and tuples?
Ordered sequence of values indexed by integer numbers. Tuples are immutable.

How to initialize an empty list/tuple?
Lists: `myList = []`
Tuples: `myTuple = ()`

Size of list/tuple?
`len(myListOrTuple)`

Get element in position x of list/tuple?
`myListOrTuple[x]` -- if not found, throws `IndexError`

Is element "x" in list/tuple?
`"x" in myListOrTuple`

Index of element "X" of list/tuple?
`myListOrTuple.index("x")` -- If not found, throws a `ValueError` exception

Number of occurrences of "x" in list/tuple?
`myListOrTuple.count("x")`

Update an item of a list/tuple?
Lists: `myList[x] = "x"`
Tuples: tuples are immutable!

Remove element in position x of list/tuple?
Lists: `del myList[x]`
Tuples: tuples are immutable!

Remove element "x" of a list/tuple?
Lists: `myList.remove("x")`. Removes the first occurrence
Tuples: tuples are immutable!

Concatenate two lists or two tuples?
Lists: `myList1 + myList2`
Tuples: `myTuple1 + myTuple2`
Concatenating a List and a Tuple will produce a `TypeError` exception

Insert element in position x of a list/tuple?
Lists: `myList.insert(x, "value")`
Tuples: tuples are immutable!

Lists and Tuples (cont)

Append "x" to a list/tuple?
Lists: `myList.append("x")`
Tuples: tuples are immutable!

Convert a list/tuple to tuple/list
List to Tuple: `tuple(myList)`
Tuple to List: `list(myTuple)`

Slicing list/tuple
`myListOrTuple[ind1:ind2:step]` -- step is optional and may be negative

Sets

What is a set?
Unordered collection with **no duplicate** elements. Sets support mathematical operations like union, intersection, difference and simmetric difference.

Initialize an empty set
`mySet = set()`

Initialize a not empty set
`mySet = set(element1, element2...)` -- Note: strings are split into their chars (duplicates are deleted). To add strings, initialize with a Tuple/List

Add element "x" to the set
`mySet.add("x")`

Remove element "x" from a set
Method 1: `mySet.remove("x")` -- If "x" is not present, raises a `KeyErorr`
Method 2: `mySet.discard("x")` -- Removes the element, if present

Remove every element from the set
`mySet.clear()`

Check if "x" is in the set
`"x" in mySet`

Union of two sets
Method 1: `mySet1.union(mySet2)`
Method 2 `mySet1 | mySet2`

Sets (cont)

Intersection of two sets
Method 1: `mySet1.intersect(mySet2)`
Method 2 `mySet1 & mySet2`

Difference of two sets
Method 1: `mySet1.difference(mySet2)`
Method 2 `mySet1 - mySet2`

Simmetric difference of two sets
Method 1: `mySet1.symmetric_difference(mySet2)`
Method 2 `mySet1 ^ mySet2`

Size of the set
`len(mySet)`

Dictionaries

What is a dictionary?
Unordered set of key:value pairs . Members are indexed by keys (immutable objects)

Initialize an empty Dict
`myDict = {}`

Add an element with key "k" to the Dict
`myDict["k"] = value`

Update the element with key "k"
`myDict["k"] = newValue`

Get element with key "k"
`myDict["k"]` -- If the key is not present, a `KeyError` is raised

Check if the dictionary has key "k"
`"k" in myDict`

Get the list of keys
`myDict.keys()`

Get the size of the dictionary
`len(myDict)`

Delete element with key "k" from the dictionary
`del myDict["k"]`

Delete all the elements in the dictionary
`myDict.clear()`

By **desmovalvo**
cheatography.com/desmovalvo/

Published 2nd March, 2018.
Last updated 2nd March, 2018.
Page 1 of 1.

Data/Item Manipulation with Python

Item Manipulation in a List, Tuple, Set, and Dictionary

- Accessing
- Adding
- Changing
- Removing
- Looping
- Listing
- Sorting
- Copying
- Joining

Accessing List and Tuples

Accessing List and Tuple Items

- List and Tuple items are indexed, and you can access them by referring to the index number inside square brackets **[]**:

```
myitems = ["apple", "banana", "cherry"]
print(myitems[1])
```

Output: **banana**

Accessing List and Tuple Items

- 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.

- Note: The search will start at **index 2 (included)** and end at **index 5 (not included)**.
- Remember that the first item has index 0.

```
myitems = ["apple", "banana", "cherry", "orange", "kiwi", "melon", "mango"]

print(myitems[2:5])
```

Output: ['cherry', 'orange', 'kiwi']

Accessing List and Tuple Items

- This example **returns** the **items** from the **beginning** to, but **NOT INCLUDING**, **"kiwi"**:

0 1 2 3 4 5 6 ← Index values

myitems = ["apple", "banana", "cherry", "orange", "kiwi", "melon", "mango"]

print(myitems[:4])

Length = 7

Output: ['apple', 'banana', 'cherry', 'orange']

Tip: You can get the length of "myitems" by using **len()** to determine which part of it you want to access or manipulate.

print(len(myitems)) #this will output a 7, which if referring to index its 0-6

How will you access the first up to third item "cherry"?

If you want to start from the second item, simply put an index value of that item. If Starting from "banana", the use 1.

Therefore, having print(myitems[1:4])

Accessing List and Tuple Items

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

Looking at the example, what do you think is the output?

Clues:

The **first** value [**first**:second] **INCLUDES** that item.

Leaving it blank takes in all.

Accessing List and Tuple Items

- The items within the list and tuples can be accessed using the Negative Indexing method.

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

```
                    -3          -2            -1
myitems = ("apple", "banana", "cherry")

print(myitems[-1])
```

Output: **cherry**

Note: Anything >-3 will raise an error “IndexError: tuple index out of range”

Accessing List and Tuple Items

- You can also specify negative indexes if you want to start the search from the end of the list:

```
myitems = ["apple", "banana", "cherry", "orange", "kiwi", "melon", "mango"]

print(myitems[-4:-1])
```

What do you think is the output?

Clues:
First key value is included, and last key value indicates the last but is not included.

Tip:

You can take the length of “myitems” and invert them negatively. Once negative indexes are identified you can use them as basis.

Accessing List and Tuple Items

- Checking **items** within the list or tuple for sanity checking.
- Using an **if** statement is a practical method.

- To determine if a specified item is present **in** a list use the in keyword:

Check **if "apple"** is present in the list:

```
myitems = ["apple", "banana", "cherry"]
if "apple" in myitems :
        print("Yes, 'apple' is in the fruits list")
```

Output: Yes, 'apple' is in the fruits list

What if **"apple"** is removed or replaced? Will there be an error?
No, there will be no error. It will simply not output nothing. Unless an else statement is provided.

Accessing List and Tuple Items

- # example list
- fruits = ['apple', 'banana', 'cherry', 'date', 'elderberry']
- # access every second element in the list
- print(fruits[::2])
- # Output: ['apple', 'cherry', 'elderberry']
- # example tuple
- numbers = (1, 2, 3, 4, 5, 6, 7, 8, 9, 10)
- # access every third element in the tuple
- print(numbers[::3])
- # Output: (1, 4, 7, 10)

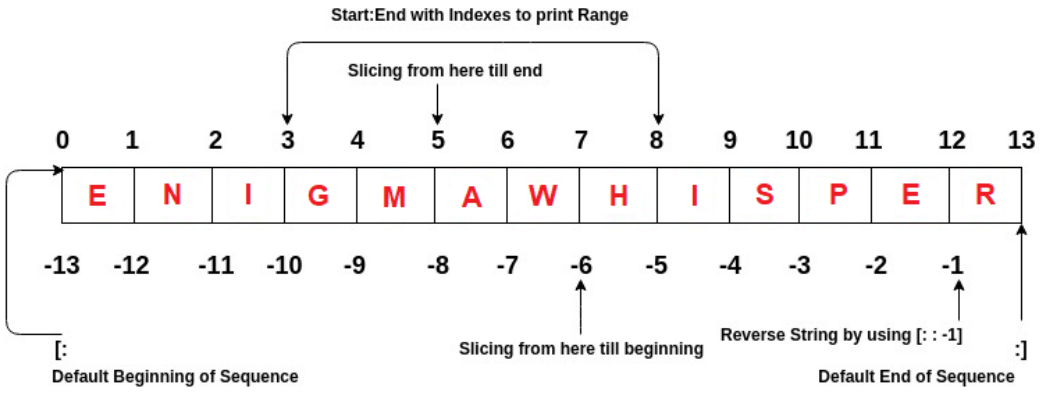
Explanation:

- The [start:end:step] syntax is used to access elements in a list or tuple.
- The first colon : indicates that we want to access elements from the start to the end of the list/tuple.
- The second colon : indicates that we want to access all elements by default.
- The third colon :step specifies the step value. In the above examples, the step value is 2 for the list and 3 for the tuple, meaning we want to access every second and third element respectively.

Python String

- Arrays of bytes representing Unicode characters.
- In simpler terms, a string is an immutable array of characters. Python does not have a character data type; a single character is simply a string with a length of 1.

- **Note: As strings are immutable, modifying a string will result in creating a new copy.**



Accessing Set Items

Accessing Set 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.

```
thisset = {"apple", "banana", "cherry"}

for x in thisset:
        print(x)
```

Output:

banana

cherry

apple

Accessing Set items

You can also check if "banana" is present **in** the set through this method:

```
thisset = {"apple", "banana", "cherry"}

print("banana" in thisset)
```

Accessing Dictionary Items

Accessing Dictionary Items

- You can access the items of a dictionary by referring to its **key name**, inside square brackets **[]**:

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

x = thisdict["model"]
```

Accessing Dictionary Items

- Another method to access the key values can be done by using the **get()** method or function will return a list of all the keys in the dictionary.

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

x = thisdict.get("model")
```

Output: **Mustang**

Accessing Dictionary Items

- To access the keys, using the **keys()** method or function will return a list of all the keys in the dictionary.

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

x = thisdict.keys()
```

Output: dict_keys(['brand', 'model', 'year'])

Accessing Dictionary Items

- The list of the keys is a view of the dictionary, meaning that any changes done to the dictionary will be reflected in the keys list.
- Add a new item to the original dictionary, and see that the keys list gets updated as well:

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

x = car.keys()
print(x) #before the change
```

Output: dict_keys(['brand', 'model', 'year'])

```
car["color"] = "white"
print(x) #after the change
```

Output: dict_keys(['brand', 'model', 'year', 'color'])

Changing items in a List

Changing Items in a List

- To **change** the value of a **specific item**, refer to the **index number** using square brackets **[]**:

```
thislist = ["apple", "banana", "cherry"]
thislist[1] = "blackcurrant“ #this will replace banana
print(thislist) #the output should have no banana instead, blackcurrant.
```

Output: ['apple', **'blackcurrant'**, 'cherry']

Changing Items in a List

Change the values **"banana"** and **"cherry"** with the item values **"blackcurrant"** and **"watermelon"**:

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

Output: ['apple', 'blackcurrant', 'watermelon', 'orange', 'kiwi', 'mango']

Changing Items in a List

- Change the **second value** by replacing it with **two new values**:

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

Output: ['apple', 'blackcurrant', 'watermelon', 'cherry']

Changing Items in a List

Change the **second and third** value by **replacing** it with **one value**:

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

Output: ['apple', 'watermelon’]

If [1:2] then it will output ['apple', 'watermelon', 'cherry’] #replaces “banana” cause its 2

If [2:1] then it will output ['apple', 'banana', 'watermelon', 'cherry’] #adds watermelon in 3

If [2:2] then it will output ['apple', 'banana', 'watermelon', 'cherry’] # adds watermelon in 3

If [2:3] then it will output ['apple', 'banana', 'watermelon’] #replaces “cherry cause its 3

If [3:3] then it will output ['apple', 'banana', 'cherry', 'watermelon’] #adds watermelon at the end

Even if you increase **y** in [**3**:**y**], it will always just add **watermelon** at the end.

If **x** in [**x**:y] gets a **value** > the current index items (which is 3) being added it will just place the **new value** at the end.

Even If you try [25:3] then it will output ['apple', 'banana', 'watermelon’] #replaces “cherry cause its 3

#THIS IS NONSENSE AND ILLOGICAL TO DO

Changing Items in a List

- Simple technique

- [x:y] read this as from x to y
- Replace the values y^n ..... y^{n-1} within the list.
- If x is 1 and y is 3, we have [1:3]
- Take values from 1 to 3
- Replace values 3, 2, EXCEPT 1. Include 1.

Changing Items in a List

- Technique application

Example:

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

Solution:

If x is 1 and y is 2, we have [1:2]

Our list is ["apple", "banana", "cherry"]

Replace value 2 which is “banana” with ["blackcurrant", "watermelon"], EXCEPT 1. Include 1. Value 3 is untouched.

Answer: ["apple", "blackcurrant", "watermelon", "cherry"]

Changing Items in a List

- Try the technique!

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

Output: ['apple', 'blackcurrant', 'watermelon', 'orange', 'kiwi', 'mango’]

The 3 and 2 values are replaced but 1 value kept and the rest in indexes >3 intact.

Changing Items in a List

- Insert "watermelon" as the third item using the **insert()** function:

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

Output: ['apple', 'banana', **'watermelon'**, 'cherry']

Changing/Updating items in a Tuple

Updating items in a Tuple

- **Once a tuple is created, you cannot change its items! However, there is work around. Therefore, we refer to it as UPDATING rather than CHANGING.**

Updating values in Tuples

- Once a tuple is created, you **cannot change** its **values**.
- Tuples are **unchangeable**, or **immutable** as it also is called.
- But there is a workaround. You can **convert** the tuple into a **list**, **change** the **list**, and **convert** the **list** back into a **tuple**.

Tuple → Convert to List → Edit the List → Covert back to Tuple

```
x = ("apple", "banana", "cherry")
y = list(x)
y[1] = "kiwi"
x = tuple(y)

print(x)
```

What is the output?

Updating values in Tuples

- Removing items in tuples is possible through a workaround.

Convert the tuple into a list by passing the tuple into the **list()** function. From there, remove **"apple“** using the **remove()** function, and convert it back into a tuple using the **tuple()** function:

thistuple = (**"apple"**, "banana", "cherry")

y = **list(**thistuple**)** *#assigns the list version of the tuple to another to prevent overriding the “thistuple” tuple.*

y.**remove("apple")** *#removes the “apple” item in the list y. Remember, you **can remove** items in a **list not a tuple**.*

thistuple = **tuple(**y**)** *#re-assigns the updated tuple converted list to variable “thistuple”*

Changing Items in a Dictionary

Changing Items in a Dictionary

- You can **change** the **value** of a **specific item** by **referring** to its **key name**:

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

thisdict["year"] = 2018

print(thisdict)
```

Output: {'brand': 'Ford', 'model': 'Mustang', 'year': **2018**}

Adding Items in a List

Adding Items in a List

- To **add** an **item** to the end of the list, use the **append()** method:

```
thislist = ["apple", "banana", "cherry"]
thislist.append("orange")

print(thislist)
```

Output: ['apple', 'banana', 'cherry', 'orange']

Removing items in a List

- The **remove()** method removes the specified **item**.

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

Adding Items in a Tuple

Adding Items in a Tuple

- Again, tuples are **unchangeable**, or **immutable** as it also is called.

- **What can you do to add entries in a Tuple? Is it possible?**
- **Yes, there is a workaround!**

Adding Items in a Set

- Once a set is created, you **cannot change** its **items**, **BUT** you **CAN ADD new items**.

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

```
thisset = {"apple", "banana", "cherry"}

thisset.add("orange")

print(thisset)
```

What is the output?

Adding Items in a Dictionary

Adding Items in a Dictionary

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

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

thisdict["color"] = "red“

print(thisdict)
```

Output: {'brand': 'Ford', 'model': 'Mustang', 'year': 1964, **'color': 'red'**}

Updating Items in a Dictionary

- **Updating** an **item** to the dictionary is done by using a **new index key** and using the .update() method:

```
#Combining directories using update method.
# Initialize the first dictionary
x= {'name': 'Link', 'race': 'Hylian', 'weapon': 'Master
Sword'}

# Initialize the second dictionary
y= {'item': 'Bombs', 'item2': 'Bow'}

# Update the first dictionary with the second dictionary
x.update(y)

# Print the updated dictionary
print(x)
```

{'name': 'Link', 'race': 'Hylian', 'weapon': 'Master Sword', 'item': 'Bombs', 'item2': 'Bow'}

List Methods

- .remove()
- .count()
- .sort()
- len()
- sorted()
- .insert()
- .pop()

Tuple Methods

- .count()
- index()
- len()
- sorted()

Set Methods

- .add()
- .remove()
- .discard()
- .pop()
- .clear()
- .copy()
- .difference()
- .union()
- .intersection()
- .symmetric_difference()
- .issubset()
- .issuperset()
- len()

Dictionary Methods

- .clear()
- .copy()
- .get()
- .items()
- .keys()
- .values()
- .pop()
- .popItem()
- .update()
- .setdefault()
- .pop()
- .fromkeys()
- dict()

Continuation Next Meeting

Working with Nested Structures

Nested List

```
MyList = [[22, 14, 16], ["Joe", "Sam", "Abel"], [True, False, True]]

print(MyList[0][1])

Output: 14

print(MyList[1])

Output: ['Joe', 'Sam', 'Abel’]

print(MyList[1][2])

Output: Abel
```

Nested List

```
MyList = [[22, 14, 16], ["Joe", "Sam", "Abel"], [True, False,
True]]

# Remove the entire sublist ["Joe", "Sam", "Abel"]
MyList.remove(["Joe", "Sam", "Abel"])

# Print the modified list
print(MyList)
```

Output: [[22, 14, 16], [True, False, True]]

Nested List

```
MyList = [[22, 14, 16], ["Joe", "Sam", "Abel"], [True, False,
True]]

# Remove the sublist at index 1
del MyList[1]

# Print the modified list
print(MyList)
```

Output: [[22, 14, 16], [True, False, True]]

Using del and remove with list

- del Statement: The del statement is a general-purpose statement to remove an element or slice from a list. It can be used to remove elements by index, i.e., del myList[index]. It can also be used to remove entire slices from a list, like del myList[start:stop]. The del statement modifies the original list in place.

- remove() Method: The remove() method is specifically designed to remove the first occurrence of a specified value from the list. It searches for the value and removes it from the list. If the value is not found, it raises a ValueError.

del with nested list via negative indexing

```
MyList = [[22, 14, 16], ["Joe", "Sam", "Abel"], [True, False,
True]]

# Remove the sublist at index 1
del MyList[:-3]

# Print the modified list
print(MyList)
```

Output: [[22, 14, 16], ['Joe', 'Sam', 'Abel'], [True, False, True]]

```
del MyList[:-2]
```

Output: [['Joe', 'Sam', 'Abel'], [True, False, True]]

```
Del MyList[:-1]
```

Output: [[True, False, True]]

Example with del in list

By using a range with the del statement, we were able to remove the items between the index number of 1 (inclusive), and the index number of 4 (exclusive), leaving us with a list of 3 items following the removal of 3 items.

The del statement allows us to remove specific items from the list data type.

```
sea_creatures =['shark', 'octopus', 'blobfish', 'mantis shrimp', 'anemone', 'yeti crab']

del sea_creatures[1:4]
print(sea_creatures)
```

Output

```
['shark', 'anemone', 'yeti crab']
```

Enumerate and Zip in Python

Enumerate

- The enumerate() function adds a counter to an iterable and returns it as an enumerate object (iterator with index and the value).
- enumerate() function takes two arguments:
- iterable - a sequence, an iterator, or objects that support iteration.
- start (optional) - enumerate() starts counting from this number. If start is omitted, 0 is taken as start.

```
enumerate(iterable, start=0)
```

Enumerate Return Values

- The enumerate() function adds a counter to an iterable and returns it. The returned object is an enumerate object.

- An enumerate object is an iterator that produces a sequence of tuples, each containing an index and the value from the iterable.

- We can convert enumerate objects to lists and tuples using list() and tuple() functions, respectively.

Enumerate Example

```
languages = ['Python', 'Java', 'JavaScript']

# enumerate the list
enumerated_languages = enumerate(languages)

# convert enumerate object to list
print(list(enumerated_languages))

# Output: [(0, 'Python'), (1, 'Java'), (2, 'JavaScript')]
```

Zip

- The zip() function takes iterables (can be zero or more), aggregates them in a tuple, and returns it.
- Zip parameter iterables can be built-in iterables (like: list, string, dict), or user-defined iterables

```
zip(*iterables)
```

Zip Return Values

- The zip() function returns an iterator of tuples based on the iterable objects.

- If we do not pass any parameter, zip() returns an empty iterator
- If a single iterable is passed, zip() returns an iterator of tuples with each tuple having only one element.
- If multiple iterables are passed, zip() returns an iterator of tuples with each tuple having elements from all the iterables.

Zip Example

```
languages = ['Java', 'Python', 'JavaScript']
versions = [14, 3, 6]

result = zip(languages, versions)
print(list(result))

# Output: [('Java', 14), ('Python', 3), ('JavaScript', 6)]
```

Merging a List

Ways to merge multiple lists

- Using the + operator.
- Using the extend() method.
- Using the += operator.

- The append() method works differently.

Using the + operator

- list1 = [1, 2, 3]
- list2 = [4, 5, 6]
- merged_list = list1 + list2
- print(merged_list)

- Output: [1, 2, 3, 4, 5, 6]

Using the extend() method

- list1 = [1, 2, 3]
- list2 = [4, 5, 6]
- list1.extend(list2)
- print(list1)

- Output: [1, 2, 3, 4, 5, 6]

Using the += operator.

- list1 = [1, 2, 3]
- list2 = [4, 5, 6]
- list1 += list2
- print(list1)

- Output: [1, 2, 3, 4, 5, 6]

Using the append() method.

- list1 = [1, 2, 3]
- list2 = [4, 5, 6]
- list1.append(list2)
- print(list1)

- [1, 2, 3, [4, 5, 6]]

Set Union and Intersection

Set Union

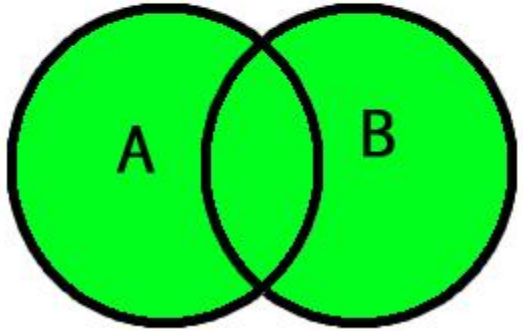
- x = {"apple", "banana", "cherry"}
- y = {"google", "microsoft", "apple"}

- z = x.union(y)

- print(z)

- Output: {'google', 'banana', 'apple', 'cherry', 'microsoft'}

A = {1,2,3}
B = {3,4,5}
AUB = {1,2,3,4,5}



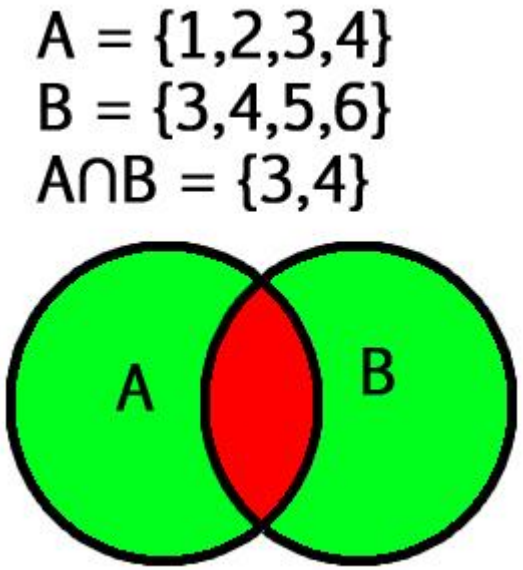
Set Intersection

- x = {"apple", "banana", "cherry"}
- y = {"google", "microsoft", "apple"}

- z = x.intersection(y)

- print(z)

- Output: {'apple'}



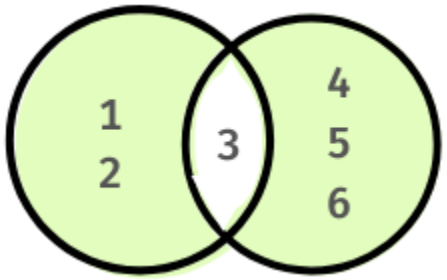
Symmetric Difference

```
x = {"apple", "banana", "cherry"}
y = {"google", "microsoft", "apple"}

z = x.symmetric_difference(y)

print(z)
```

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



$$A = \{1, 2, 3\},\ B = \{3, 4, 5, 6\}$$

$$A \,\Delta\, B = \{1, 2, 4, 5, 6\}$$

Popping

Popping an item in a list

- fruits = ['apple', 'banana', 'cherry']
- fruits.pop(1)
- Output: ['apple', 'cherry']

Popping in a Set

```
my_set = {1, 2, 3}
print(my_set)
my_set.pop()
print(my_set)
my_set2 = {"a", "b", "c"}a
print(my_set2)
my_set2.pop()
popped_element = my_set2.pop()
print("Popped element:", popped_element)
print(my_set2)
```

Dictionary Tricks

Getting Keys

- my_dict = {'a': 1, 'b': 2, 'c': 3}
- keys = my_dict.keys()

- print("Keys:", keys)

- Keys: dict_keys(['a', 'b', 'c'])

Getting Values

- my_dict = {'a': 1, 'b': 2, 'c': 3}
- values = my_dict.values()
- print("Values:", values)

- Values: dict_values([1, 2, 3])

Getting Items

- my_dict = {'a': 1, 'b': 2, 'c': 3}
- items = my_dict.items()
- print("Items:", items)

- Items: dict_items([('a', 1), ('b', 2), ('c', 3)])

Accessing Nested Dictionaries

- nested_dict = {'outer': {'inner': 42}}

- # Accessing value inside nested dictionary
- inner_value = nested_dict['outer']['inner']
- print("Inner value:", inner_value)

- Output: Inner value: 42

Safe Access

- my_dict = {'a': 1, 'b': 2}

- # Safe way to get a value with a default if key doesn't exist
- value = my_dict.get('c', 'Key not found')
- print(value)

- Output: Key not found

- Using this: value = my_dict.get('c’)
- Will output: None

Converting Keys to List

```
# Sample dictionary
my_dict = {'a': 1, 'b': 2, 'c': 3, 'd': 4}

# Convert keys into a list
keys_list = list(my_dict.keys())

# Select a specific key index
index = 2  # selecting the third key 'c'

# Assign the selected key to a variable
selected_key = keys_list[index]

# Print the selected key and its corresponding value
print("Selected Key:", selected_key)
print("Corresponding Value:", my_dict[selected_key])
```

```
Selected Key: c
Corresponding Value: 3
```

Activity

- Create different collections using the different data structures presented.
- Try the methods available for each.
- Try to access certain values which you can try printing or use as an input.
- Try combining different data structures together.