DATA SCIENCE 101: PYTHON COLLECTIONS

AGENDA

- Lab exercise 1 answer sharing
- Recap of last lesson
- Collections
  - More List Functions
  - Tuples
  - Dictionaries
- Decision structures & Boolean logic

RECAP

- Python as a language
- Variables
- Data types
  - Numeric
    - Integer
    - Float
  - String
  - Collections
    - List
    - Dictionary
- Type Conversion
- String Slicing
- List Manipulation
- Debugging

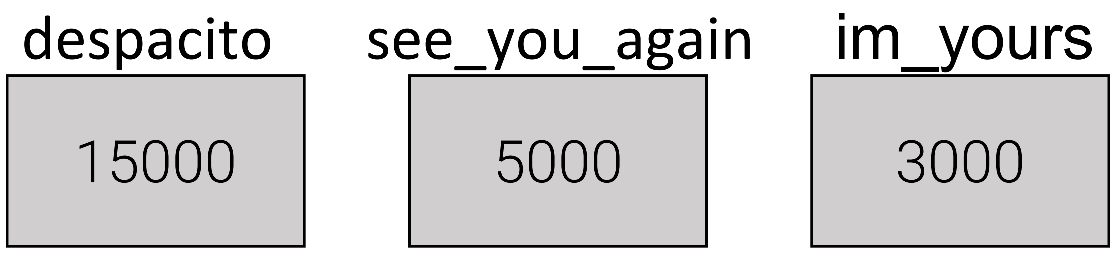
COLLECTIONS

- Why do we need collections?
- What are collections?
- Commonly used collections: Lists, Tuples & Dictionary

WHY DO WE NEED COLLECTIONS?

- There are times where we need to store multiple values in a variable for easier access

```
despacito = 15000
see_you_again = 5000
im_yours = 3000
```



Imagine YouTube have millions of videos

WHAT IS A COLLECTION?

- A data type that allows you to store many values in a single “variable”
- A collection is useful because we can carry many values around in one convenient package
- The following are lists, which is one type of collection:

```
video_views = [15000,5000,3000]
video_titles = ['despacito', 'see_you_again', 'im_yours']
```

TYPES OF COLLECTIONS

- There are three main type of commonly used Python collections:
    - Lists
    - Tuples
    - Dictionary

LISTS

- What is a list
- List manipulation
- List slicing
- Number of items in a list
- Checking item in a list
- Different list structures
- Remove vs. Delete
- Sorting

WHAT IS A LIST

- List is a collections data type that contains a collection of sequential related data items
- Think of list as a “book” that holds a series of papers (variables)
- Best-practice: List should contain data of the same type

```
video_titles = ['despacito', 'see you again', 'im yours']
video_views = [15000, 5000, 3000]
```

CREATING A LIST

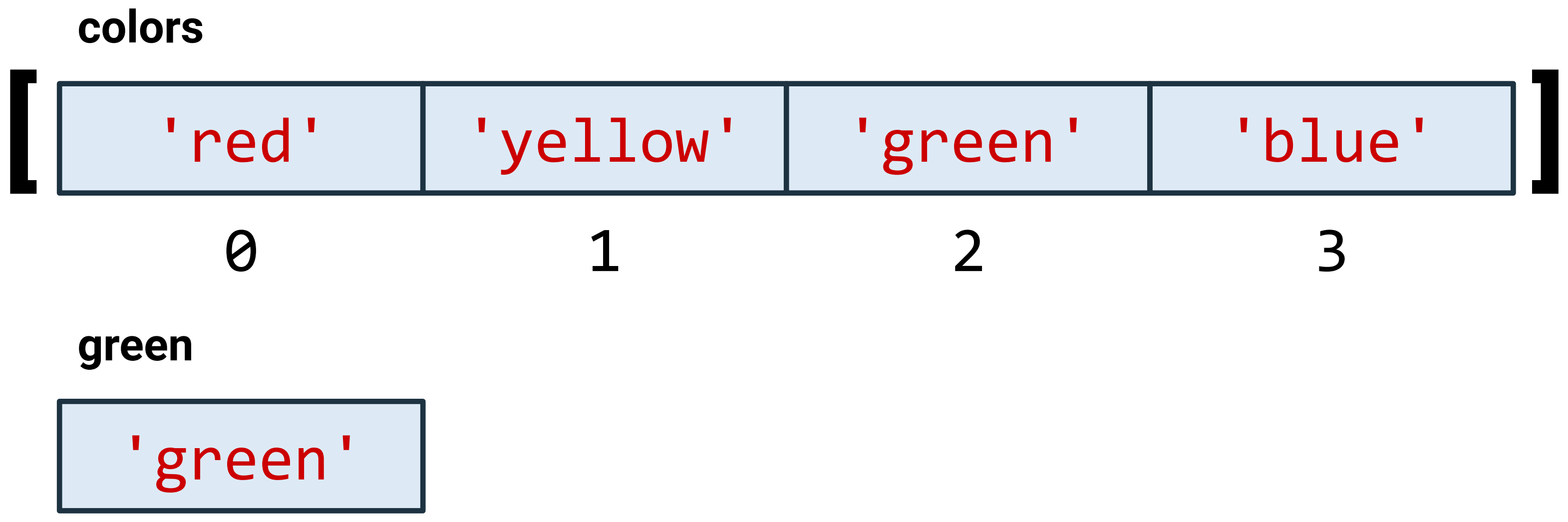
- List constants are surrounded by square brackets and the elements in the list are separated by commas.
- The lists will be created with the stipulated order specified in between in the square brackets
- List can contain mixed data type even another lists
- A list can be empty
- The values of a list can be printed using the print( ) function

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

y = ['red', 'blue', 'green']

mixture = [2.5, 'blue', ['red']]

an_empty_list = [ ]
```

EXTRACTING A VALUE FROM A LIST

- Just like in Strings, the items in Lists are arranged in an index based system. Hence we can access a data via the index.

```
colors = [red, 'yellow', 'green', 'blue']
green = colors[2]
print(green)
```



CHANGING A VALUE IN A LIST

- Strings are “immutable” - we cannot change the contents of a string - we must make a new string to make any chances
- Lists are “mutable” - we can change an element of a list using the index operator

```
x = 'despacito'
x[0] = 'B'
```

```
      1 x = 'despacito'
----> 2 x[0] = 'B'

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

```
x = ['red', 'blue', 'green']
x[0] = 'yellow'
print(x)
```

```
['yellow', 'blue', 'green']
```

ADDING VALUE TO A LIST

- The following code is how you will add data to the end of a created list

```
video_titles = ['despacito', 'see you again', 'im yours']
video_views = [15000, 5000, 3000]

print(video_titles)
video_titles.append('space jam')
print(video_titles)

print(video_views)
video_views.append(7000)
print(video_views)
```

```
['despacito', 'see you again', 'im
yours']
```

```
['despacito', 'see you again', 'im
yours', 'space jam']
```

ADDING VALUE TO A LIST

- The following code is how you will add data to the end of a created list

```
video_titles = ['despacito', 'see you again', 'im yours']
video_views = [15000, 5000, 3000]

print(video_titles)
video_titles.append('space jam')
print(video_titles)

print(video_views)
video_views.append(7000)
print(video_views)
```

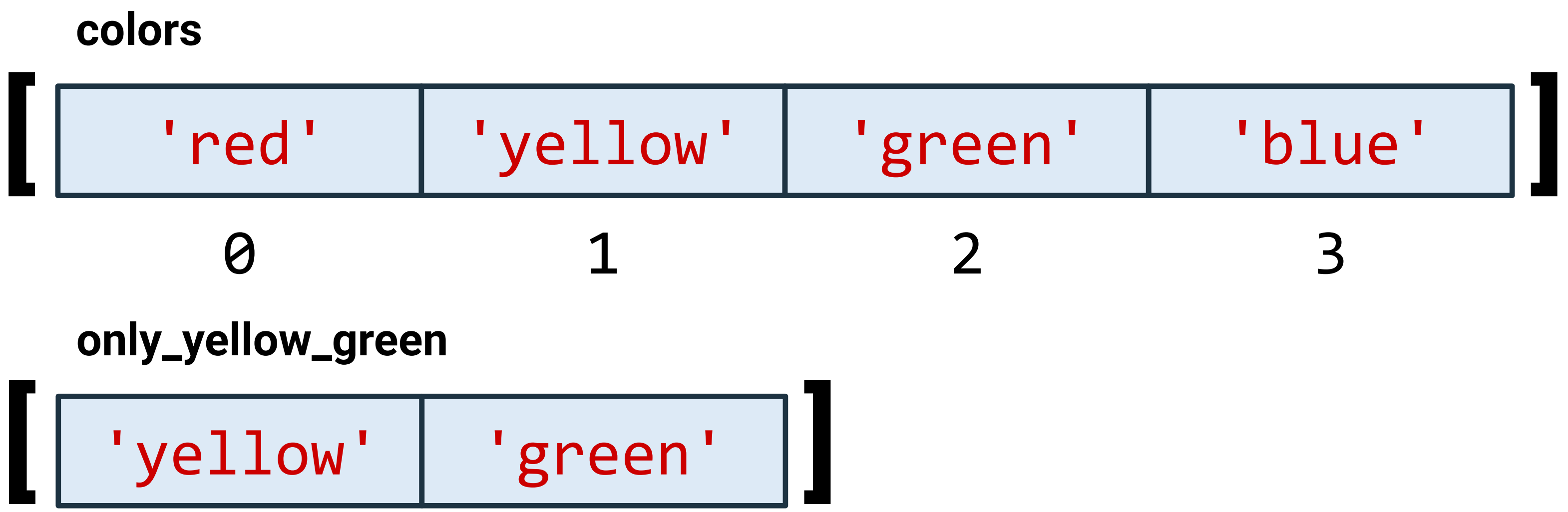
```
[15000, 5000, 3000]
```

```
[15000, 5000, 3000, 7000]
```

SLICING A LIST

- Just like Strings, Lists can be sliced according to the specified indexes. Sliced lists gives you a list.

```
colors = ['red', 'yellow', 'green', 'blue']
only_yellow_green = colors[1:3]
print(only_yellow_green)
```



CHECKING NUMBER OF ITEMS IN A LIST

- A list can contain a very large number of items and to find out how many items are there in a list, use the `len( )` function.

```
colors = ['red', 'yellow', 'green', 'blue']
total_number_of_items = len(colors)
print(total_number_of_items)
```

```
Output: 4
```

CHECKING FOR ITEM IN A LIST

- A list can contain a very large number of items and to check if a value in a list, use the **in** keyword.

```
colors = ['red', 'yellow', 'green', 'blue']
green_exists = 'green' in colors
print(green_exists)
```

```
Output: True
```

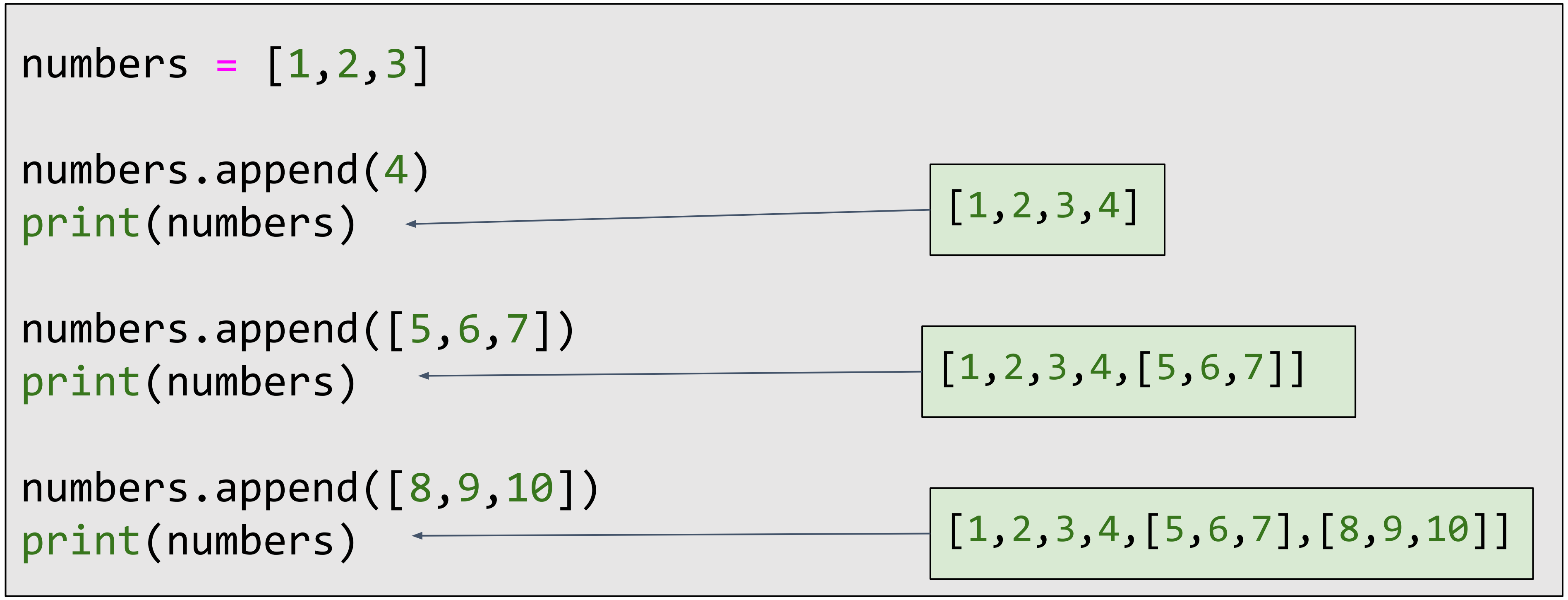
DIFFERENT STRUCTURES OF LISTS

- During the course, we will see how lists can be used in multiple ways to store the data we need for processing.
- Lists are used extensively because of its ability to hold many data types in a single variable.
- Lists can be used in the following structures:
    - List of Integers
    - List of Strings
    - List of Lists
    - List of Dictionaries

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

many_letters = ['a', 'b', 'c']

many_rows = [
   ['this is row 1'],
   ['this is row 2'],
]

many_row_details = [
   {'row': 1},
   {'row': 2}
]
```

APPEND

- The `.append()` adds one data to the back of the list.
- `.append()` retains the **integrity** of the data type when adding to a list



```
numbers = [1,2,3]

numbers.append(4)
print(numbers)                    [1,2,3,4]

numbers.append([5,6,7])
print(numbers)                    [1,2,3,4,[5,6,7]]

numbers.append([8,9,10])
print(numbers)                    [1,2,3,4,[5,6,7],[8,9,10]]
```

REMOVE

- The `.remove()` function removes the **first instance** of a given value, from index 0 to the last index.

```
video_views = [15000,5000,3000]
video_titles = ['despacito', 'see_you_again', 'im_yours']

video_views.remove(5000)
print(video_views)

video_titles.remove('see_you_again')
print(video_titles)
```

```
[15000,3000]
```

```
['despacito', 'im_yours']
```

DELETE

- The **del** function removes an item from a list based on the given **index**.

```
video_views = [15000,5000,3000]
video_titles = ['despacito', 'see_you_again', 'im_yours']

del video_views[1]
print(video_views)

del video_titles[1]
print(video_views)
```

```
[15000,3000]
```

```
['despacito', 'im_yours']
```

SORT

- The `.sort()` function helps us to sort a list in **ascending** order, by default.

```
numbers = [134,23,35,78]

numbers.sort()
print(numbers)
```

```
[23,35,78,134]
```

SORT IN DESCENDING ORDER

- To sort in descending order, we need to overwrite the default value of the `.sort()` function
- To overwrite the default value, use **`reverse=True`** within the brackets

```
numbers = [134,23,35,78]

numbers.sort(reverse=True)
print(numbers)
```

```
[134,78,35,23]
```

IN-CLASS PRACTICE: LISTS FUNCTIONS

- Complete the given exercises in the in-class notebook!

OTHER LIST METHODS

- You can make use of the method “dir()” to find out the methods inherent to Python’s list.

```
>>> x = list()
>>> type(x)
<type ‘list’>

>>> dir(x)
[‘append’, ‘count’, ‘extend’, ‘index’, ‘insert’, ‘pop’, ‘remove’,
‘reverse’, ‘sort’]
```

Source: http://docs.python.org/tutorial/datastructures.html

LIST SUMMARY

We learnt the following about lists:

- List is a reference variables that can hold multiple values
- Rules of lists (Initiating, accessing, adding, changing)
- Lists can be sliced
- Checking if something is in a list
- Length of list
- Range and list

TUPLES

- What are tuples?
- Things you cannot do with tuples
- Methods inherent to tuples
- When to use tuples over lists?

WHAT ARE TUPLES?

- Tuple functions the same as the list, except that the contents are immutable.

List

```
video_views = [15000,5000,3000]
video_views[0] = 16000
```

Tuple

```
video_views = (15000,5000,3000)
video_views[0] = 16000
```

```
      1 video_views = (15000,5000,3000)
----> 2 video_views[0] = 16000

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

THINGS YOU CANNOT DO WITH TUPLE

- Tuple functions the same as the list, except that the contents are immutable

```
       x = (3, 2, 1)
--->   x.sort()
AttributeError: 'tuple' object has no attribute 'sort'

--->   x.append(5)
AttributeError: 'tuple' object has no attribute 'append'

--->   x.reverse()
AttributeError:  'tuple' object has no attribute 'reverse'
```

METHODS INHERENT TO TUPLE

- Tuple functions the same as the list, except that the contents are immutable.

```
>>> x = list()
>>> dir(x)
['append', 'count', 'extend', 'index', 'insert', 'pop', 'remove', 'reverse', 'sort']

>>> t = tuple()
>>> dir(t)
['count', 'index']
```

WHEN TO USE TUPLES OVER LISTS?

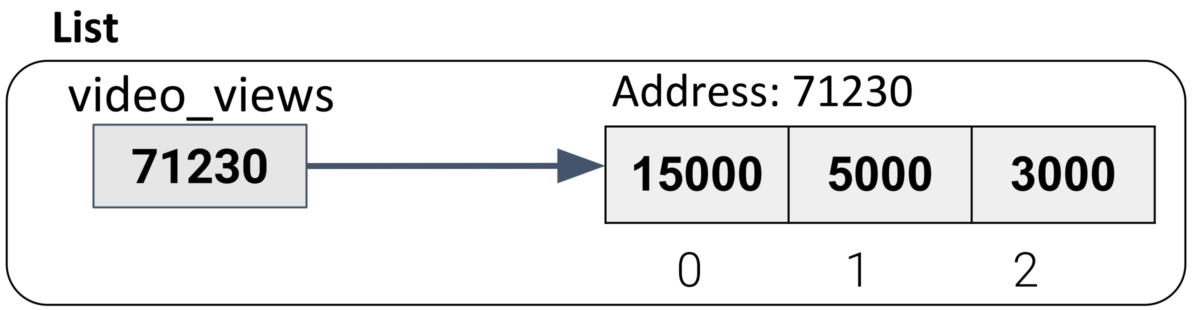
- Since Python does not have to build tuple structures to be modifiable, they are simpler and more efficient in terms of memory use and performance than lists
- So in our program when we are making "temporary variables" we prefer lists over tuples

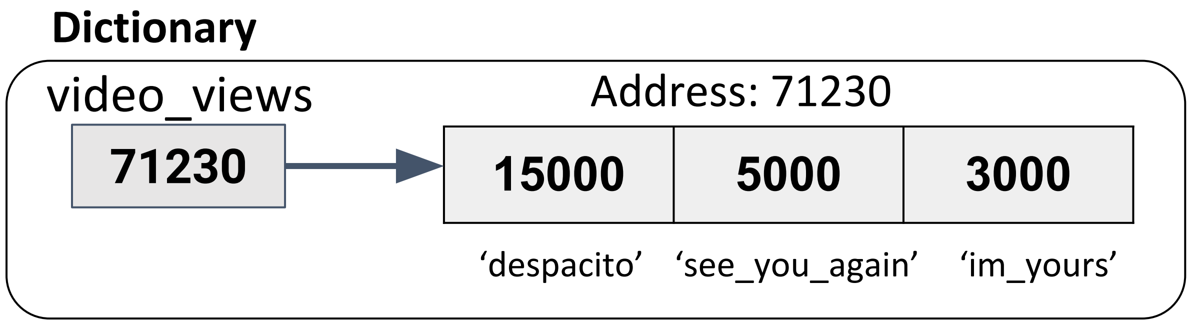
DICTIONARY

- Dictionary vs list
- What is a dictionary?
- Properties of dictionary
- Functions of dictionary

DICTIONARY VS LIST

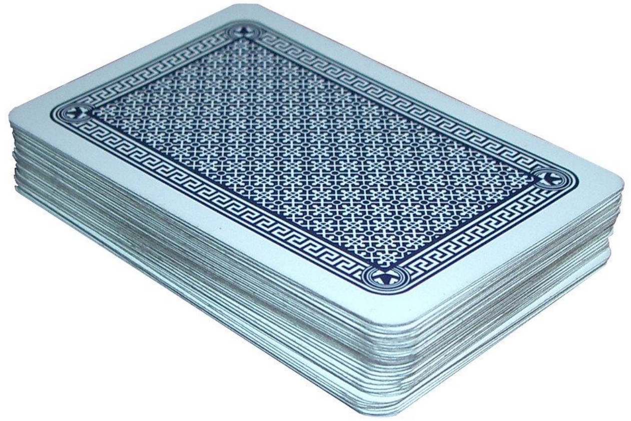
- Just like lists and tuples, dictionary is part of the collection family too
- However, there is one key difference in the way data is stored in dictionary versus lists
- Dictionaries are organized in key-value pairs
- Retrieval through lookups just like real dictionaries!





DICTIONARY VS LIST

- Think of lists as:
  - A linear collection of values that needs to stay in order

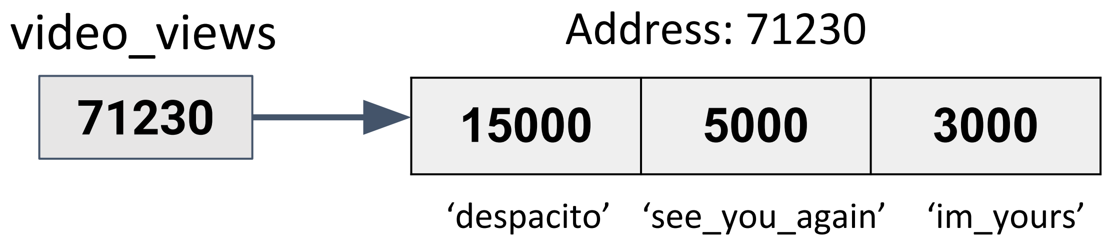


- Think of dictionaries as:
  - A “bag” of values, each with its own label

WHAT IS A DICTIONARY?

- A collection of items where each item has a label and is identified by the label
- Gaining access to that item requires us to ‘call’ its label

```
video_views = {
    'despacito': 15000,
    'see_you_again': 5000,
    'im_yours': 3000
}
print(video_views['despacito'])
```



PROPERTIES OF DICTIONARY

- Dictionaries are Python’s most powerful data collection
- Dictionaries allow us to do fast database-like operations in Python
- Dictionaries have different names in different languages
  - Associative Arrays - Perl / Php
  - Properties or Map or HashMap - Java
  - Property Bag - C# / .Net

FUNCTIONS OF DICTIONARY

- Explore the following functions:
    - Creating a new key-value pair
    - Updating key-value pair
    - Deleting a key-value pair
    - Retrieving a value
    - Checking if a key exists within a dictionary

CREATE NEW KEY-VALUE PAIR

- The following code is how you will **create** new key-value pair into a dictionary

```
video_views = {}

video_views['despacito'] = 15000
print(video_views)

video_views['see_you_again'] = 5000
print(video_views)

video_views['im_yours'] = 3000
print(video_views)
```

```
{
    'despacito':15000
}
```

```
{
    'despacito': 15000,
    'see_you_again': 5000
}
```

```
{
    'despacito': 15000,
    'see_you_again': 5000,
    'im_yours': 3000,
}
```

UPDATE KEY-VALUE PAIR

- The following code is how you will **update** the data in a dictionary

```
video_views = {
    'despacito': 15000,
    'see_you_again': 5000,
    'im_yours': 3000
}

video_views['despacito'] = 25000
print(video_views)

view_views['see_you_again'] = 10000
print(video_views)
```

```
{
    'despacito': 25000,
    'see_you_again': 5000,
    'im_yours': 3000,
}
```

UPDATE KEY-VALUE PAIR

- The following code is how you will **update** the data in a dictionary

```
video_views = {
    'despacito': 15000,
    'see_you_again': 5000,
    'im_yours': 3000
}

video_views['despacito'] = 25000
print(video_views)

view_views['see_you_again'] = 10000
print(video_views)
```

```
{
    'despacito': 25000,
    'see_you_again': 10000,
    'im_yours': 3000,
}
```

DELETING KEY-VALUE PAIR

- The following code is how you will delete a key-value pair in a dictionary

```
video_views = {
    'despacito': 15000,
    'see_you_again': 5000,
    'im_yours': 3000
}

del video_views['despacito']
print(video_views)

del video_views['see_you_again']
print(video_views)
```

```
{
    'see_you_again': 10000,
    'im_yours': 3000,
}
```

```
{
    'im_yours': 3000,
}
```

RETRIEVING A VALUE

- The following code is how you will retrieve a value from a dictionary

```
video_views = {
    'despacito': 15000,
    'see_you_again': 5000,
    'im_yours': 3000
}
despacito_views = video_views['despacito']
print(despacito_views)
```

```
15000
```

GET ALL KEYS

- To get all the keys within a dictionary, use the `.keys()` function to retrieve a **list** of all the keys

```
video_views = {
    'despacito': 15000,
    'see_you_again': 5000,
    'im_yours': 3000
}
all_titles = video_views.keys()
print(all_titles)
```

```
[
    'despacito',
    'see_you_again',
    'im_yours'
]
```

GET ALL KEY-VALUE PAIRS

- To get all key-value pairs, use the `.items()` function to get a **list of tuples**, where each tuple is a key-value.
- At index 0 of the tuple is the key; index 1 is the value.

```
video_views = {
    'despacito': 15000,
    'see_you_again': 5000,
    'im_yours': 3000
}
all_key_value_pairs = view_views.items()
print(all_key_value_pairs)
```

```
[
    (
        'despacito',
        15000
    ),
    (
        'see_you_again',
        5000
    ),
    (
        'im_yours',
        3000
    ),
]
```

CHECK IF KEY EXISTS

- You can check if a key exists within a dictionary by using the **`in`** keyword

```
video_views = {
    'despacito': 15000,
    'see_you_again': 5000,
    'im_yours': 3000
}
print('shooting star' in video_views)
```

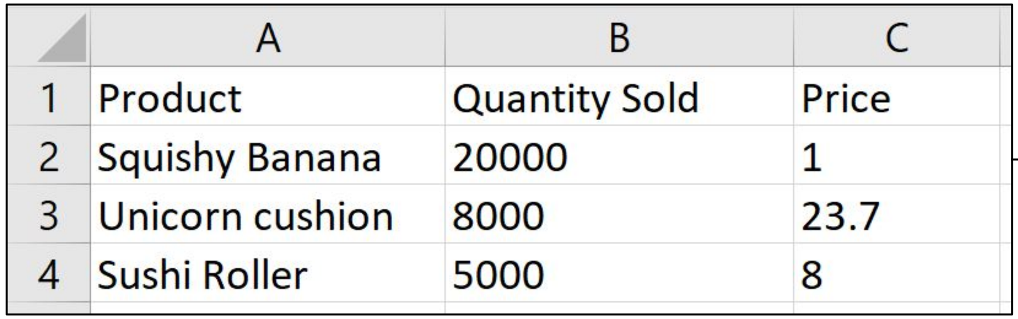
```
False
```

IN-CLASS PRACTICE: DICTIONARY FUNCTIONS

- Complete the given exercises in the in-class notebook!

APPLICATION OF COLLECTIONS: LIST / TUPLE

- An example of a list of many lists



	A	B	C
1	Product	Quantity Sold	Price
2	Squishy Banana	20000	1
3	Unicorn cushion	8000	23.7
4	Sushi Roller	5000	8

Imagine your company have the following excel data and you want to port it over to python to do further analysis

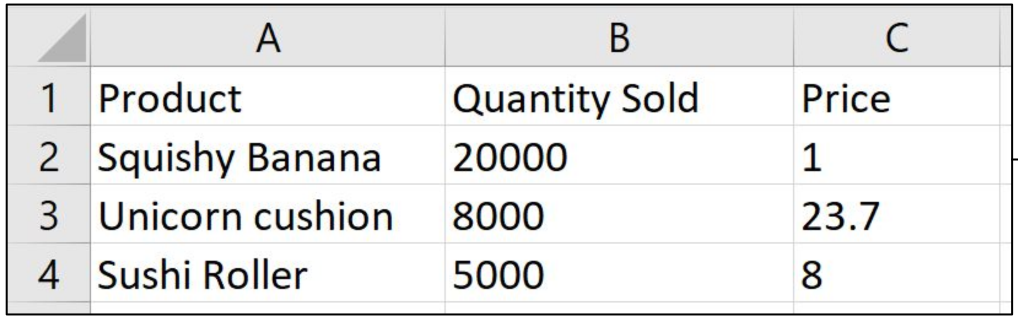
We will teach the methods to read csv excel/csv data into python for processing in week 5!

```
[
      # row 2 in excel
      [
            'Squishy Banana',
             20000,
             1
      ],
      # row 3 in excel
      [
            'Unicorn cushion',
            8000,
            23.7
      ],
      # row 4 in excel
      [
            'Sushi Roller',
            5000,
            8
      ]
]
```

We can actually port it over into a nested list, with each inner list representing one row of data!

APPLICATION OF COLLECTIONS: LIST + DICTIONARY

- An example of a list of dictionaries



	A	B	C
1	Product	Quantity Sold	Price
2	Squishy Banana	20000	1
3	Unicorn cushion	8000	23.7
4	Sushi Roller	5000	8

Imagine your company have the following excel data and you want to port it over to python to do further analysis

There are many ways of combining different collection types to store tabular data. This and the previous slides seeks to illustrate just that. Along the way we will learn more about the best way of storing data!

```
[
     # row 2 in excel
     {
          'Product': 'Squishy Banana',
          'Quantity Sold': 20000,
          'Price': 1
     },
     # row 3 in excel
     {
          'Product': 'Unicorn cushion',
          'Quantity Sold': 8000,
          'Price': 23.7
     },
     # row 4 in excel
     {
          'Product': 'Sushi Roller',
          'Quantity Sold': 5000,
          'Price': 8
     }
]
```

IN-CLASS PRACTICE: EXTRACTING DATA FROM MIXED TYPE*

- Given the following list of historical three days video views data of two videos, find the average views of each video:

```
hist_video_views = [
    {
        'video_title': 'despacito',
        'all_views': [15000, 9800, 10100]
    },
    {
        'video_title': 'im_yours',
        'all_views': [2500, 1400, 14100]
    },
]
```

OBJECT REFERENCES (SELF STUDY)

- How are variables stored?
- What are object references?
- Copying variables

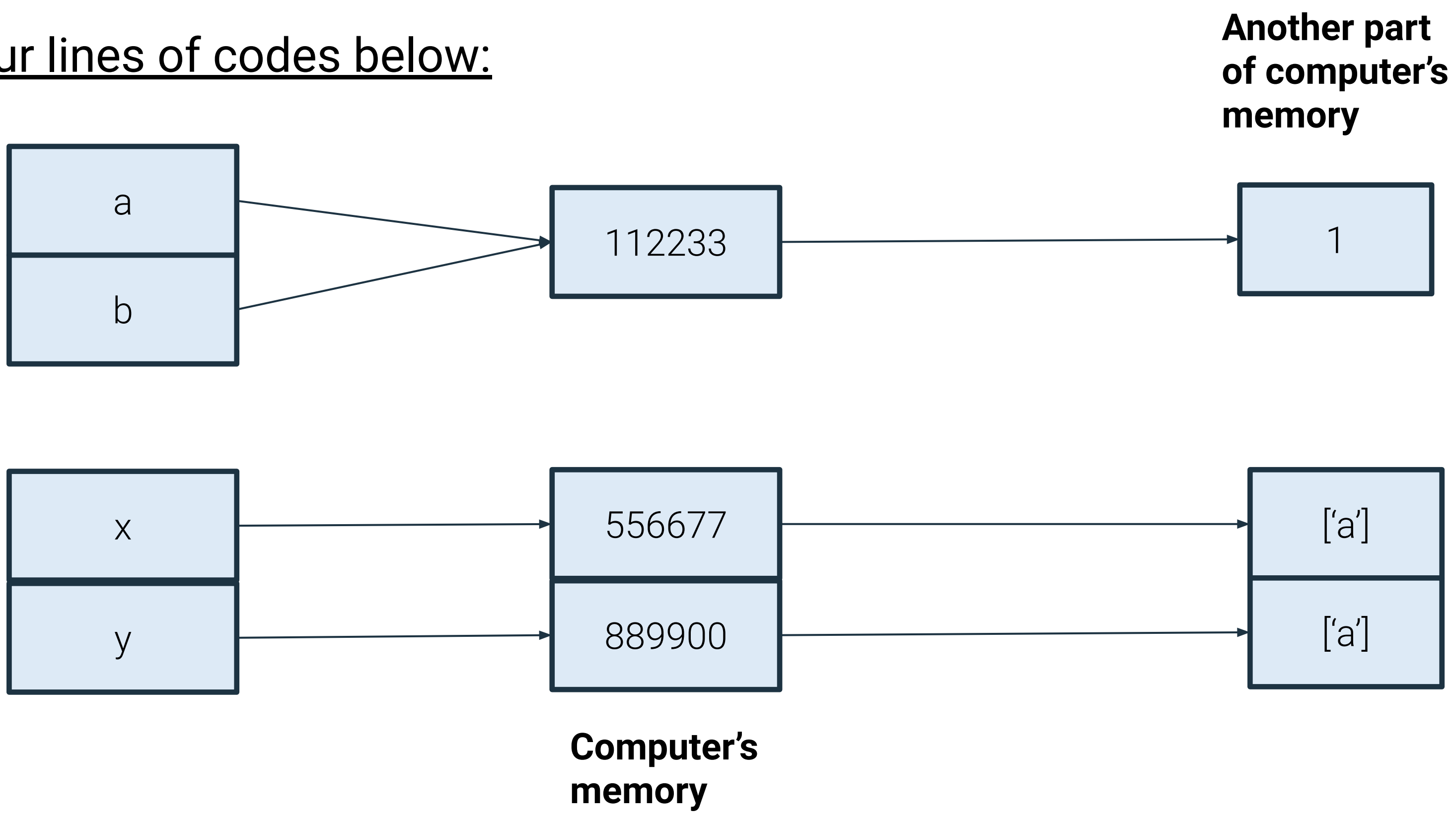
OBJECT REFERENCES

- While all variables are dynamically typed, and hence the difference between the different data types are transparent to you, it still helps to have an understanding of what goes behind the scenes.

Imagine I have the four lines of codes below:

```
a = 1
b = 1
```

```
x = ['a']
y = ['a']
```



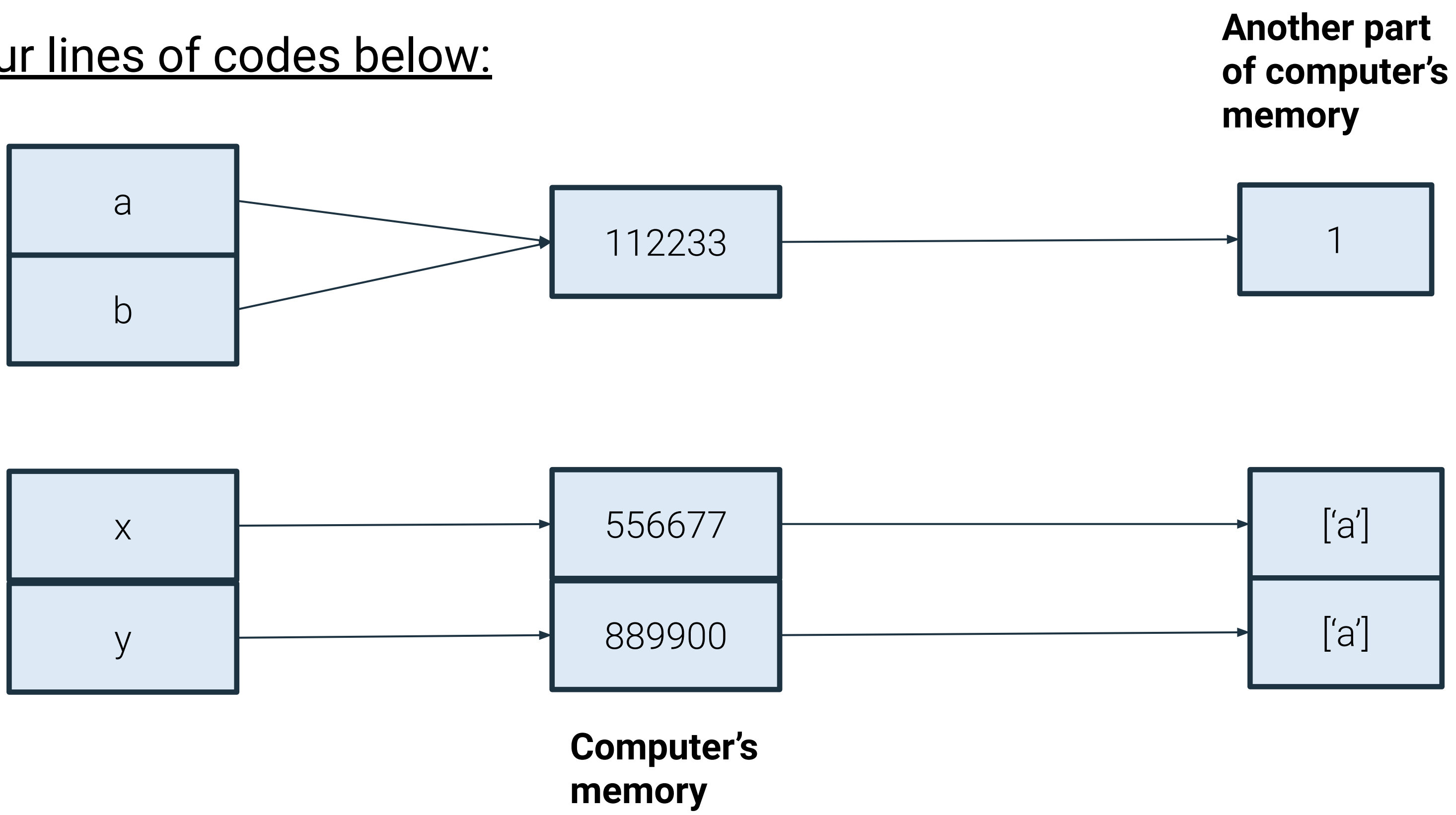
OBJECT REFERENCES

- When a variable is created, an ID is created within the computer’s memory. This ID is dependent on the variable type.

Imagine I have the four lines of codes below:

```
a = 1
b = 1
```

```
x = [‘a’]
y = [‘a’]
```



PRIMITIVE VS REFERENCE VARIABLES

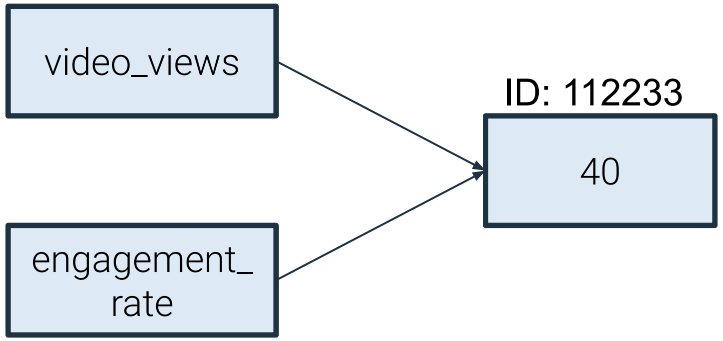
- Immutable data types share a **common** ID/address when a variable has the **same exact match**.
- Mutable data types share **differing** ID/address when a variable appear to have the same content.
- We can check the ID of a variable by using the `id( )` function.

Immutable Data Types	Mutable Data Types
Integers	Collections ● Lists ● Dictionaries
Float	
Boolean	
Strings	

HOW ARE IMMUTABLE DATA STORED?

- What happens in the memory states of CPU:

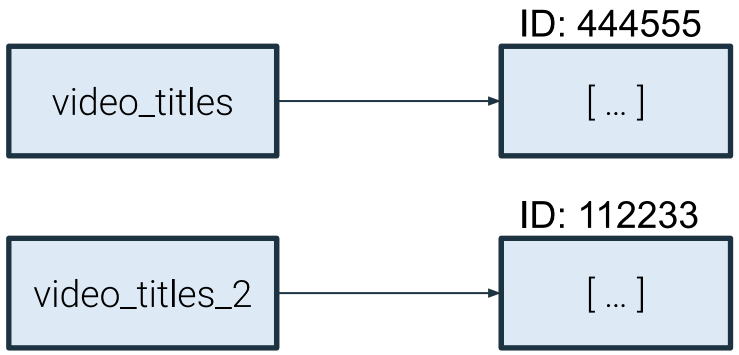
```
video_views = 40
engagement_rate = 40

print(id(video_views), id(engagement_rate))
```



HOW ARE MUTABLE DATA STORED?

- What happens in the memory states of CPU:

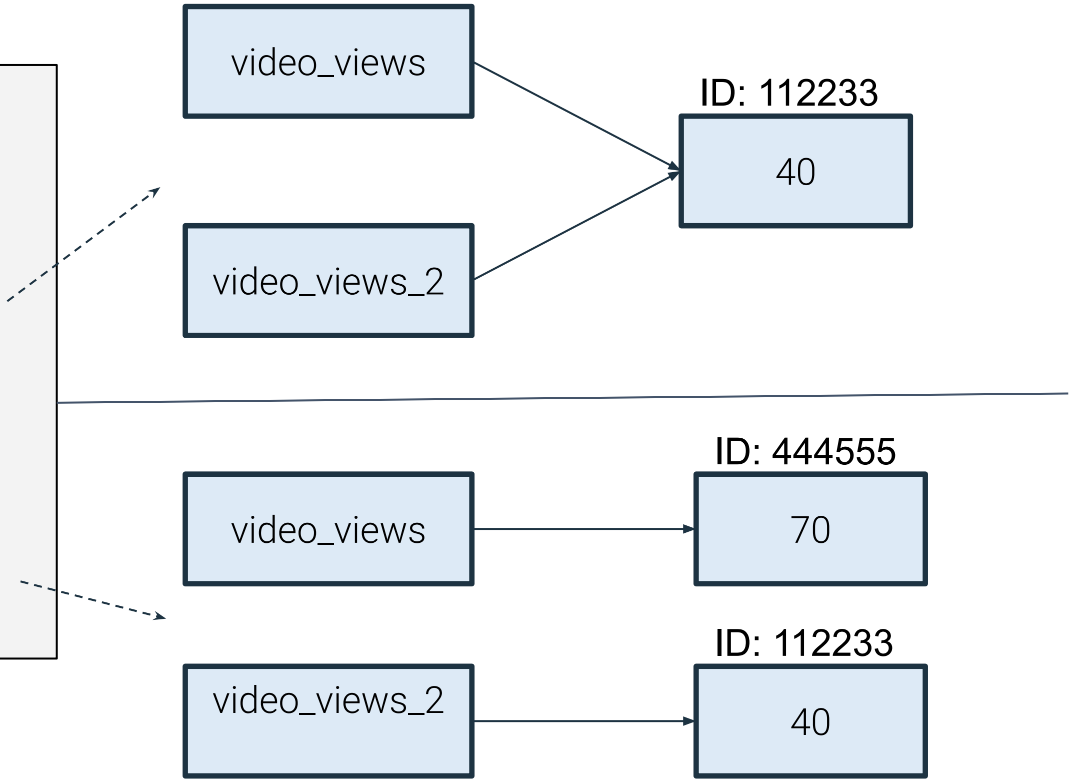
```
video_titles = ['see you again', 'im yours']
video_titles_2 = ['see you again', 'im
yours']

print(id(video_titles), id(video_titles_2))
```



COPYING VARIABLES

- When we equate (copy) a variable to another variable, that variable is pointing to the ID/reference of the original variable.

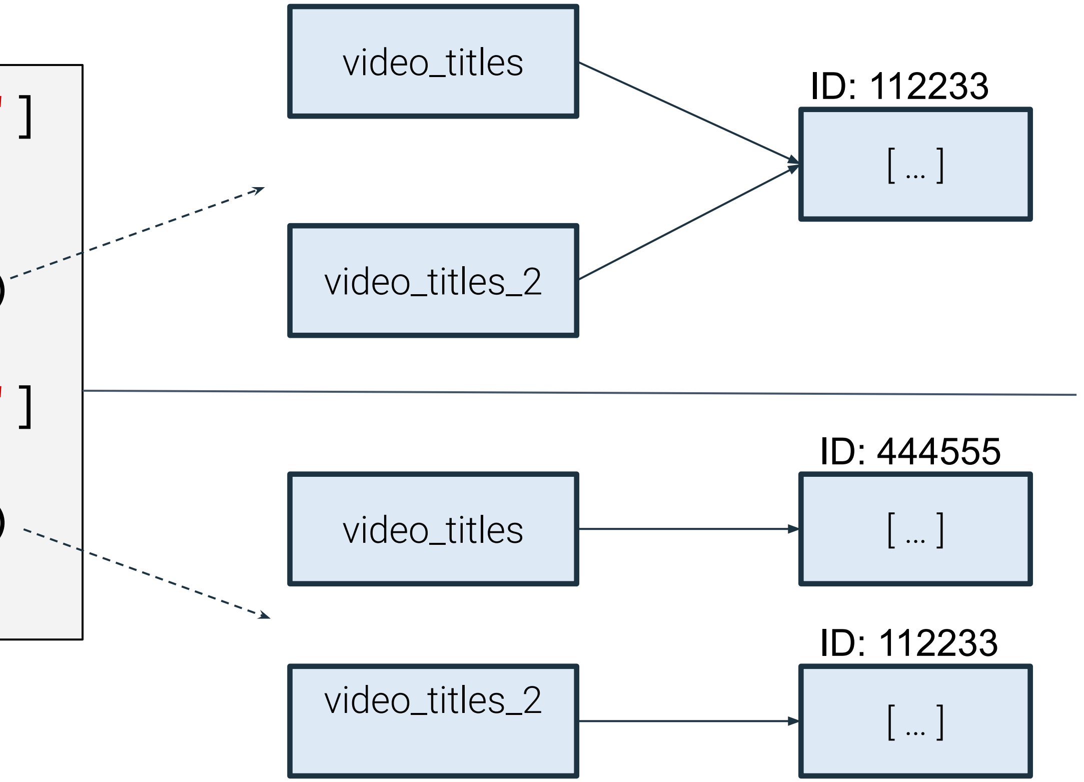
```
video_views = 40
video_views_2 = video_views

print(id(video_views), id(video_views_2))

video_views = 70

print(id(video_views), id(video_views_2))
```



COPYING VARIABLES

- This also applies to mutable data.

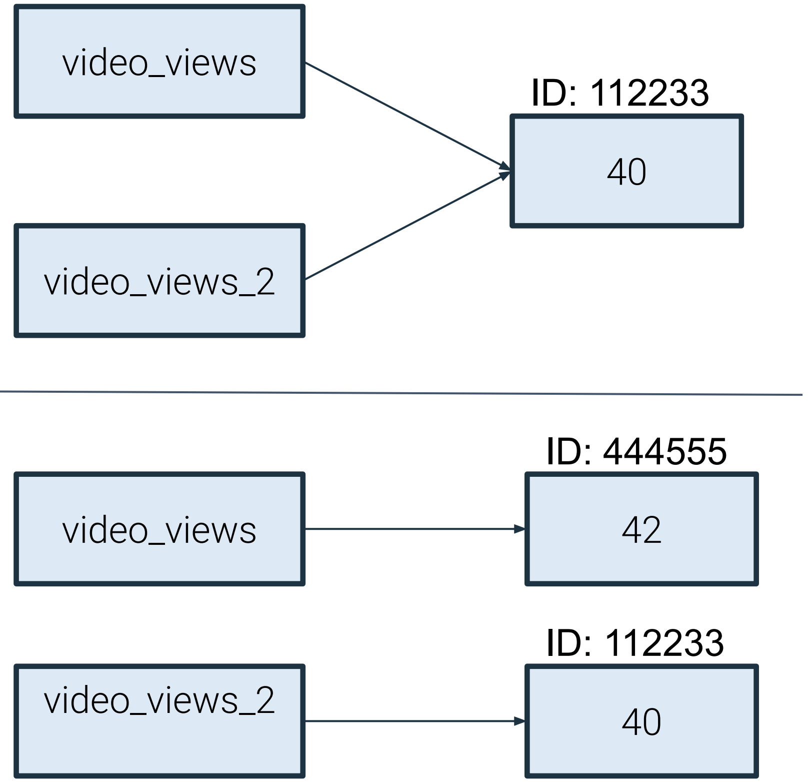
```
video_titles = ['see you again', 'im yours']
video_titles_2 = video_titles

print(id(video_titles), id(video_titles_2))

video_titles = ['see you again', 'im yours']

print(id(video_titles), id(video_titles_2))
```



COPYING VARIABLES

- However, when we change make a change to a immutable data type, the ID changes for the changed variable, as it is now pointing to a new ID.

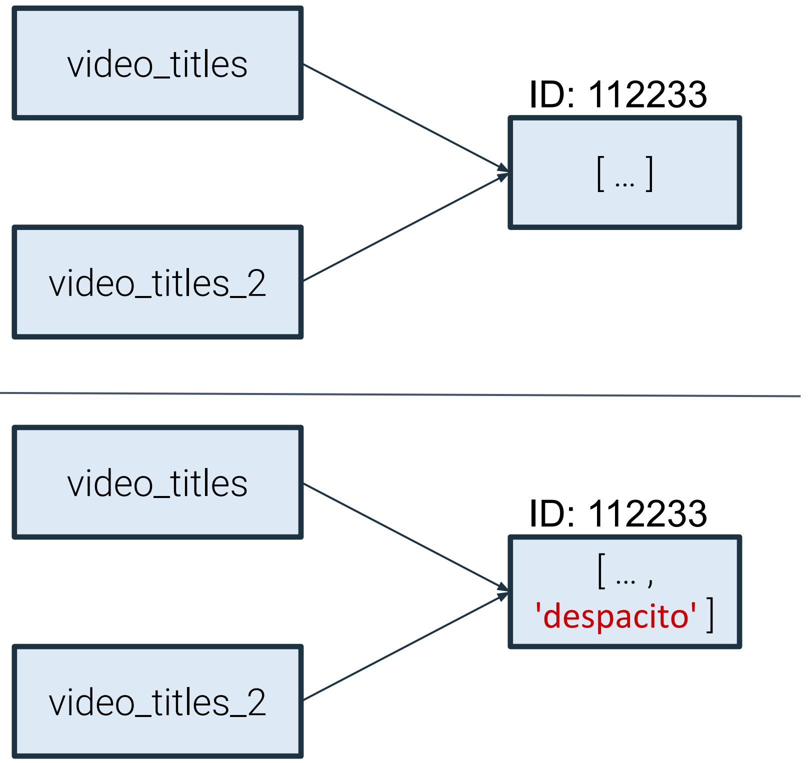
```
video_views = 40
video_views_2 = video_views

print(id(video_views), id(video_views_2))

video_views = video_views + 2

print(id(video_views), id(video_views_2))
```



COPYING VARIABLES

- Whereas with a mutable data, the ID remains the same.
- This is important to understand as it is common for people mistake their data for being independent, when it is clearly not.

```
video_titles = ['see you again', 'im yours']
video_titles_2 = video_titles

print(id(video_titles), id(video_titles_2))

video_titles.append('despacito')

print(id(video_titles), id(video_titles_2))
```



SUMMARY

- What are collections?
- Lists
- Tuples
- Dictionaries
- Application of Collections