# Python

Data Types and Structures: Lists, Tuples, Sets, Dicts

# In this lecture

- Concept of a data structure
- Array
- Strings
- Lists
- Tuples
- Sets
- Dictionaries

# Data Structures

# What is a data structure?

- Unlike a variable which stores one value at a time, **a data structure is built to store a collection of values.**
- **Indexed structures** allow for random access (RAM) – can locate an item by the index location. An array and vector allow for this.
- **Non-indexed structures** (or referenced) structures, on the other hand, are navigated sequentially. For example, a stream of data from the keyboard or from a file, or a linked list in which each node has a pointer the next in the sequence.

# Data structures and algorithms

- There are entire modules (courses) dedicated to this subject.
- The performance of typical operations (insert, delete, search and sort) vary across the structures.
- Big O notation (complexity): constant, linear, polynomial, linearithmic, quadratic etc.
- Path finding algorithms.
- Computer vision.

# Array

# Array

- The items in an array are called **elements**.
- We specify how many elements an array will have when we declare the size of the array **(if 'fixed-size'),** unlike flexible sized collections (e.g. ArrayList in Java).
- Elements are numbered and can referred to by number inside the [ ] is called the **index**. This is used when data is input and output.
- Can only store data if it **matches the type** the array is declared with.

# Array visualisation

| | |
|---|---|
| int mark1 | 28 |
| int mark2 | 76 |
| int mark3 | 54 |

int[] marks = new int[8];

| | |
|---|---|
| marks[0] | 28 |
| marks[1] | 76 |
| marks[2] | 54 |
| marks[3] | 9 |
| marks[4] | 27 |
| marks[5] | 65 |
| marks[6] | 45 |
| marks[7] | 17 |

An Array is a structure that can hold multiple values in individual elements (positions)

# Array

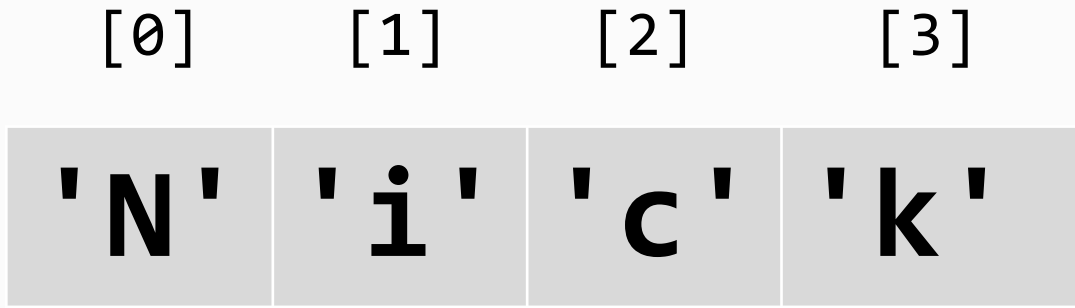
- The items in an array are called **elements**.
- We specify how many elements an array will have when we declare the size of the array **(if 'fixed-size')**, unlike flexible sized collections (ArrayList).
- Elements are numbered and can referred to by number inside the [ ] is called the **index**. This is used when data is input and output.
- Can only store data if it **matches the type** the array is declared with.

# String (str)

# String

- A String (str) object is an immutable array of characters.
- Each character has a numbered position in the array (index):
- We can make use of functions to be able to perform operations on the string.

```
name = "Nick"
```



# Characters

In[ ]:
```
1 | name = "Nick"
2 | name[0]
```

# Characters

In[ ]:
```
1 | name = "Nick"
2 | name[0]
```

Out[]: 'N'

# dir() function

```
In[ ]: 1 | dir(str)
       2 |
```

# dir() function

```
In[ ]: 1 | dir(str)
       2 |
```

```
['__add__',
'__contains__',
'__len__',
'__sizeof__',
'find',
 ... ]
```

# find function

In[ ]:
```
1 | name = "Nick"
2 | name.find('c')
```

# find function

In[ ]:
```
1 | name = "Nick"
2 | name.find('c')
```

Out[]: 2

# Case sensitive

In[ ]:

```
1 | name = "Nick"
2 | name.find('C')
```

# Case sensitive

In[ ]:
```
1 | name = "Nick"
2 | name.find('C')
```

Out[]: -1

# Reminder on Iteration

# for...in...

In[ ]:
```
1 | name = "Nick"
2 | for x in name :
3 |     print(x)
```

# for...in...

In[ ]:
```
1 | name = "Nick"
2 | for x in name :
3 |     print(x)
```

```
N
i
c
k
```

# for in range

In[ ]:

```
1 | for i in range(1,4) :
2 |     print(i)
3 |
```

# for in range

In[ ]:
```
1 | for i in range(1,4) :
2 |     print(i)
3 |
```

```
1
2
3
```

# Lists in Python

# Lists

- A list in Python does use the subscript operator [ ] typically associated with an array. Elements in this list are also indexed.
- The list will maintain a pointer (reference) to objects, rather the integer values (remember Python types are **classes**).
- Lists in python are resizable, unlike static arrays which are fixed.
- Python lists can store elements of different types, whereas arrays are declared to store values of one type.

# Lists and Arrays

## Python List

[27, "Python", 89.13]


## Array

[25, 102, 8]

# List

In[ ]:

```
1 | l = [1,2,3,4,5,6]
2 | l
3 |
```

# List

In[ ]:
```
1 | l = [1,2,3,4,5,6]
2 | l
3 |
```

Out[]: [1, 2, 3, 4, 5, 6]

# List

In[ ]:

```
1 | l = [1,2,3,4,5,6]
2 | l[0]
3 |
```

# List

In[ ]:
```
1 | l = [1,2,3,4,5,6]
2 | l[0]
3 |
```

Out[]: 1

# Access end of list

In[ ]:

```
1 | l = [1,2,3,4,5,6]
2 | l[-1]
3 |
```

# Access end of list

In[ ]:
```
1 | l = [1,2,3,4,5,6]
2 | l[-1]
3 |
```

Out[]: 6

# Access end of list

In[ ]:
```
1 | l = [1,2,3,4,5,6]
2 | l[-2]
3 |
```

# Access end of list

In[ ]:
```
1 | l = [1,2,3,4,5,6]
2 | l[-2]
3 |
```

Out[]: 5

# List slicing

In[ ]:
```
1 | l = [1,2,3,4,5,6]
2 | l[2:4]
3 |
```

# List slicing

In[ ]:
```
1 | l = [1,2,3,4,5,6]
2 | l[2:4]
3 |
```

Out[]: [3, 4]

# List append

In[ ]:
```
1 | l = [1,2,3,4,5,6]
2 | l.append(7)
3 | l
```

# List append

In[ ]:
```
1 | l = [1,2,3,4,5,6]
2 | l.append(7)
3 | l
```

Out[]: [1, 2, 3, 4, 5, 6, 7]

# List remove

In[ ]:
```
1 | l = [1,2,3,4,5,6,7]
2 | l.remove(7)
3 | l
```

# List remove

In[ ]:
```
1 | l = [1,2,3,4,5,6,7]
2 | l.remove(7)
3 | l
```

Out[]: [1, 2, 3, 4, 5, 6]

# Different types

In[ ]:

```
1 | l = [1,2.25,"Nick","N",True,obj]
2 | l
3 |
```

# Different types

In[ ]:
```
1 | l = [1,2.25,"Nick","N",True,obj]
2 | l
3 |
```

Out[]:
```
[1, 2.25, 'Nick', 'N', True, obj]
```

# Tuples in Python

# Tuples

- We've seen that a python list is indexed and can store elements of different types (heterogeneity)
- Tuples are constant (immutable) – once they are declared, they cannot be reassigned.
- A list is declared with [ ] whereas the tuple is declared with ( )
- We can still refer to elements in a tuple via the [ ]

# Tuple declaration

In[ ]:
```
1 | t = (1,2,3,4,5,6)
2 | t
3 |
```

# Tuple declaration

In[ ]:
```
1 | t = (1,2,3,4,5,6)
2 | t
3 |
```

Out[]: (1, 2, 3, 4, 5, 6)

# Tuples

In[ ]:
```
1 | t = (1,2,3,4,5,6)
2 | t[0]
3 |
```

# Tuples

In[ ]:
```
1 | t = (1,2,3,4,5,6)
2 | t[0]
3 |
```

Out[]: 1

# Re-assignment

In[ ]:
```
1 | t = (1,2,3,4,5,6)
2 | t[0] = 5
3 |
```

# Re-assignment not permitted

In[ ]:
```
1 | t = (1,2,3,4,5,6)
2 | t[0] = 5
3 |
```

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

# Tuple count method

In[ ]:
```
1 | t = (1,1,1,4,5,6)
2 | t.count(1)
3 |
```

# Tuple count method

In[ ]:
```
1 | t = (1,1,1,4,5,6)
2 | t.count(1)
3 |
```

Out[]: 3

# Tuple index method

In[ ]:
```
1 | t = (1,1,1,4,5,6)
2 | t.index(5)
3 |
```

# Tuple index method

In[ ]:
```
1 | t = (1,1,1,4,5,6)
2 | t.index(5)
3 |
```

Out[]: 4

# Tuples vs Lists

- **Tuples are immutable** (constant) – once they are declared, they cannot be reassigned.
- **A list is mutable** – elements can be reassigned.
- A list is declared with [ ] whereas the tuple is declared with ( )
- We can refer to elements in both a list and tuple via the [ ]

# Sets in Python

# Sets in Python

- Sets in mathematics refer to a set of distinct numbers – there are no duplicates.
- It is possible to store duplicates in a Python set, but only the unique values will be printed.
- Casting data to a set is a useful way to remove duplicates!
- Sets are declared with the { }
- Sets are mutable (can change)

# Sets

In[ ]:

```
1 | s = {1,2,3,4,5,6}
2 | s
3 |
```

# Sets

In[ ]:
```
1 | s = {1,2,3,4,5,6}
2 | s
3 |
```

Out[]: {1, 2, 3, 4, 5, 6}

# Add to set

In[ ]:
```
1 | s = {1,2,3,4,5,6}
2 | s.add(7)
3 | s
```

# Add to set

In[ ]:
```
1 | s = {1,2,3,4,5,6}
2 | s.add(7)
3 | s
```

Out[]: {1, 2, 3, 4, 5, 6, 7}

# Remove from set

In[ ]:
```
1 | s = {1,2,3,4,5,6,7}
2 | s.remove(7)
3 | s
```

# Remove from set

In[ ]:
```
1 | s = {1,2,3,4,5,6,7}
2 | s.remove(7)
3 | s
```

Out[]: {1, 2, 3, 4, 5, 6}

# Set duplicates

In[ ]:
```
1 | s = {1,2,3,4,5,6,1,2,3,4,5,6}
2 | s
3 |
```

# Set duplicates

In[ ]:
```
1 | s = {1,2,3,4,5,6,1,2,3,4,5,6}
2 | s
3 |
```

Out[]: {1, 2, 3, 4, 5, 6}

# Cast to set

In[ ]:
```
1 | l = [1,1,2,2,3,3,4,4,5,5,6,6]
2 | s = set(l)
3 | s
```

# Cast to set

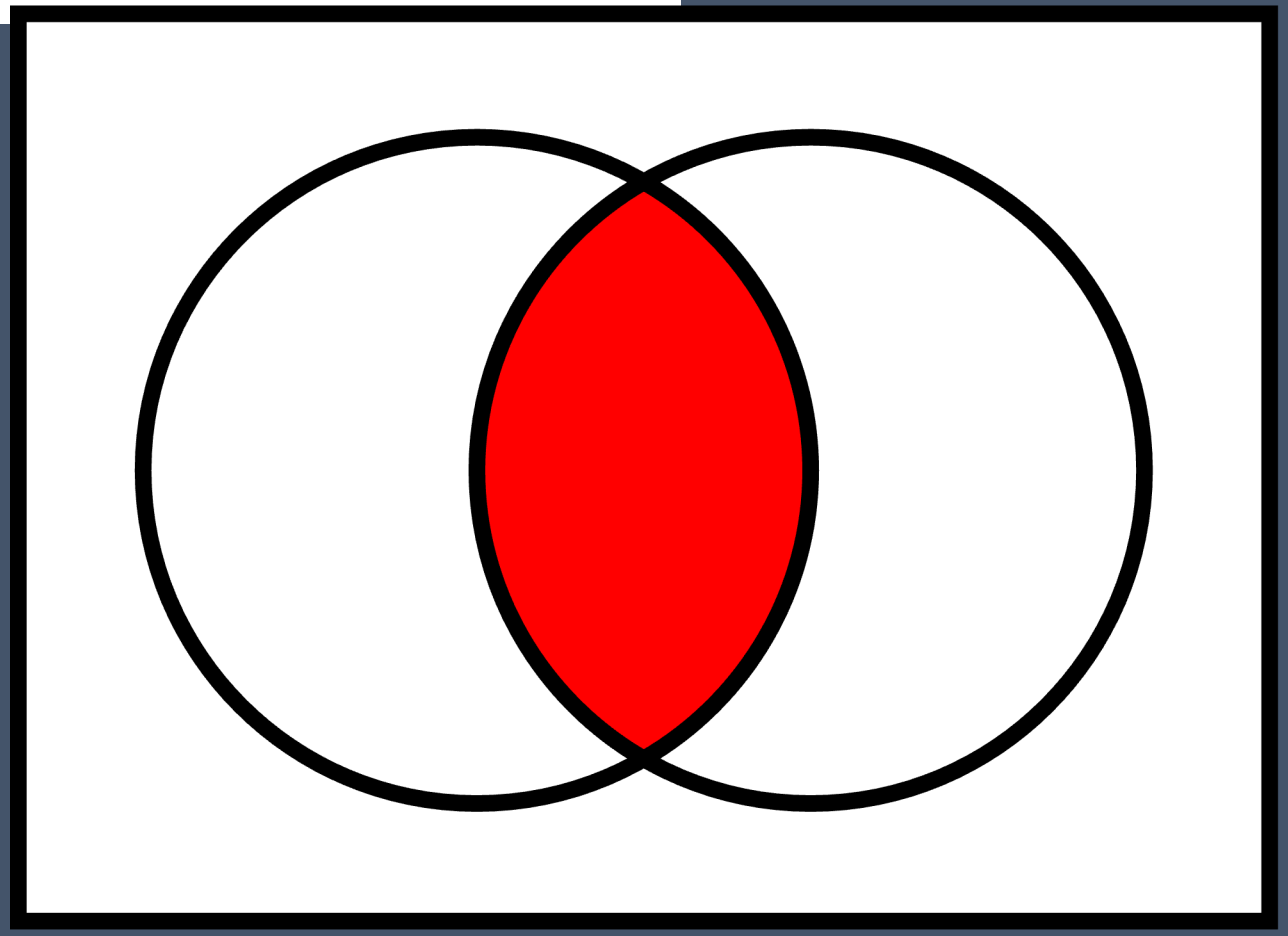
In[ ]:
```
1 | l = [1,1,2,2,3,3,4,4,5,5,6,6]
2 | s = set(l)
3 | s
```

Out[]:
```
{1, 2, 3, 4, 5, 6}
```

# Set theory

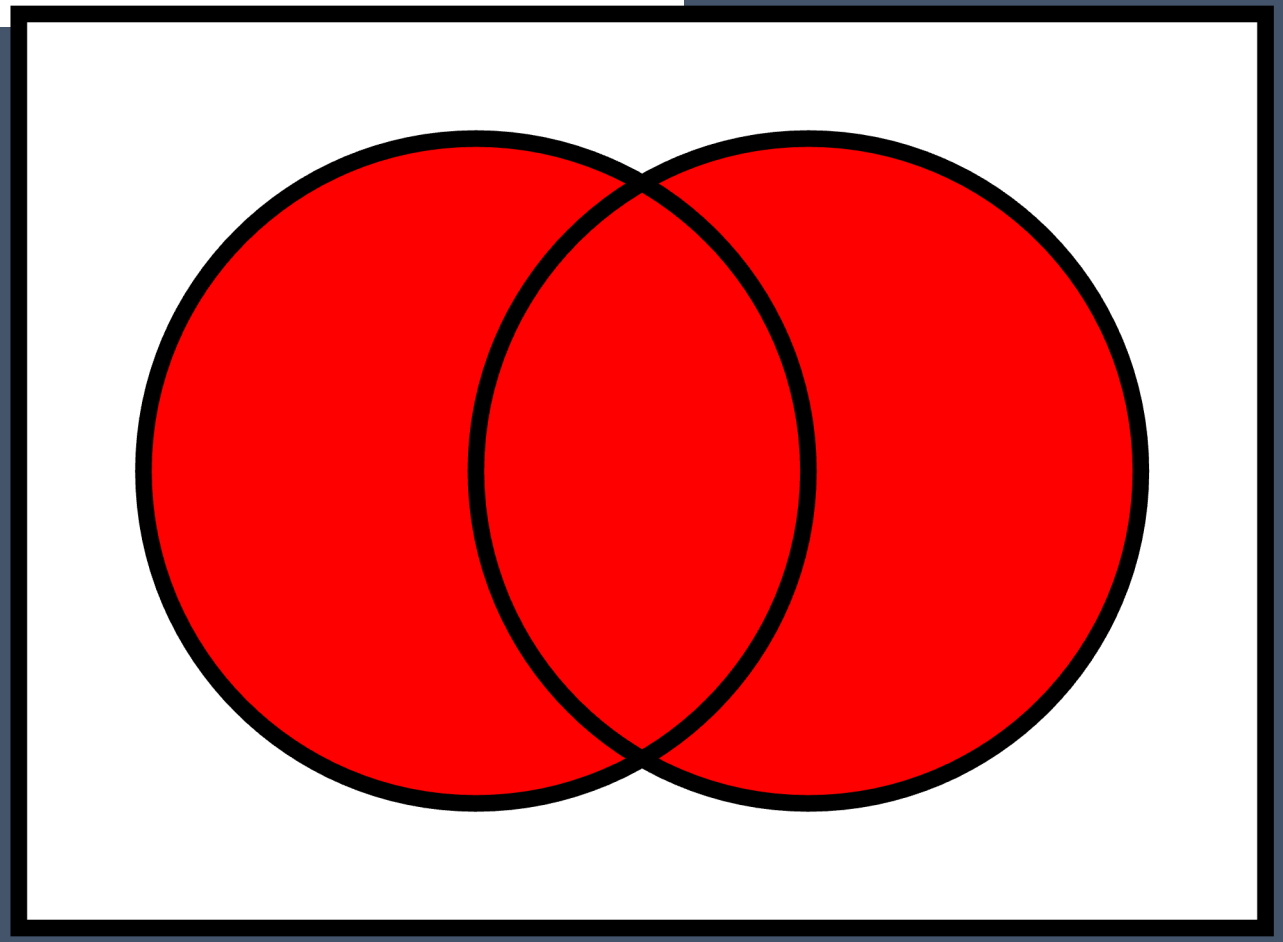
- Intersect
- Union
- Difference

# Set Intersect



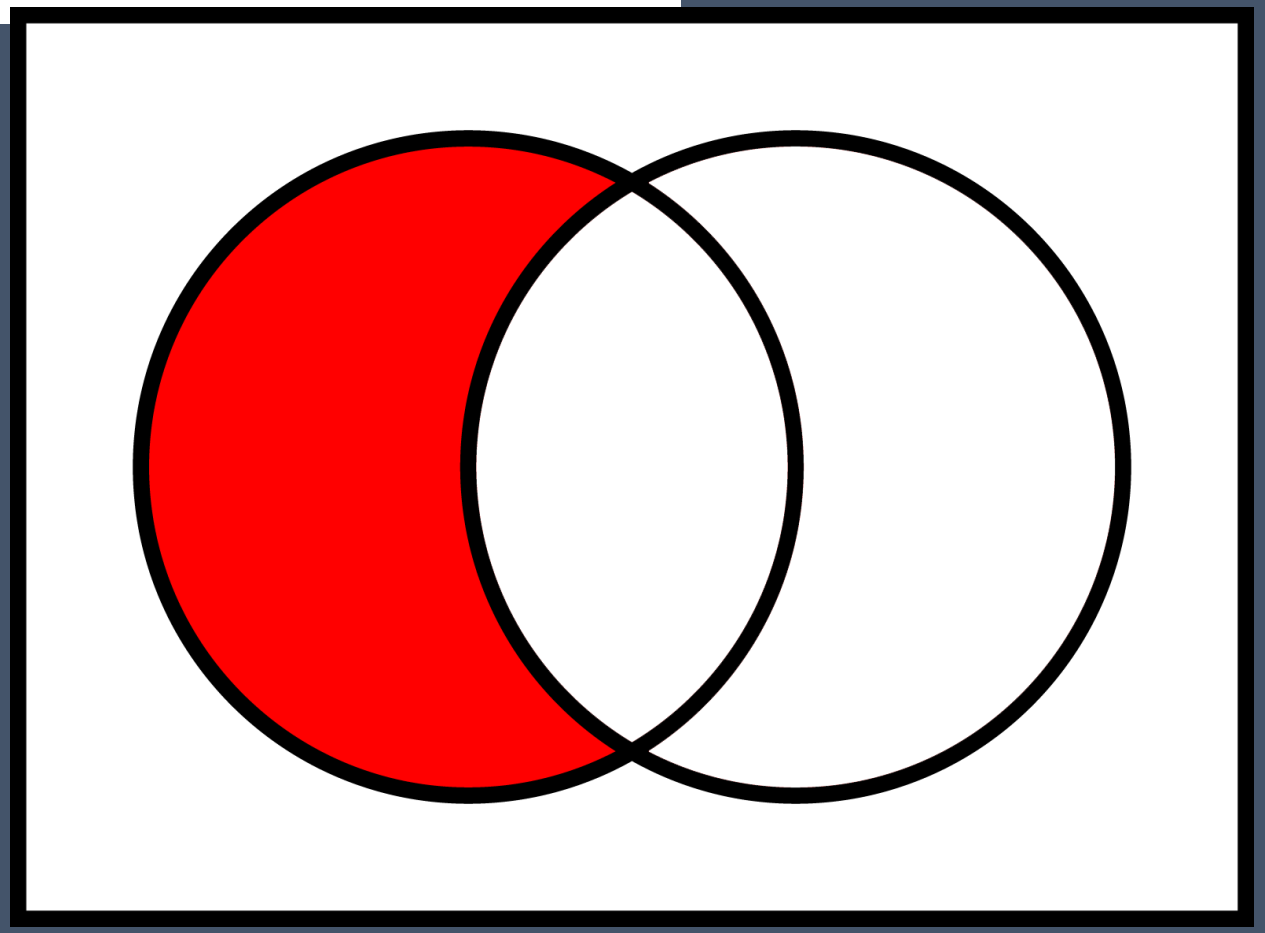
$A \cap B$

# Set Union



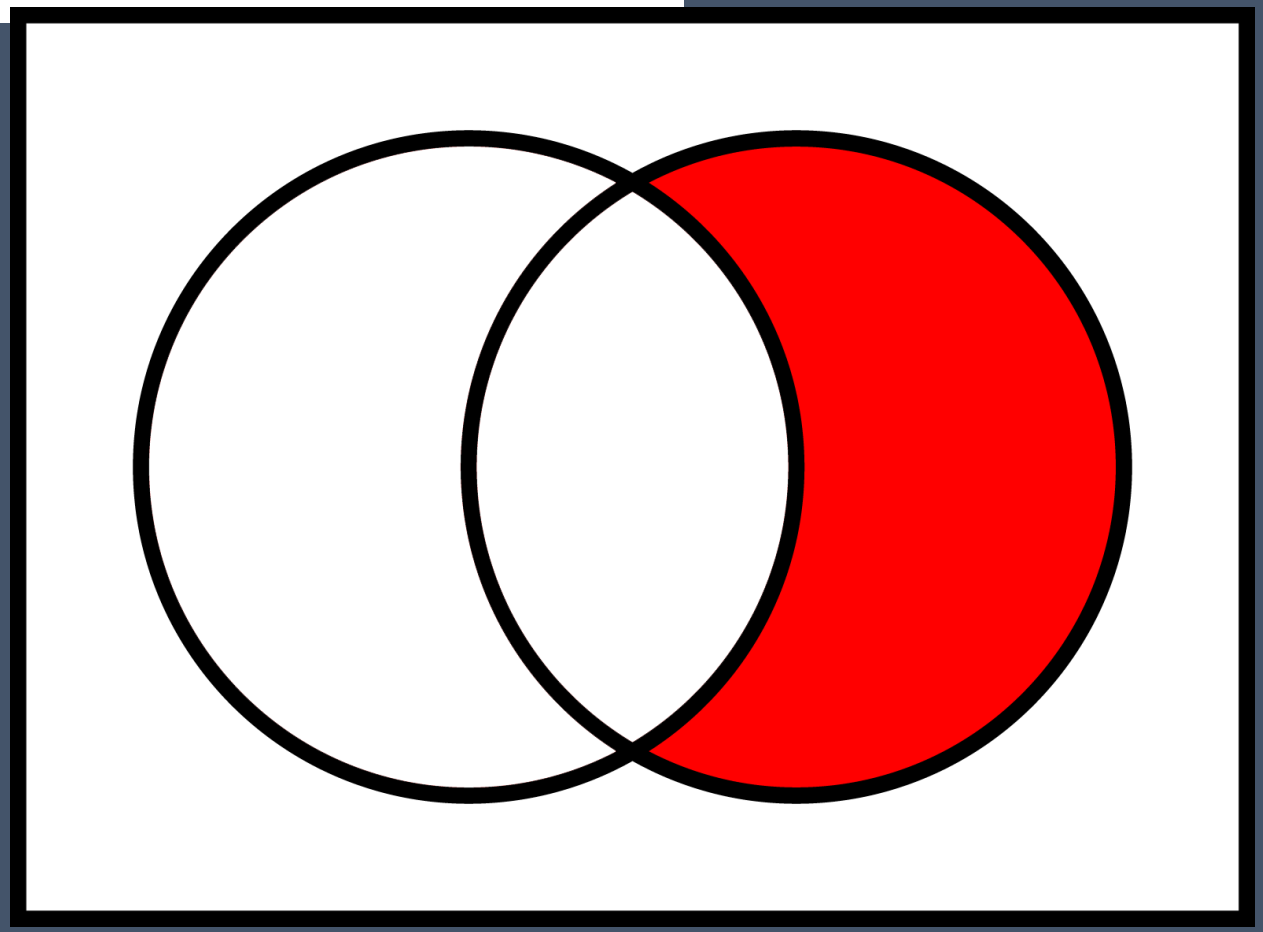
$A \cup B$

# Set Difference



$A \setminus B$

# Set Difference



$B \setminus A$

# Set intersect (mid)

In[ ]:
```
1 | s1 = {1,2,3,4,5,6}
2 | s2 = {4,5,6,7,8,9}
3 | s1 & s2
```

# Set intersect (mid)

In[ ]:
```
1 | s1 = {1,2,3,4,5,6}
2 | s2 = {4,5,6,7,8,9}
3 | s1 & s2
```

Out[]: {4, 5, 6}

# Set union (all)

In[ ]:
```
1 | s1 = {1,2,3,4,5,6}
2 | s2 = {4,5,6,7,8,9}
3 | s1 | s2
```

# Set union (all)

In[ ]:
```
1 | s1 = {1,2,3,4,5,6}
2 | s2 = {4,5,6,7,8,9}
3 | s1 | s2
```

Out[]: {1, 2, 3, 4, 5, 6, 7, 8, 9}

# Set difference (unique)

In[ ]:
```
1 | s1 = {1,2,3,4,5,6}
2 | s2 = {4,5,6,7,8,9}
3 | s1 - s2
```

# Set difference (unique)

In[ ]:
```
1 | s1 = {1,2,3,4,5,6}
2 | s2 = {4,5,6,7,8,9}
3 | s1 - s2
```

Out[]:
```
{1, 2, 3}
```

# Set difference (unique)

In[ ]:
```
1 | s1 = {1,2,3,4,5,6}
2 | s2 = {4,5,6,7,8,9}
3 | s2 - s1
```

# Set difference (unique)

In[ ]:
```
1 | s1 = {1,2,3,4,5,6}
2 | s2 = {4,5,6,7,8,9}
3 | s2 - s1
```

Out[]: {7, 8, 9}

# Dictionaries in Python

# Dictionaries

- An English Dictionary would allow us to look up the definition of a word. We search the word to locate the definition.
- In Python, we specify a key (word) to be able to get a value (definition).
- Similar to an associative array, or a Map in Java.
- Like Set, Dictionaries also use the { } but they feature : for a key and value pair  { k : v }

# Dictionary

In[ ]:
```
1 | d = {"USA": 200, "UK": 200, "EU": 200}
2 | d
3 |
```

# Dictionary

In[ ]:
```
1 | d = {"USA": 200, "UK": 200, "EU": 200}
2 | d
3 |
```

Out[]: {'USA': 200, 'UK': 200, 'EU': 200}

# Element by key

In[ ]:
```
1 | d = {"USA": 200, "UK": 200, "EU": 200}
2 | d["UK"]
3 |
```

# Element by key

In[ ]:
```
1 | d = {"USA": 200, "UK": 200, "EU": 200}
2 | d["UK"]
3 |
```

Out[]: 200

# Element by key

In[ ]:
```
1 | d = {"USA": 200, "UK": 200, "EU": 200}
2 | d["uk"]
3 |
```

# Case sensitive

```
In[ ]: 1 | d = {"USA": 200, "UK": 200, "EU": 200}
       2 | d["uk"]
       3 |
```

```
---------------------------
KeyError: 'uk'
---------------------------
```

# Append to Dict

In[ ]:
```
1 | d = {"USA": 200, "UK": 200, "EU": 200}
2 | d["Asia"] = 300
3 | d
```

# Append to Dict

In[ ]:
```
1 | d = {"USA": 200, "UK": 200, "EU": 200}
2 | d["Asia"] = 300
3 | d
```

Out[]:
```
{'USA': 200, 'UK': 200, 'EU': 200,
'Asia': 300}
```

# Remove from Dict

In[ ]:
```
1 | d = {"USA": 200, "UK": 200, "EU": 200, "Asia": 30}
2 | del d["Asia"]
3 | d
```

# Remove from Dict

In[ ]:
```
1 | d = {"USA": 200, "UK": 200, "EU": 200, "Asia": 30}
2 | del d["Asia"]
3 | d
```

Out[]:
```
{'USA': 200, 'UK': 200, 'EU': 200}
```

# Dict keys and values

In[ ]:
```
1 | d = {"USA": 200, "UK": 200, "EU": 200}
2 | print( d.keys() )
3 | print( d.values() )
```

# Dict keys and values

In[ ]:
```
1 | d = {"USA": 200, "UK": 200, "EU": 200}
2 | print( d.keys() )
3 | print( d.values() )
```

Out[]:
```
dict_keys(['USA', 'UK', 'EU'])
dict_values([200, 200, 200])
```

# Summary

# Data Structures

- You can distinguish between the key collections by the pairs of brackets used:

- Lists: [ , ] mutable
- Tuples: ( , ) immutable
- Sets: { , } unique values (duplicates not printed)
- Dict: {k : v} key and value pairs

# Exercise 1 [List]

# Exercise 1 [List]

- Write a Python function named **select_odds** to select the odd items of a **list.**
- **Note**: function should result a list with result.

```
#Test cases
>>> select_odds([24,30,44,55,12])
[55]
>>> select_odds([24,30,44,55,12,3,5,6])
[55, 3, 5]
>>>
```

# Answer 1 [List]

## Answer 1

```
def select_odds(listA):
    odd_list = []
    for number in listA:
        if number%2 != 0:
            odd_list.append(number)
    return odd_list
```

# Exercise 2 [Lists]

# Exercise 2 [Lists]

- Write function **select_unique_values** to get unique values from a list.
- **Note:** result should be in a list.

```
#Test cases
>>> select_unique_values([2,2,2,2,3,3,1,2,6,7,8,9,9,10])
[1, 2, 3, 6, 7, 8, 9, 10]

>>> select_unique_values(['one','two','one','two','three','one'])
['three', 'two', 'one']
```

# Answer 2 [Lists]

## Answer 2

```
def select_unique_values(listA):
    return list(set(listA))
```

# Exercise 3 [Dict]

# Exercise 3 [Dict]

- Write function **sum_values** to sum all the items in a dictionary.

```
#Test cases

d = {'a': 3, 'b': 4,'c':5}

>>> sum_values(d)
12
```

# Answer 3 [Dict]

# Exercise 4 [Dict]

# Exercise 4 [Dict]

- Write function **get_max_min** to get the maximum and minimum values in a dictionary and **return result in form of tuple.**
- Write function **get_max_min_dict** to get the maximum and minimum values in a dictionary and **return result in form of dictionary.**

```
>>> d = {'a': 3, 'b': 4,'c':50,'d':1}
>>> get_max_min(d)
(50, 1)
>>> get_max_min_dict(d)
{'max': 50, 'min': 1}
```

***Hint:**

use **max()** and **min()**

# Answer 4 [Dict]

## Answer 4

COPY

```
def get_max_min(dictionary):
        maximum = max(d.values())
        minimum = min(d.values())
        return maximum,minimum

def get_max_min_dict(dictionary):
        maximum = max(d.values())
        minimum = min(d.values())
        return {'max':maximum,'min':minimum}
```