COM4013 Introduction to Software Development

- Week 10
- Collections

- *Umar Arif – u.arif@leedstrinity.ac.uk*

Recap

- Previously we went over **Lists** which are a type of **container** in Python

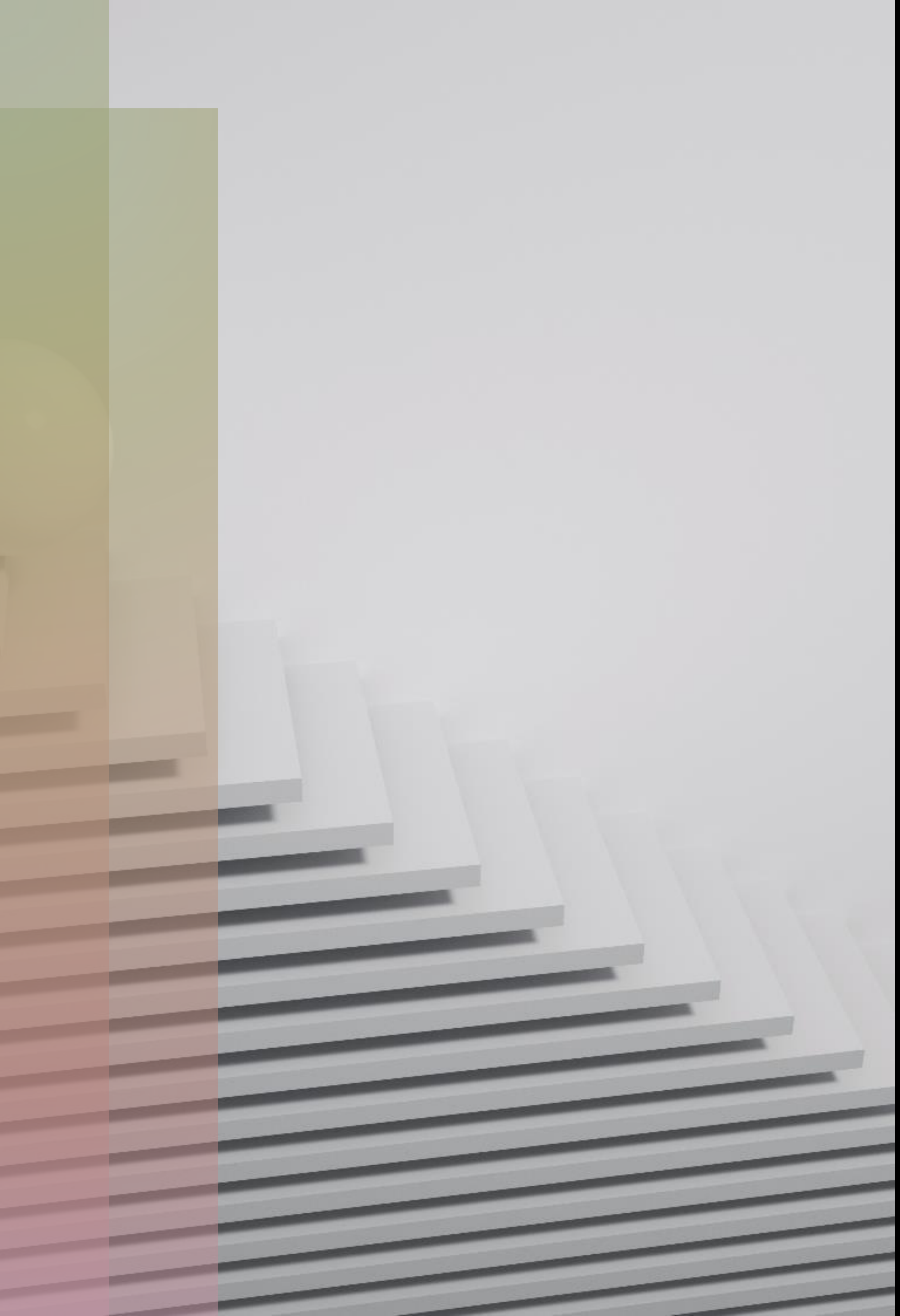
```
users = ["Liam", 12, "Ben", 11, "Allan", 12]
```

- Here we have an example of a Python List called users followed by their age.
- If we wanted to access the names only then we could do the following things

```
names = []

for i in range(0, 6, 1):
    if (i % 2 == 0):
        names.append(users[i])
```

- How would I go about grabbing the ages into their own List?



Dictionaries

- Fortunately for us there are other containers that exist (especially when we have **key-value pairs**).
- Dictionaries are versatile and a great way to store two related things. You may have used maps in other languages like Java which are like these.

```
names = { } # Empty dict – note the curly braces
example = { "name":"Umar", "Age":27, "??":"??" }
```

- Add a single item to our dictionary

```
names.update( {"Lyn", 11} )
```

- To delete a items from a dict we can use the pop method or the del keyword. Both methods require the user to provide the key which deleted both the key-value pair:

```
names.pop("Lyn")    or    del names["Lyn"]
```

Dictionaries

- Print the key component of our dict

```
for key in dict.keys():
    print(key)
```

- Print the value component of our dict

```
for value in dict.values():
    print(value)
```

- Print the key value pairs

```
for item in dict.items():
    print(item)
```

- Clear the contents of a dict

```
dict.clear()
```

It's worth noting that your keys and/or values can be of any type. This makes them a more powerful/versatile container than maps.

Tuples

- **Tuples** are another Python container (ordered)
- I find them to be the least useful of containers for they are immutable (meaning that they can't be changed once set)
- Although their immutability makes them a powerful tool to ensure that data has integrity

- Let's create a tuple

```
tuple = ("Tuple1", 10, 20)
```

- Like lists we use indices to access tuple elements.

- Which element of the tuple would we be accessing if we used the following code:

```
print(tuple[1])
```

How many elements are there in the tuple container?
Can you think of a suitable use case for tuples?

Sets

- Sets are an unordered collection of unique items

```
set = {1, 2, 3, 4, 5} # Use curly braces
```

- If you try to create a set with duplicates they will be deleted.
- What would get if we printed the following set to the console?

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

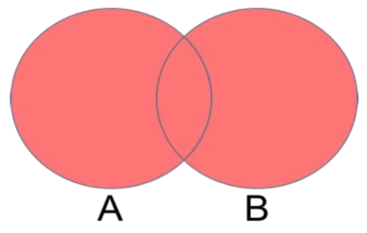
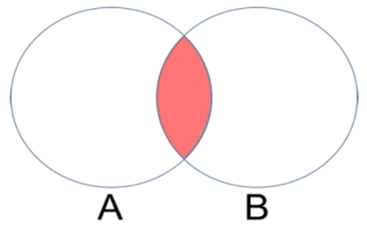
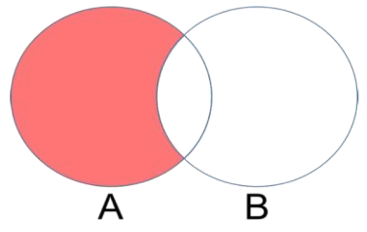
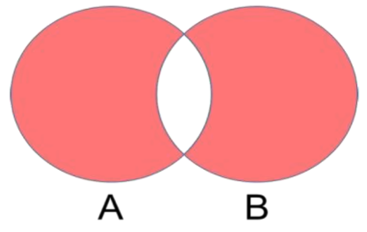
- Sets are useful for comparing and managing unique elements. Consider a scenario where you have a movie theater with a fixed number of seats. Using sets, you can associate individuals who have reserved a seat with the available seat numbers. This ensures that each seat is allocated only once, facilitating an efficient and straightforward representation of seat reservations in a way that avoids duplicates.

Sets

- A basic set
```
set = {1, 2, 3}
```
- Add an element to a set
```
set. add(4) # set - 1, 2, 3, 4
```
- Remove an element
```
set.remove(3)  # Get KeyError if it does !exist
set.discard(3) # Does not return an error
```
- Remove and return the last element of a set
```
poppedElement = set.pop()
```
- Clear the set
```
set.clear()
```

There may be other basic methods... But you can research those in your own time.

Special Set Methods

Set Operation	Venn Diagram	Interpretation
Union		*A* ∪ *B*, is the set of all values that are a member of *A*, or *B*, or both.
Intersection		*A* ∩ *B*, is the set of all values that are members of both *A* and *B*.
Difference		*A* \ *B*, is the set of all values of *A* that are not members of *B*
Symmetric Difference		*A* △ *B*, is the set of all values which are in one of the sets, but not both.

- Consider a scenario with regular parking and reserved parking.
- To identify occupied spaces, use the union operator for a set of all unique values.
- To ensure exclusive parking, use the intersect/difference operator methods, aiming for an empty set.
- The symmetric difference reveals spaces unique to each park – those unavailable in both.

- I am going to let you explore Python sets for yourselves with today's practical.

- if

Containers Iteration

- We can use for loops like the following to iterate through Python Containers
- We do not need the for in range loops – in your assignment you are asked to use the for in range loops in some instances

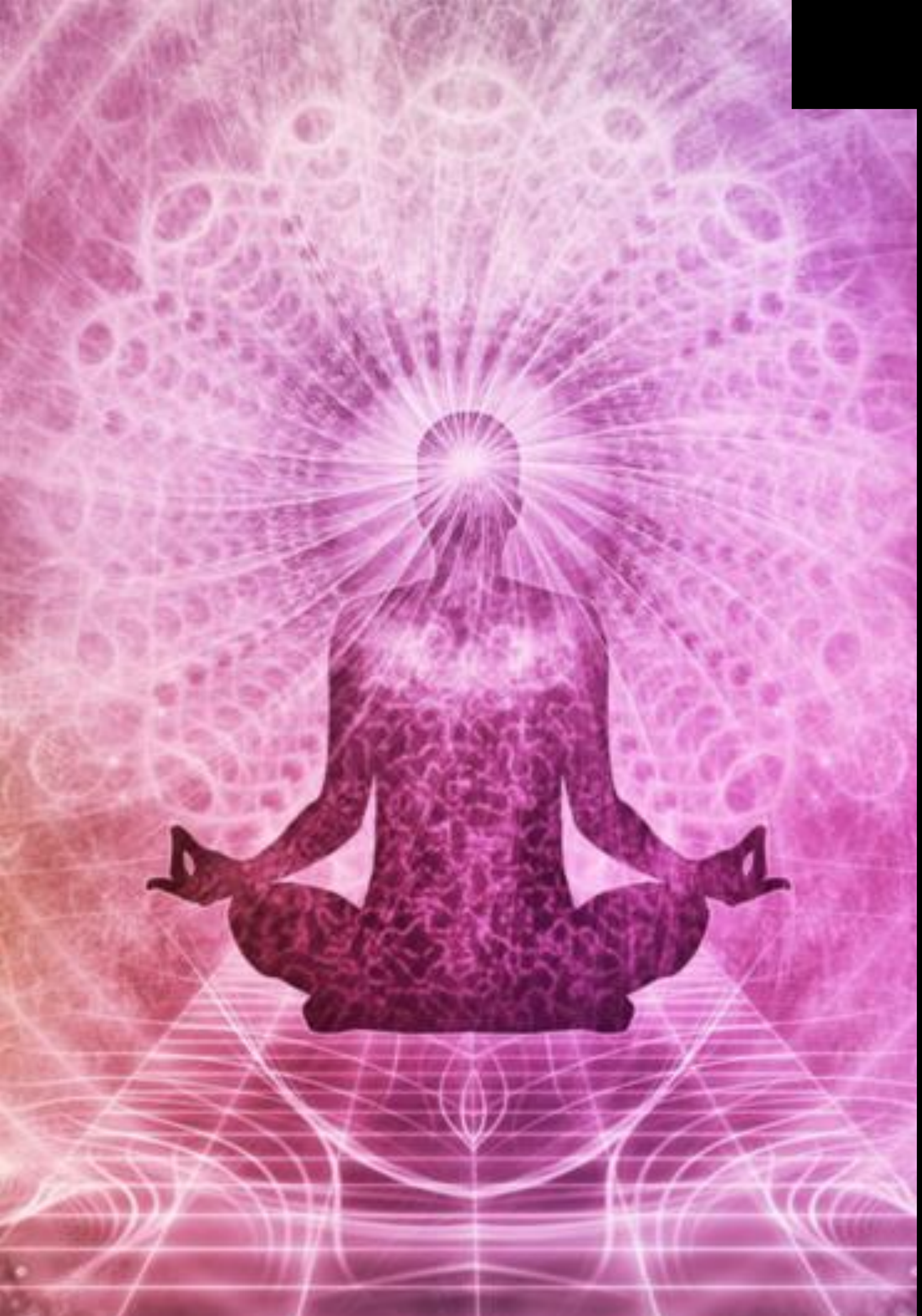
```
for element in List:
    print(element )
```

Or…

```
for i in range(0, len(List), 1):
    Print(List[i])
```

What is the end, step, and start here?

- if



In Operator

- When using if statements, especially when there are many options we may wish to use the in operator
- Note that the ( ) in the second statement is actually a Tuple, that we're creating specifically for this check.
- The second version is more readable and concise.

```
if (element = "Carbon" or element = "Zinc"):
    print(element)
```

Or...

```
if (element in ("Carbon", "Zinc")):
    print(element)
```

These do exactly the same thing

- if

Else Statement in Loop

- In Python we can have else statements after our for loops
- These are only **executed** if the statement within the loop evaluates to False

```
for i in range(5):
    print(i)
else:
    print(“Loop exited etc.”)
```

What is the start, step, and end of this for loop?

- if

Zip

- Zip is a handy function that combines two or more iterables element-wise
- Handy way of creating Tuples of information

```
users = ["Liam", "Ben", "Allan"]
Ages  = [12, 11, 12]
zipped = zip(list1, list2)
# [("Liam", 12), ("Ben", 11), ("Allan", 12)]
```

This is one way to access this zipped data

```
for pair in zipped:
    key, value = pair
    print(f"Key: {key}, Value: {value}")
```

- if

Multiple Assignment

- You can assign multiple values on one line to multiple variables
- Can be an efficient way of writing code
- It will make your code less readable and is not a sign of good coding practice

```
name, age, yOB = "Umar", 27, 1996
```

- What other issues can you see with multiple assignments?

- if

Unit Testing

- In your assignment you are asked to test a couple of components
- One way to do this is through unit testing
- Let's say we have the CValue class saved in a my_class.py file

```
class CValue:
    self.value = 0

    def __init__(self, value):
        self.value = value

    def IncrementByOne(self):
        self.value += 1
```

- What are the constructor, member variable, and class behaviour on this slide?
- Comment this class?

Unit Testing 2

- To test this, we need to create a second class. I saved it as test_my_class.py - Which test fails?

```
import unittest # class inherits functionality
from my_class import CValue

class CTestMyClass(unittest.TestCase):
    def test_add_one(self):    # Name
        obj = CValue(5)        # Create object
        obj.IncrementByOne()   # Call method
        self.assertEqual(obj.value, 6) # Assert

    def test_add_one2(self):
        obj = CValue(3)
        obj.IncrementByOne()
        self.assertEqual(obj.value, 6)

if __name__ == "__main__":
        unittest.main()
```

Unit Testing 3

- Finally, in our Jupyter Notebook we can write the following code:

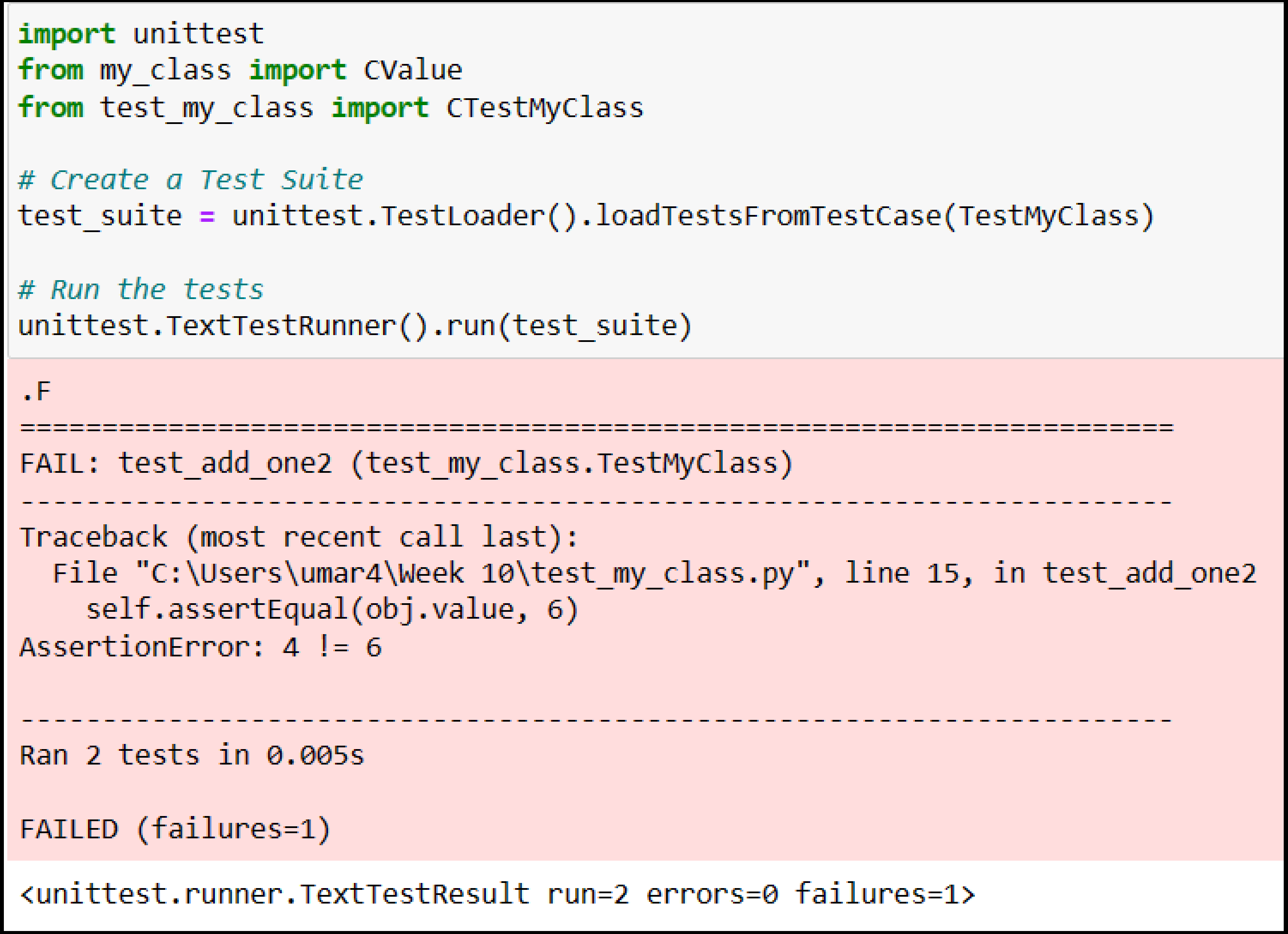
```
import unittest
from my_class import CValue
from test_my_class import CTestMyClass

# Create a Test Suite
testSuite =
unittest.TestLoader().loadTestsFromTestCase(TestMyClass)

# Run the tests
unittest.TextTestRunner().run(testSuite)
```

- As you can see from the import statements, we can call in classes from other files into our main file.
- my_class.py and test_my_class.py exist in the same folder as my Jupyer Notebook file too.

Unit Testing 4



```
import unittest
from my_class import CValue
from test_my_class import CTestMyClass

# Create a Test Suite
test_suite = unittest.TestLoader().loadTestsFromTestCase(TestMyClass)

# Run the tests
unittest.TextTestRunner().run(test_suite)
```

```
.F
======================================================================
FAIL: test_add_one2 (test_my_class.TestMyClass)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "C:\Users\umar4\Week 10\test_my_class.py", line 15, in test_add_one2
    self.assertEqual(obj.value, 6)
AssertionError: 4 != 6


----------------------------------------------------------------------
Ran 2 tests in 0.005s

FAILED (failures=1)
```

```
<unittest.runner.TextTestResult run=2 errors=0 failures=1>
```

Unit Testing 5

- We've only seen assertEqual() but there are many types of tests that we can conduct.

  assertNotEqual(a, b): Checks if a and b are not equal.
  assertIs(a, b): Checks if a and b refer to the same object.
  assertIsNot(a, b): Checks if a and b are not the same
  assertTrue(x): Checks if x is true.
  assertFalse(x): Checks if x is false.
  assertIsNone(x): Checks if x is None.
  assertIsNotNone(x): Checks if x is not None.
  assertIn(a, b): Checks if a is a member of b.
  assertNotIn(a, b): Checks if a is not a member of b.
  assertIsInstance(a, b): Checks if a is an instance of b class

There are many other too... unittest

- if

Summary

We've covered quite a few topics today:

- Mainly wrapping up our Python learning
- Introducing Unit Testing

Any questions?