At the end of this lecture, students should be able to:

- recognise sequences and the common features of sequences
- use the interactive Python interpreter to check python statements and functions
- use the interactive Python interpreter to look up Python help
my_dict = {"a": 4, "b": 6, "c": 5}
print("1.", type(my_dict.items()))

keywords = my_dict.keys()
print("2.", keywords)
print("3. It is not a list object", type(keywords))

items_list = list(my_dict.items())
keys_list = list(my_dict.keys())
values_list = list(my_dict.values())

print("4. items list:", items_list)
print("5. keys list:", keys_list)
print("6. values list:", values_list)

1. <class 'dict_items'>
2. dict_keys(['a', 'c', 'b'])
3. It is not a list object <class 'dict_keys'>
4. items list: [("c", 5), ('a', 4), ('b', 6)]
5. keys list: ['c', 'a', 'b']
6. values list: [5, 4, 6]
Sequences: strings, lists and tuples

- There are seven types of sequences in python. In CompSci 101 we use three of these: **strings**, **lists** and **tuples**.
- Sequences allow you to store multiple values in an ordered and efficient fashion.

```python
a_tuple = (3, 4, 8)
a_list = [3, 4, 8]
a_string = "348"
```

The indices of the elements of sequences start at 0. The indices can be negative (to access elements from the end of the sequence).

The **order** of the elements in a sequence is important.

Each element of a sequence can be accessed using square brackets and the index number, e.g.,

```python
#continued from the code above
print(a_tuple[2])
middle = a_list[1]
last = a_string[-1]
```
Sequences can be sliced:

```python
a_tuple = (3, 4, 8, 7, 2)
a_list = [3, 4, 8, 0, 1]
a_string = "3and 4"
a_tuple2 = a_tuple[0:3:2]
a_list2 = a_list[1:3]
print(a_tuple2, a_list2, a_string[5:1:-2])
```

The `len()`, `min()`, `max()` functions can be applied to sequences (`sum()` can be used with tuples and lists).

The `+`, `*`, and 'in' operators can all be used with sequences

```python
a_tuple = (3, 4) * 3 + (2, 1)
a_list = [3, 0, 1] + [6, 2] * 2
a_string = "3 & 4" * 2
print(a_tuple)
print(a_list)
print(a_string)
print(4 not in a_tuple, 24 in a_list, "43" in a_string)
```
Iterating through the elements of sequences

A *for* ... *in* ... *loop* can be used to visit each element of a sequence, e.g.,

```python
a_tuple = (3, 4, 8, 7, 2)
a_list = [3, 4, 8, 24, 1]

total = 0
for number in a_tuple:
    total = total + number
print("1.", total)

total = 0
for number in a_list:
    total = total + number
print("2.", total)
```

1. 24
2. 40
Iterating through the elements of strings

A `for ... in ...` loop is used to visit each character in a string sequence. Note that the elements of a string sequence are the characters making up the string.

```python
words = "wonderful to know"
number = 0

for letter in words:
    if letter in "aeiou":
        number = number + 1

print(number)
```

5
Iterating through the elements of a sequence - Exercise

Complete the get_num_uniques() function which returns the number of unique elements in the sequence (including non alphabetic characters).

```python
def get_num_uniques(a_sequence):
    green_apple = 8
    abcdefg = 7
    abbbbbbb = 2
    (3, 4, 3, 3, 4, 6, 3, 7, 8, 4) = 5
    [3, 4, 3, 3, 4, 6, 3, 7, 8, 4] = 5

def main():
    desc_1 = "(3, 4, 3, 3, 4, 6, 3, 7, 8, 4):"
    desc_2 = "[3, 4, 3, 3, 4, 6, 3, 7, 8, 4]:"
    print("green apple:", get_num_uniques("green apple"))
    print("abcdefg:", get_num_uniques("abcdefg") )
    print("abbbbbbb:", get_num_uniques("abbbbbbb") )
    print(desc_1, get_num_uniques((3, 4, 3, 3, 4, 6, 3, 7, 8, 4)))
    print(desc_2, get_num_uniques([3, 4, 3, 3, 4, 6, 3, 7, 8, 4]))
main()"
Iterating through the elements of a sequence - Exercise

Complete the count_doubles() function which returns the number of
double letters (a letter followed by the same letter) – each letter can
only be part of one double, i.e, "abbb" has only one double and
'abbbbbb' has two doubles. Assume there are no double spaces in any
strings passed as a parameter.

```python
def count_doubles(a_sequence):

def main():
    sequ_1 = "(4, 4, 3, 3, 4, 4, 3, 3, 8, 8)"
    sequ_2 = "[3, 4, 3, 3, 3, 3, 3, 7, 7, 7]"
    print("green apple: ", count_doubles("green apple"))
    print("abcdefg: ", count_doubles("abcdefg"))
    print("abbbbbb: ", count_doubles("abbbbbb"))
    print(sequ_1, count_doubles((4, 4, 3, 3, 4, 4, 3, 3, 8, 8)))
    print(sequ_2, count_doubles([3, 4, 3, 3, 3, 3, 3, 7, 7, 7]))
main()
```

green apple: 2
abcdefg: 0
abbbbbb: 3
(4, 4, 3, 3, 4, 4, 3, 3, 8, 8): 5
[3, 4, 3, 3, 3, 3, 3, 7, 7, 7]: 3
Compilers and interpreters

Compilers

• Compilers convert source code into machine code and store the machine code in a file. The machine code can then be run directly by the operating system as an executable program (a .exe file).

Interpreters

• Interpreters bypass the compilation process. They convert and execute the code directly, statement by statement.

• Python is **an interpreted language**, i.e., the python interpreter reads and executes each statement of the Python source program, statement by statement. This is the reason why the program executes until the interpreter reaches the error.
IDLE (Integrated Development Environment) is an integrated development environment for Python. This is the development environment provided when you download Python.

WIKIPEDIA states "IDLE is intended to be a simple IDE and is suitable for beginners, especially in an educational environment. To that end, it is cross-platform, and avoids feature clutter."

IDLE provides an interactive environment for checking Python code and for running Python programs.
The Python interactive interpreter (Python shell)

The interactive Python interpreter

- The Python interactive interpreter makes it easy to check Python commands.

Open the interactive interpreter

- We will use IDLE which opens a window with the interpreter prompt: `>>>`
- Once the Python interpreter has started any python command can be executed (at the prompt `>>>`).

```python
>>> word = "amazing"
>>> len(word)
7
>>> word = word * 3
>>> word
'amazingamazingamazing'
>>> another_word = word[2:3]
>>> another_word
'anmiazg'
>>> word[0:-4]
'ganmi'
>>> `}

Notice that the interpreter displays the value in the variable even though there is no print() in the statement. If the value is a string, it is displayed within single quotes.
The Python interactive interpreter cont.

The interactive Python interpreter can also be used to test functions

- The Python interactive interpreter makes it easy to check Python code.

```python
>>> def get_result(command, what_to_do, where):
    return command + " " + what_to_do + " in the " + where

>>> get_result("a", "b", "c")
'a b in the c'
>>> get_result("come", "sing", "hall")
'come sing in the hall'
>>> get_result("go", "jump", "pond")
'go jump in the pond'
```

Notice that it is necessary to insert a blank line to indicate the end of the function definition.

See the results of calling the function three times with different arguments.
The Python interactive interpreter can also be used to get help:

```python
>>> help(str.rfind)
```

```
rfind(...)  
  S.rfind(sub[, start[, end]]) -> int

  Return the highest index in S where substring sub is found, such that sub is contained within S[start:end]. Optional arguments start and end are interpreted as in slice notation.

  Return -1 on failure.
```

```python
>>> help(sum)
```

```
sum(...)  
  sum(iterable[, start]) -> value

  Return the sum of an iterable of numbers (NOT strings) plus the value of parameter 'start' (which defaults to 0). When the iterable is empty, return start.
```
In the interpreter window, print statements are used to print the arguments to the interpreter window.

A function which does not explicitly return a value, always returns None.

```python
>>> def do_little(n1, n2):
    print("Sum:", n1 + n2)

>>> do_little(3, 5)
Sum: 8

>>> print(do_little(3, 5))
Sum: 8
None
```

Notice that it is necessary to insert a blank line to indicate the end of the function definition.

The code in the function executes.

The result of calling the function is printed.
strings, lists and tuples are sequences

• The operators: +, * and in can be used with sequences
• We use a for ... in ... to iterate through each element of a sequence
• len(), min(), max() can be used with sequences
• sum() can be used with tuples and lists of numbers
• Each element of a sequence can be accessed using the index number. The index can be negative (starting from the end of the sequence)
• Sequences can be sliced using [slice_start: slice_end: step]

The Python interactive interpreter (IDLE)

• use the interactive Python interpreter to check python statements and functions
• use the interactive Python interpreter to look up Python help
Python features used in this lecture

- sequence1 = (5, 7, 2, 6, 4, 3, 9)
- sequence2 = [5, 7, 2, 6, 4, 3, 9]
- sequence3 = "This is a sequence of characters"
- for element in sequence_of_whichever_type:
  ...

- min_value = min(sequence1)
- max_value = max(sequence2)
- total = sum(sequence1)
- element_from_end1 = sequence1[-2]
- element_from_end3 = sequence3[-2]
- element = sequence1.index(4)
- position3 = sequence3.index("uen")
- part1 = sequence1[2: 5: 3]
- part2 = sequence2[2: 7: 2]
- part3 = sequence3[1: 5: 2]