Introduction to Computer Programming

Week 3.2: Data structures

Modern problems involve complex data

So far, we have encountered three types of variables (int, float, and string) and learned how to perform basic operations on these types

Now imagine that we are studying mobile phone usage and obtain this data:

Name	Number	Provider	Data limit
Matt	900 464 2453	Vodafone	10 GB
Sarah	886 964 1525	O2	Unlimited
Helen	334 362 1357	Vodafone	20 GB
Robert	556 631 2535	Three	15 GB

Question: How can we use Python to store and analyse this data?

Aims

The aim of these slides is to introduce four new data types that will extend the functionality of Python:

- Lists
- Tuples
- Sets
- Dictionaries

We will also introduce the concepts of **immutable**, **mutable**, and **iterable** data types

Lists

A **list** is an ordered sequence of values, where each value is identified and accessed by an index (which starts at 0)

Lists are created using comma-separated values contained in square brackets

Example: Define a list with three cities and print some of the entries

In [2]:
```
# define a list of cities
cities = ['Toronto', 'Barcelona', 'London']

# print the third entry
print(cities[2])

# print the first and second entry
print(cities[0:2])
```

```
London
['Toronto', 'Barcelona']
```

Lists are **iterable**, meaning we can loop through the entries

Example: Use a `for` loop to print the cities in the previous list

In [4]:
```
# print all of the values in the list of cities
for c in cities:
    print(c)
```

```
Toronto
Barcelona
London
```

Lists can contain values of different type.

Example: Create a list that contains an int, float, and string

In [5]:
```
list = [1, 1.0, 'one']
print(list)
```

```
[1, 1.0, 'one']
```

Lists are **mutable**, meaning their entries can be changed

Example: Change the last entry in a list

In [10]:
```
list = [1, 1.0, 'one']
list[2] = 'One'
print(list)
```

```
[1, 1.0, 'One']
```

Methods for lists

There are several pre-defined **methods** that perform operations on lists

- `L.append(e)` adds the object `e` to the list `L`
- `L.count(e)` returns the number of times that `e` occurs in `L`
- And many more! Type `help(list)` to see them all

We'll learn more about methods later in the course

Example: Create an empty shopping list, add Apples and Pizza to it, and print the list.

In [5]:
```
# Create an empty list
shopping_list = []

# Add some food to it:
shopping_list.append('Apples')
shopping_list.append('Pizza')

# Print what we have
print(shopping_list)
```

```
['Apples', 'Pizza']
```

Nested lists

Example: Let's create a shopping list with some foods and their prices:

Food	Price
Apples	2.00
Pizza	4.50
Bread	1.00

We can do this by creating a list of lists, also called a **nested list**

In [1]:
```
shopping_list = [["Apples", 2.00], ["Pizza", 4.50], ["Bread", 1.00]]
```

Entries in a nested list can be accessed using indices and **nested indices**

In [2]:
```
# access the third sub-list and print
print(shopping_list[2])

# access the price of bread using nested indices and print
print(shopping_list[2][1])
```

```
['Bread', 1.0]
1.0
```

Methods can be used with nested lists too

Example: Add Milk (£1.00) to your shopping list. Print the updated list.

In [3]:
```
shopping_list.append(["Milk", 1.00])
print(shopping_list)
```

```
[['Apples', 2.0], ['Pizza', 4.5], ['Bread', 1.0], ['Milk', 1.0]]
```

Loops can be applied to nested lists

Example: Use a `for` loop to print all of the foods in your shopping list

In [5]:
```
for f in shopping_list:
    print(f[0])
```

```
Apples
Pizza
Bread
Milk
```

List comprehension

List comprehension provides a concise way to apply an operation to the entries of an iterable variable (e.g. a list) that satisfy specific criteria. The output is a new list.

The syntax for a list comprehension is:

```
[op for elem in iterable if test]
```

In this syntax:

- *iterable* is an iterable object (e.g. a list)
- *elem* is an element in *iterable*
- *op* is an operation involving *elem*
- *test* is an **optional** condition that is used to determine whether to apply the operation to *elem*

Example: Create a list with the first ten square numbers

In [5]:
```
L = [e**2 for e in range(10)]
print(L)
```

```
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

Example: Create a list with the first ten square numbers based on even integers

In [6]:
```
L = [e**2 for e in range(10) if e % 2 == 0]
print(L)
```

```
[0, 4, 16, 36, 64]
```

Example: Using a list comprehension to filter out values less than 5

In [1]:
```
values = [4, 1, 6, 3, 8, 2, 7]
filtered_values = [e for e in values if e < 5]
print(filtered_values)
```

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

Example: Use a list comprehension to capitalise strings in a list. The method `capitalize` will be used.

In [4]:
```
s = 'red'
print(s.capitalize())

colours = ['red', 'blue', 'orange']
Colours = [c.capitalize() for c in colours]
print(Colours)
```

```
Red
['Red', 'Blue', 'Orange']
```

Example: Use a list comprehension to create a nested list, where each sub-list contains $[m, m^2, m^3]$ with m being an integer ranging from 1 to 5.

In [12]:
```
list = [[m, m**2, m**3] for m in range(1,6)]
print(list)
```

```
[[1, 1, 1], [2, 4, 8], [3, 9, 27], [4, 16, 64], [5, 25, 125]]
```

Tuples

A **tuple** is an ordered sequence of values and is very similar to a list.

Tuples are created using round brackets:

In [15]:
```
t = (1, 1.0, 'one')
```

A major difference is that tuples are **immutable**, meaning that their values cannot be changed:

In [16]:
```
t[2] = 'One'
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-16-6a2305685e55> in <module>
----> 1 t[2] = 'One'

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

Sets

Sets are another collection of elements. However, unlike lists and tuples, sets are **unordered** sequences of *unique* values.

Sets are created using curly brackets:

In [13]:
```
s = {1, 2, 3, 3, 4}
print(s)
```

```
{1, 2, 3, 4}
```

In the above example, that the duplicate entry of 3 is ignored.

The unordered property of sets means that the order of their elements does not matter. Two sets with the same entries, but in a different order, are equal:

In [3]:
```
s1 = {1, 2, 3, 4}
s2 = {4, 3, 2, 1}
print(s1 == s2)
```

```
True
```

Since the ordering of elements does not matter, elements cannot be accessed with indices:

In [15]:
```
print(s1[0])
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-15-110e8284c01f> in <module>
----> 1 print(s1[0])

TypeError: 'set' object is not subscriptable
```

Operations on sets

Sets in Python can be manipulated using the same operations as sets in maths.

Example: The `in` keyword can be used to test whether an element is in a set:

In [1]:
```
s = {'red', 'blue', 'green'}
print('blue' in s)
```

```
True
```

We can also compute the union, intersection, difference, and the symmetric difference of two sets.

These operations will be explored in the exercises

Dictionaries

Dictionaries (or dicts) are similar to lists, except the values are indexed by **keys** rather than integers

- You can think of dicts as key-value pairs
- The keys must be *unique* and *immutable* (e.g. strings, ints)

The syntax of a dict is:

```
dict = {Key1:Value1, Key2:Value2, ... }
```

We can access Value2 using the syntax

```
dict[Key2]
```

Dictionary example

Let's create a dictionary of country calling codes

Key	Value
UK	44
Canada	1
Spain	34
Kazakhstan	7

In [2]:
```
# creating the dictionary
countryCodes = {'UK':44, 'Canada':1, 'Spain':34, 'Kazakhstan': 7}

# let's access the country code for the UK using the key 'UK'
print(countryCodes['UK'])
```

```
44
```

Summary

Python contains several varibles types that can be used to store and operate on collections of data

Type	Ordered	Mutable
List	Yes (using ints)	Yes
Tuple	Yes (using ints)	No
Set	No	Yes
Dictionary	Yes (using keys)	Yes

Values in **ordered types** can be accessed using an index, e.g. list[0], dict[key]

Mutable variables can be modified after they are created; **immutable** variables cannot