Compound Data Structures

Overview

- So far, variables have stored one value.
- What if we want to store multiple related values?
    - We *could* just have separate variables for each…
    - But what if we don't know how many values there will be?
    - And what if we want to work with these values inside a loop?
- *Compound data structures* are data types that can store lots of values.
- There are several types of compound data structure in Python.
    - We will primarily focus on Lists.
    - We will look briefly at Sets, Tuples, and Dictionaries at the end.

Lists

- Exactly what it says on the tin: a list of data!
    - Similar to *arrays* in other languages, but somewhat more versatile in Python.
- A list is simply a comma-separated list of values, inside square brackets.
    - `["Alice", "Bob", "Carol", "Dave"]`
    - `[4, 8, 15, 16, 23, 42]`
- Unlike arrays in many languages, in Python, it is possible to mix data-types inside the list.
    - `["Emily", 42, 13, "Fred", True]`
- We can create an empty list using just square brackets.
    - `[]`
- Like other data types, we can store lists in variables.
    - `friends = ["Alice", "Bob", "Carol", "Dave"]`
    - `numbers = [4, 8, 15, 16, 23, 42]`

Operations on Lists

- There are lots of operations we can perform on lists, to build expressions using them.
- To access a specific item in a list, we can use square brackets with the position.
    - `friends[1] == "Bob"`
    - Note that lists are indexed from 0, not 1!
    - Also, you cannot access an element that has not been defined!
        - `friends[50]` will crash.
    - We can use this notation for both accessing and modifying items in the list.
    - `friends[3] = "Daniella"`
- Exercise:
    - Create a list containing the names of seven colours. Output the first, third, and last elements of the list.
    - ```
      colours = ["red", "yellow", "pink", "green", "orange", "purple", "blue"]
      print(colours[0])
      print(colours[2])
      print(colours[6])
      ```
    - Note carefully that indexing starts at 0! The first element is 0, and the last element is 6 - one less than the length of the list.
      ```

- To extract a sub-list of the list, we can use a similar syntax, specifying two positions in the list, separated by a colon.
  - `numbers[2:4] == [15, 16]`
- Lists can be attached - *concatenated* - using the addition operator.
  - `["cat", "dog"] + ["fish", "rabbit", "hamster"] == ["cat", "dog", "fish", "rabbit", "hamster"]`
- We can also repeat lists using the multiplication operator.
  - `["cat", "dog"] * 3 == ["cat", "dog", "cat", "dog", "cat", "dog"]`
- Exercise:
  - Extend the previous exercise to ask the user for a colour, and add it to the list. Then output the whole list.
  - Hint: Think carefully about how to add a string to a list. We only know how to stick together two lists!
  - ```
    colours = ["red", "yellow", "pink", "green", "orange", "purple", "blue"]
    newcolour = input("Enter a new colour: ")
    colours = colours + [newcolour]
    print(colours)
    ```
  - Note how we had to make a new list out of the `newcolour` variable. Trying to add a list and a string directly would not have worked.
  - We could also have used the `+=` operator to make this shorter.
- We can find the length of a list using `len`.
  - `len(["cat", "dog"]) == 2`
- We can test if a value is inside a list using `in`.
  - `"fish" in ["cat", "dog", "fish", "rabbit", "hamster"] == True`
  - `"sheep" in ["cat", "dog", "fish", "rabbit", "hamster"] == False`
- Exercise:
  - Extend the previous exercises further by asking the user for a colour, and telling them if it is in the list.
  - ```
    colours = ["red", "yellow", "pink", "green", "orange", "purple", "blue"]
    testcolour = input("Enter a colour: ")
    if testcolour in colours:
        print("That colour is in the list.")
    else:
        print("That colour is not in the list.")
    ```
- If the length of the list is fixed, we can *decompose* the list into individual variables.
  - ```
    x, y, z = [4, "red", "yellow"]
    print(x)        # Outputs 4
    print(y)        # Outputs red
    print(z)        # Outputs yellow
    ```
- There are some more advanced list operations (treating them as objects), but the above will be enough for now.
    ```

Iterating on Lists

- Recall that a *loop* will run a piece of code multiple times.
- We previously looked at `while` loops, which iterate a piece of code so long as a condition is true.
- We could use a `while` loop to iterate over the contents of a list, using a counter.
    - ```
      animals = ["cat", "dog", "fish", "rabbit", "hamster"]
      i = 0
      while i < len(animals):
          print(animals[i])
          i += 1
      ```
    - But this is cumbersome.
- A `for` loop is another kind of loop, for iterating over lists.
    - ```
      for variable in list:
          ... code block ...
      ```
    - The code block will be run once for every item in the list, in order, with each list item being stored in the variable.
    - ```
      animals = ["cat", "dog", "fish", "rabbit", "hamster"]
      for x in animals:
           print(x)
      ```
- Exercise:
    - Create a list of numbers. Calculate and output the sum of all the numbers in the list.
    - ```
      numbers = [4, 8, 15, 16, 23, 42]
      total = 0
      for x in numbers:
          total += x
      print(total)
      ```

Range Lists

- Python provides a special function called `range`, which generates a list of integers.
- We can use this to make counting loops more easily, since `while` loops are cumbersome.
- ```
  for x in range(10):
      print(x)
  ```

Other Data Structures

- Lists are the most useful and versatile compound data structures, but there are some useful variants to be aware of.
    - We won't cover these in much detail.
    - You can do more reading and experiments with them in your own time if you want to.
- *Sets* are like lists, but each value can only appear once.
    - They are defined using curly braces rather than square brackets.
    - If you attempt to create a set with duplicate values, the duplicates will be removed.
  ```

    - Values are not stored in a fixed order.
    - ```
      numbers = {1, 2, 4, 8, 2, 3, 2, 4}
      print(numbers)
      # Output: {8, 1, 2, 3, 4}
      ```
- *Tuples* are like lists, but cannot be modified once created.
    - They are defined using regular brackets rather than square brackets.
    - They are useful when you want to remind yourself that a set of values should be fixed.
    - We have been secretly using them for string formatting!
    - ```
      parameters = ("Alex", 14)
      print("My name is %s and I am %d years old." %
      parameters)
      ```
- *Dictionaries* let you specify a 'key' for each value stored.
    - They are defined using curly braces rather than square brackets.
    - The keys can be of any data-type you want.
    - Values stored in a dictionary are referenced by the key, not their position in the dictionary (as with lists).
    - Key-value pairs are not stored in a fixed order.
    - For each key-value pair, the key comes first, followed by a colon, followed by the value.
    - ```
      ages = {"Alice": 26, "Bob": 60, "Carol": 30, "Dave": 16}
      print(ages["Bob"])
      # Output: 60
      ```
    - When using a for loop with a dictionary, a tuple containing each key-value pair is yielded, rather than just the value. We have to use a special syntax to get the items, though (the meaning of which will be covered in a later session).
    - ```
      for (name, age) in ages.items():
          print(name)
      ```
        - Note that we use `ages.items()` rather than just `ages`.
        - This will output just the names.
    - ```
      for people in ages.items():
          print("%s is %d years old." % people)
      ```
        - Here we have been clever, and used the key-value tuple itself.

Exercise: Comparing Lists

- Create two lists, containing whatever you want.
- Write a program that outputs which values are in both lists, and which values are in one list but not the other.
- ```
  list1 = [2, 4, 6, 8, 10, 12, 14, 16, 18, 20]
  list2 = [3, 6, 9, 12, 15, 18]
  print("Items in both lists:")
  for x in list1:
      if x in list2: print(x)
  print("Items in list 1 only:")
  for x in list1:
  ```

```
    if not (x in list2): print(x)
print("Items in list 2 only:")
for x in list2:
    if not (x in list1): print(x)
```

Exercise: Bubble Sort

- Bubble sort is an inefficient but simple list sorting algorithm.
- It works as follows:
  - Iterate across each element of the list, comparing it to the following element.
    - If the two elements are out of order, swap them so they are in the correct order.
  - Once the end of the list has been reached, go back to the start and do it again.
  - Keep going until the list is sorted - i.e. until no swaps are made during one iteration across the list.
- Wikipedia has a good article on Bubble Sort if you are lost.
- Implement Bubble Sort!
- Hint: In order to compare adjacent elements of the list, you probably want to iterate a counter using `while` and access list elements directly (i.e. `mylist[counter]`), rather than using a `for` loop.
- Answer:

```
# The list to sort
mylist = [6, 2, 7, 3, 42, 6, 1]

# Repeat until the list is sorted
done = False
while not done:
    swapped = False
    # Iterate over each item in the list except the last
    for i in range(len(mylist) - 1):
      # Compare this element and the next one
        if mylist[i] > mylist[i+1]:
            # The elements are out of order - swap them
            temp = mylist[i]
            mylist[i] = mylist[i+1]
            mylist[i+1] = temp
            swapped = True
    # If no swaps were performed then we are finished
    done = not swapped

# Output the result
print(mylist)
```

Summary

- Compound data structures are good for storing datasets that we want to work on.
- We can modify and combine these structures with operators.

- We can iterate over these structures with loops.

Next lesson: Functions