

INF1-IP

Introduction to Programming

Lecture 5 – Collections 1: Lists, sets and dictionaries



> Data structures

Data structures are ways to organise and store data so your program can use it easily.

Python has four main built-in data structures:

- **List:** An ordered collection you can change. It allows duplicates.
- **Tuple:** An ordered collection you cannot change after creating it.
- **Dictionary:** Stores pairs of keys and values, so you can look up a value using its key.
- **Set:** A collection of unique items, with no guaranteed order.

Today we will look at lists, dictionaries and sets. We'll look at tuples in the next lecture on Collections.

> Lists

Lists are an extremely common data structure

- Order is meaningful
- Lists are created using `[]` and items are separated by a `,`

```
my_list = [item_1, item_2, ..., item_n]
```

- It is common for all items in the list to have the same data type, but this is not necessary

! Zero-based indexing

We can find the index (where something is in the list) using [] notation.

Note: indexes start at 0, not 1

```
my_courses = ["CRISP", "IP", "CAFI"]  
  
print(my_courses[0]) returns "CRISP"  
print(my_courses[1]) returns "IP"  
print(my_courses[2]) returns "CAFI"
```

This can easily lead to mistakes!

```
my_list = [item_1, item_2, ..., item_n]  
  
my_list[n] → IndexError!!
```

> Indexing

You can also use indexing to *count backwards* from the end of the list:

```
my_courses = ["CRISP", "IP", "CAFI"]  
print(my_courses[-1]) returns "CAFI"  
  
print(my_courses[-3]) returns "CRISP"
```

Note that we do *not* begin at 0 when counting backwards.

```
list[1] == list[-n]
```

 where n is the length of the list.

> Indexing

You can replace items in a list by assigning something new to their index.

```
my_courses = ["CRISP", "IP", "CAFI"]  
my_courses[1] = "French1"  
  
print(my_courses) returns ["CRISP", "French1", "CAFI"]
```

> Methods and functions for lists

- To add an item to the end of a list, use `.append()`

```
my_courses = ["CRISP", "IP", "CAFI"]  
my_courses.append("French1")  
  
print(my_courses) returns ["CRISP", "IP", "CAFI", "French1"]
```

- `len(list)` returns the length of a list

```
my_courses = ["CRISP", "IP", "CAFI"]  
print(len(my_courses)) returns 3
```

> Methods and functions for lists

- `.insert(index, item)` inserts *item* in the list at position *index*.

Everything after *index* then increases its index by 1.

```
my_courses = ["CRISP", "IP", "CAFI"]  
my_courses.insert(1, "French1")  
  
print(my_courses) returns ["CRISP", "French1", "IP", "CAFI"]
```

- `.remove(item)` removes *the first occurrence* of an item

```
my_courses = ["IP", "CRISP", "IP", "CAFI"]  
my_courses.remove("IP")  
  
print(my_courses) returns ["CRISP", "IP", "CAFI"]
```

> Methods and functions for lists

- `.extend()` adds one list to the end of another

```
my_courses = ["CRISP", "IP", "CAFI"]  
my_sem2_courses = ["FC", "LA", "Philosophy1"]  
  
my_courses.extend(my_sem2_courses)  
  
print(my_courses) returns ["CRISP", "IP", "CAFI", "FC", "LA", "Philosophy1"]
```

- You can do this with `.append()`, but then the new list is a single list item:

```
my_courses = ["CRISP", "IP", "CAFI"]  
my_sem2_courses = ["FC", "LA", "Philosophy1"]  
  
my_courses.append(my_sem2_courses)  
  
print(my_courses) returns ["CRISP", "IP", "CAFI", ["FC", "LA", "Philosophy1"]]
```

> Methods and functions for lists

- `.pop(index)` removes the item at index *index*

```
my_courses = ["CRISP", "IP", "CAFI"]  
  
my_courses.pop(1)  
  
print(my_courses) returns ["CRISP", "CAFI"]
```

- `.pop()` removes the final item

```
my_courses = ["CRISP", "IP", "CAFI"]  
  
my_courses.pop()  
  
print(my_courses) returns ["CRISP", "IP"]
```

> Strings as iterables

Strings are not lists – but because we can *iterate* over them, it is sometimes useful to treat them as if they were.

List methods and functions to do with *iteration* can also be used on strings:

```
my_string = "This is a string"  
print(my_string[3]) returns "s"  
print(len(my_string)) returns 16
```

List methods that are not to do with iteration - e.g., `pop()`, `append()` or `insert()` - **cannot** be used with strings.

> Dictionaries

In lists, things are identified by their position in the list → their *index*

In a dictionary, each item has a unique index called a *key*.

A dictionary is a collection of *key-value pairs*. They are contained in curly braces and use colons to separate keys and values.

```
ip_dictionary = {"name": "Intro to Programming", "credits": 20, "semester": 1}
```

> Dictionaries

Although we use curly braces to define dictionaries, we still use square brackets to retrieve information. *Keys* are used to return *values*.

```
ip_dictionary = {"name": "Intro to Programming", "credits": 20, "semester": 1}

print(ip_dictionary["name"]) returns "Intro to Programming"
print(ip_dictionary["credits"]) returns 20
```

> Dictionaries – layout

Remember that in Python, layout can be meaningful – we have already seen that *indentation* is important.

But with dictionaries, defining them over multiple lines or one is not meaningful. The two layouts below are the same. Defining them over multiple lines can make them more readable.

```
ip_dictionary = {"name": "Intro to Programming", "credits": 20, "semester": 1}

ip_dictionary = {
    "name": "Intro to Programming",
    "credits": 20,
    "semester": 1,
}
```

> Dictionaries – adding and changing items

We can assign a new value to a key in the same way we assign a new value to an index in a list. Unlike with lists, we can directly assign new key-value pairs to dictionaries.

```
ip_dictionary = {"name": "Intro to Programming", "credits": 20, "semester": 1}

ip_dictionary["name"] = "Inf1-IP"      # change the value of an existing key
ip_dictionary["year"] = 1              # add a new key-value pair to the dictionary

print(ip_dictionary) returns {"name": "Inf1-IP", "credits": 20, "semester": 1, "year": 1}
```

> Dictionaries – updating items

You can use `.update()` to change the value assigned to a key or to add a new key:

```
menu = {  
    "tea": 1.80,  
    "coffee": 4.00,  
    "sandwich": 6.50,  
    "cake": 3.50  
}  
  
menu.update({"coffee": 4.50, "juice": 3.50})  
  
print(menu) returns  
{"tea": 1.80, "coffee": 4.50, "sandwich": 6.50, "cake": 3.50, "juice": 3.50}
```

> Lists vs Dictionaries

When should you use a dictionary vs a list?

- Use lists if order is meaningful and you want to extract information based on position
 - Lists are easier because you don't have to add keys. But you can only identify an item by its position in the list.
- Use a dictionary if you want to access an item *regardless* of its position, and it's useful to have keys
 - Dictionaries mean you don't have to know what order things are in – you just need to know their key. Keys can be strings or numbers.
 - e.g., in our example, the different aspects of the course work best as a list as you don't have to remember what position, e.g., "name" is in – you can use the key to retrieve it

> Sets

Sets are used when you want a collection of things to which something applies – there is no order and no duplicates.

Our list, `my_courses`, could also be a set.

```
my_courses = {"CRISP", "IP", "CAFI"}
```

We just want to know what these courses are – we never need to list the same one more than once and we don't want duplicates.

> Lists → Sets

We can convert a list to a set using `set()`. You can convert back to a list using `list()`, but:

- Any duplicates will be removed when you convert to a set because sets do not contain duplicates
- The order in a set is random, so the reconverted list may not be in the same order.

```
my_courses = ["CRISP", "IP", "CAFI", "IP"]      # my_courses is a list
my_courses = set(my_courses)                   # now my_courses is a set
my_courses = list(my_courses)                  # now my_courses is a list again

print(my_courses) returns ["CAFI", "IP", "CRISP"] # or a different order -
                                                    # we can't tell
```

> Lists, sets and dictionaries

Songs in a playlist are better as a list because you want them in the right order (duplicates probably don't matter here).

- Better as a set: artists in the playlist - because you just want to know who is on there, you don't need an order or duplicates)
- Better as a dictionary: songs with their durations – because you can use song titles as keys to extract durations.

> Lists, sets and dictionaries

Cities mapped to their average daily temperature work best as a dictionary – the cities are the keys, the temperatures are the values.

- Better as a set: cities you have visited on a trip (you might visit the same city twice, but if you are only recorded where you have been you don't need duplication, and order doesn't matter).
- Better as a list: daily temperature readings in a city – these have an ordering and there could be repetition.

> Lists, sets and dictionaries

Rule of thumb:

List: "I need a sequence; order and duplicates may matter."

Set: "I need unique items; order doesn't matter."

Dictionary: "I need to associate each key with a value."

The main advantage of a set over a dictionary is that membership checking can be done very quickly. It is better to use a set if you don't need the functionality of a list.

> Collection membership and loops

Collections are very often used in loops: we loop through a collection and do something to every member, or to certain members.

```
my_courses = ["CRISP", "IP", "CAFI"]  
for course in my_courses:  
    print(f"I am studying {course}")
```

returns

```
"I am studying CRISP"      # predictable order because it's a list  
"I am studying IP"  
"I am studying CAFI"
```

> Collection membership and loops

Collections are very often used in loops: we loop through a collection and do something to every member, or to certain members.

```
my_courses = {"CRISP", "IP", "CAFI"}  
for course in my_courses:  
    if course.startswith("C"):  
        print(f"I am studying {course}")
```

returns

```
"I am studying CAFI"           # could be any order -  
"I am studying CRISP"         # we can't predict order because it's a set
```

> Collection membership and loops

When looping through dictionaries, the default is to loop through keys.

```
ip_dictionary = {"name": "Intro to Programming", "credits": 20, "semester": 1}
for key in ip_dictionary:                # we don't tell it what to loop
    print(f"This is a key: {key}")        # through - it assumes keys
```

returns

```
"This is a key": "name"
"This is a key": "credits"
"This is a key": "semester"
```

> Collection membership and loops

We can also loop through values, but we need to specify this.

```
ip_dictionary = {"name": "Intro to Programming", "credits": 20, "semester": 1}
for value in ip_dictionary.values():           # we have to tell it what to loop
    print(f"This is a value: {value}")        # through if we want it to be values
```

returns

```
"This is a value": "Intro to Programming"
"This is a value": 20
"This is a value": 1
```

> Collection membership and loops

With collections, you can also use `in` and `not in` to check for membership:

```
planets = {"Mercury", "Venus", "Earth", "Mars", "Jupiter", "Saturn", "Uranus", "Neptune"}

if "Pluto" in planets:
    print("An old astronomy textbook!")
else:
    print("Pluto is classified as a dwarf planet.")
```

> Collection membership and loops

With collections, you can also use `in` and `not in` to check for membership:

```
dragon_hoard = {
    "Golden crown": "Treasure",
    "Ruby necklace": "Treasure",
    "Enchanted shield": "Magic",
    "Emergency biscuit": "Snack"
}

if "Emergency biscuit" not in dragon_hoard or "Magic" not in dragon_hoard.values():
    print("The dragon refuses to leave without a snack and something magical.")
else:
    print("Adventure awaits! The dragon is ready.")
```

> Mutability and copying collections

Remember when we were looking at string manipulation:

```
message = "hello"  
message.upper()  
print(message)  
  
# Output = hello
```

Applying a method to the string `method` did not change the string – we had to assign the change to a new string:

```
message = "hello"  
message = message.upper()  
print(message)  
  
# Output = HELLO
```

> Mutability and copying collections

This is because strings are *immutable* – which means that once they have been created they can't be changed. Basic data types (int, float, bool, str) are all immutable.

But collections are *mutable* – which means they **can** be changed.

```
message = "hello"  
message.upper()  
print(message)
```

```
# Output = hello
```

```
message_list = ["hello", "goodbye"]  
message_list.append("still here")  
print(message_list)
```

```
# Output = ["hello", "goodbye", "still here"]
```

This is to do with how things are stored in memory – we'll look more at this later.

> Mutability and copying collections

What happens to mutable objects when we assign a name to a value that already has a name → *they point to the same object* and therefore behave the same.

```
my_list = ["hello", "goodbye"]  
new_list = my_list  
my_list.append("still here?")  
print(new_list)  
  
# Output =  
["hello", "goodbye", "still here?"]
```

*# new_list now points to my_list
we change my_list and not new_list
but new_list is just a pointer to
my_list, so it changes too*

We can use `my_list` and `new_list` interchangeably – there is no difference.

> Mutability and copying collections

But we don't always want this – sometimes we want to create a copy of a collection that we can then treat differently. For this, we use `copy()`.

```
my_list = ["hello", "goodbye"]
new_list = my_list.copy()

my_list.append("still here?")
print(new_list)

# Output =
["hello", "goodbye"]
```

*# new_list is a different list that
has the same values as my_list
we change my_list and not new_list*

*# now changing my_list does not change
new_list*

> Summary

In this lecture, we covered:

- Storing data in lists, sets, and dictionaries
- Creating, accessing, and updating each type of collection
- Looping over elements in each type of collection
- Different ways of copying and changing each type of collection