ILP 2022 – W3S2-3
The list type

Matthieu DE MARI – Singapore University of Technology and Design

Outline (Week3, Session2-3 – W3S2-3)

- The list type (in detail)
- Length of a list
- Indexing, slicing, updating and traversing a list
- Lists of lists
- Membership
- Concatenation, repetition and insertion
- Removing elements from a list
- Advanced methods and RTFM!

The list type

- Let us first discuss a new type of objects, called **lists**.
- **Definition (lists):** a **list** is a **sequence** of several variable elements, listed in order, between **brackets** and separated by **commas**.
- It can contain variables of any types (int, float, string, etc.).
- List can also contain mixed types of variables.

```
1 a_list = [0, 1, 2, 3, 4]
2 print(a_list)
3 print(type(a_list))
```

```
[0, 1, 2, 3, 4]
<class 'list'>
```

```
1 another_list = ['a', 'b', 'c', 'd']
2 print(another_list)
```

```
['a', 'b', 'c', 'd']
```

```
1 a_float = 3.14
2 an_int = 10
3 a_string = 'Hello'
4 mixed_list = [a_float, an_int, a_string]
5 print(mixed_list)
```

```
[3.14, 10, 'Hello']
```

Length of a list

- **Definition (length of a list):** The **length** of a list consists of the **number of elements in the list**.

- It can be simply computed with the **len()** function, which returns **a positive integer** corresponding to the **number of elements** in the list.

```
1 a_list = [0, 1, 2, 3, 4]
2 print(len(a_list))
```

```
5
```

```
1 another_list = ['a', 'b', 'c', 'd']
2 print(len(another_list))
```

```
4
```

```
1 an_empty_list = []
2 print(an_empty_list)
3 print(len(an_empty_list))
```

```
[]
0
```

Indexing/Accessing the elements of a list

- The elements in a list are **indexed**, with integers.
- They can be accessed using the **[]** notation, with an **index** value.

```
1 my_list = ['a', 'b', 'c', 'd']
2 print(my_list)
```

```
['a', 'b', 'c', 'd']
```

```
1 print(my_list[0])
```

```
a
```

```
1 print(my_list[1])
```

```
b
```

```
1 print(my_list[3])
```

```
d
```

```
1 print(my_list[4])
```

```
---------------------------------------------------------------------------
IndexError                                Traceback (most recent call last)
<ipython-input-11-df42a28462c1> in <module>
----> 1 print(my_list[4])

IndexError: list index out of range
```

Indexing/Accessing the elements of a list

- The elements in a list are **indexed**, with integers.
- They can be accessed using the **[]** notation, with an **index** value.
  - The **first element** of the list has the **index 0 (we start counting from 0 in Python!)**
  - the **second element** then has index $\mathbf{1}$, etc.
  - The last element has index $\boldsymbol{n} - \mathbf{1}$ (not $\boldsymbol{n}$!), with $\boldsymbol{n}$ being the **length** of the list.

```
1 my_list = ['a', 'b', 'c', 'd']
2 print(my_list)
```

```
['a', 'b', 'c', 'd']
```

```
1 print(my_list[0])
```

```
a
```

```
1 print(my_list[1])
```

```
b
```

```
1 print(my_list[3])
```

```
d
```

```
1 print(my_list[4])
```

```
---------------------------------------------------------------------------
IndexError                                Traceback (most recent call last)
<ipython-input-11-df42a28462c1> in <module>
----> 1 print(my_list[4])

IndexError: list index out of range
```

Indexing/Accessing the elements of a list

- When indexing/accessing elements in a list, we can also use **negative indexes**.
  - -1 refers to the **last element** of the list**,**
  - -2 the **second to last element**, etc.
  - And $-n$ is therefore the index of the **first element**.

```
1 my_list = ['a', 'b', 'c', 'd']
2 print(my_list)
```

```
['a', 'b', 'c', 'd']
```

```
1 print(my_list[-1])
```

```
1 print(my_list[-2])
```

```
c
```

```
1 print(my_list[-4])
```

```
a
```

```
1 print(my_list[-5])
```

```
---------------------------------------------------------------------------
IndexError                                Traceback (most recent call last)
<ipython-input-14-04673c904ebf> in <module>
----> 1 print(my_list[-5])

IndexError: list index out of range
```

Slicing a list

- You can retrieve a **subset** of the list, by specifying to indexes, separated with a colon (**:**) symbol.

```
1 my_list = [1, 4, 9, 14, 15, 16, 27, 38, 49, 50]
2 print(my_list)
```

```
[1, 4, 9, 14, 15, 16, 27, 38, 49, 50]
```

```
1 print(my_list[0:4])
```

```
[1, 4, 9, 14]
```

```
1 print(my_list[6:8])
```

```
[27, 38]
```

```
1 print(my_list[-7:-3])
```

```
[14, 15, 16, 27]
```

```
1 print(my_list[4:-2])
```

```
[15, 16, 27, 38]
```

Slicing a list

- You can retrieve a **subset** of the list, by specifying to indexes, separated with a colon (**:**) symbol.
- **Indexing** with **[a:b]** returns another list as a result, containing all elements of the original list:
  - Starting **from index a** (with a included),
  - All the way **to index b** (with b **not** included).

```
1 my_list = [1, 4, 9, 14, 15, 16, 27, 38, 49, 50]
2 print(my_list)
```

```
[1, 4, 9, 14, 15, 16, 27, 38, 49, 50]
```

```
1 print(my_list[0:4])
```

```
[1, 4, 9, 14]
```

```
1 print(my_list[6:8])
```

```
[27, 38]
```

```
1 print(my_list[-7:-3])
```

```
[14, 15, 16, 27]
```

```
1 print(my_list[4:-2])
```

```
[15, 16, 27, 38]
```

Slicing a list

- **Omitting** a value in **[a:b]**, either a or b, means **“take all”.**

```
1 my_list = [1, 4, 9, 14, 15, 16, 27, 38, 49, 50]
2 print(my_list)
```

```
[1, 4, 9, 14, 15, 16, 27, 38, 49, 50]
```

```
1 print(my_list[:5])
```

```
[1, 4, 9, 14, 15]
```

```
1 print(my_list[7:])
```

```
[38, 49, 50]
```

Slicing a list

- **Omitting** a value in **[a:b]**, either a or b, means **“take all”.**
- **[:b]** means:
  - **all elements from the beginning** of the list,
  - **until index b** (with b **not included**).

```
1 my_list = [1, 4, 9, 14, 15, 16, 27, 38, 49, 50]
2 print(my_list)
```

```
[1, 4, 9, 14, 15, 16, 27, 38, 49, 50]
```

```
1 print(my_list[:5])
```

```
[1, 4, 9, 14, 15]
```

```
1 print(my_list[7:])
```

```
[38, 49, 50]
```

Slicing a list

- **Omitting** a value in **[a:b]**, either a or b, means **“take all”.**
- **[:b]** means:
  - **all elements from the beginning** of the list,
  - **until index b** (with b **not included**).
- **[a:]** means:
  - all element **starting from index a**,
  - **until the end of the list** (last element included)**.**

```
1 my_list = [1, 4, 9, 14, 15, 16, 27, 38, 49, 50]
2 print(my_list)
```

```
[1, 4, 9, 14, 15, 16, 27, 38, 49, 50]
```

```
1 print(my_list[:5])
```

```
[1, 4, 9, 14, 15]
```

```
1 print(my_list[7:])
```

```
[38, 49, 50]
```

Slicing a list

- **Indexing** with **[a:b:c]** returns another list as a result, containing all elements of the original list:
  - Starting **from index a** (with a included),
  - All the way **to index b** (with b **not** included),
  - With **steps** of **size c.**

```
1 my_list = [1, 4, 9, 14, 15, 16, 27, 38, 49, 50]
2 print(my_list)
```

```
[1, 4, 9, 14, 15, 16, 27, 38, 49, 50]
```

```
1 print(my_list[:5])
```

```
[1, 4, 9, 14, 15]
```

```
1 print(my_list[7:])
```

```
[38, 49, 50]
```

```
1 print(my_list[0:6:2])
```

```
[1, 9, 15]
```

```
1 print(my_list[-1:-7:-3])
```

```
[50, 27]
```

Slicing a list

- **Indexing** with **[a:b:c]** returns another list as a result, containing all elements of the original list:
  - Starting **from index a** (with a included),
  - All the way **to index b** (with b **not** included),
  - With **steps** of **size c.**

Question: Which one of the statements below allows to display all the elements of a list my_list, in reverse order?

A. print(my_list[-1:-len(my_list)])
B. print(my_list[-len(my_list):-1])
C. print(my_list[-1:(-len(my_list)):-1])
D. print(my_list[-1:(-len(my_list)-1):-1])

Slicing a list

- **Indexing** with **[a:b:c]** returns another list as a result, containing all elements of the original list:
  - Starting **from index a** (with a included),
  - All the way **to index b** (with b **not** included),
  - With **steps** of **size c.**

Question: Which one of the statements below allows to display all the elements of a list my_list, in reverse order?

A. print(my_list[-1:-len(my_list)])
B. print(my_list[-len(my_list):-1])
C. print(my_list[-1:(-len(my_list)):-1])
D. **print(my_list[-1:(-len(my_list)-1):-1])**

Slicing a list

```python
# Original list
my_list = [1, 4, 9, 14, 15, 16, 27, 38, 49, 50]
print(my_list)
# A
print(my_list[-1:(-len(my_list))])
# B
print(my_list[-len(my_list):-1])
# C
print(my_list[-1:(-len(my_list)):-1])
# D
print(my_list[-1:(-len(my_list) - 1):-1])
```

```
[1, 4, 9, 14, 15, 16, 27, 38, 49, 50]
[]
[1, 4, 9, 14, 15, 16, 27, 38, 49]
[50, 49, 38, 27, 16, 15, 14, 9, 4]
[50, 49, 38, 27, 16, 15, 14, 9, 4, 1]
```

Slicing a list

```
1  # Original list
2  my_list = [1, 4, 9, 14, 15, 16, 27, 38, 49, 50]
3  print(my_list)
4  # A
5  print(my_list[-1:(-len(my_list))])
6  # B
7  print(my_list[-len(my_list):-1])
8  # C
9  print(my_list[-1:(-len(my_list)):-1])
10 # D
11 print(my_list[-1:(-len(my_list) - 1):-1])
```

```
[1, 4, 9, 14, 15, 16, 27, 38, 49, 50]
[]
[1, 4, 9, 14, 15, 16, 27, 38, 49]
[50, 49, 38, 27, 16, 15, 14, 9, 4]
[50, 49, 38, 27, 16, 15, 14, 9, 4, 1]
```

```
1  print(my_list[::-1])
```

```
[50, 49, 38, 27, 16, 15, 14, 9, 4, 1]
```

Activity 1 - Distributing a deck of cards

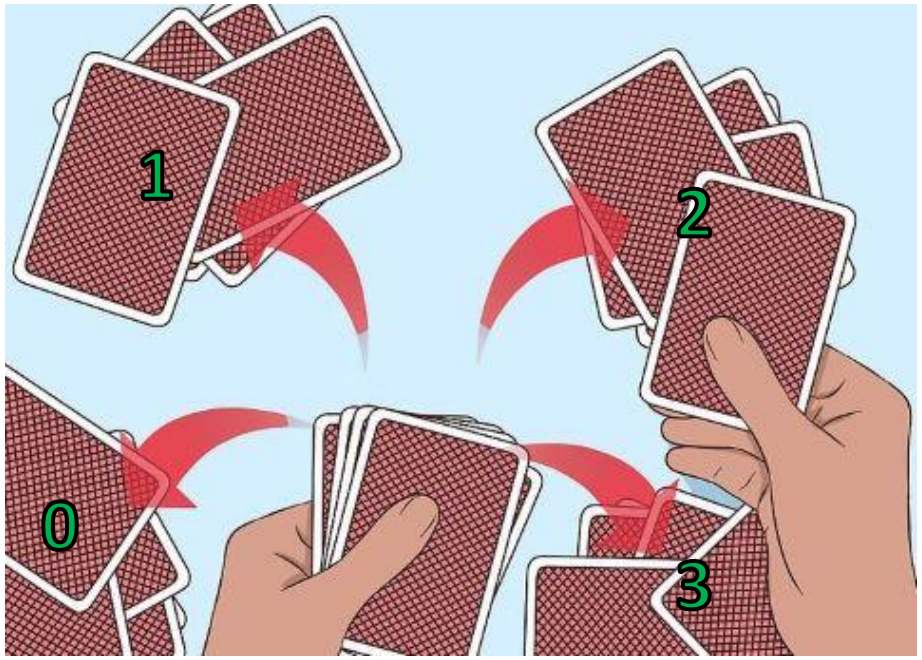
- Let us consider a standard, shuffled deck of cards, defined as a list.

```
cards_deck = ['3 of Spades', 'Jack of Diamonds', '2 of Hearts', '4 of Clubs', '8 of Clubs', \
              '4 of Spades', '9 of Hearts', 'Jack of Spades', '2 of Diamonds', '9 of Spades', \
              'Ace of Hearts', 'Jack of Hearts', '4 of Hearts', 'Queen of Clubs', '9 of Clubs', \
              'King of Hearts', 'Ace of Clubs', 'Ace of Diamonds', '8 of Diamonds', 'King of Diamonds', \
              '6 of Diamonds', '7 of Spades', 'King of Clubs', '10 of Clubs', '9 of Diamonds', \
              '3 of Hearts', '7 of Diamonds', '6 of Clubs', '4 of Diamonds', '3 of Diamonds', \
              '6 of Spades', '5 of Hearts', 'Jack of Clubs', 'King of Spades', '5 of Diamonds', \
              '10 of Spades', 'Queen of Hearts', '2 of Clubs', '2 of Spades', 'Ace of Spades', \
              'Queen of Spades', '5 of Clubs', '7 of Clubs', '6 of Hearts', '10 of Diamonds', \
              '10 of Hearts', 'Queen of Diamonds', '5 of Spades', '3 of Clubs', '7 of Hearts', \
              '8 of Hearts', '8 of Spades']
```

- In this activity, we will distribute the cards between several players.

Activity 1 - Distributing a deck of cards

- Let us now consider that we have $\boldsymbol{n}$ **players**.
  - The players are referred to as **player 0**, **player 1**, ..., **player (n-1).**
  - Players are seating around a table in a **clockwise** manner: player 1 is seating on the left of player 0, player 2 is seating on the left of player 1, etc.



Activity 1 - Distributing a deck of cards

Write a function **distribute_cards(),** which

- **receives** a shuffled deck of cards, **cards_deck**, as its first input parameter,

- **receives** a number of players seating at the table, **players_number**, as its second input parameter,

- **receives** an integer **player_index**, with value in [0, players_number - 1], as its third input parameter. This corresponds to the position in which you are sitting at the table.

- **receives** an integer **start_from**, with value in [0, players_number - 1], as its fourth input parameter. This value corresponds to the first player to receive a card.

Activity 1 - Distributing a deck of cards

The function **distribute_cards()** then

- **distributes all cards** in the deck, **one card at a time**, in **clockwise order**, between all players,

- **starting from the player** with index **start_from**.

- It is ok if some players have one more card than other players (for instance, if the number of cards is not divisible by number of players)

The function **distribute_cards()** returns a list of cards, **player_hand**, corresponding to the **hand** of player sitting in position **player_index**, after we are done distributing the cards.

Find a smart slicing for the deck!

Traversing lists (element-wise)

- You have seen how we can give lists to **for** statements to traverse and retrieve each element **one-by-one**.

1. The "**for element in list:**" approach is commonly referred to as traversing a list **element-wise**.

```
1 my_list = [1, 4, 9, 14, 15]
2 print(my_list)
```

```
[1, 4, 9, 14, 15]
```

```
1 # Element-wise
2 for element in my_list:
3     print("--")
4     print(element)
```

```
--
1
--
4
--
9
--
14
--
15
```

Traversing lists (index-wise)

- You have seen how we can give lists to **for** statements to traverse and retrieve each element **one-by-one**.

2. The "**for index in range(len(list)):**" approach is commonly referred to as traversing a list **index-wise**.

```
1  # Index-wise
2  for index in range(len(my_list)):
3      print("--")
4      print(index, my_list[index])
```

```
--
0 1
--
1 4
--
2 9
--
3 14
--
4 15
```

Traversing lists (index and element-wise)

- You have seen how we can give lists to **for** statements to traverse and retrieve each element **one-by-one**.

3. You have also seen how to use the **enumerate()** generator, and how it allows to retrieve **both the indexes and elements** at the same time, while traversing a list.

```
1  # Index and element-wise
2  for index, element in enumerate(my_list):
3      print("--")
4      print(index, element)
```

```
--
0 1
--
1 4
--
2 9
--
3 14
--
4 15
```

Traversing lists (while)

- You have seen how we can give lists to **for** statements to traverse and retrieve each element **one-by-one**.

4. You can also replace the **for** loop with a **while** loop.
   (**Not recommended** however!)

```
1  index = 0
2  while index < len(my_list):
3      print("--")
4      print(index, my_list[index])
5      index += 1
```

```
--
0 1
--
1 4
--
2 9
--
3 14
--
4 15
```

Updating an element in a list

- The indexing notation **(list[index])** can be used to update an element in the list.
- The accessed element receives (with the = sign symbol) a new value.
- A list element is just another variable after all.

```
1 my_list = [1, 4, 9, 14, 15]
2 print(my_list)
```

```
[1, 4, 9, 14, 15]
```

```
1 my_list[2] = 'Hello'
2 print(my_list)
```

```
[1, 4, 'Hello', 14, 15]
```

```
1 my_list[1:4] = 'Queen', 'of', 'Hearts'
2 print(my_list)
```

```
[1, 'Queen', 'of', 'Hearts', 15]
```

```
1 another_list = [7, 1, 9]
2 my_list[1] = another_list
3 print(my_list)
```

```
[1, [7, 1, 9], 'of', 'Hearts', 15]
```

Lists of lists

Observation: an element of a list can be another list!

- Careful with the **len()** function!
- If an element of a list is a list itself, it still only counts as 1 element!

```
1 my_list = [1, [7, 1, 9], 14, 15]
2 print(my_list)
```

```
[1, [7, 1, 9], 14, 15]
```

```
1 print(len(my_list))
```

```
4
```

```
1 print(my_list[0])
```

```
1
```

```
1 print(my_list[1])
```

```
[7, 1, 9]
```

```
1 print(len(my_list[1]))
```

```
3
```

Lists of lists

Observation: an element of a list can be another list!

- Careful with the **len()** function!
- If an element of a list is a list itself, it still only counts as 1 element!
- You can **use multiple indexing,** i.e. using [] several times in a row, to **access the elements of a sublist**.

```
1 my_list = [1, [7, 1, 9], 14, 15]
2 print(my_list)
```

```
[1, [7, 1, 9], 14, 15]
```

```
1 print(len(my_list))
```

```
4
```

```
1 print(my_list[0])
```

```
1
```

```
1 print(my_list[1])
```

```
[7, 1, 9]
```

```
1 print(len(my_list[1]))
```

```
3
```

```
1 print(my_list[1][0])
```

```
7
```

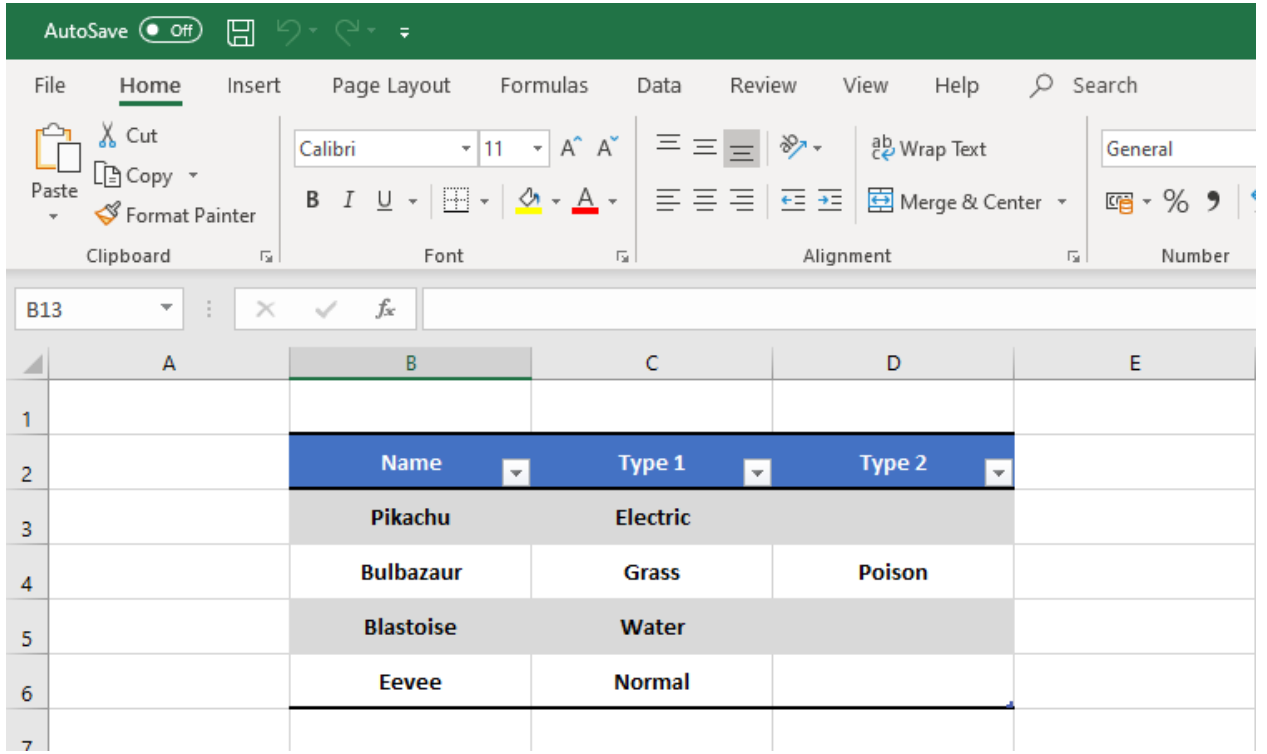
```
1 print(my_list[1][2])
```

```
9
```

Lists of lists

Lists of lists are commonly used in Python to represent **tables of values**.

Widely used in **Data Science**!




```python
my_list = [["Name", "Type 1", "Type 2"], # Columns labels
           ["Pikachu", "Electric", None], # Entry 1
           ["Bulbazaur", "Plant", "Poison"], # Entry 2
           ["Blastoise", "Water", None], # Entry 3
           ["Eevee", "Normal", None]] # Entry 4
print(my_list)
```

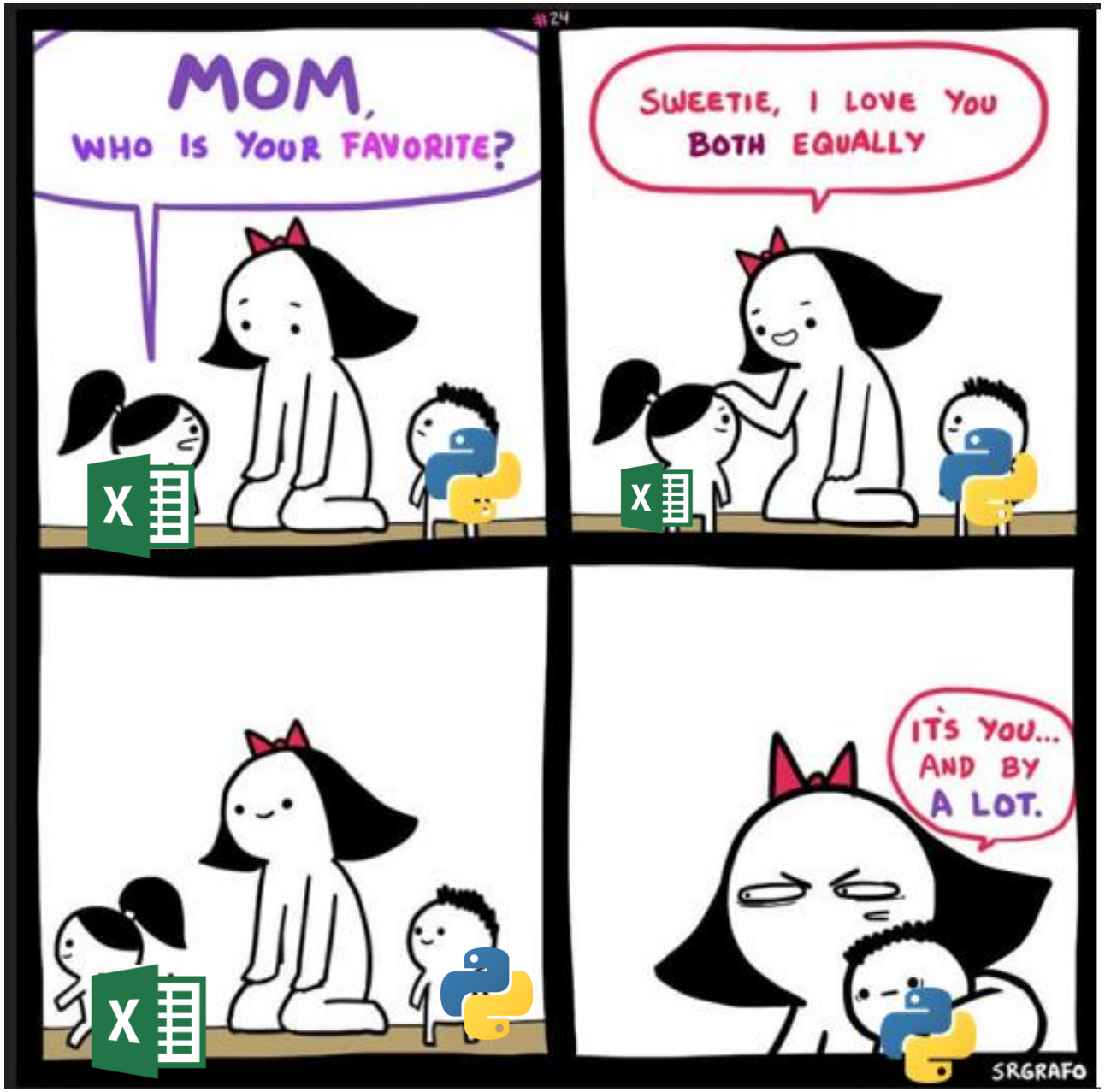
```
[['Name', 'Type 1', 'Type 2'], ['Pikachu', 'Electric', None], ['Bulbazaur', 'Plant', 'Poison'], ['Blastoise', 'Water
', None], ['Eevee', 'Normal', None]]
```

Lists of lists

Lists of lists are commonly used in Python to represent **tables of values**.

Widely used in **Data Science**!

Great news: this means that you will no longer need Excel!



Lists of lists

Traversing lists of lists requires **nested for** loops.

The **first for loop** browses through the **lines**, which are the **sublists of the list**, one line at a time.

```
for line in my_list:
    print("-----")
    print("Line: ", line)
    for element in line:
        print("-")
        print(element)
```

```
-----
Line:  ['Name', 'Type 1', 'Type 2']
-
Name
-
Type 1
-
Type 2
-----
Line:  ['Pikachu', 'Electric', None]
-
Pikachu
-
Electric
-
None
-----
Line:  ['Bulbazaur', 'Plant', 'Poison']
```

Lists of lists

Traversing lists of lists requires **nested for** loops.

The **first for loop** browses through the **lines**, which are the **sublists of the list**, one line at a time.

The **second for loop** browses through the **elements in the sublists**, one element at a time.

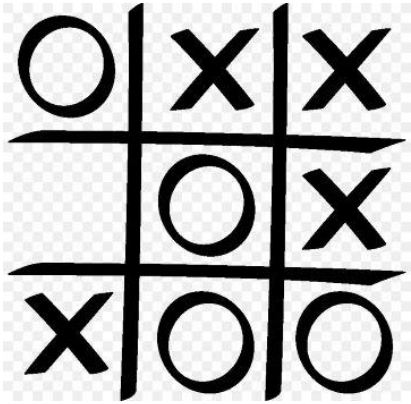
```
1 for line in my_list:
2     print("-----")
3     print("Line: ", line)
4     for element in line:
5         print("-")
6         print(element)
```

```
-----
Line:  ['Name', 'Type 1', 'Type 2']
-
Name
-
Type 1
-
Type 2
-----
Line:  ['Pikachu', 'Electric', None]
-
Pikachu
-
Electric
-
None
-----
Line:  ['Bulbazaur', 'Plant', 'Poison']
```

Activity 2 - TicTacToe

A **tic tac toe board** can be represented as a **list of lists**, by

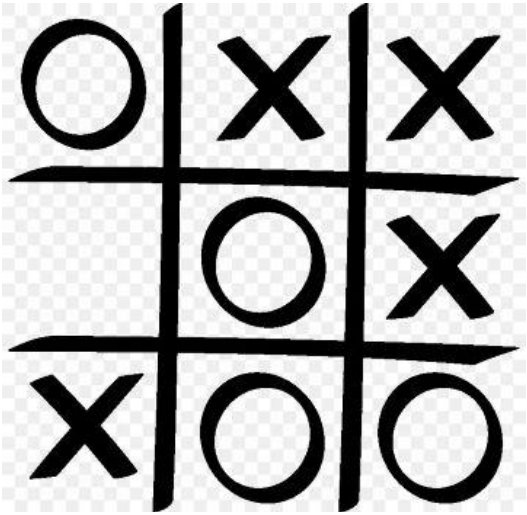
- defining a list of three elements,
- each element is a sublist of three elements corresponding to each line, from up to down,
- each sublist contains three elements, corresponding to each position on the line, from left to right,
- element will have a value **0** if the position is empty,
- element will have a value **1** if the position contains a **circle**,
- element will have a value **2** if the position contains a **cross**,



```
board = [[1, 2, 2], \
         [0, 1, 2], \
         [2, 1, 1]]
```

Activity 2 - TicTacToe

A tic tac toe board can be represented as a list of lists.



```
board = [[1, 2, 2], \
         [0, 1, 2], \
         [2, 1, 1]]
```

Write a function **board_winner()**, which:

- receives a board, defined as a list of lists, as its sole input parameter,
- returns a single output, which consists of the value
  - 1 if the circles won,
  - 2 if the crosses won,
  - and 0 if there is no clear winner.

Membership: the in keyword

- **We have seen:** the in keyword can be used to traverse lists in for loops.

```
1 my_list = [1, 7, 10, 14, 15]
2 print(my_list)
```

```
[1, 7, 10, 14, 15]
```

```
1 for element in my_list:
2     print("--")
3     print(element)
```

```
--
1
--
7
--
10
--
14
--
15
```

Membership: the in keyword

- **We have seen:** the **in** keyword can be used to traverse lists in for loops.

The **in** keyword can also be used to check for **membership**.

- If the element appears in the list, then the result of the operation is a boolean **True**.
- And **False** otherwise.

```
1 my_list = [1, 7, 10, 14, 15]
2 print(my_list)
```

```
[1, 7, 10, 14, 15]
```

```
1 for element in my_list:
2     print("--")
3     print(element)
```

```
--
1
--
7
--
10
--
14
--
15
```

```
1 bool1 =(7 in my_list)
2 print(bool1)
```

```
True
```

```
1 bool2 =(8 in my_list)
2 print(bool2)
```

```
False
```

Concatenation: + operator overload for lists

- **Concatenation:** you can merge two lists together, by using the + operator.

```
1 one_number = 6
2 another_number = 7
3 number_sum = one_number + another_number
4 print(number_sum)
```

```
13
```

```
1 a_string = '5'
2 another_string = '5'
3 string_sum = a_string + another_string
4 print(string_sum)
```

```
55
```

```
1 a_list = [0, 1, 2]
2 another_list = [1, 4, 7]
3 list_sum = a_list + another_list
4 print(list_sum)
```

```
[0, 1, 2, 1, 4, 7]
```

Concatenation: + operator overload for lists

- **Concatenation:** you can merge two lists together, by using the + operator.

- When summing two lists with +, Python does not sum the elements, as with numbers.
- Instead, it behaves as with **strings**, and concatenates both.

```
1 one_number = 6
2 another_number = 7
3 number_sum = one_number + another_number
4 print(number_sum)
```

```
13
```

```
1 a_string = '5'
2 another_string = '5'
3 string_sum = a_string + another_string
4 print(string_sum)
```

```
55
```

```
1 a_list = [0, 1, 2]
2 another_list = [1, 4, 7]
3 list_sum = a_list + another_list
4 print(list_sum)
```

```
[0, 1, 2, 1, 4, 7]
```

Adding an element with the append() method

- **Adding an element:** you can add an element to a list with the **append()** method.
- As with the **extend()** method, the **append()** method **modifies the list on which the method is applied**.

```
1 a_list = [0, 1, 2]
2 an_element = 7
3 # Append method on a list
4 a_list.append(an_element)
5 print(a_list)
```

```
[0, 1, 2, 7]
```

Adding an element with the append() method

- **Adding an element:** you can add an element to a list with the **append()** method.
- As with the **extend()** method, the **append()** method **modifies the list on which the method is applied**.
- **Adding an element #2:** You can also use the **+ operator,** with a **single element list.**

```
1 a_list = [0, 1, 2]
2 an_element = 7
3 # Append method on a list
4 a_list.append(an_element)
5 print(a_list)
```

```
[0, 1, 2, 7]
```

```
1 a_list = [0, 1, 2]
2 an_single_element_list = [7]
3 # Using the + operator on a single element list
4 list_sum = a_list + an_single_element_list
5 print(list_sum)
```

```
[0, 1, 2, 7]
```

Activity 3 – Complete, sorted deck of cards

- A standard deck of cards consists of all 52 possible combinations of 4 possible suits and 13 possible values.
- The 4 possible suits are listed in the **suits_list** below.

suits_list = ["Hearts", "Diamonds", "Spades", "Clubs"]

- The 13 possible values are listed in the **values_list** below.

values_list = ["Ace", "2", "3", "4", "5", "6", "7", "8", "9", "10", "Jack", "Queen", "King"]

Activity 3 – Complete, sorted deck of cards

Write a function **complete_deck(),**

- which receives **suits_list** and **values_list**, as input parameters,

suits_list = ["Hearts", "Diamonds", "Spades", "Clubs"]

values_list = ["Ace", "2", "3", "4", "5", "6", "7", "8", "9", "10", "Jack", "Queen", "King"]

- And returns a standard, sorted deck, as defined on the right.

deck = ['Ace of Hearts', '2 of Hearts', '3 of Hearts', '4 of Hearts', …, 'Queen of Hearts', 'King of Hearts', 'Ace of Diamonds', '2 of Diamonds', '3 of Diamonds', '4 of Diamonds', …, 'Queen of Diamonds', 'King of Diamonds', 'Ace of Spades', '2 of Spades', '3 of Spades', '4 of Spades', …, 'Queen of Spades', 'King of Spades', 'Ace of Clubs', '2 of Clubs', '3 of Clubs', '4 of Clubs', … 'Queen of Clubs', 'King of Clubs']

Activity 4 - Party merger

- In many video games, you might form a group of friends and play together. In this activity, we will define a function, which attempts to merge two groups of players into a single one.
- A group will simply consist of a list of names, as shown below.

  group = ["Matt", "Oka", "Tony", "Norman"]

- Write a function **merge_groups()**, which receives two groups of people as lists of names.
- We will call these two groups variables, **group1** and **group2** respectively.
- This function will **merge both groups** into a new one containing the members of both groups, and will return the **merged_group** list as its only output.
- However, it will only do so, **if the resulting merged group does not contain more than 6 people**. Otherwise, the function will return both group1 and group2, therefore refusing to merge them.

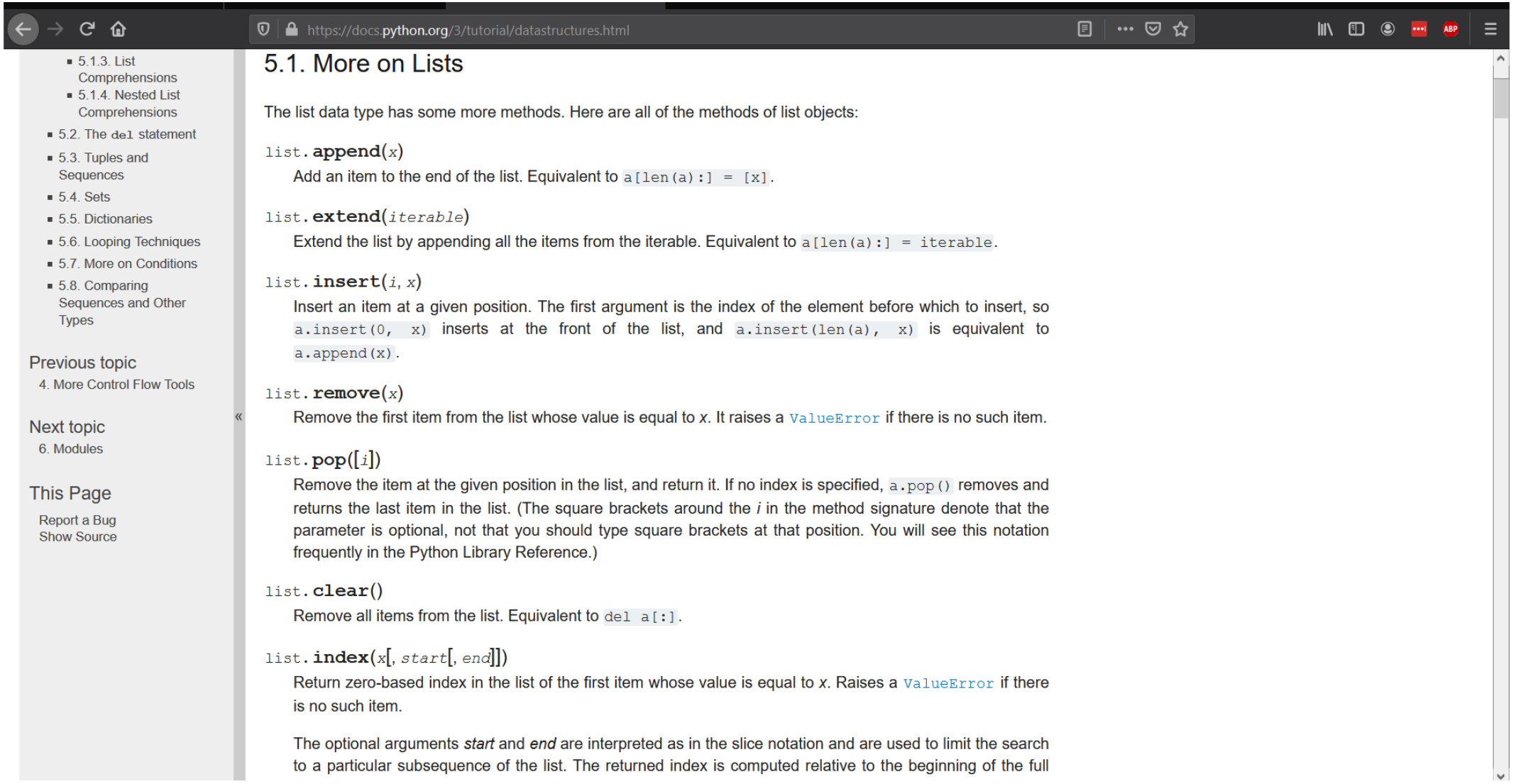
More methods and functions on lists

The lists objects in Python have **many more built-in methods and functions** you could use.

- We **do not expect you to know** about all these methods and functions, nor about how they work in detail.
- Just know where to find the information if you need it.
- Also, do not be afraid to **read the Python documentation!**

- **List documentation:** https://docs.python.org/3/tutorial/datastructures.html

More methods and functions on lists



RTFM

- **Definition (RTFM):** The acronym **RTFM** stands for

RTFM

- **Definition (RTFM):** The acronym **RTFM** stands for **“Read.**

RTFM

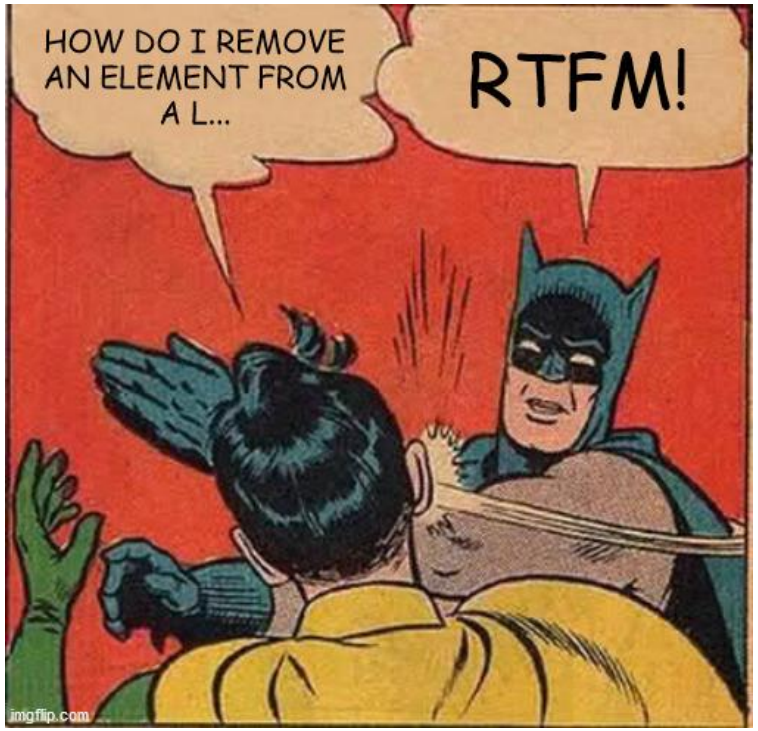
- **Definition (RTFM):** The acronym **RTFM** stands for **“Read. The.**

RTFM

- **Definition (RTFM):** The acronym **RTFM** stands for **“Read. The. F***ing.**

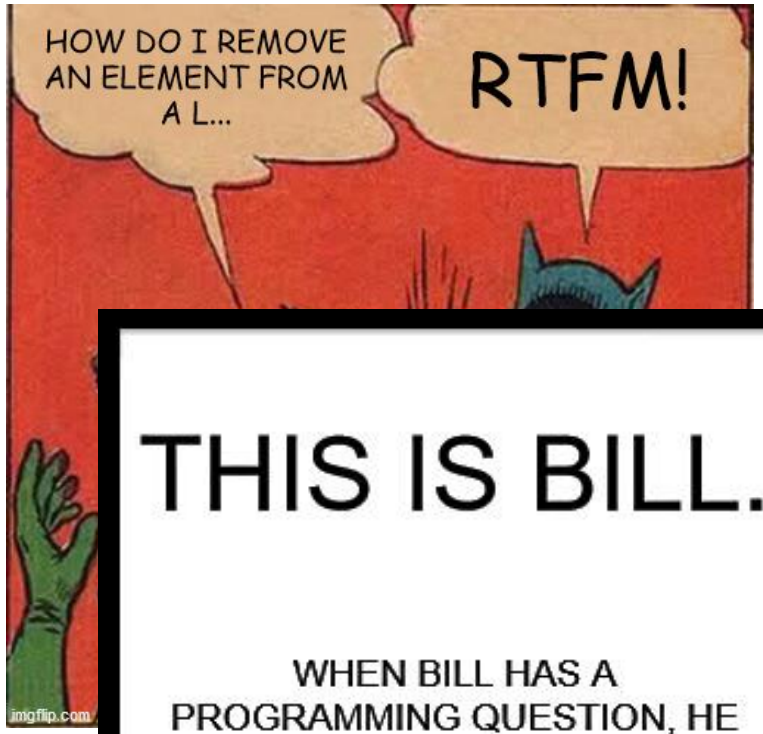
RTFM

- **Definition (RTFM):** The acronym **RTFM** stands for **“Read. The. F***ing. Manual.”**.



RTFM

- **Definition (RTFM):** The acronym **RTFM** stands for **“Read. The. F***ing. Manual.”**.
- While it may sound rude, **RTFM** is positively used in computer science communities, to encourage people to try to help themselves first, before seeking assistance from others.



KEEP

Matt's Great advice #9

Matt's Great Advice #9: Do not fear the Python documentation.

No programmer in the world knows everything about every function, method and variable type in Python.

Instead, we know **where and how to find the information online**, when needed.

Make it a habit to look for the Python documentation when in doubt or looking for something specific!

Conclusion

- The list type (in detail)
- Length of a list
- Indexing, slicing, updating and traversing a list
- Lists of lists
- Membership
- Concatenation, repetition and insertion
- Removing elements from a list
- Advanced methods and RTFM!

“Pythonic” loops

Definition (Pythonic loop):
A Pythonic loop is a programming concept, where a list of elements is generated by integrating a **for** loop **inside** a list definition.

```
# Let us consider the list below
my_list = [1, 3, 5, 7, 9, 11]
print(my_list)
```

```
# We could create it by telling Python to use
# a range and make a list out of it, like so.
my_list = list(range(1, 13, 2))
print(my_list)
```

```
# What if we wanted a list of squared values instead?
my_list = [1, 4, 9, 16, 25, 36]
print(my_list)
```

“Pythonic” loops

Definition (Pythonic loop):
A Pythonic loop is a programming concept, where a list of elements is generated by integrating a **for** loop **inside** a list definition.

```
# We would have to do a for loop and progressively assemble all elements together
# Start with an empty list
my_list = []
# For loop on all values to be squared (1-6)
for value in range(1, 7):
    # Squaring value
    element = value**2
    # Appending newly calculated element to the list
    my_list.append(element)
print(my_list)
```

```
[1, 4, 9, 16, 25, 36]
```

“Pythonic” loops

Definition (Pythonic loop):
A Pythonic loop is a programming concept, where a list of elements is generated by integrating a **for** loop **inside** a list definition.

```
# We would have to do a for loop and progressively assemble all elements together
# Start with an empty list
my_list = []
# For loop on all values to be squared (1-6)
for value in range(1, 7):
    # Squaring value + Appending newly calculated element to the list
    my_list.append(value**2)
print(my_list)
```

```
[1, 4, 9, 16, 25, 36]
```

“Pythonic” loops

Definition (Pythonic loop):
A Pythonic loop is a programming concept, where a list of elements is generated by integrating a **for** loop **inside** a list definition.

```
# We would have to do a for loop and progressively assemble all elements together
# Start with an empty list
my_list = []
# For loop on all values to be squared (1-6)
for value in range(1, 7):
    # Squaring value + Appending newly calculated element to the list
    my_list.append(value**2)
print(my_list)
```

```
[1, 4, 9, 16, 25, 36]
```

```
# Pythonic loop version of the list!
my_list = [value**2 for value in range(1, 7)]
print(my_list)
```

“Pythonic” loops

Definition (Pythonic loop):
A Pythonic loop is a programming concept, where a list of elements is generated by integrating a **for** loop **inside** a list definition.

```
# We would have to do a for loop and progressively assemble all elements together
# Start with an empty list
my_list = []
# For loop on all values to be squared (1-6)
for value in range(1, 7):
    # Squaring value + Appending newly calculated element to the list
    my_list.append(value**2)
print(my_list)
```

```
[1, 4, 9, 16, 25, 36]
```

```
# Pythonic loop version of the list!
my_list = [value**2 for value in range(1, 7)]
print(my_list)
```

“Pythonic” loops

Definition (Pythonic loop):
A Pythonic loop is a programming concept, where a list of elements is generated by integrating a **for** loop **inside** a list definition.

```
# We would have to do a for loop and progressively assemble all elements together
# Start with an empty list
my_list = []
# For loop on all values to be squared (1-6)
for value in range(1, 7):
    # Squaring value + Appending newly calculated element to the list
    my_list.append(value**2)
print(my_list)
```

```
[1, 4, 9, 16, 25, 36]
```

```
# Pythonic loop version of the list!
my_list = [value**2 for value in range(1, 7)]
print(my_list)
```

“Pythonic” loops

Definition (Pythonic loop):
A Pythonic loop is a programming concept, where a list of elements is generated by integrating a **for** loop **inside** a list definition.

```
# We would have to do a for loop and progressively assemble all elements together
# Start with an empty list
my_list = []
# For loop on all values to be squared (1-6)
for value in range(1, 7):
    # Squaring value + Appending newly calculated element to the list
    my_list.append(value**2)
print(my_list)
```

```
[1, 4, 9, 16, 25, 36]
```

```
# Pythonic loop version of the list!
my_list = [value**2 for value in range(1, 7)]
print(my_list)
```

```
[1, 4, 9, 16, 25, 36]
```

“Pythonic” loops

Definition (Pythonic loop):
A Pythonic loop is a programming concept, where a list of elements is generated by integrating a **for** loop **inside** a list definition.

```
# We would have to do a for loop and prog
# Start with an empty list
my_list = []
# For loop on all values to be squared (1
for value in range(1, 7):
    # Squaring value + Appending newly ca
    my_list.append(value**2)
print(my_list)
```

```
[1, 4, 9, 16, 25, 36]
```

Why use pythonic loops?

- Convenient when the elements of the list can be defined regularly (e.g. all odd numbers from a to b, etc.)
- Less cumbersome than typing the elements, manually, especially for large lists.

```
# Pythonic loop version of the list!
my_list = [value**2 for value in range(1, 7)]
print(my_list)
```

```
[1, 4, 9, 16, 25, 36]
```

Repetition: * operator overloading

- **Repetition:** you can **repeat** a list n times (with n being an **int**), by multiplying a list with an int, using the * operator.

```
1  a_list = [0, 1, 2]
2  a_number = 3
3  list_product = a_list*a_number
4  print(list_product)
```

```
[0, 1, 2, 0, 1, 2, 0, 1, 2]
```

Repetition: * operator overloading

- **Repetition:** you can **repeat** a list n times (with n being an **int**), by multiplying a list with an int, using the * operator.
- The result of this repetition is a list where the elements are repeated n times.
- As if we add summed (+) the list with itself n times in a row.

```
1  a_list = [0, 1, 2]
2  a_number = 3
3  list_product = a_list*a_number
4  print(list_product)
```

```
[0, 1, 2, 0, 1, 2, 0, 1, 2]
```

Repetition: * operator overloading

- **Repetition:** you can **repeat** a list n times (with n being an **int**), by multiplying a list with an int, using the * operator.
- The result of this repetition is a list where the elements are repeated n times.
- As if we add summed (+) the list with itself n times in a row.
- Similar to **multiplying** a **string** with an **int**.

```
1  a_list = [0, 1, 2]
2  a_number = 3
3  list_product = a_list*a_number
4  print(list_product)
```

```
[0, 1, 2, 0, 1, 2, 0, 1, 2]
```

```
1  a_string = "pika"
2  a_number = 3
3  some_stuff = a_string*a_number + "chu"
4  print(some_stuff)
```

```
pikapikapikachu
```

Element-wise removal

- **Element-wise removal:** the **remove()** method removes the **element** passed as its **parameter,** from the list on which the method is applied.

```
1 my_list = [1, 3, 5, 7, 9, 11]
2 print(my_list)
```

```
[1, 3, 5, 7, 9, 11]
```

```
1 # Element-wise removal
2 an_element = 7
3 my_list.remove(an_element)
4 print(my_list)
```

```
[1, 3, 5, 9, 11]
```

Element-wise removal

- **Element-wise removal:** the **remove()** method removes the **element** passed as its **parameter,** from the list on which the method is applied.
- The **remove()** method changes the list as a result.

```
1  my_list = [1, 3, 5, 7, 9, 11]
2  print(my_list)
```

```
[1, 3, 5, 7, 9, 11]
```

```
1  # Element-wise removal
2  an_element = 7
3  my_list.remove(an_element)
4  print(my_list)
```

```
[1, 3, 5, 9, 11]
```

Element-wise removal

- **Element-wise removal:** the **remove()** method removes the **element** passed as its **parameter,** from the list on which the method is applied.
- The **remove()** method changes the list as a result.
- **Note:** if the element appear multiple times, it is only removed once, when it is first encountered.

```
1 my_list = [1, 3, 5, 7, 9, 11]
2 print(my_list)
```

```
[1, 3, 5, 7, 9, 11]
```

```
1 # Element-wise removal
2 an_element = 7
3 my_list.remove(an_element)
4 print(my_list)
```

```
[1, 3, 5, 9, 11]
```

```
1 my_list = [1, 3, 5, 7, 9, 11,
2 print(my_list)
```

```
[1, 3, 5, 7, 9, 11, 7]
```

```
1 # Element-wise removal
2 an_element = 7
3 my_list.remove(an_element)
4 print(my_list)
```

```
[1, 3, 5, 9, 11, 7]
```

Index-wise removal

- **Index-wise removal:** you can remove an element at a given index by using the **del function** on the element itself.

```
1 my_list = [1, 3, 5, 7, 9, 11, 7]
2 print(my_list)
```

```
[1, 3, 5, 7, 9, 11, 7]
```

```
1 # Index-wise removal
2 an_index = 4
3 del(my_list[an_index])
4 print(my_list)
```

```
[1, 3, 5, 7, 11, 7]
```

Index-wise removal

- **Index-wise removal:** you can remove an element at a given index by using the **del function** on the element itself.

- **Note:** the **del function** can be used on **any variable, of any type,** to remove it from the Python memory.

```
1 my_list = [1, 3, 5, 7, 9, 11, 7]
2 print(my_list)
```

```
[1, 3, 5, 7, 9, 11, 7]
```

```
1 # Index-wise removal
2 an_index = 4
3 del(my_list[an_index])
4 print(my_list)
```

```
[1, 3, 5, 7, 11, 7]
```

```
1 a_number = 10
2 print("First print: ", a_number)
3 del(a_number)
4 print("Second print: ", a_number)
```

```
First print:  10

---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-94-ee50ccff233f> in <module>
      2 print("First print: ", a_number)
      3 del(a_number)
----> 4 print("Second print: ", a_number)

NameError: name 'a_number' is not defined
```

Index-wise extraction

- **Index-wise extraction:** you can **extract** the **element** of a **list**, at the **index** given as its **parameter**, by using the **pop() method**.

```
1  a_deck_of_cards = ["7 of Hearts", "8 of Diamonds", \
2                     "10 of Hearts", "6 of Clubs"]
3  print(a_deck_of_cards)
```

```
['7 of Hearts', '8 of Diamonds', '10 of Hearts', '6 of Clubs']
```

```
1  an_index = 2
2  # Index-wise Pop method on list
3  drawn_card = a_deck_of_cards.pop(an_index)
4  print(drawn_card)
5  print(a_deck_of_cards)
```

```
10 of Hearts
['7 of Hearts', '8 of Diamonds', '6 of Clubs']
```

Index-wise extraction

- **Index-wise extraction:** you can **extract** the **element** of a **list**, at the **index** given as its **parameter**, by using the **pop() method**.
- As with other methods, **it changes the list** on which it is applied.

```
1  a_deck_of_cards = ["7 of Hearts", "8 of Diamonds", \
2                     "10 of Hearts", "6 of Clubs"]
3  print(a_deck_of_cards)
```

```
['7 of Hearts', '8 of Diamonds', '10 of Hearts', '6 of Clubs']
```

```
1  an_index = 2
2  # Index-wise Pop method on list
3  drawn_card = a_deck_of_cards.pop(an_index)
4  print(drawn_card)
5  print(a_deck_of_cards)
```

```
10 of Hearts
['7 of Hearts', '8 of Diamonds', '6 of Clubs']
```

Index-wise extraction

- **Index-wise extraction:** you can **extract** the **element** of a **list**, at the **index** given as its **parameter**, by using the **pop() method**.
- As with other methods, **it changes the list** on which it is applied.
- The **output** of the **pop()** method consists of the **element**, which is extracted from the list, at the given index.

```
a_deck_of_cards = ["7 of Hearts", "8 of Diamonds", \
                   "10 of Hearts", "6 of Clubs"]
print(a_deck_of_cards)
```

```
['7 of Hearts', '8 of Diamonds', '10 of Hearts', '6 of Clubs']
```

```
an_index = 2
# Index-wise Pop method on list
drawn_card = a_deck_of_cards.pop(an_index)
print(drawn_card)
print(a_deck_of_cards)
```

```
10 of Hearts
['7 of Hearts', '8 of Diamonds', '6 of Clubs']
```

The index() method

- The **index()** method is also useful to check for membership and find an element in a list.
- It returns the **index** of the first occurrence of the element passed in parameter, in the list it applies to.
- Careful though, as it fails if no element in the list has the value passed as parameter.

```
1 my_list = [1, 7, 10, 7, 14]
2 print(my_list.index(7))
```

```
1
```

```
1 my_list = [1, 7, 10, 7, 14]
2 print(my_list.index(4))
```

```
---------------------------------
ValueError
<ipython-input-4-2ed34508db49> in <mc
      1 my_list = [1, 7, 10, 7, 14]
----> 2 print(my_list.index(4))

ValueError: 4 is not in list
```

List **alteration** with the **extend()** method

- **Concatenation:** A second list can be added to a first one with the **extend()** method.

```
1  first_list = [0, 1, 2]
2  second_list = [1, 4, 7]
3  print(first_list)
4  print(second_list)
5  # Extend method on a list
6  first_list.extend(second_list)
7  print("-")
8  print(first_list)
9  print(second_list)
```

```
[0, 1, 2]
[1, 4, 7]
-
[0, 1, 2, 1, 4, 7]
[1, 4, 7]
```

List **alteration** with the **extend()** method

- **Concatenation:** A second list can be added to a first one with the **extend()** method.

This is slightly different, as

- It **changes the list** on which the **extend()** method is applied,
- And adds the elements of the second lists to the first one.
- At the end of the operation, **the first list will be changed**.

```
1  first_list = [0, 1, 2]
2  second_list = [1, 4, 7]
3  print(first_list)
4  print(second_list)
5  # Extend method on a list
6  first_list.extend(second_list)
7  print("-")
8  print(first_list)
9  print(second_list)
```

```
[0, 1, 2]
[1, 4, 7]
-
[0, 1, 2, 1, 4, 7]
[1, 4, 7]
```

Inserting an element in a list

- **Insertion:** You can **insert** an **element**, at a given **index**, by using the **insert()** method.
- As with the **extend()** and **append()** methods, the **insert()** method is applied to a list.
- It receives an **index** as its **first parameter** and an **element** as its **second parameter**.
- And it **changes the list as a result**.

```
1 my_list = [1, 3, 5, 7, 9, 11]
2 print(my_list)
```

```
[1, 3, 5, 7, 9, 11]
```

```
1 an_element = 'Hello'
2 an_index = 4
3 my_list.insert(an_index, an_element)
4 print(my_list)
```

```
[1, 3, 5, 7, 'Hello', 9, 11]
```

Additional methods and functions on lists

Many other additional methods and functions can be used on lists objects…

- **Maximum:** the **max()** function returns the element with maximal value.
- **Minimum:** the **min()** function returns the element with maximal value.
- **Reverse:** the **reverse()** method reverses the elements in a list.

```
1  # Max and min values
2  grades_list = [85, 70, 80, 70, 65]
3  print("Max: ", max(grades_list))
4  print("Min: ", min(grades_list))
```

```
Max:  85
Min:  65
```

```
1  # Reverse
2  a_list = [7, 8, 4, 0, 1, 5]
3  a_list.reverse()
4  print(a_list)
```

```
[5, 1, 0, 4, 8, 7]
```

Additional methods and functions on lists

Many other additional methods and functions can be used on lists objects…

- **Sorting:** the **sort()** method can be used to sort the elements,

```
1  # Sorting
2  a_list = [7, 8, 4, 0, 1, 5]
3  a_list.sort()
4  print(a_list)
```

```
[0, 1, 4, 5, 7, 8]
```

Additional methods and functions on lists

Many other additional methods and functions can be used on lists objects…

- **Sorting:** the **sort()** method can be used to sort the elements,
- In **ascending order** for **numerical** elements,

```
1  # Sorting
2  a_list = [7, 8, 4, 0, 1, 5]
3  a_list.sort()
4  print(a_list)
```

```
[0, 1, 4, 5, 7, 8]
```

Additional methods and functions on lists

Many other additional methods and functions can be used on lists objects…

- **Sorting:** the **sort()** method can be used to sort the elements,
- In **ascending order** for **numerical** elements,
- And in **alphabetical order** for **string** elements.

```
1  # Sorting
2  a_list = [7, 8, 4, 0, 1, 5]
3  a_list.sort()
4  print(a_list)
```

```
[0, 1, 4, 5, 7, 8]
```

```
1  # Sorting
2  a_list = ["Matt", "Tony", "Oka", "Norman"]
3  a_list.sort()
4  print(a_list)
```

```
['Matt', 'Norman', 'Oka', 'Tony']
```

Additional methods and functions on lists

Many other additional methods and functions can be used on lists objects…

- **Sorting:** the **sort()** method can be used to sort the elements,
- In **ascending order** for **numerical** elements,
- And in **alphabetical order** for **string** elements.
- **Note:** does not work on lists with mixed types elements.

```
1  # Sorting
2  a_list = [7, 8, 4, 0, 1, 5]
3  a_list.sort()
4  print(a_list)
```

```
[0, 1, 4, 5, 7, 8]
```

```
1  # Sorting
2  a_list = ["Matt", "Tony", "Oka", "Norman"]
3  a_list.sort()
4  print(a_list)
```

```
['Matt', 'Norman', 'Oka', 'Tony']
```

```
1  # Sorting
2  a_list = ["Matt", "Tony", 4, 7, 8]
3  a_list.sort()
4  print(a_list)
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-111-4fcbc28e0be2> in <module>
      1 # Sorting
      2 a_list = ["Matt", "Tony", 4, 7, 8]
----> 3 a_list.sort()
      4 print(a_list)

TypeError: '<' not supported between instances of 'int' and 'str'
```

Additional methods and functions on lists

Many other additional methods and functions can be used on lists objects…

- **Counting:** the **count()** method can be used to count how many times an element appears in a list.

```
1  # Counting
2  a_list = ["Potion", "Sword", "Shield", "Potion", "Potion"]
3  number = a_list.count("Potion")
4  print(number)
```

```
3
```