

Lists and Dictionaries

MAD 102



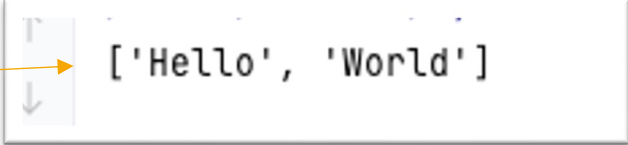
Lists

- Lists are containers
 - Containers are objects that group related objects together
- Lists are mutable
 - Mutability allows a list to be changed – it can grow, it can shrink
- Lists are sequenced
 - Sequenced items have a left-to-right positional order
 - Can be accessed using index values

Lists

- Lists can be created using `[]` and with literal values

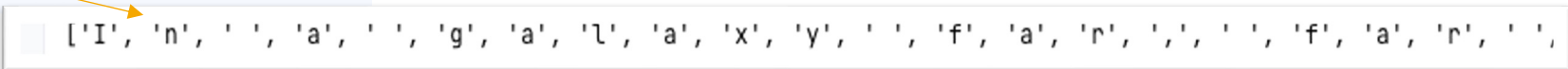
```
3 example = ['Hello', 'World']  
4 print(example)  
5
```



`['Hello', 'World']`

- Lists can be created with `list()` function
 - Accepts a single, iterable object (string, list, tuple) as an argument

```
2 title = 'In a galaxy far, far away'  
3 print(list(title))  
4
```



`['I', 'n', ' ', 'a', ' ', 'g', 'a', 'l', 'a', 'x', 'y', ' ', 'f', 'a', 'r', ' ', 'f', 'a', 'r', ' ', 'a', 'w', 'a', 'y']`

Lists

- Members are accessed using the index
 - First element has an index value of 0

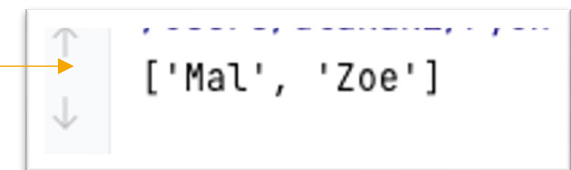
```
crew = ['Mal', 'Zoe', 'Washburn', 'Jayne', 'Simon']  
print(crew[0])  
|
```



Mal

- Can access a range of members
 - Index starting from (inclusive) and going to (exclusive)

```
3 crew = ['Mal', 'Zoe', 'Washburn', 'Jayne', 'Simon']  
4 print(crew[0:2])  
5 |
```

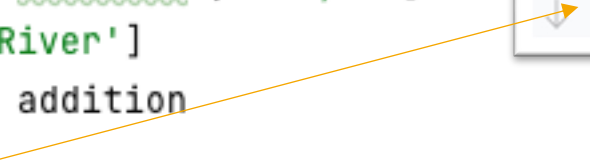


['Mal', 'Zoe']

Lists

- Can join lists

```
3 crew = ['Mal', 'Zoe', 'Washburn', 'Jayne']
4 addition = ['Simon', 'River']
5 serenity_crew = crew + addition
6 print(serenity_crew)
```




```
['Mal', 'Zoe', 'Washburn', 'Jayne', 'Simon', 'River']
```

Lists

- Growing and shrinking of lists is known as **in-place modification**
 - The contents can be modified without making a completely new list

```
3 crew = ['Mal', 'Zoe', 'Washburn', 'Jayne']
4 print(crew)
5 crew[2] = 'Book'
6 print(crew)
7
```

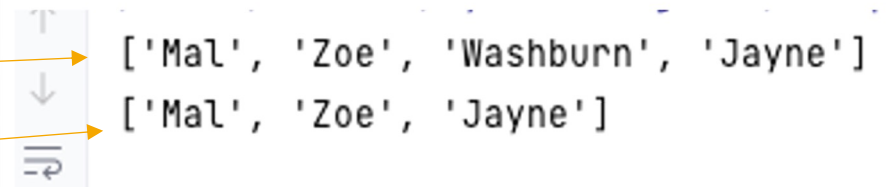


```
['Mal', 'Zoe', 'Washburn', 'Jayne']
['Mal', 'Zoe', 'Book', 'Jayne']
```

Lists

- Items can be deleted

```
2  
3 crew = ['Mal', 'Zoe', 'Washburn', 'Jayne']  
4 print(crew)  
5 del crew[2]  
6 print(crew)  
7
```



['Mal', 'Zoe', 'Washburn', 'Jayne']
['Mal', 'Zoe', 'Jayne']



List methods

- Although many operations can be performed without using these methods – it is recommended to use as they provide more readable code
- List methods help perform common operations of adding, removing, sorting , etc.

Adding elements

- The **append()** method adds an element to the end of the list

```
3 crew = ['Mal', 'Zoe', 'Washburn']
4 print(crew)
5 crew.append('Jayne')
6 print(crew)
```

```
↑ /usr/local/lib/python3.9/site-packages/example.py
↓ ['Mal', 'Zoe', 'Washburn']
  ['Mal', 'Zoe', 'Washburn', 'Jayne']
  ⚡
```

- The **extend()** method adds a series of items to a list

```
3 crew = ['Mal', 'Zoe', 'Washburn']
4 print(crew)
5 crew.extend(['Simon', 'River'])
6 print(crew)
```

```
↑ /usr/local/lib/python3.9/site-packages/example.py
↓ ['Mal', 'Zoe', 'Washburn']
  ['Mal', 'Zoe', 'Washburn', 'Simon', 'River']
  ⚡
```

Adding elements

- Elements can be added to a list before a specified index

```
3 crew = ['Mal', 'Zoe', 'Washburn', 'Simon', 'River']
4 print(crew)
5 crew.insert(3, 'Jayne')
6 print(crew)
7
```

['Mal', 'Zoe', 'Washburn', 'Simon', 'River']

['Mal', 'Zoe', 'Washburn', 'Jayne', 'Simon', 'River']

Removing Elements

- Elements can be removed from a list by name

```
3 crew = ['Mal', 'Zoe', 'Washburn', 'Jayne', 'Simon', 'River']
4 print(crew)
5 crew.remove('Jayne')
6 print(crew)
```

```
['Mal', 'Zoe', 'Washburn', 'Jayne', 'Simon', 'River']
['Mal', 'Zoe', 'Washburn', 'Simon', 'River']
```

- The last element can be removed and returned

```
3 crew = ['Mal', 'Zoe', 'Washburn', 'Jayne', 'Simon', 'River']
4 print(crew)
5 last = crew.pop()
6 print(crew)
7 print(last)
```

```
['Mal', 'Zoe', 'Washburn', 'Jayne', 'Simon', 'River']
['Mal', 'Zoe', 'Washburn', 'Jayne', 'Simon']
River
```

Removing Elements

- Remove and return an item at a specific index position

```
3 crew = ['Mal', 'Zoe', 'Washburn', 'Jayne', 'Simon', 'River']
4 print(crew)
5 member = crew.pop(2)
6 print(crew)
7 print(member)
8
```

```
['Mal', 'Zoe', 'Washburn', 'Jayne', 'Simon', 'River']
['Mal', 'Zoe', 'Jayne', 'Simon', 'River']
Washburn
```

Sorting Lists

- Lists can be sorted

- Alphabet

```
['Mal', 'Zoe', 'Washburn', 'Jayne', 'Simon', 'River']  
['River', 'Simon', 'Jayne', 'Washburn', 'Zoe', 'Mal']
```

```
3 crew = ['Mal', 'Zoe', 'Washburn', 'Jayne', 'Simon', 'River']  
4 print(crew)  
5 crew.sort()  
6 print(crew)  
7
```

```
['Mal', 'Zoe', 'Washburn', 'Jayne', 'Simon', 'River']  
['Jayne', 'Mal', 'River', 'Simon', 'Washburn', 'Zoe']
```

- Placed in reverse order

```
3 crew = ['Mal', 'Zoe', 'Washburn', 'Jayne', 'Simon', 'River']  
4 print(crew)  
5 crew.reverse()  
6 print(crew)  
7
```

```
['Mal', 'Zoe', 'Washburn', 'Jayne', 'Simon', 'River']  
['River', 'Simon', 'Jayne', 'Washburn', 'Zoe', 'Mal']
```

Counting Elements

- Can determine the number of times a specific element appears in a list
 - This does not modify the list

```
3 crew = ['Mal', 'Zoe', 'Washburn', 'Jayne', 'Simon', 'River', 'Mal']
4 print(crew.count('Mal'))
5 |
```

2

Iterating over a list

- A **for loop** is used for looping over a list
- Each iteration allows for the element to a new variable

```
3 crew = ['Mal', 'Zoe', 'Washburn', 'Jayne', 'Simon', 'River', 'Mal']
4 for member in crew:
5     print(member, end=' ')
```

```
Mal Zoe Washburn Jayne Simon River Mal
Process finished with exit code 0
```

Iterating over a list – value and index

- You can get the value of each element and their index using the `enumerate()` function

```
3 crew = ['Mal', 'Zoe', 'Washburn', 'Jayne', 'Simon', 'River']
4 for position, member in enumerate(crew):
5     print(f'{member} is a index {position}')
```

```
Mal is a index 0
Zoe is a index 1
Washburn is a index 2
Jayne is a index 3
Simon is a index 4
River is a index 5
```



Iterate over list - numerical values

- There are built-in functions to handle common numerical calculations
- Is a list empty or does it contain any 0 values – use **all(list)**
 - **True** if there is not a 0 value
- **any(list)** returns True if any value is True.

Iterate over list - numerical values

- To get the min or max value in a list, use **min(list)** or **max(list)**
- To get the sum of all elements in a list, use **sum(list)**

```
3 purchases = [34.12, 45.23, 12.34, 10.01, 12.22]
4
5 print(sum(purchases))
6
```



113.92

Nested lists

- Lists can contain lists.
 - These are known as nested lists.

The diagram illustrates a nested list structure. A code snippet is shown with a light blue background and a vertical line on the left. The code is: `crews = [['Mal', 'Washburne', 'Zoe'], ['Han', 'Chewie'], ['Kirk', 'Spock', 'McCoy']]`. Above the code, the text "Nested Lists" is centered. Below it, three labels "List 0", "List 1", and "List 2" are positioned above the first, second, and third sublists respectively. A bracket labeled "Main List" spans the entire width of the code. Another bracket labeled "Nested Lists" spans the width of the three sublists.

```
~
3 crews = [['Mal', 'Washburne', 'Zoe'], ['Han', 'Chewie'], ['Kirk', 'Spock', 'McCoy']]
4 |
```

Nested Lists

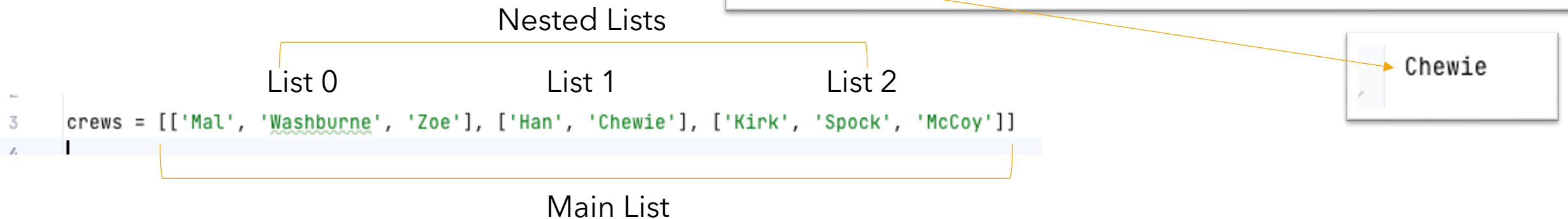
List 0 List 1 List 2

Main List

Nested Lists

- To access the nested list, use the index to get the location of the nested list, and another index value to get the element in the nested list
 - For example – to access Chewie – it is in list index 1, at index position 1

```
3 crews = [['Mal', 'Washburne', 'Zoe'], ['Han', 'Chewie'], ['Kirk', 'Spock', 'McCoy']]
4 print(crews[1][1])
```



Nested Lists

- Can use a for loop, with a nested for loop, to iterate over the entire contents


```
3 crews = [['Mal', 'Washburne', 'Zoe'], ['Han', 'Chewie'], ['Kirk', 'Spock', 'McCoy']]
4
5 → for position, crew in enumerate(crews):
6     print('=' * 20)
7     print(f'Crew #{position + 1}')
8     print('=' * 20)
9     → for member in crew:
10         print(member)
11
```

```
=====
Crew #1
=====
Mal
Washburne
Zoe
=====
Crew #2
=====
Han
Chewie
=====
Crew #3
=====
Kirk
Spock
McCoy
```

List slicing

- Using **slice notation**, you can create new lists with just the elements you want
- **list[firstIndex : lastIndex]** – where the **firstIndex** is included, but the **lastIndex** is not
- -ve values start at the end

```
3 crew = ['Kirk', 'Spock', 'McCoy', 'Uhura', 'Checkov', 'Scotty', 'Sulu']  
4 print(crew[0:3])
```



```
['Kirk', 'Spock', 'McCoy']
```

List slicing

- Slice notation allows you to not provide an index as part of the argument values
 - Absence of a value means – *from start or to end*

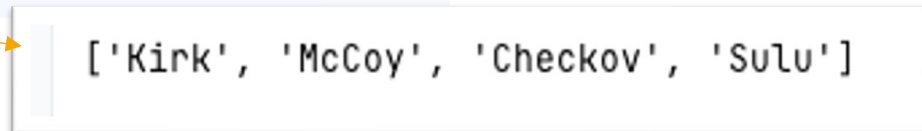
list[: 2] – *means from the start to index 2*

list[3:] – *means from the index position 4 to the end*

List slicing

- Slice notation also allows for a **stride** value
 - Indicates how many elements are skipped

```
3 crew = ['Kirk', 'Spock', 'McCoy', 'Uhura', 'Checkov', 'Scotty', 'Sulu']  
4 print(crew[::2])  
5
```



```
['Kirk', 'McCoy', 'Checkov', 'Sulu']
```




List comprehensions

- A construct called list comprehensions allows you to modify every element in a list the same way
 - The construct iterates over a list
 - Modifies each element
 - Returns a new list of modified elements

List comprehensions

- Made up of three components
 1. Expression used to evaluate each element
 2. Loop variable to bind the the current element
 3. Iterable object to iterate over
- Is surrounded by []
- Contains the keywords for and in to separate the expression from the loop and the loop variable from the iterable object

List comprehensions

- We want to increase all our prices by 5%

```
3 prices = [23.45, 22.99, 19.99, 9.95]
4
5 price_increase = [price * 1.05 for price in prices]
6
7 print(price_increase)
8
```

```
[24.6225, 24.139499999999998, 20.9895, 10.4475]
```

List comprehensions

- List comprehensions can be applied based on a condition
 - Remember - this makes a new list, but only uses the elements that meet the current condition

```
prices = [23.45, 22.99, 19.99, 9.95]
```

```
price_increase = [price * 1.05 for price in prices if price > 20]
```

```
print(price_increase)
```

```
[24.6225, 24.139499999999998]
```

List sorting

- Two methods can be used for sorting a list
 - The **sort()** method will place the list in alphabetical or numerical order
 - The **sorted()** method provides the same sorting - but it returns a new list instead of just modifying it in place
- Optional key argument that specified a function to be applied to each element **prior** to being compared
 - Useful to use `.lower` or `.upper` or `.capitalize` as a key

List sorting

- Note order without the capitalize and with

```
3 crew = ['rey', 'finn', 'Poe']
4 crew.sort()
5 print(crew)
6 crew.sort(key=str.capitalize)
7 print(crew)
```

```
['Poe', 'finn', 'rey']
['finn', 'Poe', 'rey']
```

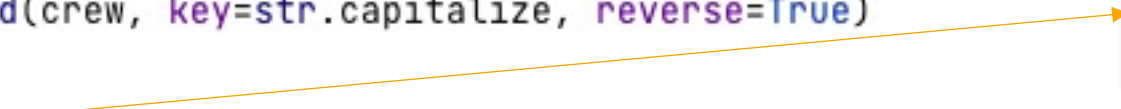
```
3 crew = ['rey', 'finn', 'Poe']
4 results = sorted(crew, key=str.capitalize)
5 print(crew)
6 print(results)
```

```
['rey', 'finn', 'Poe']
['finn', 'Poe', 'rey']
```

List sorting

- To sort highest to lowest or reverse alphabetical, use the reverse keyword with a value of true

```
3 crew = ['rey', 'finn', 'Poe']  
4 results = sorted(crew, key=str.capitalize, reverse=True)  
5 print(crew)  
6 print(results)  
7
```



```
['rey', 'finn', 'Poe']  
['rey', 'Poe', 'finn']
```

Dictionaries

- Container object that store content in key and value pairs
 - The dict type implements a dictionary in Python
- Dictionaries are created by placing key : value in { }
- Using dict() function – passing in keys assigned to values or an array of tuples

```
example = {'Kirk': 'Captain', 'Spock': 'Commander', 'McCoy': 'Lt. Commander'}  
function_example = dict(Kirk='Captain', Spock='Commander', McCoy='Lt. Commander')  
function_example_with_tuples = dict([('Kirk', 'Captain'), ('Spock', 'Commander'), ('McCoy', 'Lt. Commander')])  
  
print(example)  
print(function_example)  
print(function_example_with_tuples)
```

```
{'Kirk': 'Captain', 'Spock': 'Commander', 'McCoy': 'Lt. Commander'}  
{'Kirk': 'Captain', 'Spock': 'Commander', 'McCoy': 'Lt. Commander'}  
{'Kirk': 'Captain', 'Spock': 'Commander', 'McCoy': 'Lt. Commander'}
```


Dictionaries

- Access elements using the key to get the value

```
example = {'Kirk': 'Captain', 'Spock': 'Commander', 'McCoy': 'Lt. Commander'}  
print(example['Kirk'])
```

Captain

- Add an entry using the key (will modify if the key exists)

```
example = {'Kirk': 'Captain', 'Spock': 'Commander', 'McCoy': 'Lt. Commander'}  
example['Sulu'] = 'Lieutenant'  
print(example)
```

```
{'Kirk': 'Captain', 'Spock': 'Commander', 'McCoy': 'Lt. Commander', 'Sulu': 'Lieutenant'}
```

Dictionaries

- Delete the entry using del and the key

```
3 example = {'Kirk': 'Captain', 'Spock': 'Commander', 'McCoy': 'Lt. Commander', 'Sulu': 'Lieutenant'}  
4 del example['Sulu']  
5 print(example)
```

```
{'Kirk': 'Captain', 'Spock': 'Commander', 'McCoy': 'Lt. Commander'}
```

- Checks to see if the key is present using in

```
example = {'Kirk': 'Captain', 'Spock': 'Commander', 'McCoy': 'Lt. Commander', 'Sulu': 'Lieutenant'}  
print('Sulu' in example)
```

```
True
```

Common dictionary methods

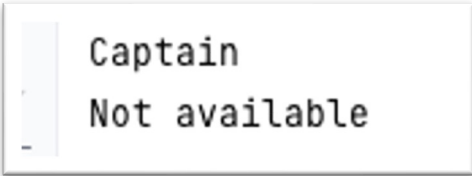
- The **clear()** method will remove all items from a dictionary

```
example = {'Kirk': 'Captain', 'Spock': 'Commander', 'McCoy': 'Lt. Commander', 'Sulu': 'Lieutenant'}  
example.clear()  
print(example)
```



- The get(key, default) method reads the value associated with the key – or returns the default value

```
3 example = {'Kirk': 'Captain', 'Spock': 'Commander', 'McCoy': 'Lt. Commander', 'Sulu': 'Lieutenant'}  
4 print(example.get('Kirk', 'Not available'))  
5 print(example.get('Chapel', 'Not available'))  
,
```



Common dictionary methods

- The **update()** method allows you to merge two dictionaries into one

```
3 example = {'Kirk': 'Captain', 'Spock': 'Commander', 'McCoy': 'Lt. Commander'}
4 example.update({'Sulu': 'Lieutenant'})
5 print(example)
```

```
{'Kirk': 'Captain', 'Spock': 'Commander', 'McCoy': 'Lt. Commander', 'Sulu': 'Lieutenant'}
```

- The **pop(key, default)** removes and returns the value or the default if not found

```
3 example = {'Kirk': 'Captain', 'Spock': 'Commander', 'McCoy': 'Lt. Commander'}
4 print(example.pop('Kirk', 'Not found'))
5 print(example.pop('Scott', 'Not found'))
```

```
Captain
Not found
;
```

Iterating over a dictionary

- The for loop can be used to iterate over a dictionary
 - Ordering is determined by the hash value created by the Python interpreter for the key values

```
3 example = {'Kirk': 'Captain', 'Spock': 'Commander', 'McCoy': 'Lt. Commander'}
4 for key in example:
5     print(key)
```

Kirk
Spock
McCoy

```
example = {'Kirk': 'Captain', 'Spock': 'Commander', 'McCoy': 'Lt. Commander'}
for key in example:
    print(example[key])
```

Captain
Commander
Lt. Commander

Iterating over a dictionary

- A **view object** is created by three useful dictionary methods
 - This is read-only access to keys and values
- They are the **items()**, **keys()**, **values()** methods
 - Items() returns a tuple of key and value
 - Keys() returns just the keys
 - Values returns just the values

Iterating over a dictionary

- Handling items

```
3 example = {'Kirk': 'Captain', 'Spock': 'Commander', 'McCoy': 'Lt. Commander'}  
4 for name, rank in example.items():  
5     print(f'{name} holds the rank of {rank}.')  
6 |
```

```
Kirk holds the rank of Captain.  
Spock holds the rank of Commander.  
McCoy holds the rank of Lt. Commander.
```

Iterating over a dictionary

- Handling keys

```
3 example = {'Kirk': 'Captain', 'Spock': 'Commander', 'McCoy': 'Lt. Commander'}  
4 for name in example.keys():  
5     print(f'{name} is part of the crew.')  
6
```

```
Kirk is part of the crew.  
Spock is part of the crew.  
McCoy is part of the crew.
```


Iterating over a dictionary

- Handling values

```
3 example = {'Kirk': 'Captain', 'Spock': 'Commander', 'McCoy': 'Lt. Commander'}  
4 for rank in example.values():  
5     print(f'{rank} is a rank on our crew.')  
6
```

```
Captain is a rank on our crew.  
Commander is a rank on our crew.  
Lt. Commander is a rank on our crew.
```

Dictionary nesting

- Dictionaries can be nested

```
students = {'Harry': {'Potions': 88, 'Dark Arts': 83}, 'Hermione': {'Dark Arts': 100, 'Potions': 100},  
           'Ron': {'Potions': 67, 'Dark Arts': 74}}
```

- Lookup values using keys

```
3 students = {'Harry': {'Potions': 88, 'Dark Arts': 83}, 'Hermione': {'Dark Arts': 100, 'Potions': 100},  
4           'Ron': {'Potions': 67, 'Dark Arts': 74}}  
5  
6 print(students['Harry']['Potions'])
```

88