Lists

Reference:

- https://docs.python.org/3/library/stdtypes.html#lists
- https://docs.python.org/3/tutorial/datastructures.html?highlight=lists#more-on-lists

A `List` represents a numbered, ordered collection of items. A List may contain zero or more items. A list can contain items of any datatype, but as a best practice, all items in a list should share a datatype and structure:

```
# DO:

[]

[1, 2, 3, 4]

[100, 75, 33]

["fun", "times", "right?"]

[{"a": 1, "b": 2}, {"a": 5, "b": 6}] # lists can contain dictionaries

[[1, 2, 3], [4, 5, 6], [7, 8, 9]] # lists can be "nested" inside other lists

# DON'T:

[100, "fun"] # mixed datatypes

[{"a": 1, "b": 2}, {"x": 5, "z": 6}] # non-standard dictionary keys
```

Like other languages, individual list items can be accessed by their index. List item indices are zero-based, meaning the index of the first list item is 0.

```
arr = ["a", "b", "c", "d"]
arr[0] #> "a"
arr[1] #> "b"
arr[2] #> "c"
arr[3] #> "d"
arr[4] #> IndexError: list index out of range

arr.index("a") #> 0
arr.index("b") #> 1
```

```
arr.index("c") #> 2
arr.index("z") #> -1 (applies to any item not found in the list)
```

Equality operators apply:

```
[1,2,3] == [1,2,3] #> True
[1,2,3] == [3,2,1] #> False
```

Inclusion operators apply:

```
arr = [1,2,3,4,5]

3 in arr #> True

3 not in arr #> False
```

Common list functions and operators include the following built-in Python functions:

```
arr = [6,3,9,7]

len(arr) #> 4

min(arr) #> 3

max(arr) #> 9
```

Add an element to the end of a list:

```
arr = ["a", "b", "c", "d"]
arr.append("e") # this is a mutating operation
arr #> ["a", "b", "c", "d", "e"]
```

Remove an element from a list by specifying the index of the item you would like to remove:

```
arr = ["a", "b", "c", "d"]
del arr[1] # this is a mutating operation
arr #> ['a', 'c', 'd']
```

Concatenate two lists:

```
arr = ["a", "b", "c", "d"]
arr2 = ["x", "y", "z"]
arr3 = arr + arr2
arr3 #> ["a", "b", "c", "d", "x", "y", "z"]
```

Remove duplicate values in a list by converting it to another datatype called a "Set" (which rejects non-unique values), and then converting it back to a "List":

```
arr = [1,2,2,2,3]
arr2 = list(set(arr))
arr2 #> [1, 2, 3]

list(set(["hello", "hello", "hello"])) #> ['hello']
```

Sorting

Sort a list:

```
arr = [6,3,8]
arr.sort() # this is mutating
arr #> [3, 6, 8]

arr.reverse() # this is mutating
arr #> [8, 6, 3]
```

If you have a list of dictionaries, you should be able to sort it based on dictionary values:

```
teams = [
    {"city": "New York", "name": "Yankees"},
    {"city": "New York", "name": "Mets"},
    {"city": "Boston", "name": "Red Sox"},
    {"city": "New Haven", "name": "Ravens"}
]

def team_name(team):
    return team["name"]

def sort_by_hometown(team):
    return team["city"]
```

```
def sort_special(team):
    return team["city"] + "-" + team["name"]

teams2 = sorted(teams, key=team_name)
teams2 #> [{'city': 'New York', 'name': 'Mets'}, {'city': 'New Haven',
'name': 'Ravens'}, {'city': 'Boston', 'name': 'Red Sox'}, {'city': 'New
York', 'name': 'Yankees'}]

teams3 = sorted(teams, key=sort_by_hometown)
teams3 #> [{'city': 'Boston', 'name': 'Red Sox'}, {'city': 'New Haven',
'name': 'Ravens'}, {'city': 'New York', 'name': 'Yankees'}, {'city': 'New
York', 'name': 'Mets'}]

teams4 = sorted(teams, key=sort_special)
teams4 #> [{'city': 'Boston', 'name': 'Red Sox'}, {'city': 'New Haven',
'name': 'Ravens'}, {'city': 'New York', 'name': 'Mets'}, {'city': 'New York',
'name': 'Yankees'}]
```

Alternatively for simple attribute-based sorting, you could use the `operator` module's `itemgetter()` function, for example:

```
import operator

teams = [
    {"city": "New York", "name": "Yankees"},
    {"city": "New York", "name": "Mets"},
    {"city": "Boston", "name": "Red Sox"},
    {"city": "New Haven", "name": "Ravens"}
]

teams = sorted(teams, key=operator.itemgetter('city'))
teams #> [{'city': 'Boston', 'name': 'Red Sox'}, {'city': 'New Haven',
'name': 'Ravens'}, {'city': 'New York', 'name': 'Yankees'}, {'city': 'New
York', 'name': 'Mets'}]
```

Iteration

Reference:

- https://docs.python.org/3/tutorial/datastructures.html?#list-comprehensions
- https://docs.python.org/3/tutorial/datastructures.html?highlight=lists#list-comprehensions
- https://docs.python.org/3/tutorial/controlflow.html#for-statements

A list can be iterated, or "looped" using a `for ... in ...` statement:

```
```

```
for letter in ["a", "b", "c", "d"]:
    print(letter)

#> a
#> b
#> c
#> d
```

> TIP: If it helps, you can vocalize this like "for each item in the list of items, do something with that item"

A common pattern is to loop through one list to populate the contents of another:

```
arr = [1, 2, 3, 4]
arr2 = []

for i in arr:
  arr2.append(i * 100)

arr #> [1, 2, 3, 4]
arr2 #> [100, 200, 300, 400]
```

Mapping

Lists can be looped "in-place" using Python's built-in `map()` function. The `map()` function takes two parameters. The first parameter is the name of a pre-defined function to perform on each item in the list. The function should accept a single parameter representing a single list item. The second parameter is the actual list to be operated on:

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

def enlarge(num):
    return num * 100

arr2 = map(enlarge, arr)
arr2 #> <map object at 0x10c62e710>
list(arr2) #> [100, 200, 300, 400]
```

> NOTE: remember to use the `return` keyword in your mapping function!

Another way of mapping is to use a list comprehension:

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

[i * 100 for i in arr] #> [100, 200, 300, 400]
```

```
teams = [
    {"city": "New York", "name": "Yankees"},
    {"city": "New York", "name": "Mets"},
    {"city": "Boston", "name": "Red Sox"},
    {"city": "New Haven", "name": "Ravens"}
]

[team["name"] for team in teams] #> ['Yankees', 'Mets', 'Red Sox', 'Ravens']
```

Filtering

Reference: https://docs.python.org/3/library/functions.html#filter.

Use the `filter()` function to select a subset of items from a list - only those items matching a given condition. The filter function accepts the same parameters as the `map()` fuction:

```
arr = [1,2,4,8,16]

def all_of_them(i):
    return True # same as ... return i == i

def equals_two(i):
    return i == 2

def greater_than_two(i):
    return i > 2

def really_big(i):
    return i > 102

filter(all_of_them, arr) #> <filter at 0x103fa71d0>
list(filter(all_of_them, arr)) #> [1, 2, 4, 8, 16]
list(filter(equals_two, arr)) #> [2]
list(filter(greater_than_two, arr)) #> [4, 8, 16]
list(filter(really_big, arr)) #> []
```

> Note: depending on how many items matched the filter condition, the resulting filtered list may be empty, or it may contain one item, or it may contain multiple items

When using the filter function, observe this alternative filtering syntax involving the keyword `lambda`:

```
arr = [1,2,4,8,16]
list(filter(lambda i: i > 2, arr)) #> #> [4, 8, 16]
```

If your list is full of dictionaries, you can `filter()` based on their attribute values:

```
teams = [
    {"city": "New York", "name": "Yankees"},
    {"city": "New York", "name": "Mets"},
    {"city": "Boston", "name": "Red Sox"},
    {"city": "New Haven", "name": "Ravens"}
]

def yanks(obj):
    return obj["name"] == "Yankees"

def from_new_york(obj):
    return obj["city"] == "New York"

def from_new_haven(obj):
    return obj["city"] == "New Haven"

def from_new_something(obj):
    return "New" in obj["city"]

list(filter(yanks, teams)) #> [{...}]
list(filter(from_new_york, teams)) #> [{...}, {...}]
list(filter(from_new_haven, teams)) #> [{...}]
list(filter(from_new_something, teams)) #> [{...}, {...}, {...}]
```

If you need to implement complex filtering conditions, consider using a list comprehension, or "lambda" syntax, or consider writing out your function the long way:

```
teams = [
    {"city": "New York", "name": "Yankees"},
    {"city": "New York", "name": "Mets"},
    {"city": "Boston", "name": "Red Sox"},
    {"city": "New Haven", "name": "Ravens"}
]

# using a list comprehension
def teams_from(city):
  return [team for team in teams if team["city"] == city]

# using "lambda" syntax
def teams_from2(city):
  return list(filter(lambda team: team["city"] == city, teams))

# the long way
def teams_from3(city):
  matches = []
```

```
    for team in teams:
        if team["city"].upper() == city.upper():
            matches.append(team)
    return matches

print(teams_from("New York")) #> [{'city': 'New York', 'name': 'Yankees'},
{'city': 'New York', 'name': 'Mets'}]
print(teams_from2("New York")) #> [{'city': 'New York', 'name': 'Yankees'},
{'city': 'New York', 'name': 'Mets'}]
print(teams_from3("New York")) #> [{'city': 'New York', 'name': 'Yankees'},
{'city': 'New York', 'name': 'Mets'}]
```

Grouping

Reference the `itertools` module for additional operations.