Mastering Python Data Structure

Lists

Cahya Alkahfi

Table of contents

What is a list?

In Python, a list is a data type that serves as a **collection** of elements. Lists are defined by enclosing elements within square brackets [ ], and elements are separated by commas. Lists are highly dynamic data structures that can hold elements of any data type, including numbers, strings, objects, or even other lists.

mutable: lists are mutable which means we can modify, add, or remove elements from a list.

ordered: the elements are arranged in a specific sequence, and this order is maintained when accessing the list's elements.

duplicate elements: a list can store duplicate elements and the order of elements is maintained

Creating a list

Lists can be created using square brackets **[]**, with each element written inside them and separated by commas. Lists can also be created using the **list** function. This function is typically used to convert other iterable objects into lists.

```
# empty list using []
empty_list = []
print(empty_list)

# empty list using list()
empty_list = list()
print(empty_list)

# list with 5 identical elements
identical_list = [100] * 5
print(identical_list)

# list with 4 elements
pokemons = ["Pikachu", "Bulbasaur", "Charmander", "Squirtle"]
print(pokemons)

# list with different data types
mixed_list = ["Pikachu", "Electric", 0.4, 6.0, 35, True]
print(mixed_list)
```

```
[]
[]
[100, 100, 100, 100, 100]
['Pikachu', 'Bulbasaur', 'Charmander', 'Squirtle']
['Pikachu', 'Electric', 0.4, 6.0, 35, True]
```

More complex lists

Lists can hold elements of any data type including lists or dictionaries. Here are examples of lists of lists and lists of dictionaries.

```
# list of lists
pokemons = [
    ["Pikachu", "Raichu"],
    ["Charmander", "Charmeleon", "Charizard"],
    ["Bulbasaur", "Ivysaur", "Venusaur"],
    ["Squirtle", "Wartortle", "Blastoise"],
]
```

```
# list of dicts
pokemons = [
    { 'name': 'Pikachu',    'type': 'Electric',
      'HP': 60, 'atk': 55, 'def': 40 },
    { 'name': 'Charmander', 'type': 'Fire',
      'HP': 58, 'atk': 52, 'def': 43 },
    { 'name': 'Squirtle',   'type': 'Water',
      'HP': 56, 'atk': 48, 'def': 65 },
    { 'name': 'Bulbasaur',  'type': 'Grass',
      'HP': 64, 'atk': 49, 'def': 49 }
]
```

Accessing elements (1/3)

List elements can be accessed based on their index order using the syntax name_list[index]. Indexing in lists starts from 0. This means the 1st element will have an index of 0, the 2nd element with an index of 1, and so on. Lists can also be accessed using negative indexes. Index -1 indicates the last element in the list, index -2 the second-to-last element, and so forth.

```
pokemons = ["Pikachu", "Bulbasaur", "Charmander", "Squirtle", "Snorlax"]

first_el = pokemons[0]
print(first_el)             # Pikachu

third_el = pokemons[2]
print(third_el)             # Charmander

# negative index
last_el = pokemons[-1]
print(last_el)              # Snorlax (last element)

fourth_el = pokemons[-2]
print(fourth_el)            # Squirtle (2nd last element)
```

```
Pikachu
Charmander
Snorlax
Squirtle
```

Accessing elements (2/3)

In a list of lists, we can access elements in the same way. We can further access elements from its main elements.

```
# List of lists
pokemons = [
    ["Pikachu", "Raichu"],
    ["Charmander", "Charmeleon", "Charizard"],
    ["Bulbasaur", "Ivysaur", "Venusaur"],
    ["Squirtle", "Wartortle", "Blastoise"],
]

# first element
first_el = pokemons[0]          # ["Pikachu", "Raichu"]
print(first_el)

last_el = pokemons[-1]          # ["Squirtle", "Wartortle", "Blastoise"]
print(last_el)

# accessing elements of element
pikachu = pokemons[0][0]        # Pikachu
print(pikachu)

venusour = pokemons[2][-1]      # Venusaur
print(venusour)

# slicing elements
slice_el = pokemons[2][:2]      # ['Bulbasaur', 'Ivysaur']
print(slice_el)
```

Accessing elements (3/3)

In a list of dictionaries, we can access elements in the same way. We can further access any key value from its elements.

```
# list of dicts
pokemons = [
    { 'name': 'Pikachu',    'type': 'Electric',
      'HP': 60, 'atk': 55, 'def': 40 },
    { 'name': 'Charmander', 'type': 'Fire',
      'HP': 58, 'atk': 52, 'def': 43 },
    { 'name': 'Squirtle',   'type': 'Water',
      'HP': 56, 'atk': 48, 'def': 65 },
    { 'name': 'Bulbasaur',  'type': 'Grass',
      'HP': 64, 'atk': 49, 'def': 49 }
]

# first element
first_el = pokemons[0]                  # { 'name': 'Pikachu', ... }
print(first_el)

last_el = pokemons[-1]                  # {'name': 'Bulbasaur', ...}
print(last_el)

# accessing elements of element
pikachu = pokemons[0]['name']           # Pikachu
print(pikachu)

bulbasaur_type = pokemons[-1]['type']   # Grass
print(bulbasaur_type)
```

Slicing list

A list can be sliced to take only a few elements. Slicing a list is done using a starting index and an ending index. The starting index is inclusive, while the ending index is exclusive. For example, to slice the first to the third element, the starting index is 0, and the ending index is 3. We can omit the starting index when slicing from the first element, and likewise, we can omit the ending index when slicing to the last element. Slicing can also be performed using negative indices.

```
pokemons = ["Pikachu", "Bulbasaur", "Charmander", "Squirtle", "Snorlax"]

# for slicing from 0-2 use [0:3]
slice_el_a = pokemons[0:3]
print(slice_el_a)                 # ['Pikachu', 'Bulbasaur', 'Charmander']

# slicing from the first element
# `start` index can be omited
slice_el_b = pokemons[:3]
print(slice_el_b)                 # ['Pikachu', 'Bulbasaur', 'Charmander']

# slicing 3rd & 4th elements
# (2nd & 3rd indices)
slice_el_c = pokemons[2:4]
print(slice_el_c)                 # ['Charmander', 'Squirtle']

# slicing to the last element
# `end` index can be omited
slice_el_d = pokemons[2:]
print(slice_el_d)                 # ['Charmander', 'Squirtle', `Snorlax]

# slicing with negative indices
slice_el_e = pokemons[-4:-2]
print(slice_el_e)                 # ['Bulbasaur', 'Charmander']
```

Iterating through list

List elements can be accessed sequentially through a for-loop iteration. During each iteration, we can access the list elements directly. We can also access the index (counter) and value of each element by using enumerate function

```
pokemons = ["Pikachu", "Bulbasaur", "Charmander", "Squirtle", "Snorlax"]

for pokemon in pokemons:
    print(f"Current pokemon -> {pokemon}")
```

```
Current pokemon -> Pikachu
Current pokemon -> Bulbasaur
Current pokemon -> Charmander
Current pokemon -> Squirtle
Current pokemon -> Snorlax
```

```
pokemons = ["Pikachu", "Bulbasaur", "Charmander", "Squirtle", "Snorlax"]

for idx, pokemon in enumerate(pokemons):
  print(f"{idx+1} -> {pokemon}")
```

```
1 -> Pikachu
2 -> Bulbasaur
3 -> Charmander
4 -> Squirtle
5 -> Snorlax
```

Adding elements

Adding elements to a list can be done using the **append** method, **insert** method, or adding with another list object.

```
pokemons = ["Pikachu"]
print(f"Pokemons : {pokemons}\n")
```

```
Pokemons : ['Pikachu']
```

```
# appending an element
print("Append : Bulbasaur & Squirtle")
pokemons.append("Bulbasaur")
pokemons.append("Squirtle")
print(f"Pokemons : {pokemons}\n")
```

```
Append : Bulbasaur & Squirtle
Pokemons : ['Pikachu', 'Bulbasaur', 'Squirtle']
```

```
# inserting an element
print("Insert : Charmander at 2nd index")
pokemons.insert(2, "Charmander")
print(f"Pokemons : {pokemons}\n")
```

```
Insert : Charmander at 2nd index
Pokemons : ['Pikachu', 'Bulbasaur', 'Charmander', 'Squirtle']
```

```
# combining 2 lists
print("Adding Snorlax & Onyx")
new_pokemons = ["Snorlax", "Onyx"]
pokemons = pokemons + new_pokemons
print(f"Pokemons : {pokemons}\n")
```

```
Adding Snorlax & Onyx
Pokemons : ['Pikachu', 'Bulbasaur', 'Charmander', 'Squirtle', 'Snorlax', 'Onyx']
```

Modifying elements

Modifying list elements can be done in the same way as accessing or slicing those elements. Select the element based on the desired index and assign a new value to it

```
pokemons = ["Pikachu", "Bulbasaur", "Charmander", "Squirtle"]
print(f"Pokemon : {pokemons}\n")

# modifying single element
pokemons[0] = "Raichu"
pokemons[-1] = "Wartortle"
print(f"Pokemon : {pokemons}\n")

# modifying multiple elements
pokemons[1:3] = ["Ivysaur", "Charmeleon"]
print(f"Pokemon : {pokemons}")
```

```
Pokemon : ['Pikachu', 'Bulbasaur', 'Charmander', 'Squirtle']

Pokemon : ['Raichu', 'Bulbasaur', 'Charmander', 'Wartortle']

Pokemon : ['Raichu', 'Ivysaur', 'Charmeleon', 'Wartortle']
```

Deleting elements

Deleting an element at a specific index can be done using the **pop** method. If we want to remove based on a specific value, then we use the **remove** method.

```
pokemons = ['Pikachu', 'Bulbasaur', 'Charmander', 'Squirtle', 'Snorlax', 'Onyx']

# deleting element at spesific index using `pop`
popped_pokemon = pokemons.pop(-2)    # -> pop Snorlax
print(f"Popped pokemon: {popped_pokemon}")
print(f"Pokemons: {pokemons}\n")

# if the index is not specify, it will remove the last element
popped_pokemon = pokemons.pop()      # pop Onyx
print(f"Popped pokemon : {popped_pokemon}")
print(f"Pokemons: {pokemons}\n")

# removing element based on any value
pokemons.remove("Bulbasaur")         # remove Bulbasaur
print(f"Pokemons: {pokemons}\n")

# Clearing all elements
pokemons.clear()
print("Clearing all elements:")
print(f"Pokemons: {pokemons}")
```

```
Popped pokemon: Snorlax
Pokemons: ['Pikachu', 'Bulbasaur', 'Charmander', 'Squirtle', 'Onyx']

Popped pokemon : Onyx
Pokemons: ['Pikachu', 'Bulbasaur', 'Charmander', 'Squirtle']

Pokemons: ['Pikachu', 'Charmander', 'Squirtle']

Clearing all elements:
Pokemons: []
```

Reordering elements (1/2)

The order of elements in a list can be rearranged using **reverse** or **sort** methods.

- **reverse**: method to reverse the order of the elements.
- **sort**: method to sort the elements in ascending or descending order

```python
# Reordering elements
pokemons = ["Pikachu", "Bulbasaur", "Charmander", "Squirtle"]
print(f"Original : {pokemons}\n")

# reverse
pokemons.reverse()
print(f"Reversed : {pokemons}\n")

# sort ASC
pokemons.sort()
print(f"Sorted (Ascending): {pokemons}\n")

# sort DESC
pokemons.sort(reverse=True)
print(f"Sorted (Descending): {pokemons}\n")
```

```
Original : ['Pikachu', 'Bulbasaur', 'Charmender', 'Squirtle']

Reversed : ['Squirtle', 'Charmender', 'Bulbasaur', 'Pikachu']

Sorted (Ascending): ['Bulbasaur', 'Charmender', 'Pikachu', 'Squirtle']

Sorted (Descending): ['Squirtle', 'Pikachu', 'Charmender', 'Bulbasaur']
```

Reordering elements (2/2)

When using the **sort** method, we can customize how the elements are sorted by specifying the key parameter.

```
# Reordering elements
pokemons = ["Pikachu", "Bulbasaur", "Charmander", "Squirtle"]
print(f"Original : {pokemons}\n")

# sort custom by passing a function to `key` parameter
# let's say we want to sort it by the length of its name descendingly
pokemons.sort(key=lambda el : len(el), reverse=True)
print(f"Sorted (custom): {pokemons}")
```

```
Original : ['Pikachu', 'Bulbasaur', 'Charmender', 'Squirtle']

Sorted (custom): ['Charmender', 'Bulbasaur', 'Squirtle', 'Pikachu']
```

```
# list of dictionaries
pokemons = [
    {"name": "Pikachu",   "hp": 35},
    {"name": "Bulbasaur", "hp": 45},
    {"name": "Charmander","hp": 39},
    {"name": "Squirtle",  "hp": 44},
]

# we can sort it by any keys (example by `hp`)
pokemons.sort(key=lambda el: el["hp"], reverse=True)
print(pokemons)
```

```
[{'name': 'Bulbasaur', 'hp': 45}, {'name': 'Squirtle', 'hp': 44},
 {'name': 'Charmander', 'hp': 39}, {'name': 'Pikachu', 'hp': 35}]
```

Checking Elements

count: method to count how many occurrences of a specific value exist in a list

```
data = [90, 80, 80, 60, 80, 90, 70]

count_80 = data.count(80)
print(f"There are {count_80} elements with a value of 80")
```

```
There are 3 elements with a value of 80
```

index: method to find the first index that contains a specific value

```
data = [90, 80, 80, 60, 80, 90, 70]

idx_60 = data.index(60)
print(f"60 is at index {idx_60}")

idx_80 = data.index(80)
print(f"80 is at index {idx_80}")
```

```
60 is at index 3
80 is at index 1
```

in (not in): operator to check whether one or more elements exist in a list

```
pokemons = ["Pikachu", "Bulbasaur", "Charmander", "Squirtle"]

print(f"Pikachu exists?        {'Pikachu' in pokemons}")
print(f"Snorlax exists?        {'Snorlax' in pokemons}")
print(f"Snorlax doesn't exist? {'Snorlax' not in pokemons}")
```

```
Pikachu exists?        True
Snorlax exists?        False
Snorlax doesn't exist? True
```

len & sum

len: function to get the number of elements in a list (or other iterable)

```
data = [90, 80, 80, 60, 80, 90, 70]

print(f"Number of elements: {len(data)}")
```

```
Number of elements: 7
```

sum: function to get the sum of values in the elements of a list (or other iterable)

```
data = [90, 80, 80, 60, 80, 90, 70]

print(f"Sum of elements: {sum(data)}")
```

```
Sum of elements: 550
```

min & max

min: function to get the element with the smallest value from a list (or other iterable)
max: function to get the element with the largest value from a list (or other iterable)

```
data = [90, 80, 80, 60, 80, 90, 70]

print(f"Sum of elements: {sum(data)}")
```

```
Minimum value: 60
Maximum value: 90
```

The min and max functions have a `key` parameter where we can pass a function how the smallest or largest values are measured. This allows us to find the minimum and maximum values in a list with more complex elements

```
pokemons = [
    {"name": "Pikachu",    "hp": 35},
    {"name": "Bulbasaur", "hp": 45},
    {"name": "Charmander","hp": 39},
    {"name": "Squirtle",  "hp": 44},
]

min_hp = min(pokemons, key=lambda el: el['hp'])
max_hp = max(pokemons, key=lambda el: el['hp'])

print(f"Pokemon with lowest HP: {min_hp['name']}")
print(f"Pokemon with highest HP: {max_hp['name']}")
```

```
Pokemon with lowest HP: Pikachu
Pokemon with highest HP: Bulbasaur
```

reversed & sorted

reversed: function to reverse the order of the elements (similar to reverse method, but the reversed function doesn't change the original list and It returns a new iterator object)

```
pokemons = ["Pikachu", "Bulbasaur", "Charmander", "Squirtle"]

reversed_pokemons = list(reversed(pokemons))

print(f"Original : {pokemons}\n")
print(f"Reversed : {reversed_pokemons}\n")
```

```
Original : ['Pikachu', 'Bulbasaur', 'Charmander', 'Squirtle']
Reversed : ['Squirtle', 'Charmander', 'Bulbasaur', 'Pikachu']
```

sorted: function to sort the elements (similar to sort method, but the sorted function doesn't change the original list and It returns a new iterator object)

```
pokemons = ["Pikachu", "Bulbasaur", "Charmander", "Squirtle"]

# sort ASC
pokemons_asc = sorted(pokemons)
print(f"Sorted (Ascending) : {pokemons_asc}\n")

# sort DESC
pokemons_desc = sorted(pokemons, reverse=True)
print(f"Sorted (Descending): {pokemons_desc}\n")

# sort custom by passing a function to `key` parameter
# let's say we want to sort it by the length of its name descendingly
pokemons_custom = sorted(pokemons, key=lambda el : len(el), reverse=True)
print(f"Sorted (custom)    : {pokemons_custom}")
```

```
Sorted (Ascending) : ['Bulbasaur', 'Charmander', 'Pikachu', 'Squirtle']
Sorted (Descending): ['Squirtle', 'Pikachu', 'Charmander', 'Bulbasaur']
Sorted (custom)    : ['Charmander', 'Bulbasaur', 'Squirtle', 'Pikachu']
```

map (1/2)

The **map** function is used to apply a given function to each element of a list (or other iterable objects) and returns a new iterable with the results. It allows us to perform the same operation on each element without the need to write a loop explicitly.

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

# creating a new list by squaring `numbers` list
squared = list(map(lambda x: x**2, numbers))
print(squared)
```

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

```
scores = [40, 60, 30, 80, 60]
tot = sum(scores)

# function to calculate the proportion of each element
calculate_props = lambda x: round(x / tot, 2)

props = list(map(calculate_props, scores))
print(f"Proportions: {props}")
```

```
Proportions: [0.15, 0.22, 0.11, 0.3, 0.22]
```

```
# function to check whether one has passed or not
# based on a specific threshold value
check_status = lambda x: "PASSED" if x > 50 else "FAILED"

status = list(map(check_status, scores))
print(status)
```

```
['FAILED', 'PASSED', 'FAILED', 'PASSED', 'PASSED']
```

map (2/2)

The map function can be used to extract specific information from complex elements within a list.

```
pokemons = [
    {"name": "Pikachu", "type": "Electric", "hp": 35},
    {"name": "Bulbasaur", "type": "Grass", "hp": 45},
    {"name": "Charmander", "type": "Fire", "hp": 39},
    {"name": "Squirtle", "type": "Water", "hp": 44},
]

pokemons_name = list(map(lambda x: x["name"], pokemons))
pokemons_hp = list(map(lambda x: x["hp"], pokemons))

print(f"Name: {pokemons_name}")
print(f"HP: {pokemons_hp}")
```

```
Name: ['Pikachu', 'Bulbasaur', 'Charmander', 'Squirtle']
HP: [35, 45, 39, 44]
```

```
def info(pokemon):
  return f"{pokemon['name']} is {pokemon['type']} pokemon"

pokemon_info = list(map(info, pokemons))
print(pokemon_info)
```

```
['Pikachu is Electric pokemon', 'Bulbasaur is Grass pokemon',
 'Charmander is Fire pokemon', 'Squirtle is Water pokemon']
```

filter

The **filter** function selects elements within an iterable based on specific criteria. It takes two arguments: the first argument is a function containing the filter criteria, and the second argument is the iterable to be filtered.

```
scores = [40, 60, 30, 80, 60]

# filtering `scores` > 50
passed_scores = list(filter(lambda x: x > 50, scores))
print(passed_scores)
```

```
[60, 80, 60]
```

```
pokemons = [
    {"name": "Pikachu", "type": "Electric", "hp": 35},
    {"name": "Bulbasaur", "type": "Grass", "hp": 45},
    {"name": "Charmander", "type": "Fire", "hp": 39},
    {"name": "Squirtle", "type": "Water", "hp": 44},
]

# filtering `pokemons` list based on `hp` key of each element
strong_pokemons= list(filter(lambda x: x["hp"] > 40 , pokemons))

print("Strong pokemon:\n")
print(strong_pokemons)
```

```
Strong pokemon:

[{'name': 'Bulbasaur', 'type': 'Grass', 'hp': 45},
 {'name': 'Squirtle', 'type': 'Water', 'hp': 44}]
```

zip (1/2)

The **zip** function is used to combine elements from one or more iterables into a set of paired tuples. This merging allows us to perform paired operations between elements at the same positions in each list. This function can accommodate two or more iterables at once. If the number of elements in each iterable is different, the zip function will stop after the shortest iterable.

```
mid_test   = [60, 90, 85, 75]
final_test = [80, 85, 70, 80]

all_test = list(zip(mid_test, final_test))
print(all_test)
```

```
[(60, 80), (90, 85), (85, 70), (75, 80)]
```

zip + **map** example:

```
average_score = list(
    map(lambda x: sum(x)/len(x), zip(mid_test, final_test)),
)

print(average_score)
```

```
[70.0, 87.5, 77.5, 77.5]
```

zip + **filter** example:

```
all_75_above = list(
    filter(lambda x: x[0] >= 75 and x[1] >= 75,
                zip(mid_test, final_test)),
)

print(all_75_above)
```

```
[(90, 85), (75, 80)]
```

zip (2/2)

zip with more than 2 list example:

```
mid_test   = [60, 90, 85, 75]
final_test = [80, 85, 70, 80]
assignment = [90, 95, 85, 90]

all_scores = list(zip(mid_test, final_test, assignment))
print(all_scores)
```

```
[(60, 80, 90), (90, 85, 95), (85, 70, 85), (75, 80, 90)]
```

zip + map example:

```
def calculate_final_score(scores):
  final_score = 0.3*scores[0] + 0.5*scores[1] + 0.2*scores[2]
  return final_score

final_scores = list(map(calculate_final_score, all_scores))
print(final_scores)
```

```
[76.0, 88.5, 77.5, 80.5]
```

all and any

all: returns **True** if all elements in an iterable are **True**; otherwise, it returns **False**.

any: returns **True** if at least one element in an iterable is **True**; otherwise, it returns **False**.

```
scores = [90, 45, 75, 80, 70, 95]

passed = list(map(lambda x: x > 50, scores))
print(f"Did everyone pass? {all(passed)}")

failed = list(map(lambda x: x <= 50, scores))
print(f"Did anyone fail? {any(failed)}")
```

```
Did everyone pass? False
Did anyone fail? True
```

List comprehension

List comprehension is a concise and powerful feature in Python that allows us to create lists with a compact and readable syntax. It enables us to generate new lists by applying expressions to each item in an existing iterable, such as a list or a range of values. This approach is often more efficient and expressive than traditional for-loops. List comprehensions make code cleaner and easier to understand, facilitating the creation and transformation of lists in a more intuitive and elegant manner.

There are three parts in a list comprehension:

- **existing_iterable**: the iterable object that will be used in the creation of a new list.
- **expression**: the code used to process each element from the existing_list to generate new elements.
- **condition** (optional): conditions can be set to filter elements from the existing_list that meet specific criteria.

new**_list = [expression for** item **in existing_iterable if condition**]

Example:

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

doubled = [2*x for x in numbers]
print(doubled)        # [2, 4, 6, 8, 10]

# using other iterable as the source
odds = [x for x in range(1, 6) if x % 2 != 0]
print(odds)        # [1, 3, 5]
```

Mapping iterable using list comprehension (1/2)

List comprehension can be used as a replacement for the **map** method with a more concise syntax.

The following examples are syntax for performing mapping using **list comprehensions** and their comparison with the **map** function.

```python
numbers = [1, 2, 3, 4, 5]

# creating a new list by squaring `numbers` list
squared = [x**2 for x in numbers]

# squared = list(map(lambda x: x**2, numbers)) # using map

print(squared)
```

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

```python
scores = [40, 60, 30, 80, 60]
tot = sum(scores)

# function to calculate the proportion of each element
calculate_props = lambda x: round(x / tot, 2)

props = [calculate_props(x) for x in scores]
print(f"Proportions: {props}")

# props = list(map(calculate_props, scores))  # using map
```

```
Proportions: [0.15, 0.22, 0.11, 0.3, 0.22]
```

Mapping iterable using list comprehension (2/2)

```python
scores = [40, 60, 30, 80, 60]

# function to check whether one has passed or not
# based on a specific threshold value
check_status = lambda x: "PASSED" if x > 50 else "FAILED"

status = [check_status(x) for x in scores]
print(f"Status: {status}")

# status = list(map(check_status, scores))  # using map
```

```
Status: ['FAILED', 'PASSED', 'FAILED', 'PASSED', 'PASSED']
```

```python
pokemons = [
    {"name": "Pikachu", "type": "Electric", "hp": 35},
    {"name": "Bulbasaur", "type": "Grass", "hp": 45},
    {"name": "Charmander", "type": "Fire", "hp": 39},
    {"name": "Squirtle", "type": "Water", "hp": 44},
]

pokemons_name = [x["name"] for x in pokemons]
pokemons_hp = [x["hp"] for x in pokemons]

print(f"Name: {pokemons_name}")
print(f"HP: {pokemons_hp}")

# pokemons_name = list(map(lambda x: x["name"], pokemons))  # using map
# pokemons_hp = list(map(lambda x: x["hp"], pokemons))      # using map
```

```
Name: ['Pikachu', 'Bulbasaur', 'Charmander', 'Squirtle']
HP: [35, 45, 39, 44]
```

Filtering iterable using list comprehension

List comprehension can also be used as a replacement for the **filter** function.

The following examples are syntax for performing filtering using **list comprehensions** and their comparison with the **filter function.**

```
scores = [40, 60, 30, 80, 60]

# filtering `scores` > 50
passed_scores = [x for x in scores if x > 50]
print(passed_scores)

# passed_scores = list(filter(lambda x: x > 50, scores))  # using filter
```

```
[60, 80, 60]
```

```
pokemons = [
    {"name": "Pikachu", "type": "Electric", "hp": 35},
    {"name": "Bulbasaur", "type": "Grass", "hp": 45},
    {"name": "Charmander", "type": "Fire", "hp": 39},
    {"name": "Squirtle", "type": "Water", "hp": 44},
]

# filtering `pokemons` list based on `hp` key of each element
strong_pokemons = [p for p in pokemons if p["hp"] > 40]

print("Strong pokemon:\n")
print(strong_pokemons)

# strong_pokemons = list(filter(lambda x: x["hp"] > 40 , pokemons))
```

```
Strong pokemon:

[{'name': 'Bulbasaur', 'type': 'Grass', 'hp': 45},
 {'name': 'Squirtle', 'type': 'Water', 'hp': 44}]
```

Mapping + Filtering

We can perform mapping and filtering simultaneously using **list comprehension**.

```
students = [
    {"name": "Andy",   "pre": 70, "post": 80},
    {"name": "Brian",  "pre": 80, "post": 85},
    {"name": "Charly", "pre": 85, "post": 90},
    {"name": "Drake",  "pre": 75, "post": 80},
    {"name": "Emily",  "pre": 75, "post": 75}
]

def student_final_score(student):
    final_score = 0.25*student["pre"] + 0.75*student["post"]
    return {
        "name": student["name"],
        "score": final_score
    }
```

map + filter

```
# using map and filter function
final_scores = list(map(student_final_score, students))
passed_students = list(filter(lambda s: s["score"] >=80, final_scores))
```

list comprehension

```
# using list comprehension
final_scores = [student_final_score(s) for s in students]
passed_students = [s for s in final_scores if s["score"] >= 80]
```

```
Passed students:
-> {'name': 'Brian', 'score': 83.75}
-> {'name': 'Charly', 'score': 88.75}
```

THANK YOU

sainsdata.id