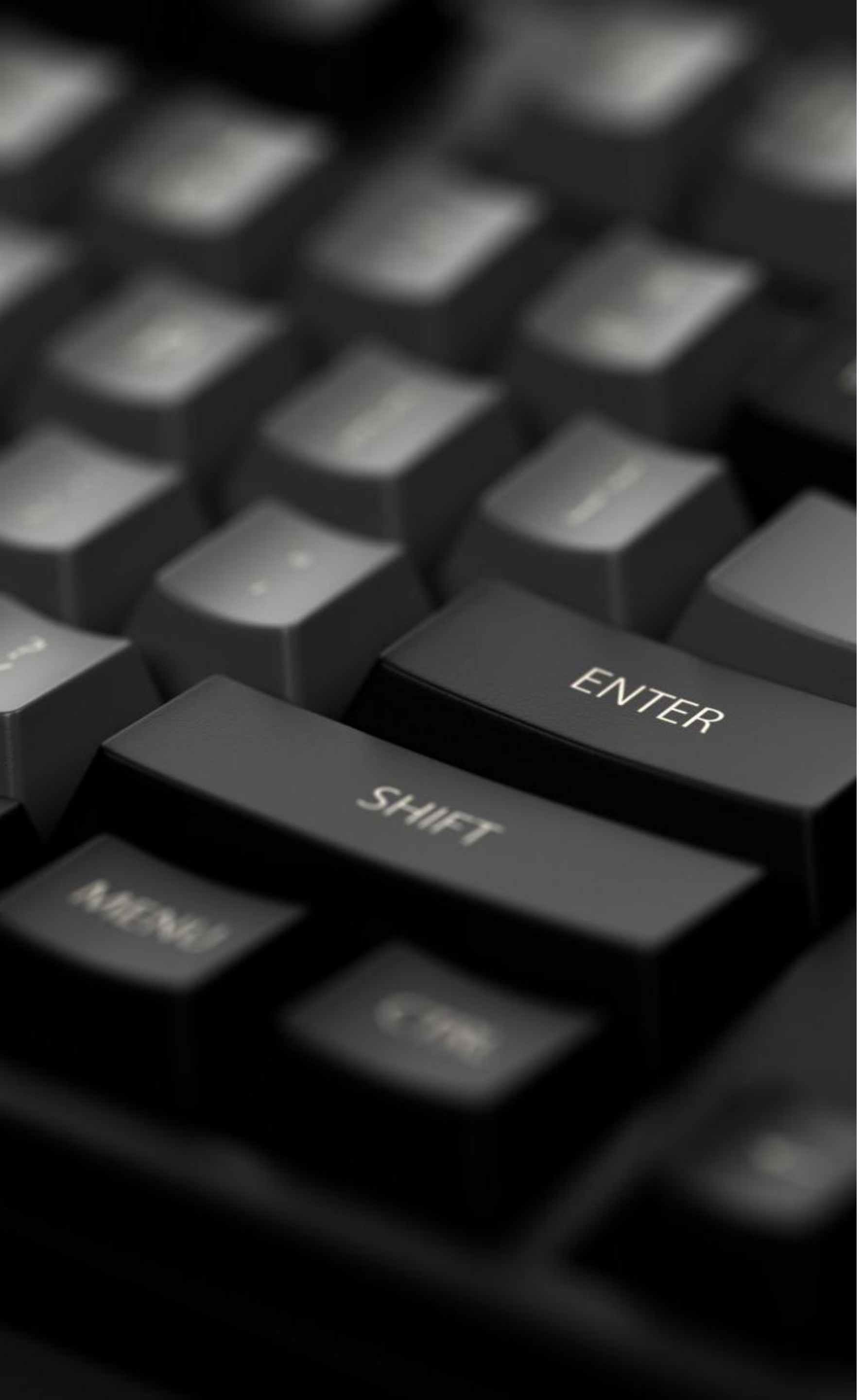
A gamified introduction to Python Programming

Lecture 14
Dictionaries and Object-Oriented Thinking

Matthieu DE MARI – Singapore University of Technology and Design

SINGAPORE UNIVERSITY OF TECHNOLOGY AND DESIGN



Outline (Chapter 14)

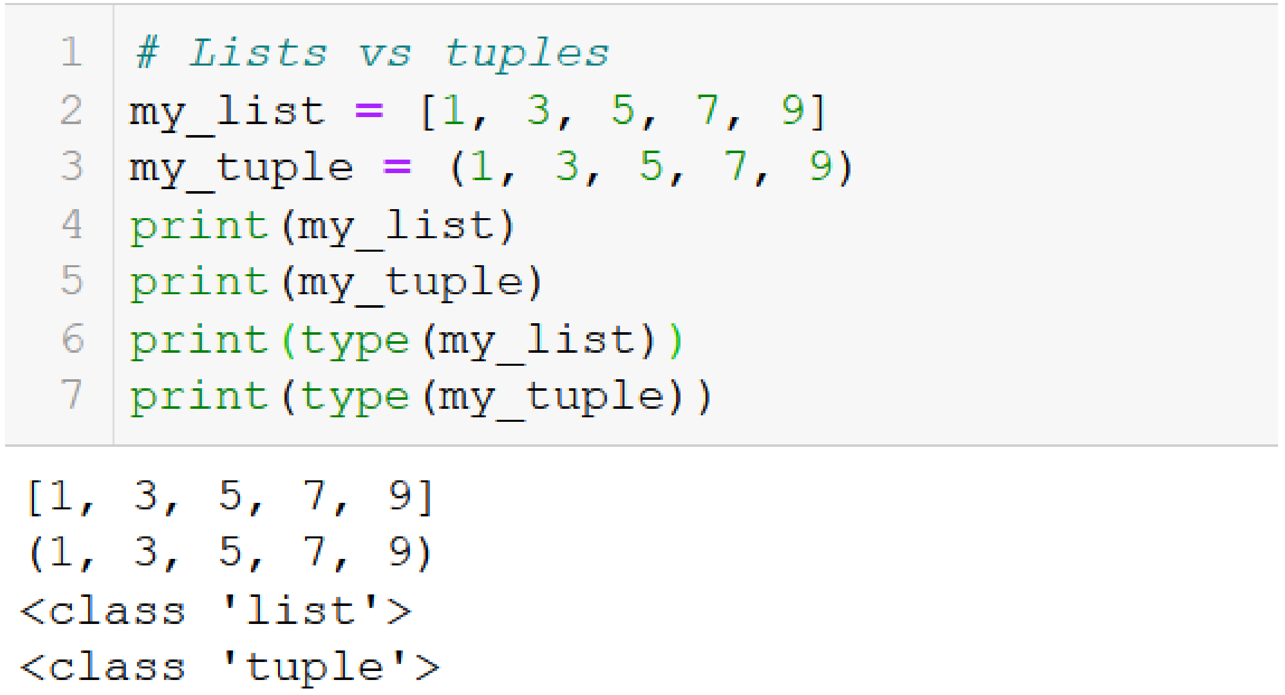
- What is the **tuple** type? (fast version)
- What is the **dictionary** type?
- What are **typical operations on dictionaries**?
- How are **dictionaries** and **the concept of object-oriented programming** related?
- What is “object-oriented thinking”?
- **Practice** on dictionaries.
- *(If time allows, what is the set types?)*

The tuple type

Definition (the tuple type):

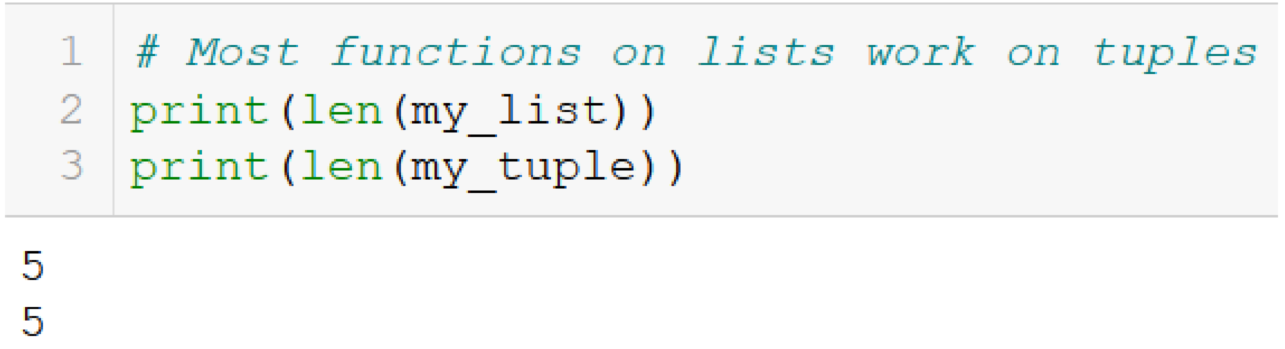
Tuples are **collections of variables**, very similar to lists, but they have their elements listed between **parentheses ()**, instead of square brackets [].

- They share many functions with lists, such as the len**()** function for instance.
- Tuples are indexed/accessed, sliced and traversed as in lists.



```python
# Lists vs tuples
my_list = [1, 3, 5, 7, 9]
my_tuple = (1, 3, 5, 7, 9)
print(my_list)
print(my_tuple)
print(type(my_list))
print(type(my_tuple))
```

```
[1, 3, 5, 7, 9]
(1, 3, 5, 7, 9)
<class 'list'>
<class 'tuple'>
```



```python
# Most functions on lists work on tuples
print(len(my_list))
print(len(my_tuple))
```

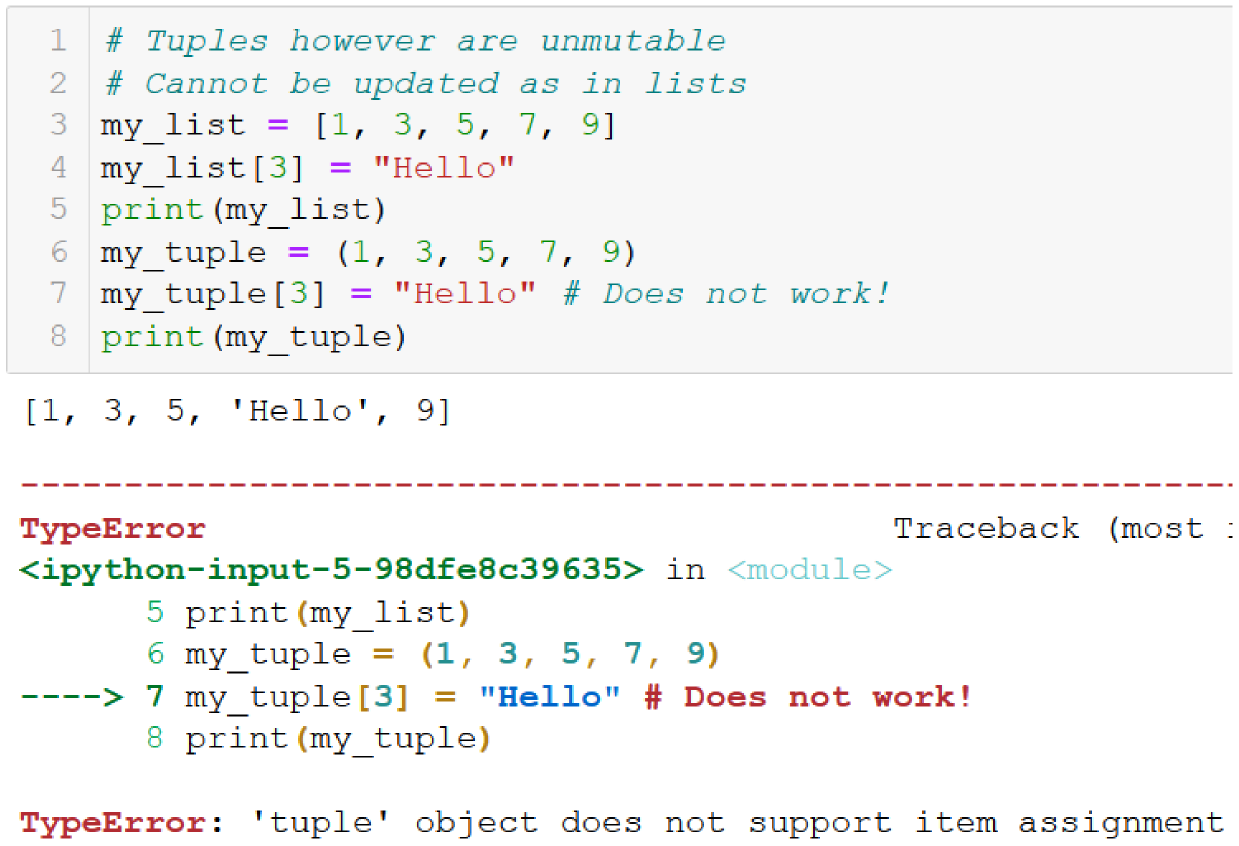
```
5
5
```

The tuple type

Definition (the tuple type):

Tuples are **collections of variables**, very similar to **lists**, but they have their elements listed between **parentheses ()**, instead of square brackets **[]**.

However, the **difference** between lists and tuples is that tuples, just like strings, are **immutable**.



```
1  # Tuples however are unmutable
2  # Cannot be updated as in lists
3  my_list = [1, 3, 5, 7, 9]
4  my_list[3] = "Hello"
5  print(my_list)
6  my_tuple = (1, 3, 5, 7, 9)
7  my_tuple[3] = "Hello" # Does not work!
8  print(my_tuple)
```

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

---------------------------------------------------------------------------
TypeError                                 Traceback (most
<ipython-input-5-98dfe8c39635> in <module>
      5 print(my_list)
      6 my_tuple = (1, 3, 5, 7, 9)
----> 7 my_tuple[3] = "Hello" # Does not work!
      8 print(my_tuple)

TypeError: 'tuple' object does not support item assignment
```

→ Added practice on tuples (for those interested) in the extra practice folder on today's materials!

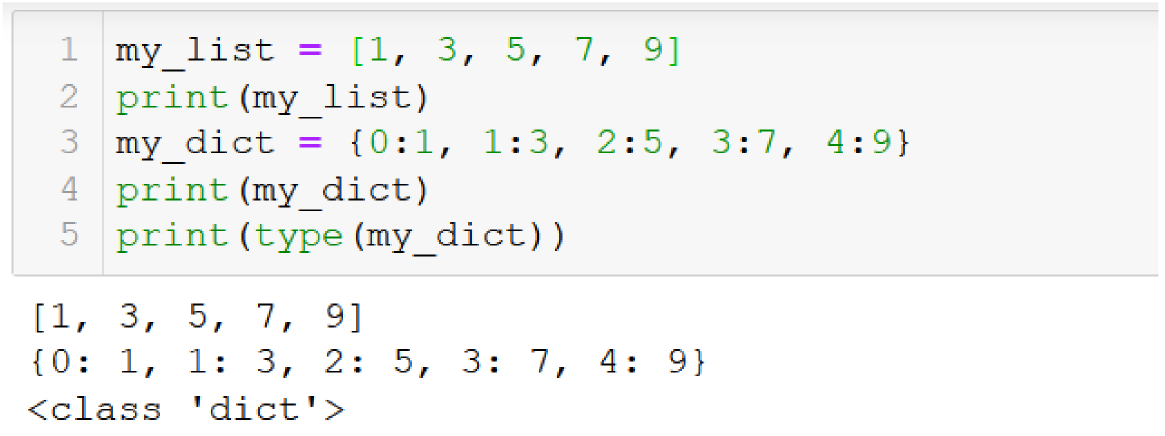
The dict type

Definition (the dictionary type):

Dictionaries (dict) are the last type of built-in variables we will cover during this course.

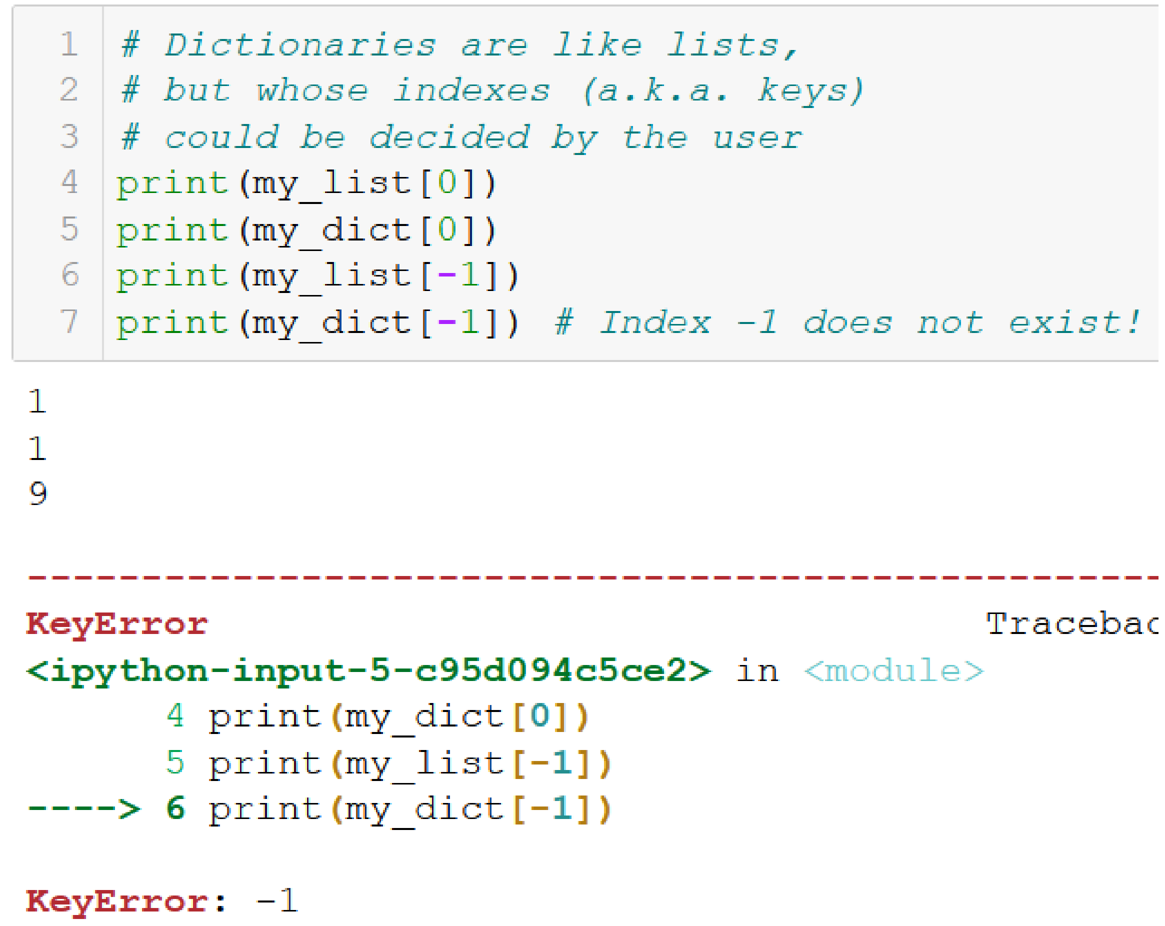
They operate as lists.

But they give the user the possibility to choose **<u>custom indexes/keys</u>** to use for elements instead of the default list values for positional indexes.



```python
my_list = [1, 3, 5, 7, 9]
print(my_list)
my_dict = {0:1, 1:3, 2:5, 3:7, 4:9}
print(my_dict)
print(type(my_dict))
```

```
[1, 3, 5, 7, 9]
{0: 1, 1: 3, 2: 5, 3: 7, 4: 9}
<class 'dict'>
```



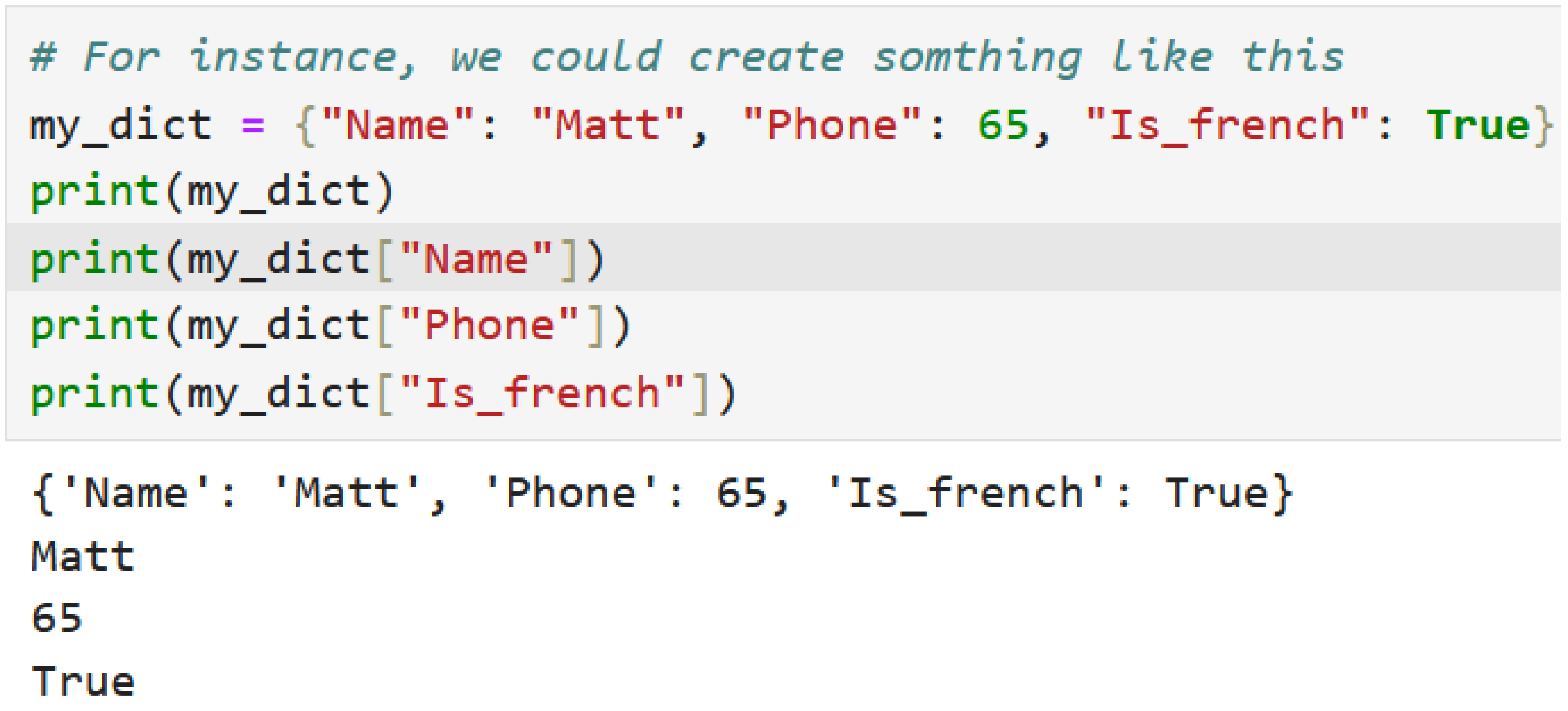
```python
# Dictionaries are like lists,
# but whose indexes (a.k.a. keys)
# could be decided by the user
print(my_list[0])
print(my_dict[0])
print(my_list[-1])
print(my_dict[-1]) # Index -1 does not exist!
```

```
1
1
9

---------------------------------------------------------
KeyError                                         Tracebac
<ipython-input-5-c95d094c5ce2> in <module>
      4 print(my_dict[0])
      5 print(my_list[-1])
----> 6 print(my_dict[-1])

KeyError: -1
```

The dict type

Dictionaries (**dict**) operate as **lists**. But they give the user the possibility to choose **custom indexes/keys** to use for elements.

- Indexes do not have to be integers, they could be strings, for instance.



```python
# For instance, we could create somthing like this
my_dict = {"Name": "Matt", "Phone": 65, "Is_french": True}
print(my_dict)
print(my_dict["Name"])
print(my_dict["Phone"])
print(my_dict["Is_french"])
```

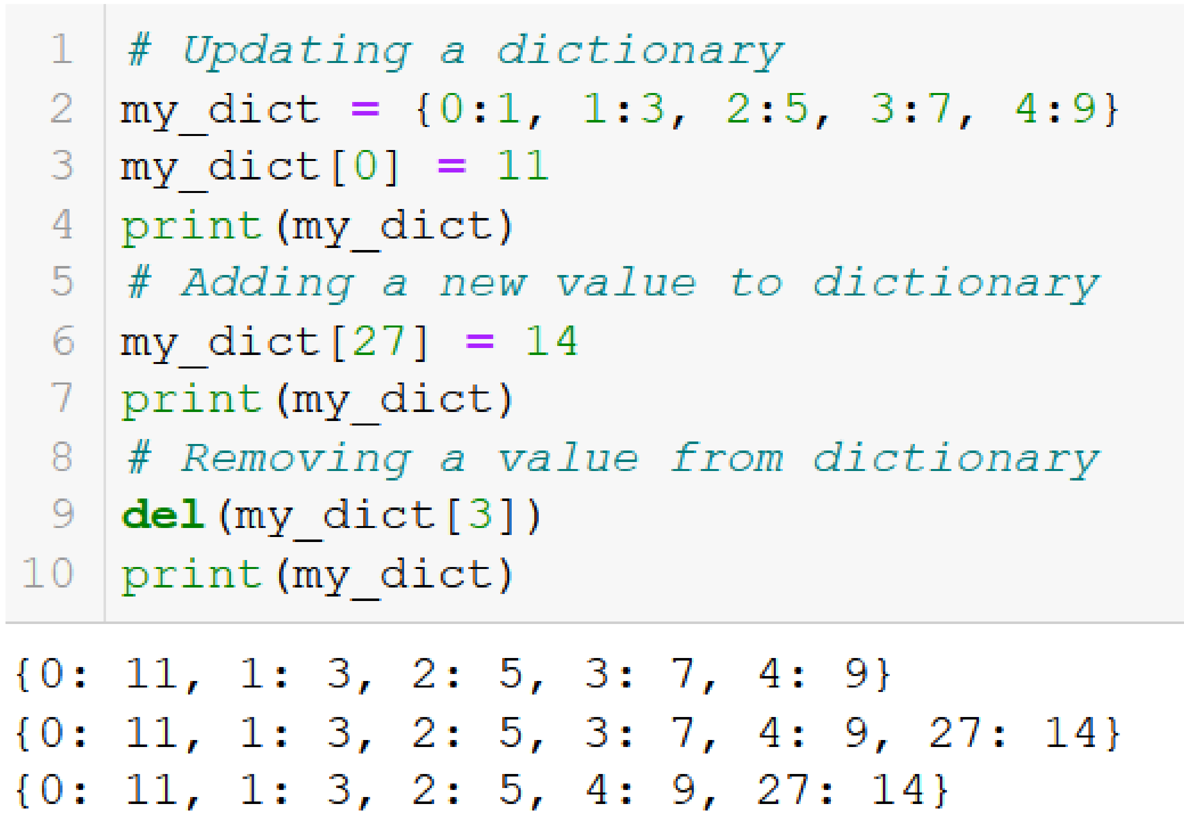
```
{'Name': 'Matt', 'Phone': 65, 'Is_french': True}
Matt
65
True
```

Adding, updating and removing elements

As with lists, we can update elements using the squared brackets **[]** notation.

- During an update, If the **key** did not exist in the dictionary, it will **add a new entry to the dictionary**.
- Otherwise, it will **replace the current value assigned to this key.**

As with lists, we can also remove an element from the dictionary with the **del()** function.



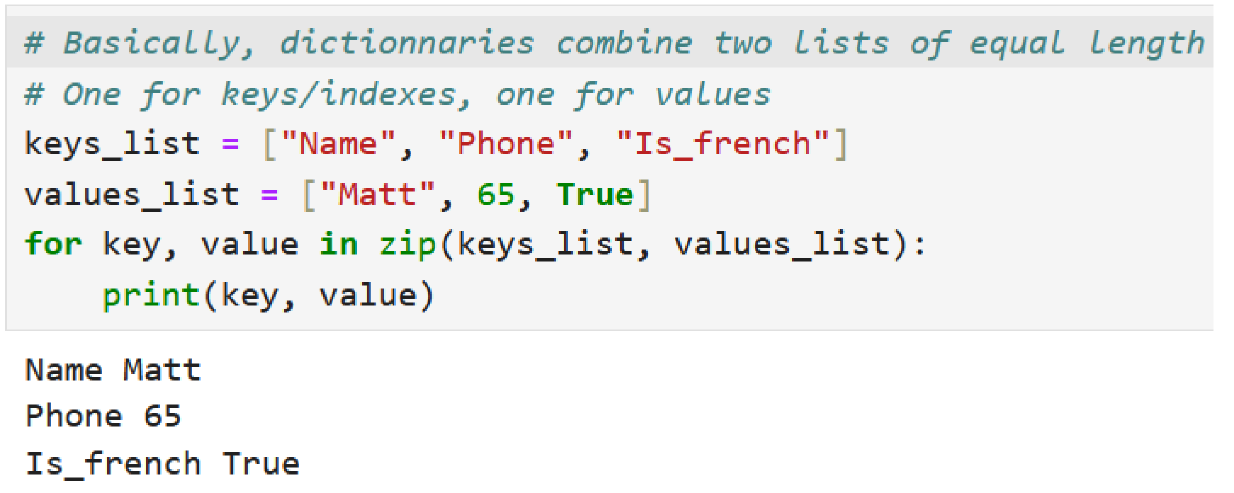
```python
# Updating a dictionary
my_dict = {0:1, 1:3, 2:5, 3:7, 4:9}
my_dict[0] = 11
print(my_dict)
# Adding a new value to dictionary
my_dict[27] = 14
print(my_dict)
# Removing a value from dictionary
del(my_dict[3])
print(my_dict)
```

```
{0: 11, 1: 3, 2: 5, 3: 7, 4: 9}
{0: 11, 1: 3, 2: 5, 3: 7, 4: 9, 27: 14}
{0: 11, 1: 3, 2: 5, 4: 9, 27: 14}
```

Defining dictionaries with zip

A dictionary simply consists of two lists with identical lengths:

- One list of **indexes/keys**, used to refer to elements (replacing the list of positional indexes we can use to refer to the elements).
- One list of **values**, assigned to each index/key.



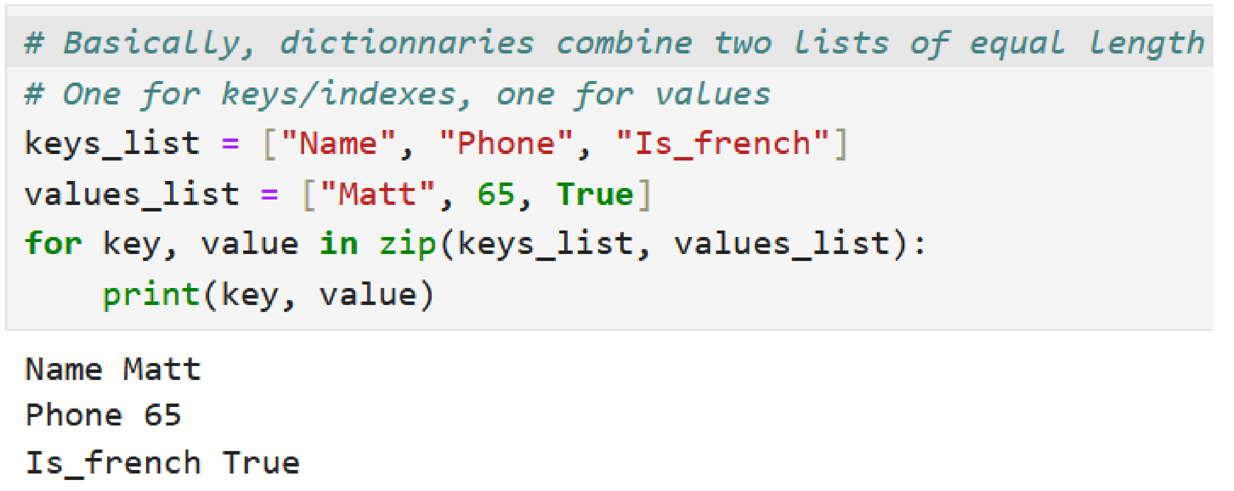
```
# Basically, dictionnaries combine two lists of equal length
# One for keys/indexes, one for values
keys_list = ["Name", "Phone", "Is_french"]
values_list = ["Matt", 65, True]
for key, value in zip(keys_list, values_list):
    print(key, value)
```

```
Name Matt
Phone 65
Is_french True
```

Defining dictionaries with zip

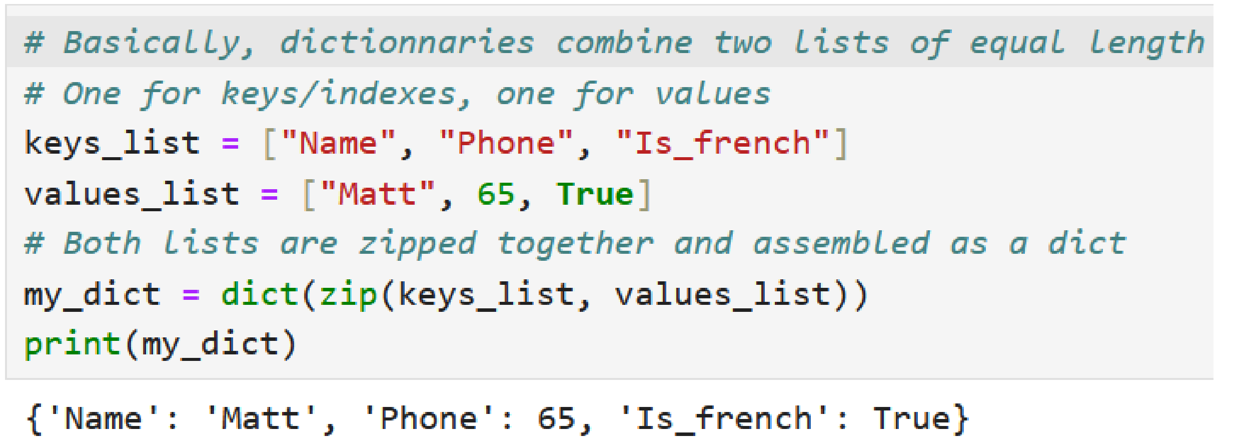
A dictionary simply consists of two lists with identical lengths:

- Dictionaries can be created with the **zip**() generator, which we used earlier to **combine two lists of equal length in a for loop.**
- Turns out that if you have a list of keys and a list of values to assign to each of these keys, then you can use **zip**() to generate a dictionary easily!



```python
# Basically, dictionnaries combine two lists of equal length
# One for keys/indexes, one for values
keys_list = ["Name", "Phone", "Is_french"]
values_list = ["Matt", 65, True]
for key, value in zip(keys_list, values_list):
    print(key, value)
```

```
Name Matt
Phone 65
Is_french True
```



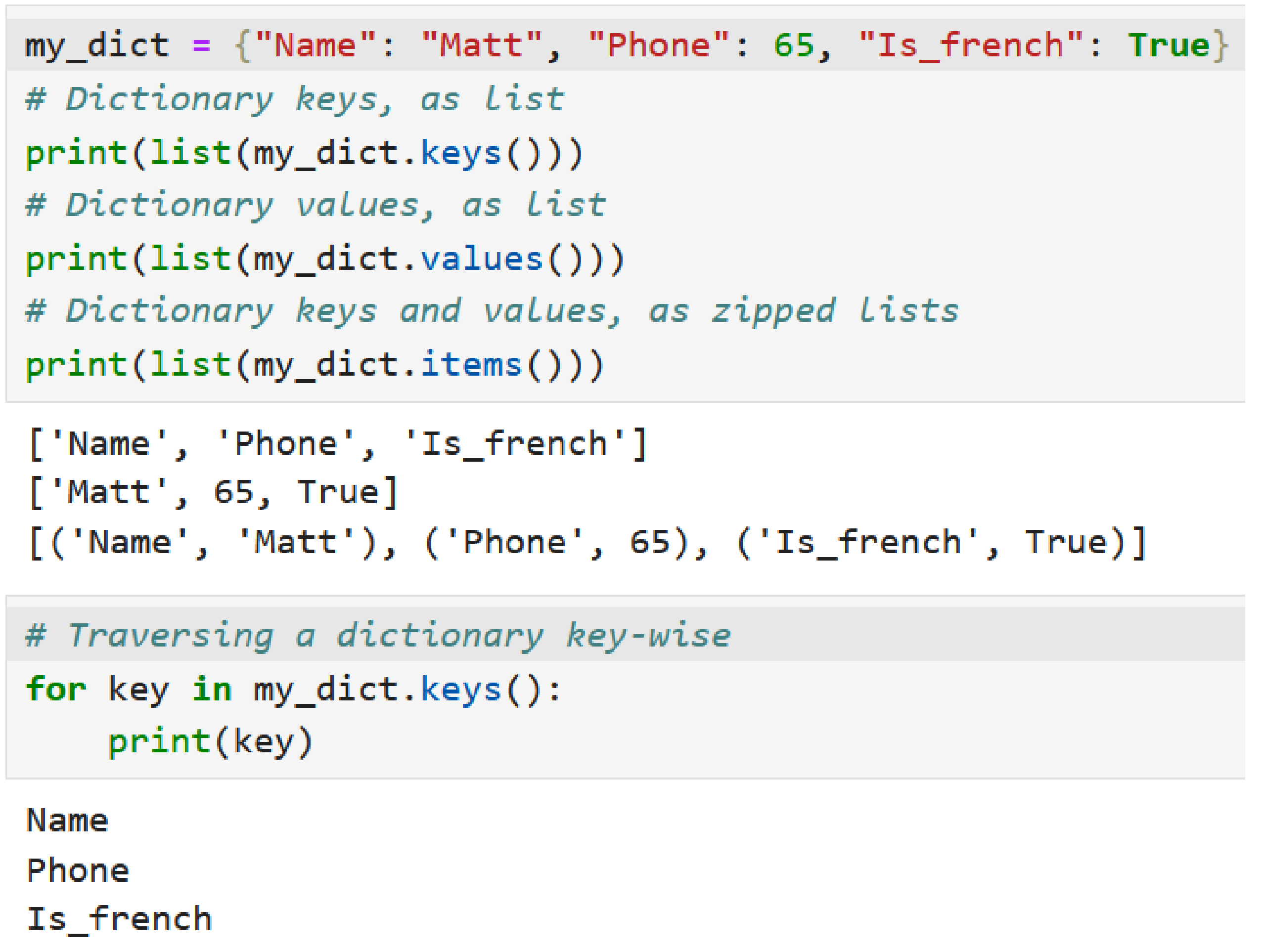
```python
# Basically, dictionnaries combine two lists of equal length
# One for keys/indexes, one for values
keys_list = ["Name", "Phone", "Is_french"]
values_list = ["Matt", 65, True]
# Both lists are zipped together and assembled as a dict
my_dict = dict(zip(keys_list, values_list))
print(my_dict)
```

```
{'Name': 'Matt', 'Phone': 65, 'Is_french': True}
```

Keys, values and items in a dict

Once a dictionary is defined, we can retrieve

- Its **list of indexes/keys**, using the **keys()** method,



```
my_dict = {"Name": "Matt", "Phone": 65, "Is_french": True}
# Dictionary keys, as list
print(list(my_dict.keys()))
# Dictionary values, as list
print(list(my_dict.values()))
# Dictionary keys and values, as zipped lists
print(list(my_dict.items()))
```

```
['Name', 'Phone', 'Is_french']
['Matt', 65, True]
[('Name', 'Matt'), ('Phone', 65), ('Is_french', True)]
```

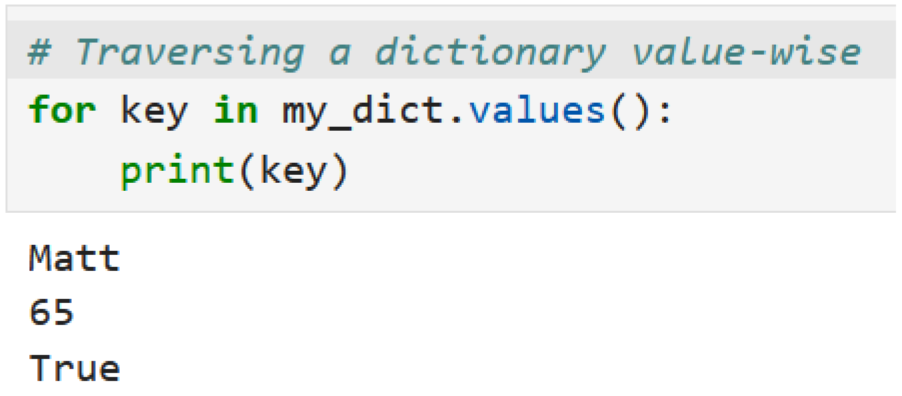
```
# Traversing a dictionary key-wise
for key in my_dict.keys():
    print(key)
```

```
Name
Phone
Is_french
```

Keys, values and items in a dict

Once a dictionary is defined, we can retrieve

- Its **list of indexes/keys**, using the **keys()** method,
- Its **list of values**, using the **values()** method,



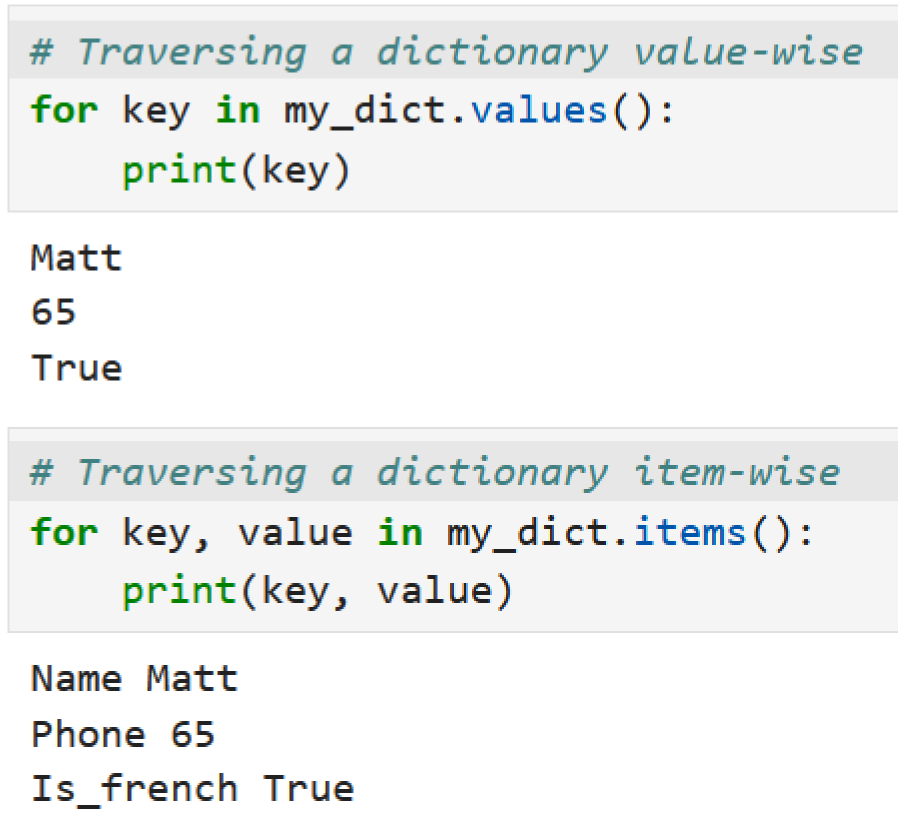
```
# Traversing a dictionary value-wise
for key in my_dict.values():
    print(key)
```

```
Matt
65
True
```

Keys, values and items in a dict

Once a dictionary is defined, we can retrieve

- Its **list of indexes/keys**, using the **keys()** method,
- Its **list of values**, using the **values()** method,
- A list of all the possible **of keys/indexes and values pairs**, using the **items()** method.



```
# Traversing a dictionary value-wise
for key in my_dict.values():
    print(key)
```

```
Matt
65
True
```

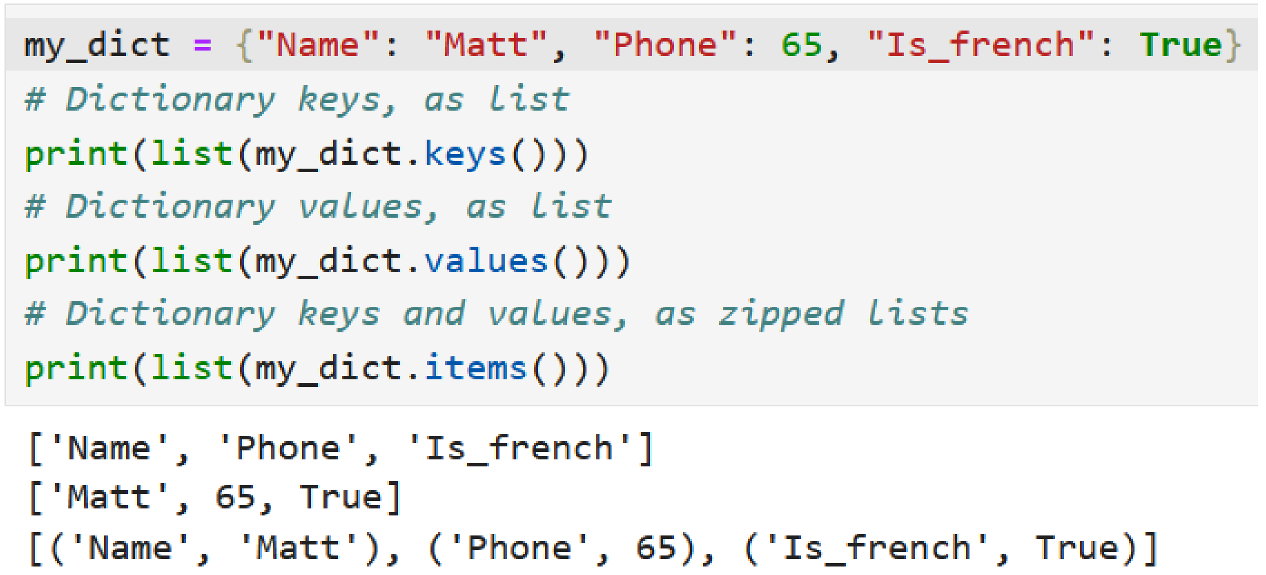
```
# Traversing a dictionary item-wise
for key, value in my_dict.items():
    print(key, value)
```

```
Name Matt
Phone 65
Is_french True
```

Keys, values and items in a dict

Once a dictionary is defined, we can retrieve

- Its **list of indexes/keys**, using the **keys()** method,
- Its **list of values**, using the **values()** method,
- A list of all the possible **of keys/indexes and values pairs,** using the **items()** method.



```
my_dict = {"Name": "Matt", "Phone": 65, "Is_french": True}
# Dictionary keys, as list
print(list(my_dict.keys()))
# Dictionary values, as list
print(list(my_dict.values()))
# Dictionary keys and values, as zipped lists
print(list(my_dict.items()))
```

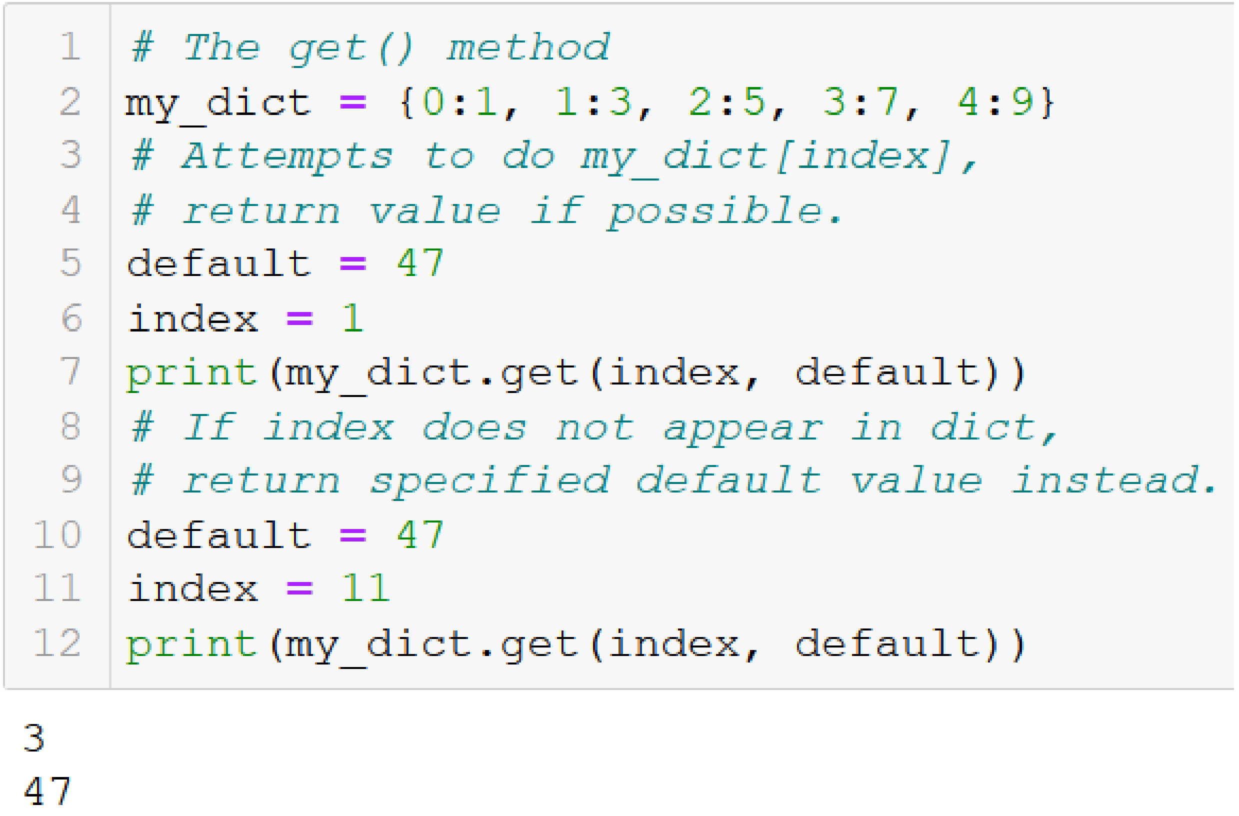
```
['Name', 'Phone', 'Is_french']
['Matt', 65, True]
[('Name', 'Matt'), ('Phone', 65), ('Is_french', True)]
```

Note: all three methods give **generators** (as range, zip, enumerate, etc.).
Convert them to **lists** to visualize the values in these generators!

The get() method

Finally, dictionaries comes with a **get()** method, which receives two arguments.

- The first argument is an index/key. The **get()** method attempts to retrieve the value in the dict for the given index/key.
- If the index/key does not exist, it returns the **default value** specified as the second argument of **get()** instead.



```python
# The get() method
my_dict = {0:1, 1:3, 2:5, 3:7, 4:9}
# Attempts to do my_dict[index],
# return value if possible.
default = 47
index = 1
print(my_dict.get(index, default))
# If index does not appear in dict,
# return specified default value instead.
default = 47
index = 11
print(my_dict.get(index, default))
```

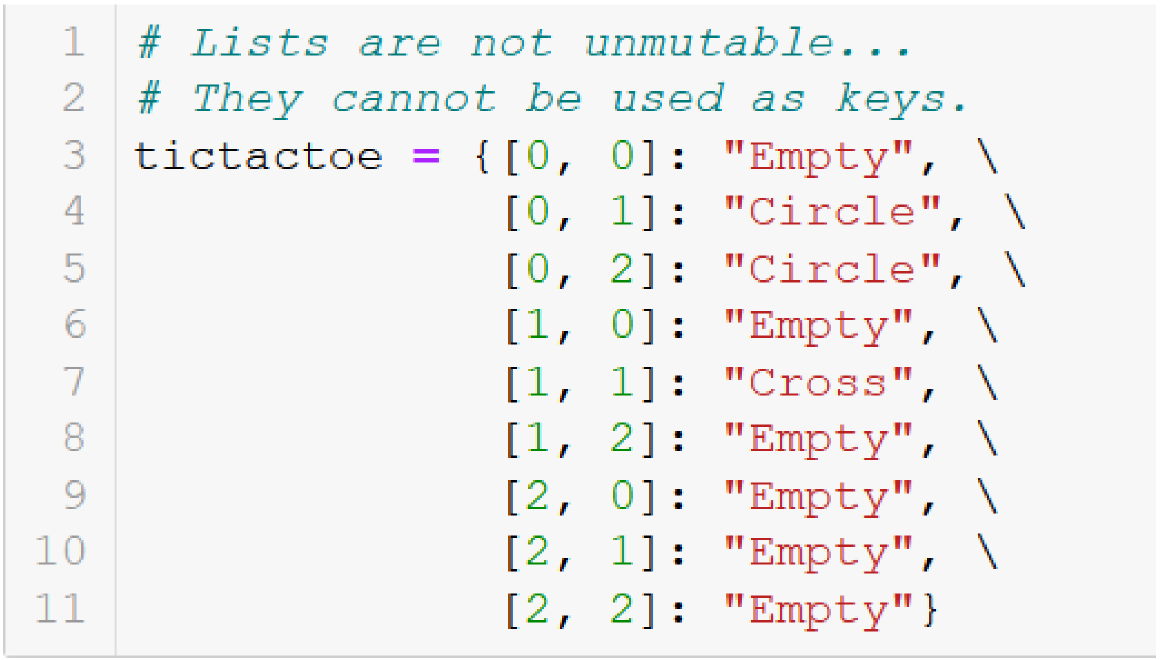
```
3
47
```

A note on keys

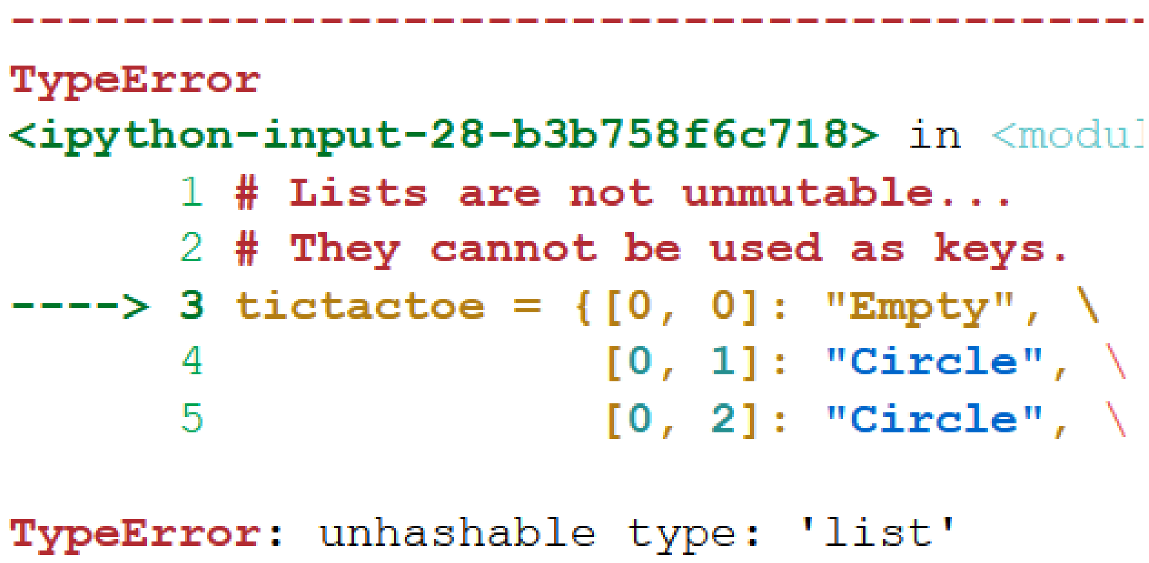
Important note on keys:
The keys of a dictionary need to be **immutable** types of variables.

- **Immutables types:** int, float, string, tuple, set, etc.
- **Mutable types:** list, etc.

The dictionary on the right uses lists as keys (not acceptable!)



```python
# Lists are not unmutable...
# They cannot be used as keys.
tictactoe = {[0, 0]: "Empty", \
             [0, 1]: "Circle", \
             [0, 2]: "Circle", \
             [1, 0]: "Empty", \
             [1, 1]: "Cross", \
             [1, 2]: "Empty", \
             [2, 0]: "Empty", \
             [2, 1]: "Empty", \
             [2, 2]: "Empty"}
```

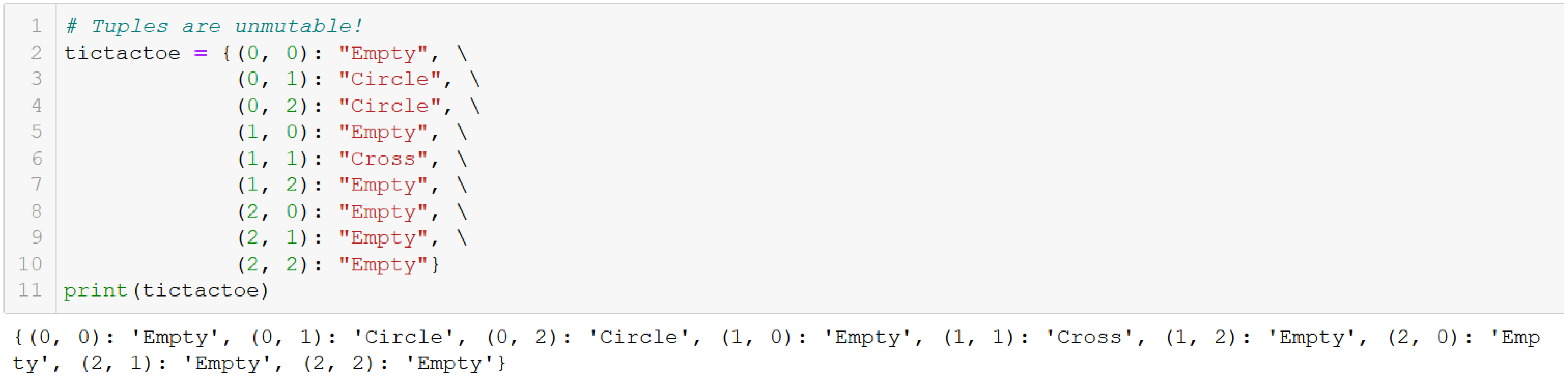


```
---------------------------------------------
TypeError
<ipython-input-28-b3b758f6c718> in <modul
      1 # Lists are not unmutable...
      2 # They cannot be used as keys.
----> 3 tictactoe = {[0, 0]: "Empty", \
      4              [0, 1]: "Circle", \
      5              [0, 2]: "Circle", \

TypeError: unhashable type: 'list'
```

A note on keys

Important note on keys:
The keys of a dict need to be **immutable** types of variables.

- We can therefore make a dictionary if we use tuples as indexes/keys.



```python
# Tuples are unmutable!
tictactoe = {(0, 0): "Empty", \
             (0, 1): "Circle", \
             (0, 2): "Circle", \
             (1, 0): "Empty", \
             (1, 1): "Cross", \
             (1, 2): "Empty", \
             (2, 0): "Empty", \
             (2, 1): "Empty", \
             (2, 2): "Empty"}
print(tictactoe)
```

```
{(0, 0): 'Empty', (0, 1): 'Circle', (0, 2): 'Circle', (1, 0): 'Empty', (1, 1): 'Cross', (1, 2): 'Empty', (2, 0): 'Emp
ty', (2, 1): 'Empty', (2, 2): 'Empty'}
```

Matt’s Great advice

Matt’s Great Advice: Get comfortable with the dictionaries type.

Dictionaries are one of the most important types in Python.

While they might seem abstract at first, they are pretty easy to work with, especially if you understood the concepts behind lists.

Practice until you get comfortable with them, as we will be using them a lot!

Activity 1 - Find the missing card v2

Write a function **find_missing_card()**, which receives a deck, in variable **deck**, as its sole parameter.

The deck is a standard deck that has been shuffled and is missing a card. The function **find_missing_card()** should then return the one card that is missing in the deck, as a tuple.

Note that the deck is always missing a single card and will contain no duplicates.

Objective: try using a dictionary to count the number of hearts, spades, diamonds, clubs in deck. The one suit with a different number of cards is going to be the suit of the missing card!

```
suits_dict = {"Hearts": 12, "Spades": 13, "Diamonds": 13, "Clubs": 13}
```

Activity 2 - 1337 speak translators v2

- You now know about dictionaries.
- We will use a translation dictionary, defined below, whose keys are English characters and whose values are the numbers which should replace the said letter, when we translate English sentences into leet speak.

```
eng_leet_dict = {'I': '1', 'Z': '2', 'E': '3', 'A': '4',
                 'S': '5', 'T': '7', 'B': '8', 'O': '0’}
```

- This translation dictionary should help you simplify your translation function drastically.

Activity 2 - 1337 speak translators v2

Your objective is then to write a second version of the function **eng_to_leet_v2()**, which receives two parameters:

- the first one is **eng_str,** which corresponds to a sentence in English, written in capital letters,
- the second one is the translation dictionary **eng_leet_dict**, defined earlier.

The function returns the equivalent sentence in leet speak, modifying the letters with numbers, by following the rules defined earlier.

The translation dictionary should help you simplify your function drastically.

Matt’s Great advice

Matt’s Great Advice: Everything around you can be described as an object.

Everything around us can be represented, in some way, as a dictionary with various attributes

Everything.

This approach is known as object-oriented programming. It conceptualizes all entities as collections of attributes, where each attribute is a key-value pair.

For instance, as a person, I have a name (str), an age (int), a birth year (int), etc.

Object oriented programming and thinking

Definition (Object-Oriented Programming): Object Oriented Programming (OOP) is a programming paradigm that relies on the concept of objects.

- It is used to structure a software program into simple, reusable pieces of code/blueprints (usually called objects)
- These are then used to create individual instances of self-contained objects.

This paradigm is an absolute game-changer, more to follow.

- For now, I propose an introduction to OOP, called **"Object Oriented Thinking"**, which uses dictionaries instead.
- Let us demonstrate with an example!

I believe this Object-Oriented Thinking calls for a demo…

Here is what happens in the mind of a programmer playing video games

SQUIRTLE♂ Lv5
HP
CHARMANDER♂ Lv5
HP
19/ 19
EXP
What will
CHARMANDER do?
▶FIGHT
BAG
POKéMON
RUN

In the mind of a programmer…

Following the OOP paradigm, Pokemons can be viewed and described as programming objects, with several attributes, such as:

- A name: as a string variable
- A level: as an int
- An elemental type: as a string variable
- Some hitpoints (HP), strength and defense points.
- Etc. (Surely you can come up with more ideas of attributes!)

Therefore, each Pokemon in the game could be described, by using a dictionary with several entries, following the same blueprint.
Each Pokemon dictionary would have the same keys, but not necessarily the same values for each attribute (different names/stats).

…Pokemons are programming objects!

```python
# Everything can be defined as an object,
# i.e. a collection a keys and values.
pokemon1 = {"Name": "Charmander", \
            "Type": "Fire", \
            "Level": 5, \
            "HP": 19, \
            "Attack": 25, \
            "Defense": 12}
print(pokemon1)
```

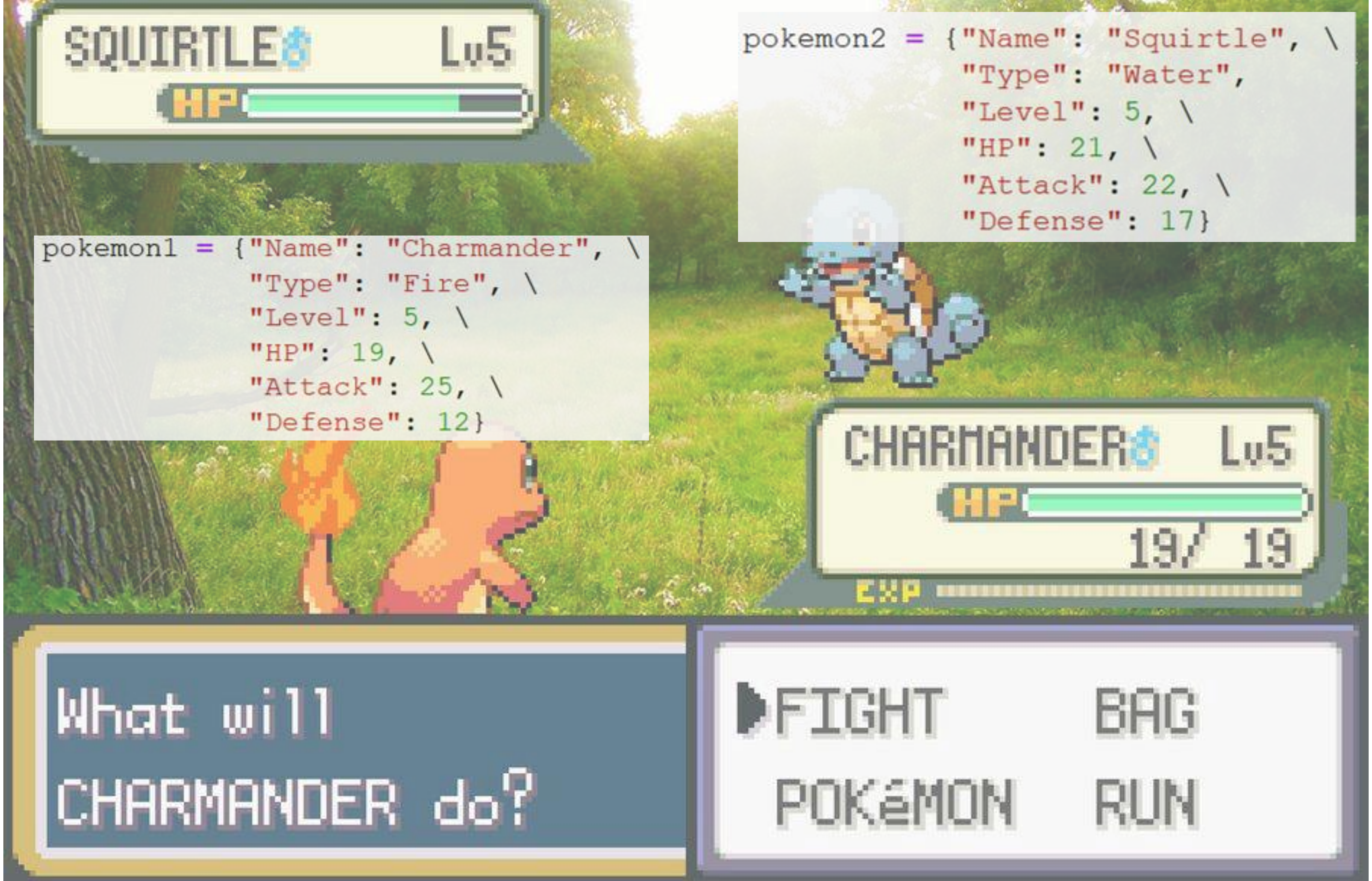
```
{'Name': 'Charmander', 'Type': 'Fire', 'Level': 5, 'HP': 19, 'Attack': 25, 'Defense': 12}
```

…Pokemons are programming objects!

In fact, each Pokemon in the game could be described, by using a dictionary with several entries, following the same blueprint.

- Each Pokemon dictionary would have the same keys, but not necessarily the same values assigned to each of these keys.
- After all, the Pokemons could have different names and stats.

More concepts in the game could also be described using objects of some sort, such as (human character/player, items, environment objects, etc.)



```python
pokemon2 = {"Name": "Squirtle", \
            "Type": "Water",
            "Level": 5, \
            "HP": 21, \
            "Attack": 22, \
            "Defense": 17}
```

```python
pokemon1 = {"Name": "Charmander", \
            "Type": "Fire", \
            "Level": 5, \
            "HP": 19, \
            "Attack": 25, \
            "Defense": 12}
```

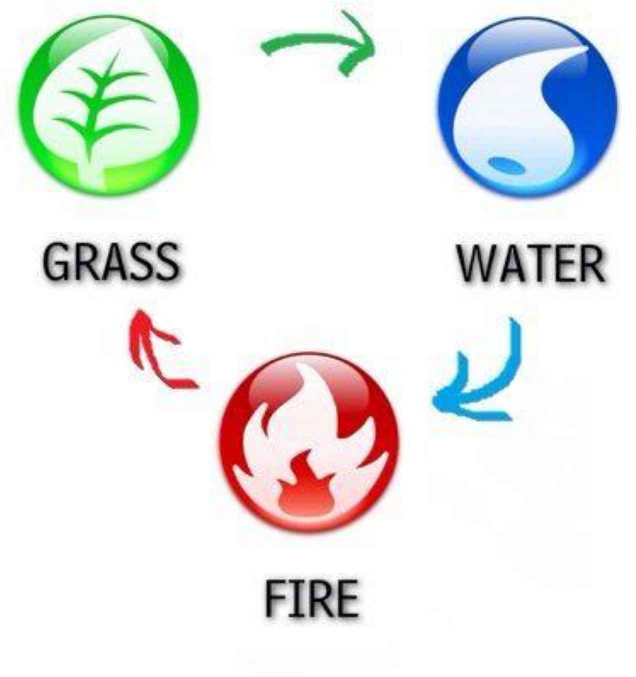
Pokemons types tables

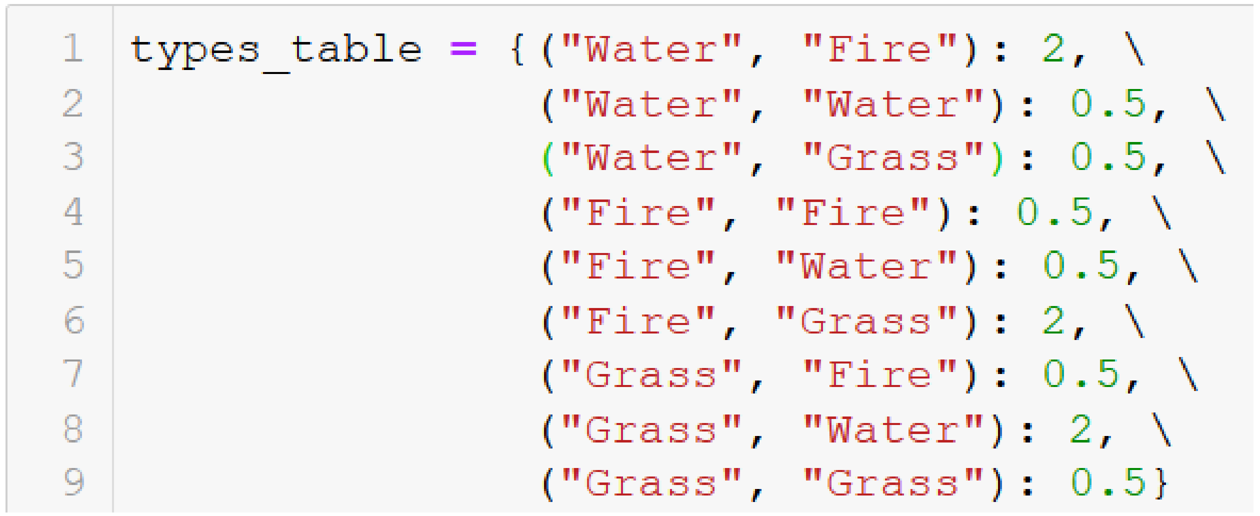
Pokemons have an **elemental type** (Fire, Water, Grass, etc.)

The Pokemon game has a **rock-paper-scissor** concept.

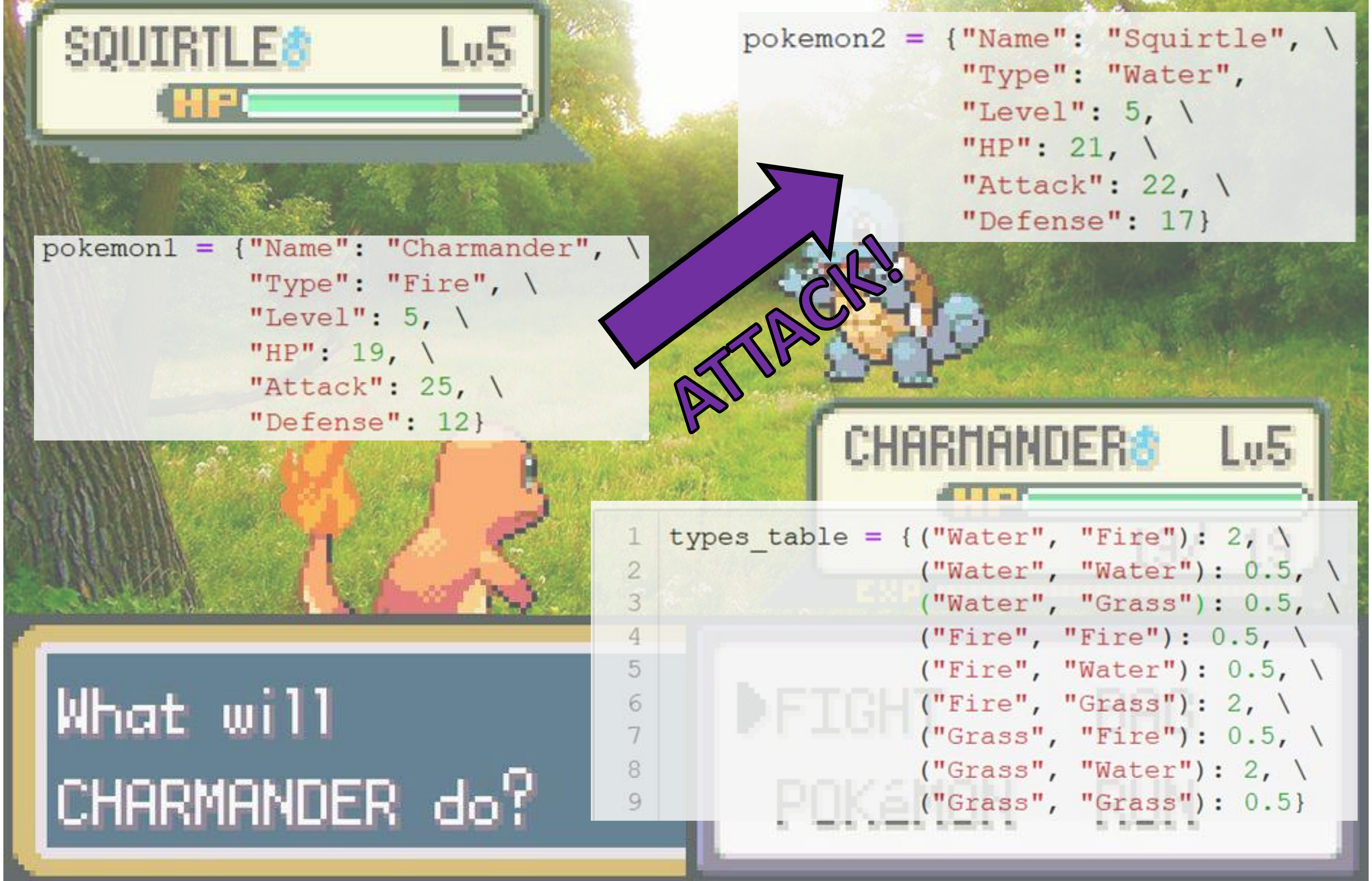
- Fire strong vs. Grass,
- Water strong vs. Fire,
- Grass strong vs. Water.

Strong means you deal **double damage** in combat, **weak** means you deal **half damage** only.





```
types_table = {("Water", "Fire"): 2, \
               ("Water", "Water"): 0.5, \
               ("Water", "Grass"): 0.5, \
               ("Fire", "Fire"): 0.5, \
               ("Fire", "Water"): 0.5, \
               ("Fire", "Grass"): 2, \
               ("Grass", "Fire"): 0.5, \
               ("Grass", "Water"): 2, \
               ("Grass", "Grass"): 0.5}
```



```
pokemon2 = {"Name": "Squirtle", \
            "Type": "Water",
            "Level": 5, \
            "HP": 21, \
            "Attack": 22, \
            "Defense": 17}
```

```
pokemon1 = {"Name": "Charmander", \
            "Type": "Fire", \
            "Level": 5, \
            "HP": 19, \
            "Attack": 25, \
            "Defense": 12}
```

```
types_table = {("Water", "Fire"): 2, \
               ("Water", "Water"): 0.5, \
               ("Water", "Grass"): 0.5, \
               ("Fire", "Fire"): 0.5, \
               ("Fire", "Water"): 0.5, \
               ("Fire", "Grass"): 2, \
               ("Grass", "Fire"): 0.5, \
               ("Grass", "Water"): 2, \
               ("Grass", "Grass"): 0.5}
```

The attack function

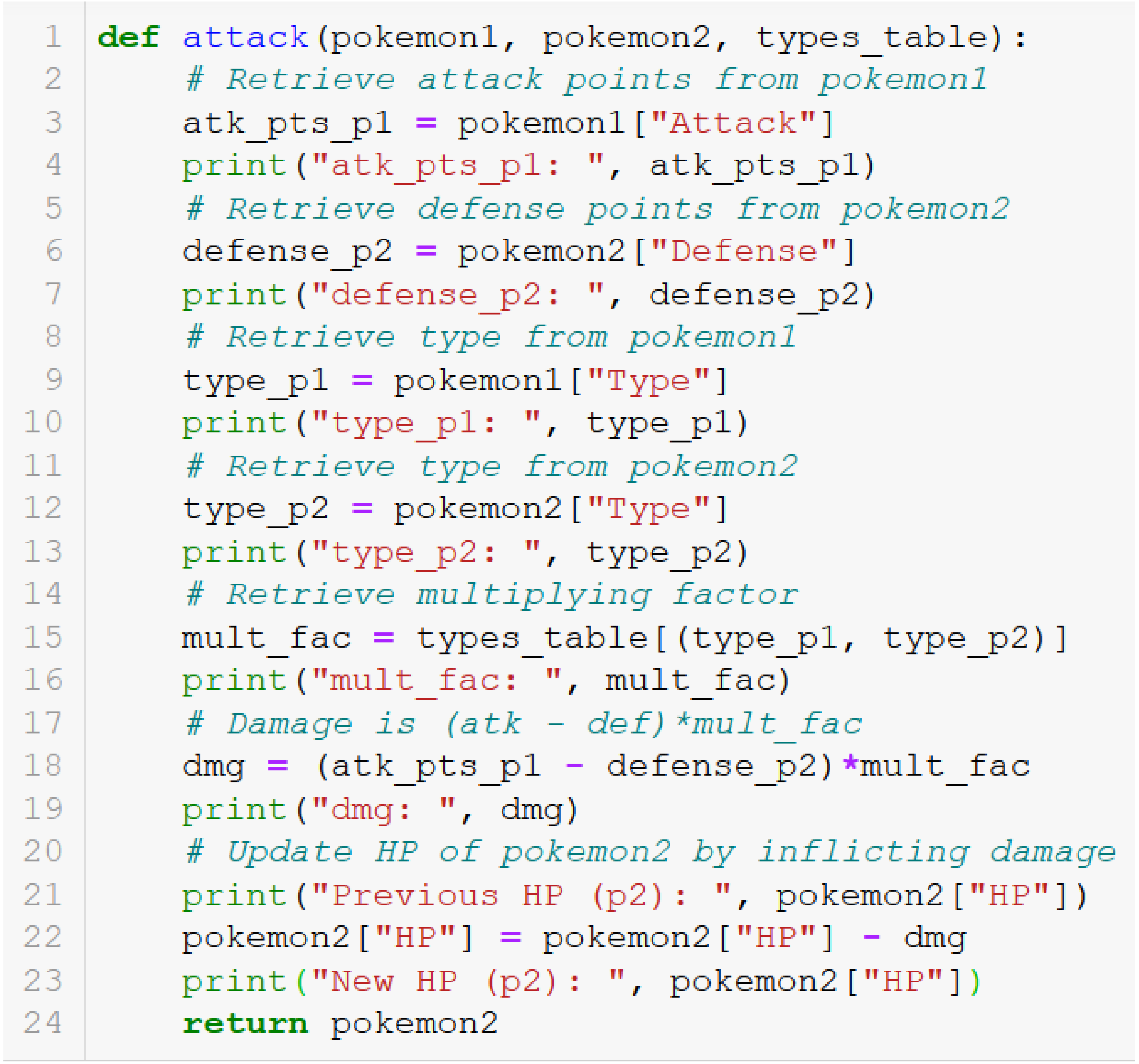
And then, our gameplay could be described as functions using and/or updating dictionaries corresponding to each of these entities.

For instance, when pokemon1 attacks pokemon2, it inflicts damage to pokemon2, which is calculated as

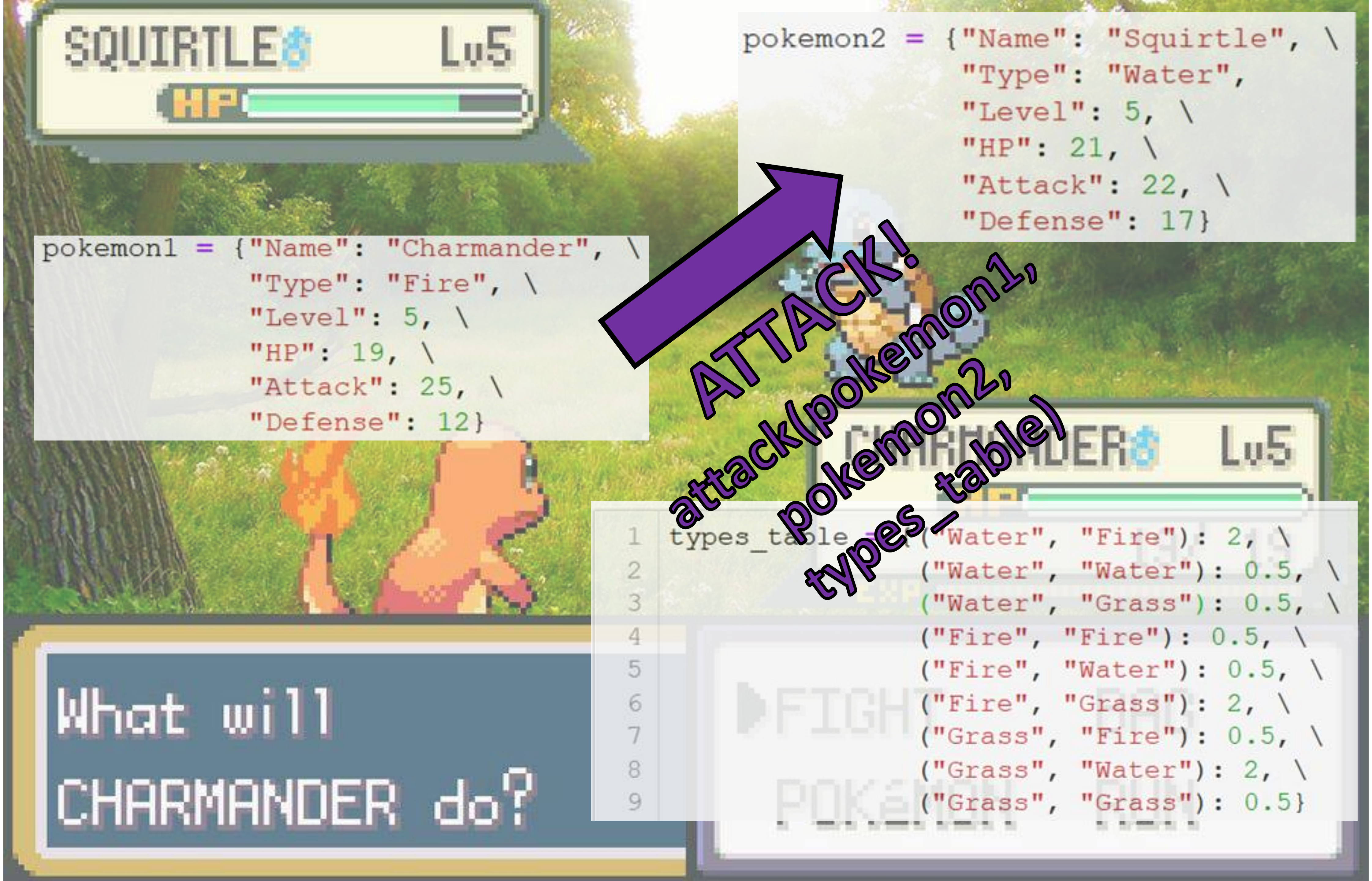
$$\left(\boldsymbol{attack_{pokemon1}} - \boldsymbol{defense_{pokemon2}}\right) \times \boldsymbol{m}$$

Where $\boldsymbol{m}$ is the multiplying factor (either 2 or 0.5), which

- depends on the elemental types of the attacker (pokemon1) and the defender (pokemon2),
- and is given by the types_table.



```python
def attack(pokemon1, pokemon2, types_table):
    # Retrieve attack points from pokemon1
    atk_pts_p1 = pokemon1["Attack"]
    print("atk_pts_p1: ", atk_pts_p1)
    # Retrieve defense points from pokemon2
    defense_p2 = pokemon2["Defense"]
    print("defense_p2: ", defense_p2)
    # Retrieve type from pokemon1
    type_p1 = pokemon1["Type"]
    print("type_p1: ", type_p1)
    # Retrieve type from pokemon2
    type_p2 = pokemon2["Type"]
    print("type_p2: ", type_p2)
    # Retrieve multiplying factor
    mult_fac = types_table[(type_p1, type_p2)]
    print("mult_fac: ", mult_fac)
    # Damage is (atk - def)*mult_fac
    dmg = (atk_pts_p1 - defense_p2)*mult_fac
    print("dmg: ", dmg)
    # Update HP of pokemon2 by inflicting damage
    print("Previous HP (p2): ", pokemon2["HP"])
    pokemon2["HP"] = pokemon2["HP"] - dmg
    print("New HP (p2): ", pokemon2["HP"])
    return pokemon2
```



```python
pokemon1 = {"Name": "Charmander", \
            "Type": "Fire", \
            "Level": 5, \
            "HP": 19, \
            "Attack": 25, \
            "Defense": 12}
```

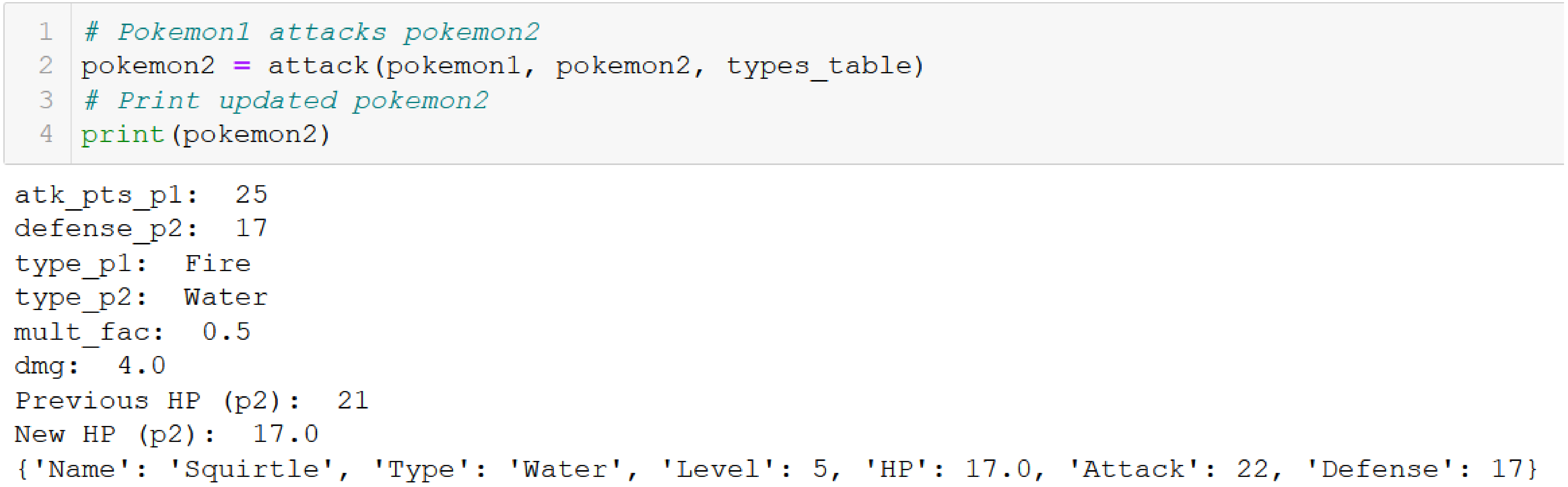
```python
pokemon2 = {"Name": "Squirtle", \
            "Type": "Water", \
            "Level": 5, \
            "HP": 21, \
            "Attack": 22, \
            "Defense": 17}
```

ATTACK!

attack(pokemon1, pokemon2, types_table)

```python
types_table = {("Water", "Fire"): 2, \
               ("Water", "Water"): 0.5, \
               ("Water", "Grass"): 0.5, \
               ("Fire", "Fire"): 0.5, \
               ("Fire", "Water"): 0.5, \
               ("Fire", "Grass"): 2, \
               ("Grass", "Fire"): 0.5, \
               ("Grass", "Water"): 2, \
               ("Grass", "Grass"): 0.5}
```

The attack function, at work



```python
# Pokemon1 attacks pokemon2
pokemon2 = attack(pokemon1, pokemon2, types_table)
# Print updated pokemon2
print(pokemon2)
```

```
atk_pts_p1:  25
defense_p2:  17
type_p1:  Fire
type_p2:  Water
mult_fac:  0.5
dmg:  4.0
Previous HP (p2):  21
New HP (p2):  17.0
{'Name': 'Squirtle', 'Type': 'Water', 'Level': 5, 'HP': 17.0, 'Attack': 22, 'Defense': 17}
```

Going deeper

I believe that this example helped visualize how Object-Oriented Thinking can help improve the structure of your code!

You have seen here how it could be done using a dictionary. In practice however, **OOP goes far deeper**, and gives the possibility to create:

- Your **own custom types of variables**, so that you can represent any object you want.
- Your **own custom methods**, i.e. functions that will only apply to these custom objects.

We will learn more about the Object-Oriented Programming concepts in the next coming lecture!

Activity 3 - Hero stats

In OOP, a function, which creates an object and its dictionary, based on a few variables, is called a **constructor function for the object**.

In this activity, we design a **constructor function** for a **hero object**.

Write a function **create_hero()**, which receives three parameters:

- str_pts: an int corresponding to the strength points of the hero,
- agi_pts: an int corresponding to the agility points of the hero,
- int_pts: an int corresponding to the intelligence points of the hero.

Activity 3 - Hero stats

It returns a dictionary containing the following keys and values:

- str_pts: an int corresponding to the strength points of the hero,
- agi_pts: an int corresponding to the agility points of the hero,
- int_pts: an int corresponding to the intelligence points of the hero,
- HP_pts: an int corresponding to the life points of the hero, which is 10 points, plus 10 times the number of strength points,
- MP_pts: an int corresponding to the magic points (MP) of the hero, which is 5 times the number of intelligence points,
- atk_pts: an int corresponding to the attack points of the hero, which is the sum of the strength, agility and intelligence points.

Activity 4 - Equipping a weapon

- Let us consider the hero defined below as an object, with its own dictionary.

```
hero_dict = {'str_pts': 10, 'equipped_weapon': None, 'atk_pts': None}
```

- Notice how no attack points and equipped weapon have yet been defined (they instead have the None value).
- We can also define **weapons objects**, as dictionaries as well. Below we have defined two weapon objects: a sword and a bow.

```
sword = {"weapon_name": "Sword of Blazing Justice", "weapon_atk": 10}
bow = {"weapon_name": "Bow of Impeccable Accuracy", "weapon_atk": 8}
```

Activity 4 - Equipping a weapon

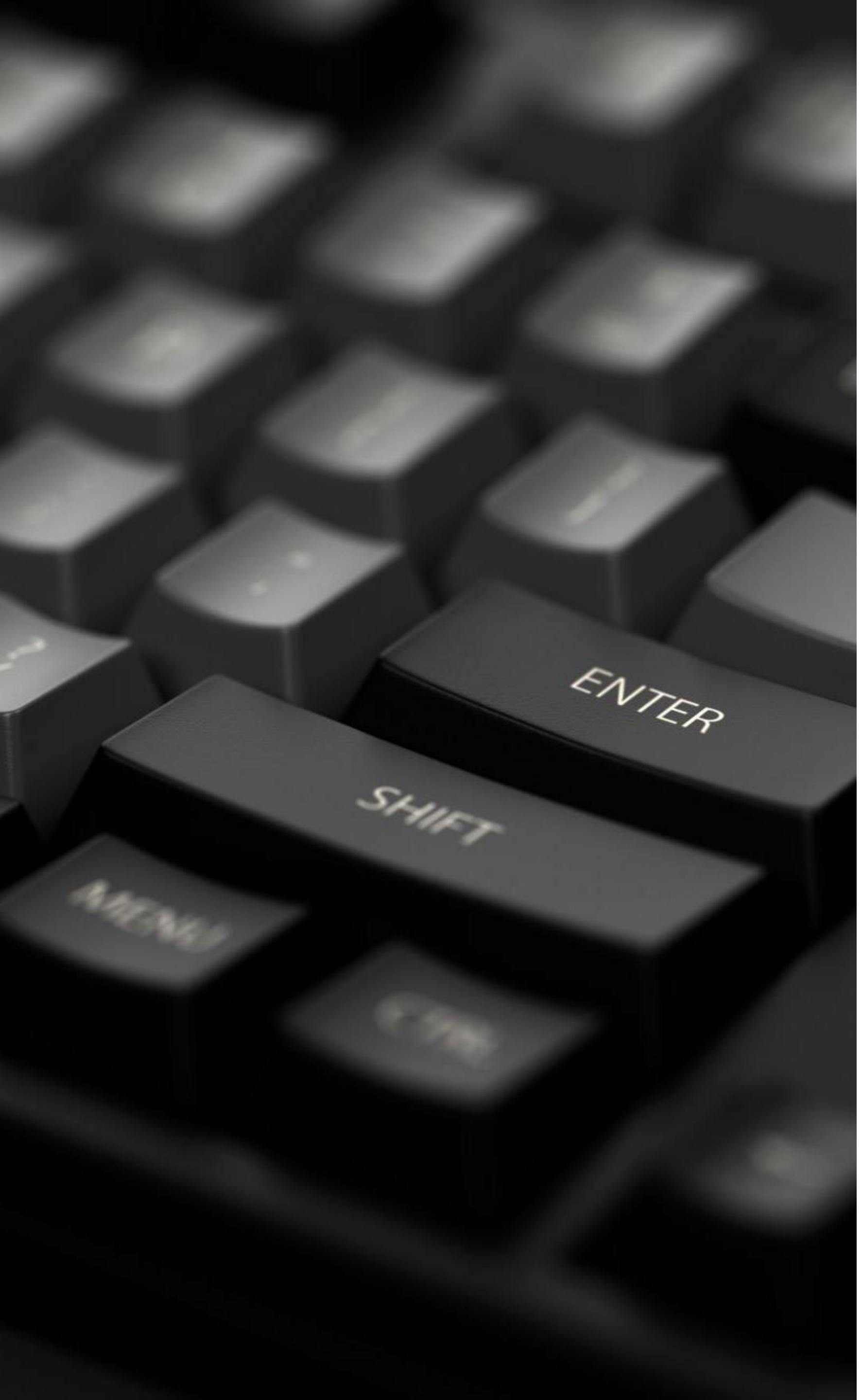
Write a function **equip_weapon()**, which receives two dictionaries:

- The first one is a hero dictionary,
- The second one is a weapon dictionary (either the sword or bow).

It will return the hero dictionary, whose keys, 'equipped_weapon' and 'atk_pts', have been updated.

- The value assigned to the 'equipped_weapon' of the hero dictionary, will be the dictionary corresponding to the equipped weapon.
- The value assigned to the 'atk_pts' of the hero dictionary, will consist of the hero strength points (in 'str_pts'), plus the weapon attack points (in the weapon dictionary, key 'weapon_atk').

Note: in practice, this OOP concept defines a **'has-a' relationship**, because one of the attributes of the hero object is a weapon object.



Conclusion (Chapter 14)

- What is the **tuple** type? (fast version)
- What is the **dictionary** type?
- What are **typical operations on dictionaries**?
- How are **dictionaries** and **the concept of object-oriented programming** related?
- What is “object-oriented thinking”?
- **Practice** on dictionaries.
- *(If time allows, what is the set types?)*

slido

Please download and install the Slido app on all computers you use

What will be printed?

ⓘ Start presenting to display the poll results on this slide.

slido

Please download and install the Slido app on all computers you use

Which operation should we use on line 2 to obtain ['gaming', 'running'] as a result on line 5?

ⓘ Start presenting to display the poll results on this slide.

slido

Please download and install the Slido app on all computers you use

Which of the following checks if the key 'name' exists in the dictionary d?

ⓘ Start presenting to display the poll results on this slide.

slido

Please download and install the Slido app on all computers you use

What will be the outcome of this code?

ⓘ Start presenting to display the poll results on this slide.

slido

Please download and install the Slido app on all computers you use

Which of these four prints will display the string "gaming"?

ⓘ Start presenting to display the poll results on this slide.