ILP 2023 – W6S1 Dictionaries and Object-oriented thinking

Matthieu DE MARI – Singapore University of Technology and Design

SINGAPORE UNIVERSITY OF TECHNOLOGY AND DESIGN

Outline (Week6, Session1 – W6S1)

- The tuple type (fast version)
- The dictionary type
- Dictionary and objects
- Object-oriented thinking and programming
- (If time allows, tuples and sets types extra practice)

The tuple type

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

- They share many functions with lists, such as the **len()** function for instance.

```
# 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'>
```

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

```
5
5
```

The **tuple** type

As with **lists**, **tuples** objects can be

- **Indexed**, **Sliced**,
- **Traversed** using **for** loops,
- Etc.

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

Unmutable: values can only be changed on creation. No updates.

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

The tuple type

As with **lists**, **tuples** objects can be

- **Indexed**, **Sliced**,
- **Traversed** using **for** loops,
- Etc.

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

Unmutable: values can only be changed on creation. No updates.

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

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

The dict type

- **Dictionaries** (**dict**) are the last type of built-in object 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.

```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.
- This opens plenty of new possibilities, for instance

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

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

The dict type

- **Dictionaries** (**dict**) operate as **lists**. But they give the user the possibility to choose **custom indexes/keys** to use for elements.
- This opens plenty of new possibilities, for instance

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

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

- More on this later!

Adding, updating and removing elements

- As with lists, we can update elements using the brackets [] notation.
- If the **key/index** did not exist in the dictionary, it will add a new entry to the dictionary.

```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}
```

Adding, updating and removing elements

- As with lists, we can update elements using the brackets [] notation.
- If the **key/index** did not exist in the dictionary, it will add a new entry to the dictionary.

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

```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
- One list of **values**, assigned to each index/key.

- Dictionaries can be defined as with the **zip()** generator, which we used earlier to combine lists.

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

```
Name Matt
Phone 65
Is_the_best_teacher True
```

```python
# Basically, dictionnaries combine two lists of equal length
# One for keys/indexes, one for values
keys_list = ["Name", "Phone", "Is_the_best_teacher"]
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_the_best_teacher': 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,

```python
my_dict = {"Name": "Matt", "Phone": 65, "Is_the_best_teacher": 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_the_best_teacher']
['Matt', 65, True]
[('Name', 'Matt'), ('Phone', 65), ('Is_the_best_teacher', True)]
```

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

```
Name
Phone
Is_the_best_teacher
```

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,

```python
my_dict = {"Name": "Matt", "Phone": 65, "Is_the_best_teacher": 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_the_best_teacher']
['Matt', 65, True]
[('Name', 'Matt'), ('Phone', 65), ('Is_the_best_teacher', True)]
```

```python
# Traversing a dictionary value-wise
for value 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 **combined zip of keys/indexes and values**, using the **items()** method.

```python
my_dict = {"Name": "Matt", "Phone": 65, "Is_the_best_teacher": 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_the_best_teacher']
['Matt', 65, True]
[('Name', 'Matt'), ('Phone', 65), ('Is_the_best_teacher', True)]
```

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

```
Name Matt
Phone 65
Is_the_best_teacher 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 **combined zip of keys/indexes and values**, using the **items()** method.

```python
my_dict = {"Name": "Matt", "Phone": 65, "Is_the_best_teacher": 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_the_best_teacher']
['Matt', 65, True]
[('Name', 'Matt'), ('Phone', 65), ('Is_the_best_teacher', True)]
```

Note: all three methods give **generators** (as zip, enumerate, etc.).

Convert them to **lists** to visualize the values in these generators!

The **get**() method

Finally, dictionaries provide 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.

```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
```

The **get**() method

Finally, dictionaries provide 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 a the second argument of **get()**.

```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 dict need to be **UNMUTABLE** types of variables.

- **Unmutables types:** ints, floats, strings, tuples, sets, etc.
- **Mutable types:** lists, etc.

```
# 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 **UNMUTABLE** types of variables.

```
# 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 #12

Matt’s Great Advice #12: Get comfortable with dictionaries.

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 #13

Matt’s Great Advice #13: Everything is an object.

Everything in the world can be described, to some extent, using a dictionary with several attributes.

EVERYTHING.

This school of thought/paradigm is commonly referred to as **object-oriented programming**.

Object oriented programming and thinking

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

Object oriented programming and thinking

- **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) which are used to create individual instances of self-contained objects.
- This paradigm is a game-changer, but we will merely cover the basic concepts during this Summer School.
- Here, I propose an introduction to OOP, as **“Object Oriented Thinking”**.

I believe this calls for a demo…

I believe this calls for a demo…

Time to discuss about these “guys”.


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

Pokemons as programming objects

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.

Therefore, they could be described, by using a dictionary with several entries, one for each attribute.

Pokemons as 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}
```

```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

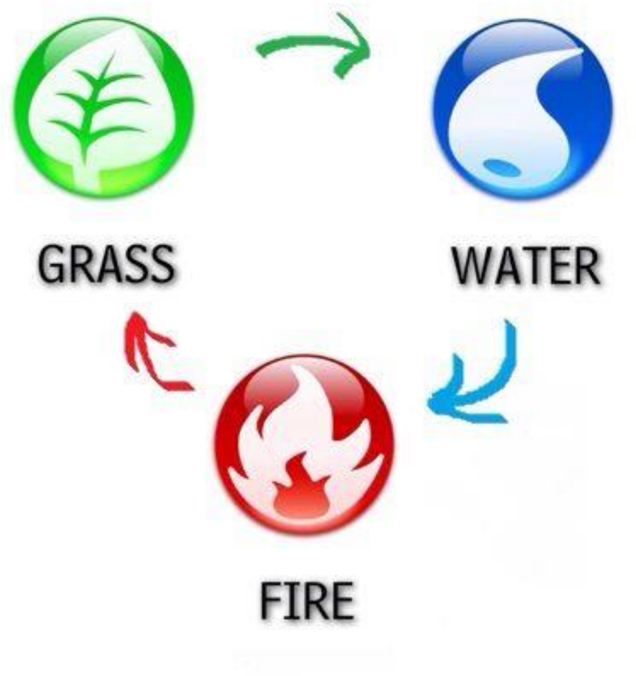
Pokemons have an **elemental type** (Fire, Water, Grass, etc.)

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.



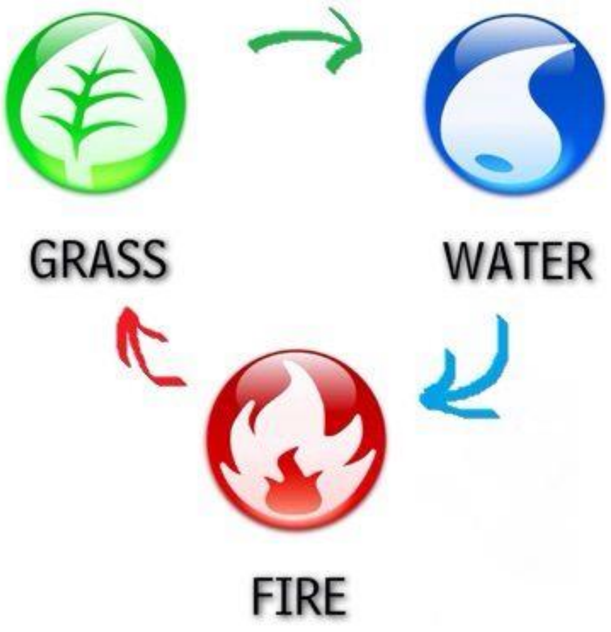
Pokemons types tables

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

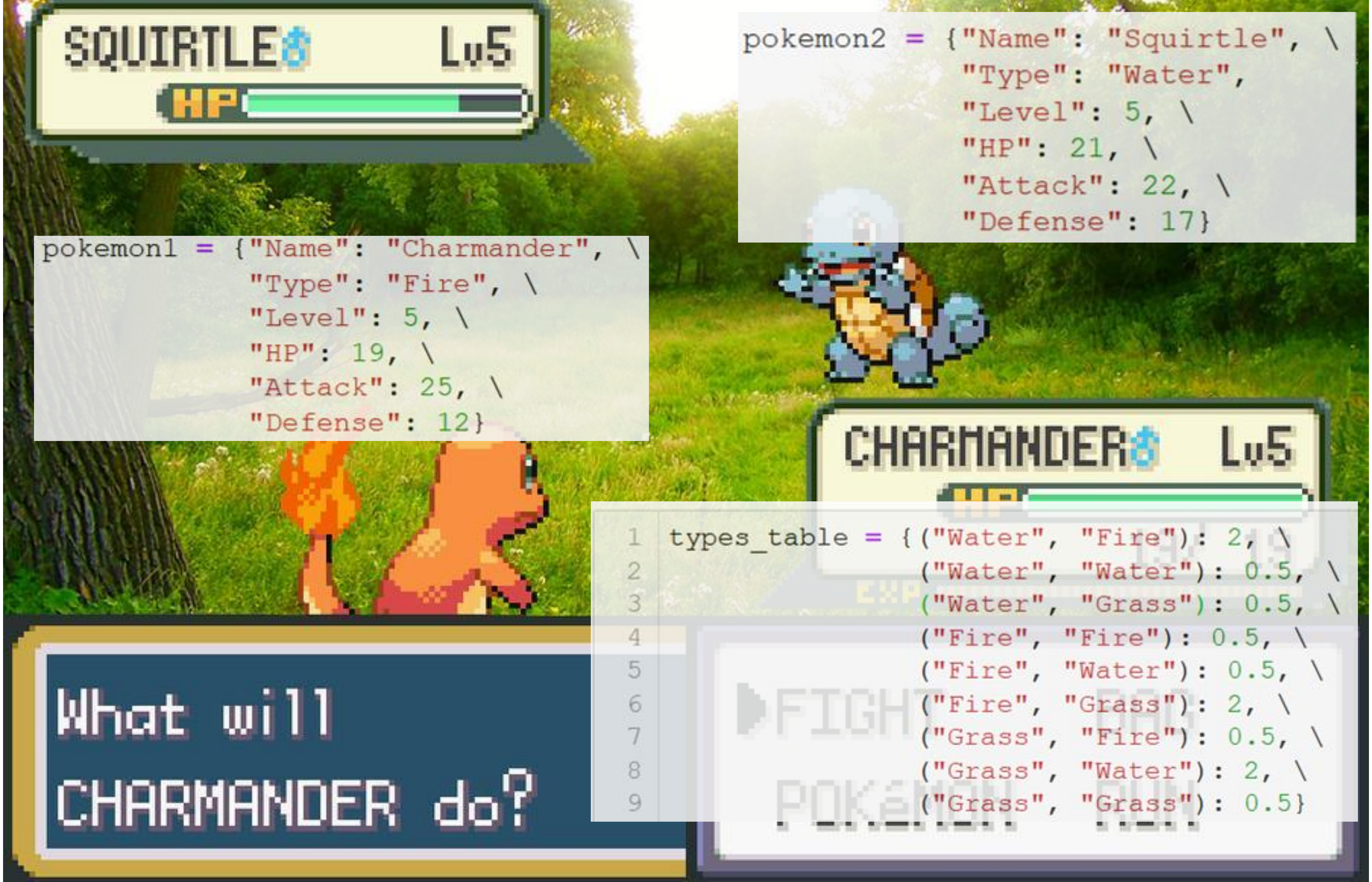
The Pokemon game has a **rock-paper-scissor** concept.

- 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.



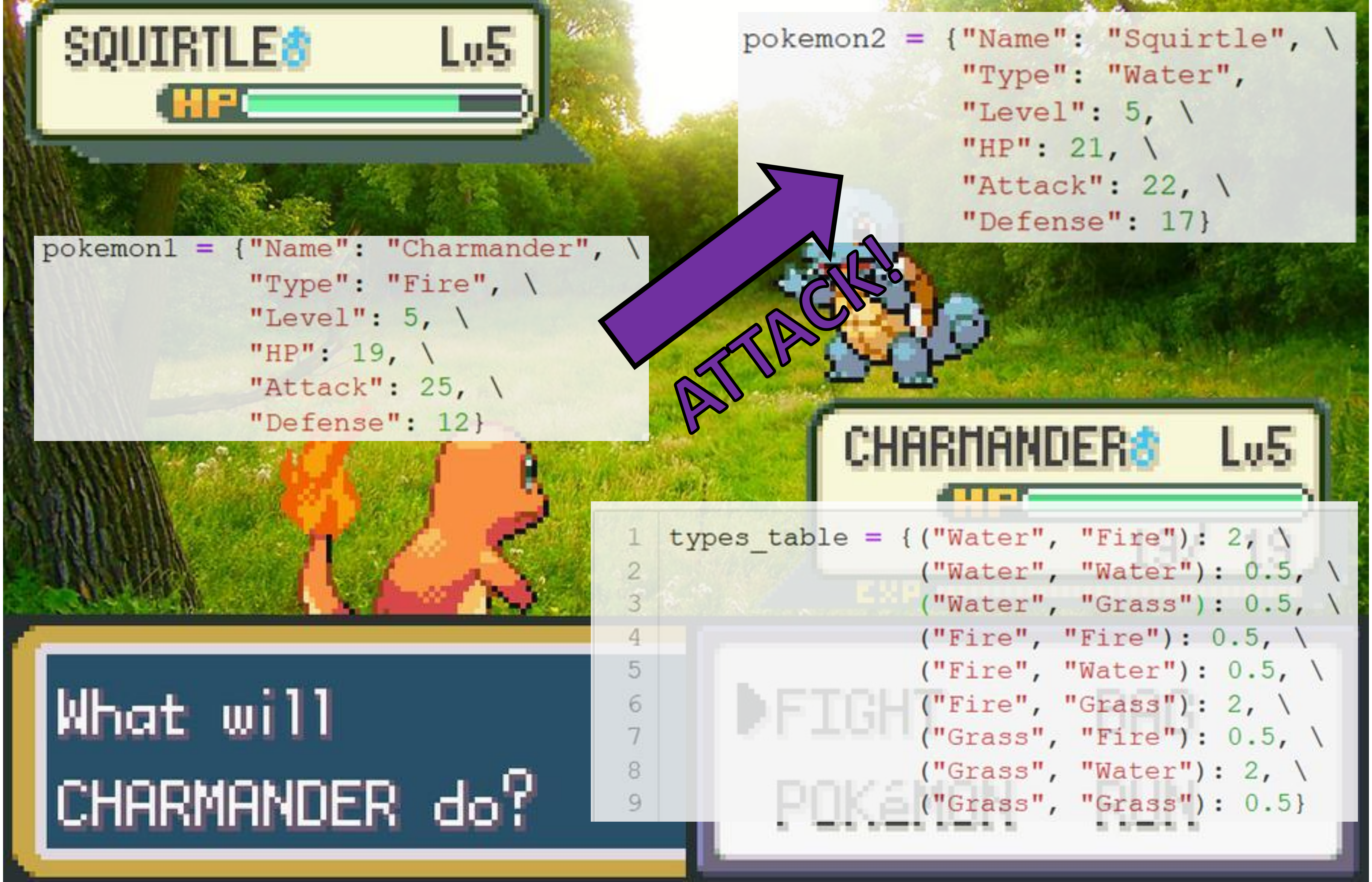

```
types_table = {("Water", "Fire"): 2, \
               ("Water", "Water"): 0.5, \
               ("Water", "Grass"): 0.5, \
               ("Fire", "Fire"): 0.5, \
               ("Fire", "Water"): 0.5, \
               ("Fire", "Grass"): 2, \
               ("Grass", "Fire"): 0.5, \
            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




```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}
```

```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}
```




```
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

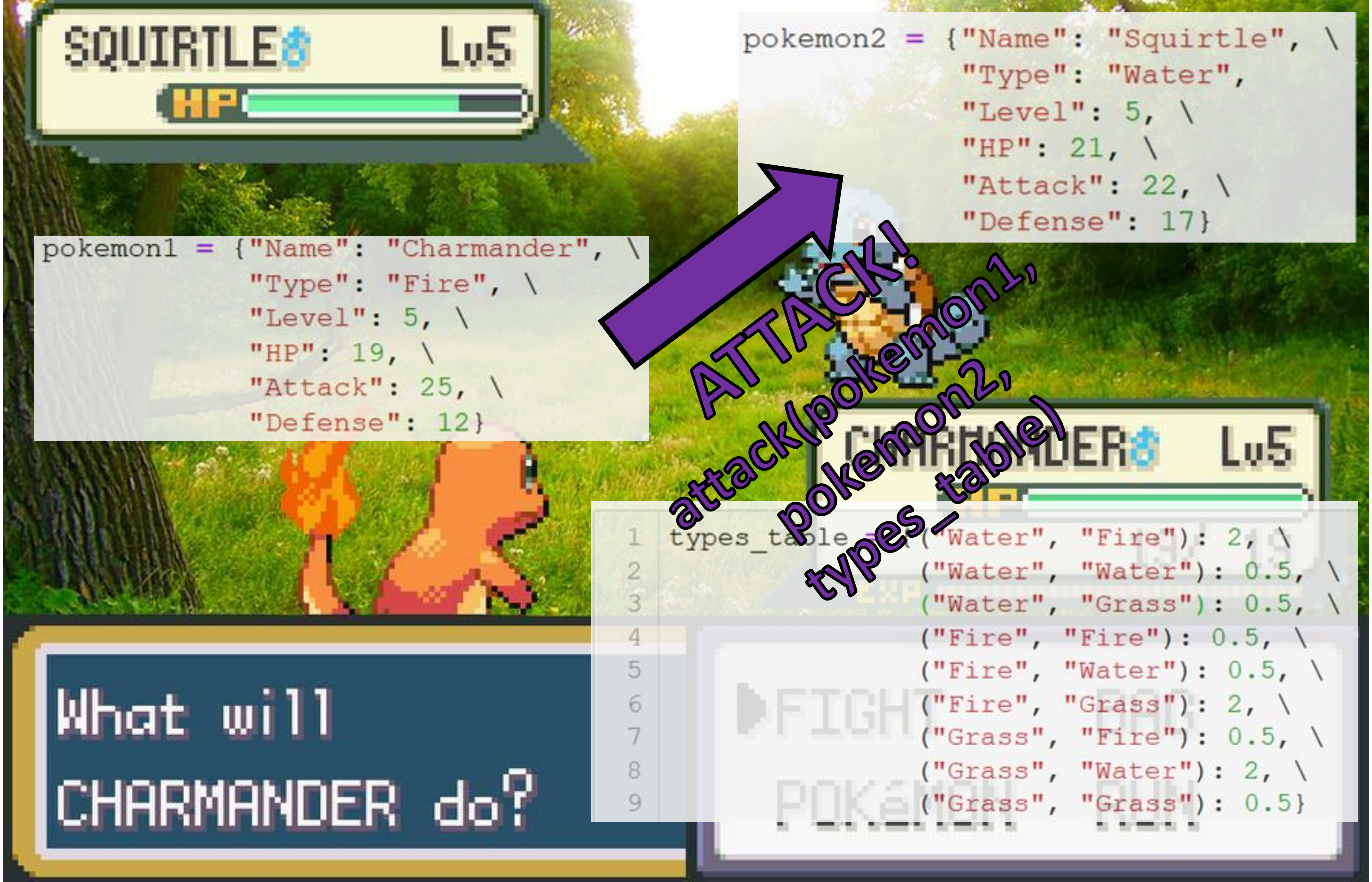
- When pokemon1 attacks pokemon2, it inflicts damage to pokemon2, which is calculated as

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

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
```


SQUIRTLE Lv5
HP
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}
ATTACK!
attack(pokemon1,
pokemon2,
types_table)
CHARMANDER Lv5
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}
What will
CHARMANDER do?
FIGHT

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

- Pokemon is a game with a gameplay that goes far deeper than just this basic “attack” gameplay mechanic.

- However, I believe that this example helped visualize how OOP can help improve the structure of your code!

About OOP

- **Object Oriented Programming** (OOP) is a programming paradigm that relies on the concept of objects.
- In OOP, everything can be described as an object.

About OOP

- **Object Oriented Programming** (OOP) is a programming paradigm that relies on the concept of objects.
- In OOP, everything can be described as an object.

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.

About OOP

- **Object Oriented Programming** (OOP) is a programming paradigm that relies on the concept of objects.
- In OOP, everything can be described as an object.

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.

TO BE COVERED IN W6S1 AND W6S2!

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

- The tuple type (fast version)
- The dictionary type
- Dictionary and objects
- Object-oriented thinking and programming
- (If time allows, tuples and sets types extra practice)