Python Essentials 1:
Module 4
Functions, Tuples, Dictionaries, and Data Processing

In this module, you will cover the following topics:

code structuring and the concept of function;
function invocation and returning a result from a function;
name scopes and variable shadowing;
tuples and their purpose, constructing and using tuples;
dictionaries and their purpose, constructing and using dictionaries.

(part 6)

Sequence types and mutability

Before we start talking about tuples and dictionaries, we have to introduce two important concepts: sequence types and mutability.

A sequence type is a type of data in Python which is able to store more than one value (or less than one, as a sequence may be empty), and these values can be sequentially (hence the name) browsed, element by element.

As the for loop is a tool especially designed to iterate through sequences, we can express the definition as: a sequence is data which can be scanned by the for loop.

You've encountered one Python sequence so far - the list. The list is a classic example of a Python sequence, although there are some other sequences worth mentioning, and we're going to present them to you now.

The second notion - mutability - is a property of any of Python's data that describes its readiness to be freely changed during program execution. There are two kinds of Python data: mutable and immutable.

Mutable data can be freely updated at any time - we call such an operation in situ.

In situ is a Latin phrase that translates as literally in position. For example, the following instruction modifies the data in situ:

```
list.append(1)
```

Immutable data cannot be modified in this way.

Imagine that a list can only be assigned and read over. You would be able neither to append an element to it, nor remove any element from it. This means that appending an element to the end of the list would require the recreation of the list from scratch.

You would have to build a completely new list, consisting of the all elements of the already existing list, plus the new element.

The data type we want to tell you about now is a tuple. A tuple is an immutable sequence type. It can behave like a list, but it mustn't be modified in situ.

What is a tuple?
The first and the clearest distinction between lists and tuples is the syntax used to create them - tuples prefer to use parenthesis, whereas lists like to see brackets, although it's also possible to create a tuple just from a set of values separated by commas.

Look at the example:

```
tuple_1 = (1, 2, 4, 8)
tuple_2 = 1., .5, .25, .125
```

There are two tuples, both containing four elements.
Let's print them:

```
tuple_1 = (1, 2, 4, 8)
tuple_2 = 1., .5, .25, .125
print(tuple_1)
print(tuple_2)
```

This is what you should see in the console:

(1, 2, 4, 8)
(1.0, 0.5, 0.25, 0.125)
output

Note: each tuple element may be of a different type (floating-point, integer, or any other not-as-yet-introduced kind of data).

How to create a tuple?
It is possible to create an empty tuple - parentheses are required then:

```
empty_tuple = ()
```

If you want to create a one-element tuple, you have to take into consideration the fact that, due to syntax reasons (a tuple has to be distinguishable from an ordinary, single value), you must end the value with a comma:

```
one_element_tuple_1 = (1, )
one_element_tuple_2 = 1.,
```

Removing the commas won't spoil the program in any syntactical sense, but you will instead get two single variables, not tuples.

How to use a tuple?

If you want to get the elements of a tuple in order to read them over, you can use the same conventions to which you're accustomed while using lists.

Take a look at the code in the editor.

```
my_tuple = (1, 10, 100, 1000)

print(my_tuple[0])
print(my_tuple[-1])
print(my_tuple[1:])
print(my_tuple[:-2])

for elem in my_tuple:
    print(elem)
```

The program should produce the following output - run it and check:

```
1
1000
(10, 100, 1000)
(1, 10)
1
10
100
1000
output
```

The similarities may be misleading - don't try to modify a tuple's contents! It's not a list!

All of these instructions (except the topmost one) will cause a runtime error:

```
my_tuple = (1, 10, 100, 1000)

my_tuple.append(10000)
del my_tuple[0]
my_tuple[1] = -10
```

This is the message that Python will give you in the console window:

```
AttributeError: 'tuple' object has no attribute 'append'
```

How to use a tuple: continued

What else can tuples do for you?

the len() function accepts tuples, and returns the number of elements contained inside;
the + operator can join tuples together (we've shown you this already)
the * operator can multiply tuples, just like lists;
the in and not in operators work in the same way as in lists.
The snippet in the editor presents them all.

```
my_tuple = (1, 10, 100)

t1 = my_tuple + (1000, 10000)
t2 = my_tuple * 3

print(len(t2))
print(t1)
print(t2)
print(10 in my_tuple)
print(-10 not in my_tuple)
```

The output should look as follows:

```
9
(1, 10, 100, 1000, 10000)
(1, 10, 100, 1, 10, 100, 1, 10, 100)
True
True
```

output

One of the most useful tuple properties is their ability to appear on the left side of the assignment operator. You saw this phenomenon some time ago, when it was necessary to find an elegant tool to swap two variables' values.

Take a look at the snippet below:

```
var = 123

t1 = (1, )
t2 = (2, )
t3 = (3, var)

t1, t2, t3 = t2, t3, t1

print(t1, t2, t3)
```

It shows three tuples interacting - in effect, the values stored in them "circulate" - t1 becomes t2, t2 becomes t3, and t3 becomes t1.

Note: the example presents one more important fact: a tuple's elements can be variables, not only literals. Moreover, they can be expressions if they're on the right side of the assignment operator.

What is a dictionary?

The dictionary is another Python data structure. It's not a sequence type (but can be easily adapted to sequence processing) and it is mutable.

To explain what the Python dictionary actually is, it is important to understand that it is literally a dictionary.

The Python dictionary works in the same way as a bilingual dictionary. For example, you have an English word (e.g., cat) and need its French equivalent. You browse the dictionary in order to find the word (you may use different techniques to do that - it doesn't matter) and eventually you get it. Next, you check the French counterpart and it is (most probably) the word "chat".

In Python's world, the word you look for is named a key. The word you get from the dictionary is called a value.

This means that a dictionary is a set of key-value pairs. Note:

each key must be unique - it's not possible to have more than one key of the same value;
a key may be any immutable type of object: it can be a number (integer or float), or even a string, but not a list;
a dictionary is not a list - a list contains a set of numbered values, while a dictionary holds pairs of values;
the len() function works for dictionaries, too - it returns the numbers of key-value elements in the dictionary;
a dictionary is a one-way tool - if you have an English-French dictionary, you can look for French equivalents of English terms, but not vice versa.

Now we can show you some working examples.

How to make a dictionary?
If you want to assign some initial pairs to a dictionary, you should use the following syntax:

```
dictionary = {"cat": "chat", "dog": "chien", "horse": "cheval"}
phone_numbers = {'boss': 5551234567, 'Suzy': 22657854310}
empty_dictionary = {}

print(dictionary)
print(phone_numbers)
print(empty_dictionary)
```

In the first example, the dictionary uses keys and values which are both strings. In the second one, the keys are strings, but the values are integers. The reverse layout (keys → numbers, values → strings) is also possible, as well as number-number combination.

The list of pairs is surrounded by curly braces, while the pairs themselves are separated by commas, and the keys and values by colons.

The first of our dictionaries is a very simple English-French dictionary. The second - a very tiny telephone directory. The empty dictionaries are constructed by an empty pair of curly braces - nothing unusual.

The dictionary as a whole can be printed with a single print() invocation. The snippet may produce the following output:

```
{'dog': 'chien', 'horse': 'cheval', 'cat': 'chat'}
{'Suzy': 5557654321, 'boss': 5551234567}
{}
```
output

Have you noticed anything surprising? The order of the printed pairs is different than in the initial assignment. What does that mean?

First of all, it's a confirmation that dictionaries are not lists - they don't preserve the order of their data, as the order is completely meaningless (unlike in real, paper dictionaries). The order in which a dictionary stores its data is completely out of your control, and your expectations. That's normal. (*)

NOTE

(*) In Python 3.6x dictionaries have become ordered collections by default. Your results may vary depending on what Python version you're using.

How to use a dictionary?

If you want to get any of the values, you have to deliver a valid key value:

```
print(dictionary['cat'])
print(phone_numbers['Suzy'])
```

Getting a dictionary's value resembles indexing, especially thanks to the brackets surrounding the key's value.

Note:

- if the key is a string, you have to specify it as a string;
- keys are case-sensitive: 'Suzy' is something different from 'suzy'.

The snippet outputs two lines of text:

```
chat
5557654321
```

output

And now the most important news: you mustn't use a non-existent key. Trying something like this:

```
print(phone_numbers['president'])
```

will cause a runtime error. Try to do it.

Fortunately, there's a simple way to avoid such a situation. The in operator, together with its companion, not in, can salvage this situation.

The following code safely searches for some French words:

```
dictionary = {"cat": "chat", "dog": "chien", "horse": "cheval"}
words = ['cat', 'lion', 'horse']

for word in words:
    if word in dictionary:
        print(word, "->", dictionary[word])
    else:
        print(word, "is not in dictionary")
```

The code's output looks as follows:

```
cat -> chat
lion is not in dictionary
horse -> cheval
```

output

NOTE

When you write a big or lengthy expression, it may be a good idea to keep it vertically aligned. This is how you can make your code more readable and more programmer-friendly, e.g.:

```
# Example 1:
dictionary = {
             "cat": "chat",
             "dog": "chien",
             "horse": "cheval"
             }
# Example 2:
phone_numbers = {'boss': 5551234567,
                 'Suzy': 22657854310
                 }
```

Such ways of formatting code are called hanging indents.

How to use a dictionary: the keys()

Can dictionaries be browsed using the for loop, like lists or tuples?

No and yes.

No, because a dictionary is not a sequence type - the for loop is useless with it.

Yes, because there are simple and very effective tools that can adapt any dictionary to the for loop requirements (in other words, building an intermediate link between the dictionary and a temporary sequence entity).

The first of them is a method named keys(), possessed by each dictionary. The method returns an iterable object consisting of all the keys gathered within the dictionary. Having a group of keys enables you to access the whole dictionary in an easy and handy way.

Just like here:

```
dictionary = {"cat": "chat", "dog": "chien", "horse": "cheval"}

for key in dictionary.keys():
    print(key, "->", dictionary[key])
```

The code's output looks as follows:

```
horse -> cheval
dog -> chien
cat -> chat
```

output

The sorted() function

Do you want it sorted? Just enrich the for loop to get such a form:

```
for key in sorted(dictionary.keys()):
```

The sorted() function will do its best - the output will look like this:

```
cat -> chat
dog -> chien
horse -> cheval
```

How to use a dictionary: The items() and values() methods

Another way is based on using a dictionary's method named items(). The method returns tuples (this is the first example where tuples are something more than just an example of themselves) where each tuple is a key-value pair.

This is how it works:

```
dictionary = {"cat": "chat", "dog": "chien", "horse": "cheval"}

for english, french in dictionary.items():
    print(english, "->", french)
```

Note the way in which the tuple has been used as a for loop variable.

The example prints:

```
cat -> chat
dog -> chien
horse -> cheval
```
output

There is also a method named values(), which works similarly to keys(), but returns values.

Here is a simple example:

```
dictionary = {"cat": "chat", "dog": "chien", "horse": "cheval"}

for french in dictionary.values():
    print(french)
```

As the dictionary is not able to automatically find a key for a given value, the role of this method is rather limited.

Here is the expected output:

```
cheval
chien
```
chat

How to use a dictionary: modifying and adding values

Assigning a new value to an existing key is simple - as dictionaries are fully mutable, there are no obstacles to modifying them.

We're going to replace the value "chat" with "minou", which is not very accurate, but it will work well with our example.

Look:

```
dictionary = {"cat": "chat", "dog": "chien", "horse": "cheval"}

dictionary['cat'] = 'minou'
print(dictionary)
```

The output is:

```
{'cat': 'minou', 'dog': 'chien', 'horse': 'cheval'}
```
output

Adding a new key

Adding a new key-value pair to a dictionary is as simple as changing a value - you only have to assign a value to a new, previously non-existent key.

Note: this is very different behavior compared to lists, which don't allow you to assign values to non-existing indices.

Let's add a new pair of words to the dictionary - a bit weird, but still valid:

```
dictionary = {"cat": "chat", "dog": "chien", "horse": "cheval"}

dictionary['swan'] = 'cygne'
print(dictionary)
```

The example outputs:

```
{'cat': 'chat', 'dog': 'chien', 'horse': 'cheval', 'swan': 'cygne'}
```
output

EXTRA

You can also insert an item to a dictionary by using the update() method, e.g.:

```
dictionary = {"cat": "chat", "dog": "chien", "horse": "cheval"}

dictionary.update({"duck": "canard"})
print(dictionary)
```

Removing a key

Can you guess how to remove a key from a dictionary?

Note: removing a key will always cause the removal of the associated value. Values cannot exist without their keys.

This is done with the del instruction.

Here's the example:

```
dictionary = {"cat": "chat", "dog": "chien", "horse": "cheval"}

del dictionary['dog']
print(dictionary)
```

Note: removing a non-existing key causes an error.

The example outputs:

```
{'cat': 'chat', 'horse': 'cheval'}
```
output

EXTRA

To remove the last item in a dictionary, you can use the popitem() method:

```
dictionary = {"cat": "chat", "dog": "chien", "horse": "cheval"}

dictionary.popitem()
print(dictionary)    # outputs: {'cat': 'chat', 'dog': 'chien'}
```

In the older versions of Python, i.e., before 3.6.7, the popitem() method removes a random item from a dictionary.

Tuples and dictionaries can work together

We've prepared a simple example, showing how tuples and dictionaries can work together.

Let's imagine the following problem:

you need a program to evaluate the students' average scores;
the program should ask for the student's name, followed by her/his single score;
the names may be entered in any order;
entering an empty name finishes the inputting of the data (note 1: entering an empty score will raise the ValueError exception, but don't worry about that now, you'll see how to handle such cases when we talk about exceptions in the second part of the Python Essentials course series)
a list of all names, together with the evaluated average score, should be then emitted.

Look at the code in the editor. This how to do it.

```
school_class = {}

while True:
    name = input("Enter the student's name: ")
    if name == '':
        break

    score = int(input("Enter the student's score (0-10): "))
    if score not in range(0, 11):
          break

    if name in school_class:
        school_class[name] += (score,)
    else:
        school_class[name] = (score,)

for name in sorted(school_class.keys()):
    adding = 0
    counter = 0
    for score in school_class[name]:
        adding += score
        counter += 1
    print(name, ":", adding / counter)
```

Now, let's analyze it line by line:

line 1: create an empty dictionary for the input data; the student's name is used as a key, while all the associated scores are stored in a tuple (the tuple may be a dictionary value - that's not a problem at all)
line 3: enter an "infinite" loop (don't worry, it'll break at the right moment)
line 4: read the student's name here;
line 5-6: if the name is an empty string (), leave the loop;
line 8: ask for one of the student's scores (an integer from the range 0-10)
line 9-10: if the score entered is not within the range from 0 to 10, leave the loop;
line 12-13: if the student's name is already in the dictionary, lengthen the associated tuple with the new score (note the += operator)
line 14-15: if this is a new student (unknown to the dictionary), create a new entry - its value is a one-element tuple containing the entered score;
line 17: iterate through the sorted students' names;
line 18-19: initialize the data needed to evaluate the average (sum and counter)
line 20-22: we iterate through the tuple, taking all the subsequent scores and updating the sum, together with the counter;
line 23: evaluate and print the student's name and average score.

This is a record of a conversation we had with our program:

```
Enter the student's name: Bob
Enter the student's score (0-10): 7
Enter the student's name: Andy
Enter the student's score (0-10): 3
Enter the student's name: Bob
Enter the student's score (0-10): 2
Enter the student's name: Andy
Enter the student's score (0-10): 10
Enter the student's name: Andy
Enter the student's score (0-10): 3
Enter the student's name: Bob
Enter the student's score (0-10): 9
Enter the student's name:
Andy : 5.333333333333333
Bob : 6.0
```

Key takeaways: tuples

1. Tuples are ordered and unchangeable (immutable) collections of data. They can be thought of as immutable lists. They are written in round brackets:

```
my_tuple = (1, 2, True, "a string", (3, 4), [5, 6], None)
print(my_tuple)

my_list = [1, 2, True, "a string", (3, 4), [5, 6], None]
print(my_list)
```

Each tuple element may be of a different type (i.e., integers, strings, booleans, etc.). What is more, tuples can contain other tuples or lists (and the other way round).

2. You can create an empty tuple like this:

```
empty_tuple = ()
print(type(empty_tuple))    # outputs: <class 'tuple'>
```

3. A one-element tuple may be created as follows:

```
one_elem_tuple_1 = ("one", )    # Brackets and a comma.
one_elem_tuple_2 = "one",       # No brackets, just a comma.
```

If you remove the comma, you will tell Python to create a variable, not a tuple:

```
my_tuple_1 = 1,
print(type(my_tuple_1))    # outputs: <class 'tuple'>

my_tuple_2 = 1             # This is not a tuple.
print(type(my_tuple_2))    # outputs: <class 'int'>
```

4. You can access tuple elements by indexing them:

```
my_tuple = (1, 2.0, "string", [3, 4], (5, ), True)
print(my_tuple[3])    # outputs: [3, 4]
```

5. Tuples are immutable, which means you cannot change their elements (you cannot append tuples, or modify, or remove tuple elements). The following snippet will cause an exception:

```
my_tuple = (1, 2.0, "string", [3, 4], (5, ), True)
my_tuple[2] = "guitar"    # The TypeError exception will be raised.
```

However, you can delete a tuple as a whole:

```
my_tuple = 1, 2, 3,
del my_tuple
print(my_tuple)    # NameError: name 'my_tuple' is not defined
```

6. You can loop through a tuple elements (Example 1), check if a specific element is (not)present in a tuple (Example 2), use the len() function to check how many elements there are in a tuple (Example 3), or even join/multiply tuples (Example 4):

```
# Example 1
tuple_1 = (1, 2, 3)
for elem in tuple_1:
    print(elem)

# Example 2
tuple_2 = (1, 2, 3, 4)
print(5 in tuple_2)
print(5 not in tuple_2)

# Example 3
tuple_3 = (1, 2, 3, 5)
print(len(tuple_3))

# Example 4
tuple_4 = tuple_1 + tuple_2
tuple_5 = tuple_3 * 2

print(tuple_4)
print(tuple_5)
```

EXTRA

You can also create a tuple using a Python built-in function called tuple(). This is particularly useful when you want to convert a certain iterable (e.g., a list, range, string, etc.) to a tuple:

```
my_tuple = tuple((1, 2, "string"))
print(my_tuple)

my_list = [2, 4, 6]
print(my_list)    # outputs: [2, 4, 6]
print(type(my_list))    # outputs: <class 'list'>
tup = tuple(my_list)
print(tup)    # outputs: (2, 4, 6)
print(type(tup))    # outputs: <class 'tuple'>
```

By the same fashion, when you want to convert an iterable to a list, you can use a Python built-in function called list():

```
tup = 1, 2, 3,
my_list = list(tup)
print(type(my_list))    # outputs: <class 'list'>
```

Key takeaways: dictionaries

1. Dictionaries are unordered*, changeable (mutable), and indexed collections of data. (*In Python 3.6x dictionaries have become ordered by default.

Each dictionary is a set of key: value pairs. You can create it by using the following syntax:

```
my_dictionary = {
    key1: value1,
    key2: value2,
    key3: value3,
    }
```

2. If you want to access a dictionary item, you can do so by making a reference to its key inside a pair of square brackets (ex. 1) or by using the get() method (ex. 2):

```
pol_eng_dictionary = {
    "kwiat": "flower",
    "woda": "water",
    "gleba": "soil"
    }

item_1 = pol_eng_dictionary["gleba"]    # ex. 1
print(item_1)    # outputs: soil

item_2 = pol_eng_dictionary.get("woda")
print(item_2)    # outputs: water
```

3. If you want to change the value associated with a specific key, you can do so by referring to the item's key name in the following way:

```
pol_eng_dictionary = {
    "zamek": "castle",
    "woda": "water",
    "gleba": "soil"
    }

pol_eng_dictionary["zamek"] = "lock"
item = pol_eng_dictionary["zamek"]
print(item)  # outputs: lock
```

4. To add or remove a key (and the associated value), use the following syntax:

```
phonebook = {}    # an empty dictionary

phonebook["Adam"] = 3456783958    # create/add a key-value pair
print(phonebook)    # outputs: {'Adam': 3456783958}

del phonebook["Adam"]
print(phonebook)    # outputs: {}
```

You can also insert an item to a dictionary by using the update() method, and remove the last element by using the popitem() method, e.g.:

```
pol_eng_dictionary = {"kwiat": "flower"}

pol_eng_dictionary.update({"gleba": "soil"})
print(pol_eng_dictionary)    # outputs: {'kwiat': 'flower', 'gleba': 'soil'}

pol_eng_dictionary.popitem()
print(pol_eng_dictionary)    # outputs: {'kwiat': 'flower'}
```

5. You can use the for loop to loop through a dictionary, e.g.:

```
pol_eng_dictionary = {
    "zamek": "castle",
    "woda": "water",
    "gleba": "soil"
    }

for item in pol_eng_dictionary:
    print(item)

# outputs: zamek
#          woda
#          gleba
```

6. If you want to loop through a dictionary's keys and values, you can use the items() method, e.g.:

```
pol_eng_dictionary = {
    "zamek": "castle",
    "woda": "water",
    "gleba": "soil"
    }

for key, value in pol_eng_dictionary.items():
    print("Pol/Eng ->", key, ":", value)
```

7. To check if a given key exists in a dictionary, you can use the in keyword:

```
pol_eng_dictionary = {
    "zamek": "castle",
    "woda": "water",
    "gleba": "soil"
    }

if "zamek" in pol_eng_dictionary:
    print("Yes")
else:
    print("No")
```

8. You can use the del keyword to remove a specific item, or delete a dictionary. To remove all the dictionary's items, you need to use the clear() method:

```
pol_eng_dictionary = {
    "zamek": "castle",
    "woda": "water",
    "gleba": "soil"
    }

print(len(pol_eng_dictionary))    # outputs: 3
del pol_eng_dictionary["zamek"]    # remove an item
print(len(pol_eng_dictionary))    # outputs: 2

pol_eng_dictionary.clear()   # removes all the items
print(len(pol_eng_dictionary))    # outputs: 0

del pol_eng_dictionary    # removes the dictionary
```

9. To copy a dictionary, use the copy() method:

```
pol_eng_dictionary = {
    "zamek": "castle",
    "woda": "water",
    "gleba": "soil"
    }

copy_dictionary = pol_eng_dictionary.copy()
```

Key takeaways: tuples and dictionaries

Exercise 1

What happens when you attempt to run the following snippet?

```
my_tup = (1, 2, 3)
print(my_tup[2])
```

Exercise 2

What is the output of the following snippet?

```
tup = 1, 2, 3
a, b, c = tup

print(a * b * c)
```

Exercise 3

Complete the code to correctly use the count() method to find the number of duplicates of 2 in the following tuple.

```
tup = 1, 2, 3, 2, 4, 5, 6, 2, 7, 2, 8, 9
duplicates = # Write your code here.

print(duplicates)    # outputs: 4
```

Exercise 4

Write a program that will "glue" the two dictionaries (d1 and d2) together and create a new one (d3).

```
d1 = {'Adam Smith': 'A', 'Judy Paxton': 'B+'}
d2 = {'Mary Louis': 'A', 'Patrick White': 'C'}
d3 = {}

for item in (d1, d2):
    # Write your code here.

print(d3)
```

Exercise 5

Write a program that will convert the my_list list to a tuple.

```
my_list = ["car", "Ford", "flower", "Tulip"]

t =  # Write your code here.
print(t)
```

Exercise 6

Write a program that will convert the colors tuple to a dictionary.

```
colors = (("green", "#008000"), ("blue", "#0000FF"))

# Write your code here.

print(colors_dictionary)
```

Exercise 7

What will happen when you run the following code?

```
my_dictionary = {"A": 1, "B": 2}
copy_my_dictionary = my_dictionary.copy()
my_dictionary.clear()

print(copy_my_dictionary)
```

Exercise 8

What is the output of the following program?

```
colors = {
    "white": (255, 255, 255),
    "grey": (128, 128, 128),
    "red": (255, 0, 0),
    "green": (0, 128, 0)
    }

for col, rgb in colors.items():
    print(col, ":", rgb)
```

PROJECT

Estimated time
30-120 minutes

Level of difficulty
Medium/Hard

Objectives
perfecting the student's skills in using Python for solving complex problems,
integration of programming techniques in one program consisting of many various parts.
Scenario
Your task is to write a simple program which pretends to play tic-tac-toe with the user. To make it all easier for you, we've decided to simplify the game. Here are our assumptions:

the computer (i.e., your program) should play the game using 'X's;
the user (e.g., you) should play the game using 'O's;
the first move belongs to the computer - it always puts its first 'X' in the middle of the board;
all the squares are numbered row by row starting with 1 (see the example session below for reference)
the user inputs their move by entering the number of the square they choose - the number must be valid, i.e., it must be an integer, it must be greater than 0 and less than 10, and it cannot point to a field which is already occupied;
the program checks if the game is over - there are four possible verdicts: the game should continue, or the game ends with a tie, your win, or the computer's win;
the computer responds with its move and the check is repeated;
don't implement any form of artificial intelligence - a random field choice made by the computer is good enough for the game.

The example session with the program may look as follows:

```
+-------+-------+-------+
|       |       |       |
|   1   |   2   |   3   |
|       |       |       |
+-------+-------+-------+
|       |       |       |
|   4   |   X   |   6   |
|       |       |       |
+-------+-------+-------+
|       |       |       |
|   7   |   8   |   9   |
|       |       |       |
+-------+-------+-------+
Enter your move: 1
+-------+-------+-------+
|       |       |       |
|   0   |   2   |   3   |
|       |       |       |
+-------+-------+-------+
|       |       |       |
|   4   |   X   |   6   |
|       |       |       |
+-------+-------+-------+
|       |       |       |
|   7   |   8   |   9   |
|       |       |       |
+-------+-------+-------+
```

```
+-------+-------+-------+
|       |       |       |
|   0   |   X   |   3   |
|       |       |       |
+-------+-------+-------+
|       |       |       |
|   4   |   X   |   6   |
|       |       |       |
+-------+-------+-------+
|       |       |       |
|   7   |   8   |   9   |
|       |       |       |
+-------+-------+-------+
Enter your move: 8
+-------+-------+-------+
|       |       |       |
|   0   |   X   |   3   |
|       |       |       |
+-------+-------+-------+
|       |       |       |
|   4   |   X   |   6   |
|       |       |       |
+-------+-------+-------+
|       |       |       |
|   7   |   0   |   9   |
|       |       |       |
+-------+-------+-------+
+-------+-------+-------+
|       |       |       |
|   0   |   X   |   3   |
|       |       |       |
+-------+-------+-------+
|       |       |       |
|   4   |   X   |   X   |
|       |       |       |
+-------+-------+-------+
|       |       |       |
|   7   |   0   |   9   |
|       |       |       |
+-------+-------+-------+
Enter your move: 4
+-------+-------+-------+
|       |       |       |
|   0   |   X   |   3   |
|       |       |       |
+-------+-------+-------+
|       |       |       |
|   0   |   X   |   X   |
|       |       |       |
+-------+-------+-------+
|       |       |       |
|   7   |   0   |   9   |
|       |       |       |
+-------+-------+-------+
```

```
+-------+-------+-------+
|       |       |       |
|   O   |   X   |   X   |
|       |       |       |
+-------+-------+-------+
|       |       |       |
|   O   |   X   |   X   |
|       |       |       |
+-------+-------+-------+
|       |       |       |
|   7   |   O   |   9   |
|       |       |       |
+-------+-------+-------+
Enter your move: 7
+-------+-------+-------+
|       |       |       |
|   O   |   X   |   X   |
|       |       |       |
+-------+-------+-------+
|       |       |       |
|   O   |   X   |   X   |
|       |       |       |
+-------+-------+-------+
|       |       |       |
|   O   |   O   |   9   |
|       |       |       |
+-------+-------+-------+
You won!
```

Requirements
Implement the following features:

the board should be stored as a three-element list, while each element is another three-element list (the inner lists represent rows) so that all of the squares may be accessed using the following syntax:

```
board[row][column]
```

each of the inner list's elements can contain 'O', 'X', or a digit representing the square's number (such a square is considered free)
the board's appearance should be exactly the same as the one presented in the example.
implement the functions defined for you in the editor.

Drawing a random integer number can be done by utilizing a Python function called randrange(). The example program below shows how to use it (the program prints ten random numbers from 0 to 8).

Note: the from-import instruction provides an access to the randrange function defined within an external Python module callled random.

```
from random import randrange

for i in range(10):
    print(randrange(8))
```

Congratulations! You have completed Module 4.

Well done! You've reached the end of Module 4 and completed a major milestone in your Python programming education. Here's a short summary of the objectives you've covered and got familiar with in Module 4:

the defining and using of functions - their rationale, purpose, conventions, and traps;
the concept of passing arguments in different ways and setting their default values, along with the mechanisms of returning the function's results;
name scope issues;
new data aggregates: tuples and dictionaries, and their role in data processing.

You are now ready to take the module quiz and attempt the final challenge: Module 4 Test, which will help you gauge what you've learned so far.