Lesson_6 Functions and First-Class Functions Lambdas

September 8, 2020

1 Functions, First-Class Functions and Lambdas

1.0.1 Agenda

- Code blocks. Functions as code blocks
- Syntax and conventions
- Code reuseability
- Types of functions: UDF, build-in, annonymous
- Arguments and kward arguments, default values
- Return statement
- First-class objects. Functions as first-class objects
- Flexible arguments: * args, ** kargs
- Let's practice

1.0.2 Resources

- Python Reference Manual - 4.Execution model
- Fluent Python - Chapter 5: First-cass functions
- PEP-3107: Function Annotations
- PEP-3102: Keyword-Only Arguments

1.0.3 Code blocks. Functions as code blocks

A code block is a piece of Python program text that can be executed as a unit, such as a module, a class definition or a function body. Some code blocks (like modules) are normally executed only once, others (like function bodies) may be executed many times. Code blocks may textually contain other code blocks. Code blocks may invoke other code blocks (that may or may not be textually contained in them) as part of their execution, e.g., by invoking (calling) a function.

The following are code blocks:

- A module is a code block.
- A function body is a code block.

- A class definition is a code block.
- Each command typed interactively is a separate code block
- A script file (a file given as standard input to the interpreter or specified on the interpreter command line the first argument) is a code block;
- A script command (a command specified on the interpreter command line with the '-c' option) is a code block.
- The file read by the built-in function execfile() is a code block.
- The string argument passed to the built-in function eval() and to the exec statement is a code block.
- Expression read and evaluated by the built-in function input() is a code block.

4.1 Code blocks, execution frames, and namespaces

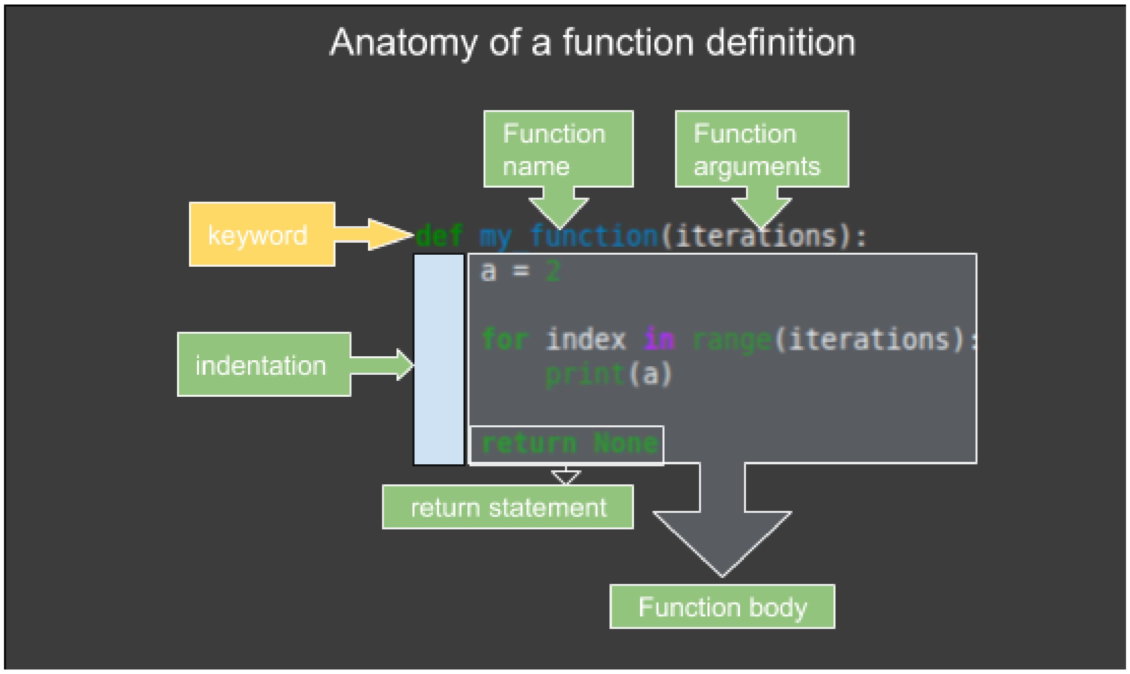
```
[ ]: def i_am_a_function():
         number = 1
         string = 'a'
         dictionary = {string:number}
         print(f"I evaluated the following lines in my code block:
      ↪\n{number}\n{string}\n{dictionary}")


     for index in range(5):
         print(f"Iteration: {index}")
         i_am_a_function()
```

1.0.4 Syntax and conventions

All the functions that are written by us come under the category of user-defined functions. Below are the steps for writing user-defined functions:

- **def** keyword is used to declare user defined functions.
- an indented block of statements follows the function name and arguments which contains the body of the function. > - return statement inside of the function body: values that we want to return (if the return statement is not present, the function will return None by default)



```
def my_function(iterations):
    a = 2

    for index in range(iterations):
        print(a)
    return None
```

[ ]:
```
def my_function(iterations):
    a = 2
    for index in range(iterations):
        print(a)
    return None
```

Calling a functions Defined functions can be called by using the function_name and (). If the function requires arguments those arguments will be passed to the function inside of (iterations). For multiple arguments, we will use commas to separate them, similar to the argument definition when creating the function.

[ ]:
```
my_function(5)
```

Positional and keyword arguments We have two ways of passng arguments to a function:

- Positional : `my_function(arg_1, arg_2, arg_3)`
- Keyword: `my_function(arg_1_name=arg_1, arg_2_name=arg_2)`

[ ]:
```
def my_function(iterations, print_value):
    for index in range(iterations):
        print(print_value)
    return None
```

[ ]:
```
# positional
my_function(5, 'test')
```

[ ]:
```
# keyword
my_function(iterations=5, print_value='test')
```

Are arguments passed by Reference or passed by value?

[ ]:
```
def my_function(x):
    print(f"Recived: {x} with id: {id(x)}")

x = 10
print(f"Passing: {x} with id: {id(x)}")

my_function(x)
```

In the case above, my_function got the reference of the variable x as an argument. What happens if we modify the variable x inside the function?

[ ]:
```
def my_function(x):
    x += 10
    print(f'Recived: {x} with id: {id(x)}')

x = 10
print(f'Passing: {x} with id: {id(x)}')

my_function(x)

print(f'After the function call: x: {x} with id: {id(x)}')
```

Because variables of type int are immutable, inside the function a new object is created taking the value of x (passed as an argument to the function) and adds 10. If you look closely at the id of x inside the function block you will see that is different than the value of x before and after the function call.

In the case of immutable types, such as lists, changing the list inside a function will reflect outside.

[ ]:
```
# Multiple variables can point to the same referance
def my_list_function(lst):
    print(f'The reference of lst before change is: {id(lst)}')
    print(f'Lst before change contains: {lst}')
    lst[0] = 'test'

    print(f'The reference of lst after change is: {id(lst)}')
    print(f'Lst after change contains: {lst}')
    print('-----')


elements = [1,2,3,4,5]
print(f'Elements contains: {elements}')
print(f'The reference of elements is: {id(elements)}')
print('-----')

my_list_function(elements)

print(f'Elements contains: {elements}')
print(f'The reference of elements is: {id(elements)}')
```

You notice that the reference to our list never changed. And the changes made to the list inside the function reflected outside.

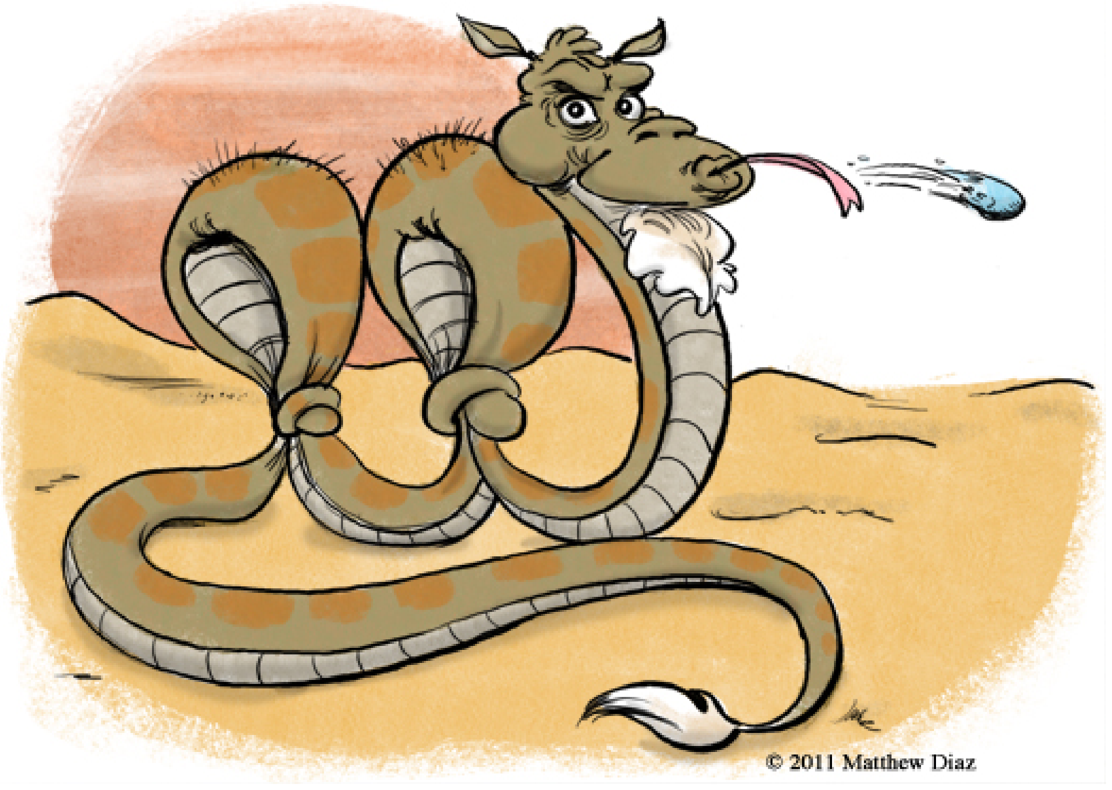
Conventions Function names should be lowercase, with words separated by underscores as necessary to improve readability.

Variable names follow the same convention as function names.

PEP-8: Function and Variable Names

Python functions use snake_case not CamelCase Following the convention is considered good practice and tramandesly improves readability. The code will work either way but as Guido says:

> *The code is read much more often than it is written.*



1.0.5 Code reusability

Python function is a group of related statements that perform a specific task. Functions provide better modularity for our application and code reusability. Function achieve two important purposes:

- Group logic and functionality
- Reuse the logic as many times as required and in different contexts

[ ]:
```
from datetime import datetime
import time

def log_variable_value(variable):
    print(f'At this moment: {datetime.now()} The variable value is {variable}␣
 ↪and referance: {id(variable)}')

log_variable_value(5)

for index in range(10):
    time.sleep(2)
    log_variable_value(index)
```

[ ]:
```
def get_user_name():
    name = input('Please write your name:')
    return name
```

[ ]:
```
name = get_user_name()
log_variable_value(name)
```

[ ]:
```
required_users = 3
users = {}
for user_no in range(required_users):
    name = get_user_name()
    users[name] = f'{name} is the user number: {user_no}'

log_variable_value(users)
```

!Challenge 1!: Let's make the input message configurable for each iteration. Change get_user_name function in order to be able to have a custom input message.

example messages: "Please write the name for user 0:", "Please write the name for user 1:", "Please write the name for user 2:"

1.0.6 Types of functions

A function is also known as a callable object. From the examples above you can see that when we call a function we use the call operator `()`. Besides the functions we define in our code, there are multiple objects that can make use of call operator, those objects are callable. To determine whether an object is callable we can use the `callable()` built-in function

[ ]:
```
callable(log_variable_value)
```

[ ]:
```
callable(users)
```

[ ]:
```
callable(print)
```

[ ]:
```
callable(callable)
```

In python we have three types of functions:

- User-defined functions and anonymous functions > Created with the `def` statement or `lambda expressions` in the case of anonymous functions
- Built-in functions > a function implemented in C (for CPython), like `len`.
- Generator functions > Functions or methods that use the yield keyword. When called, generator functions return a generator object. > > *We will look into generator object in the next lectures*

Besides functions there multiple kinds of callables that are worth mentioning:

- Built-in methods > Methods implemented in C, like `dict.get` or `dict.keys`.
- Methods > Functions defined in the body of a class.
- Classes > When invoked, a class runs its `__new__` method to create an instance, then `__in it__` to initialize it, and finally the instance is returned to the caller. Because there is no new operator in Python, calling a class is like calling a function
- Class instances > If a class defines a `__call__` method, then its instances may be invoked as functions. Those are also called User-defined callable types

The seven flavours of callable objects, Chapter 5: First-class functions, page 144

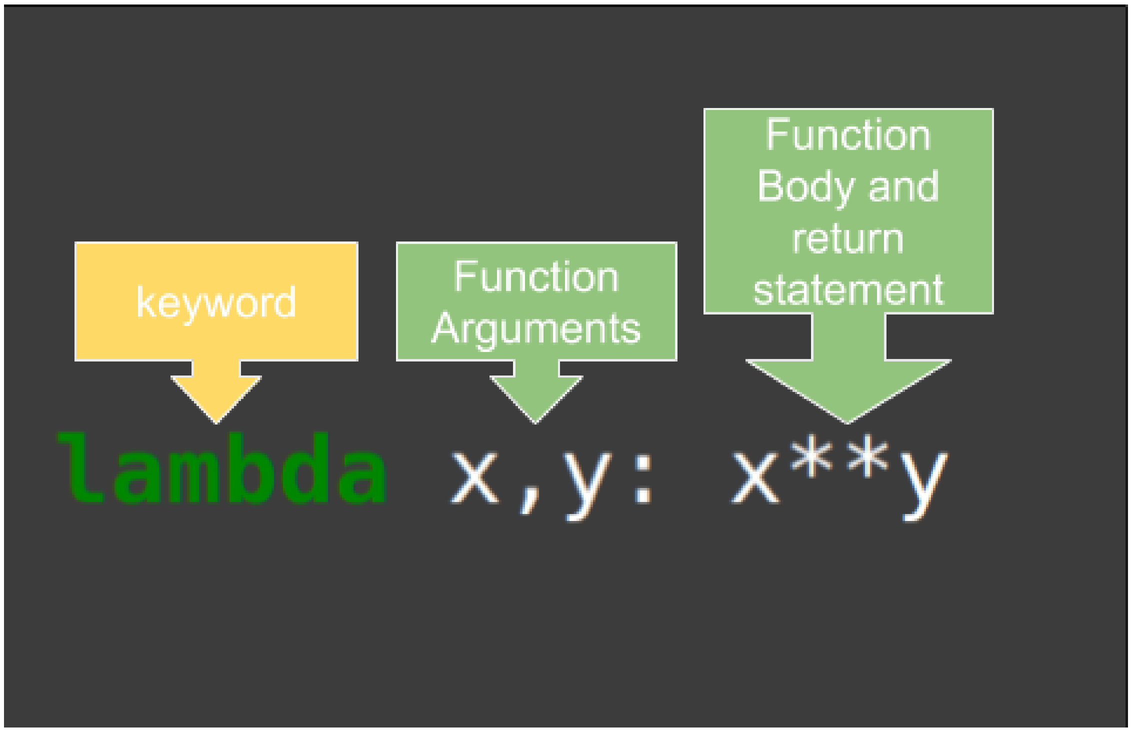
We will talk more in-depth about classes, methods and instances in the next lectures. I am mentioning them here as a reference for callable objects

[ ]:
```
fruits = ['strawberry', 'fig', 'apple', 'cherry', 'raspberry', 'banana']

# This is a built-in function
sorted(fruits)
```

[ ]:
```
# Using a built-in function as a sort key
sorted(fruits, key=len)
```

[ ]:
```
# Using anonymous function as a sort key
sorted(fruits, key= lambda word:word[::-1])
```



```
lambda x,y: x**y
```

Anonymous function syntax

```
[ ]: # giving a name to a function like this kind of defits the purpose of␣
      ↪annonymous functions
     func = lambda x,y:x**y
     func(2,2)
```

```
[ ]: # lambda function used inside map built-in
     lst = [2, 4, 6, 8, 10]
     squared_lst = list(map(lambda x: x**2, lst))
     print(squared_lst)
```

!Chalange 2!: Transform the map and lambda example using list comprehensions.

Starting with `lst = [2, 4, 6, 8, 10]` we need optain the resoult `squared_lst = [4, 16, 36, 64, 100]`

1.0.7 First-class objects. Function as first class object

First-class objects are defined as a program entity that can be:

- created at runtime
- assigned to a variable or element in a data structure
- passed as an argument to a function
- returned as the result of a function.

Functions in Python are first-class objects and have all the above properties. Integers, strings and dictionaries are other examples of first-class objects. Functions being *first-class citizens* of the language is no common and is one of the things that makes Python a powerful programing language

Next we will look at how function are objects:

- access function object attribute **doc**
- assign a function to a variable
- pass function as an argument
- return a function as a result of another function

[ ]:
```
def mul(x, y):
    '''returns x*y'''
    return x*y

def add(x, y):
    '''returns x+y'''
    return x+y
```

[ ]:
```
mul(5, 6)

mul.__doc__
```

[ ]:
```
add(5, 6)

add.__doc__
```

[ ]:
```
# same as in the case of lambda we can assign an user-definde function to a␣
 ↪variable name
func = mul
print(func(5,6))
func.__doc__
```

[ ]:
```
def compute(operation, arg_1, arg_2):
    '''run the provided operation with the provided arguments'''
    return operation(arg_1, arg_2)
```

[ ]:
```
compute(add, 5,6)
```

[ ]:
```
compute(mul, 5,6)
```

[ ]:
```
def special_func():
    def nested_func(a,b):
        return a*b
    return nested_func
```

[ ]:
```
my_special_func = special_func()
callable(my_special_func)
```

[ ]:
```
my_special_func(5,6)
```

The possibility of passing a function as arguments and returning a function as a result led to a lot of powerful features such as decorators, that we will look into in the next lectures

1.0.8 Flexible arguments: * args, ** kargs

One of the best features of Python functions is the extremely flexible parameter handling mechanism, enhanced with keyword-only arguments in Python 3. Closely related is the use of * and ** to “explode” iterables and mappings into separate arguments when we call a function.

Remember the * from tuple unpacking tricks?

We can collect extra elements from tuples using *

[ ]:
```
address = ('Memorandumului', 56, 'Cluj-Napoca','Cluj', 'Romania')
street, *rest = address
```

Same as tuple unpacking * can be used when defining a function to collect an unlimited number of arguments. The convention, in this case, is to use `*args` but any variable name after `*` will work: `*rest`

[ ]:
```
def my_func(*args):
    return args
```

[ ]:
```
my_func(1,2,3,4,5,6,7,8)
```

[ ]:
```
type(my_func(1,2,3,4,5,6,7,8))
```

[ ]:
```
my_func('a','b', [1,2,3,4])
```

[ ]:
```
type(my_func('a','b', [1,2,3,4]))
```

Same as `*args` we have `**kwargs` that can collect an unlimited number of keyword arguments. The convention, in this case, is to use `*kwargs` but any variable name after `*` will work: `**rest`

[ ]:
```
def my_keyword_func(**kwargs):
    return kwargs
```

[ ]:
```
my_keyword_func(city='Cluj',street='Memorandumului')
```

[ ]:
```
type(my_keyword_func(city='Cluj',street='Memorandumului'))
```

Combine `*args` and `**kwarg`

[ ]:
```python
def my_func(*args, **kwargs):
    arguments = {
        'args': args,
        'kwargs': kwargs
    }
    return arguments
```

[ ]:
```python
my_func(1, 2, 3, 4, 5, 6, 7, 8, city='Cluj',street='Memorandumului')
```

[ ]:
```python
my_func(city='Cluj')
```

[ ]:
```python
my_func(city='Cluj',street='Memorandumului', 1, 2, 3, 4, 5, 6, 7, 8,)
```

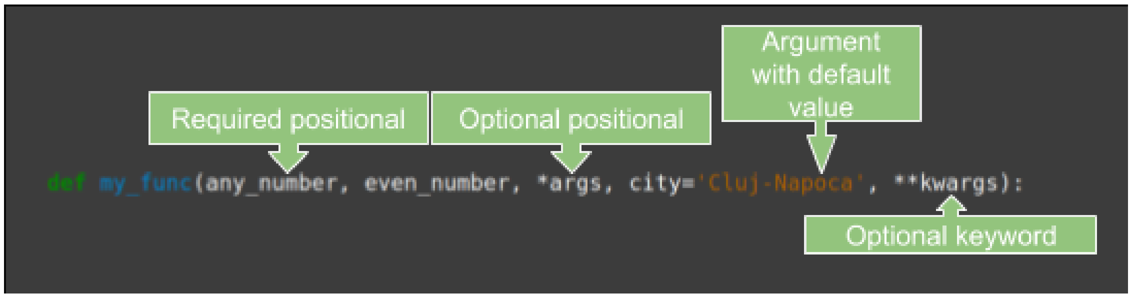
We can also require some arguments for our function and collect the rest. If we want to specify a default for our function we can do it in the function definition as follows: `my_func(any_number, even_number, *args, city='Cluj-Napoca' **kwargs)`

[ ]:
```python
def my_func(any_number, even_number, *args, city='Cluj-Napoca', **kwargs):
    arguments = {
        'args': args,
        'kwargs': kwargs
    }
    return arguments
```

[ ]:
```python
my_func(1000,2002,3,4,5,6,7, city='Cluj-Napoca', country='Romania')
```

As you can see `*args` and `**kwargs` are collecting only the extra arguments

[ ]:
```python
my_func(city='Cluj')
```



```python
def my_func(any_number, even_number, *args, city='Cluj-Napoca', **kwargs):
```

1.0.9 Let's practice

For this exercise we will perform some data aggregation tasks, on data from eurostat regarding internet access coverage.

Tasks:

- Average/year

- Average/country

Raw dataset:

```
description = ('Country', [
    '2011 ', '2012 ', '2013 ', '2014 ', '2015 ', '2016 ', '2017 ', '2018 ',
    '2019 '
])

raw_data = [
    ('AL', [': ', ': ', ': ', ': ', ': ', ': ', ': ', '84 ', ': ']),
    ('AT', ['75 ', '79 ', '81 ', '81 ', '82 ', '85 ', '89 ', '89 ', '90 ']),
    ('BA', [': ', ': ', ': ', ': ', ': ', ': ', ': ', '69 ', '72 ']),
    ('BE', ['77 ', '78 ', '80 ', '83 ', '82 ', '85 ', '86 ', '87 ', '90 ']),
    ('BG', ['45 ', '51 ', '54 ', '57 ', '59 ', '64 ', '67 ', '72 ', '75 ']),
    ('CH', [': ', ': ', ': ', '91 ', ': ', ': ', '93 b', ': ', '96 ']),
    ('CY', ['57 ', '62 ', '65 ', '69 ', '71 ', '74 ', '79 ', '86 ', '90 ']),
    ('CZ', ['67 ', '73 ', '73 ', '78 ', '79 ', '82 ', '83 ', '86 ', '87 ']),
    ('DE', ['83 ', '85 ', '88 ', '89 ', '90 ', '92 ', '93 ', '94 ', '95 ']),
    ('DK', ['90 ', '92 ', '93 ', '93 ', '92 ', '94 ', '97 ', '93 ', '95 ']),
    ('EA', ['74 ', '76 ', '79 ', '81 ', '83 ', '85 ', '87 ', '89 ', '90 ']),
    ('EE', ['69 ', '74 ', '79 ', '83 ', '88 ', '86 ', '88 ', '90 ', '90 ']),
    ('EL', ['50 ', '54 ', '56 ', '66 ', '68 ', '69 ', '71 ', '76 ', '79 ']),
    ('ES', ['63 ', '67 ', '70 ', '74 ', '79 ', '82 ', '83 ', '86 ', '91 ']),
    ('FI', ['84 ', '87 ', '89 ', '90 ', '90 ', '92 ', '94 ', '94 ', '94 ']),
    ('FR', ['76 ', '80 ', '82 ', '83 ', '83 ', '86 ', '86 ', '89 ', '90 ']),
    ('HR', ['61 ', '66 ', '65 ', '68 ', '77 ', '77 ', '76 ', '82 ', '81 ']),
    ('HU', ['63 ', '67 ', '70 ', '73 ', '76 ', '79 ', '82 ', '83 ', '86 ']),
    ('IE', ['78 ', '81 ', '82 ', '82 ', '85 ', '87 ', '88 ', '89 ', '91 ']),
    ('IS', ['93 ', '95 ', '96 ', '96 ', ': ', ': ', '98 ', '99 ', '98 ']),
    ('IT', ['62 ', '63 ', '69 ', '73 ', '75 ', '79 ', '81 ', '84 ', '85 ']),
    ('LT', ['60 ', '60 ', '65 ', '66 ', '68 ', '72 ', '75 ', '78 ', '82 ']),
    ('LU', ['91 ', '93 ', '94 ', '96 ', '97 ', '97 ', '97 ', '93 b', '95 ']),
    ('LV', ['64 ', '69 ', '72 ', '73 ', '76 ', '77 b', '79 ', '82 ', '85 ']),
    ('ME', [': ', '55 ', ': ', ': ', ': ', ': ', '71 ', '72 ', '74 ']),
    ('MK', [': ', '58 ', '65 ', '68 ', '69 ', '75 ', '74 ', '79 ', '82 ']),
    ('MT', ['75 ', '77 ', '78 ', '80 ', '81 ', '81 ', '85 ', '84 ', '86 ']),
    ('NL', ['94 ', '94 ', '95 ', '96 ', '96 ', '97 ', '98 ', '98 ', '98 ']),
    ('NO', ['92 ', '93 ', '94 ', '93 ', '97 ', '97 ', '97 ', '96 ', '98 ']),
    ('PL', ['67 ', '70 ', '72 ', '75 ', '76 ', '80 ', '82 ', '84 ', '87 ']),
    ('PT', ['58 ', '61 ', '62 ', '65 ', '70 ', '74 ', '77 ', '79 ', '81 ']),
    ('RO', ['47 ', '54 ', '58 ', '61 b', '68 ', '72 ', '76 ', '81 ', '84 ']),
    ('RS', [': ', ': ', ': ', ': ', '64 ', ': ', '68 ', '73 ', '80 ']),
    ('SE', ['91 ', '92 ', '93 ', '90 ', '91 ', '94 b', '95 ', '93 ', '96 ']),
    ('SI', ['73 ', '74 ', '76 ', '77 ', '78 ', '78 ', '82 ', '87 ', '89 ']),
    ('SK', ['71 ', '75 ', '78 ', '78 ', '79 ', '81 ', '81 ', '81 ', '82 ']),
    ('TR', [': ', '47 ', '49 ', '60 ', '70 ', '76 ', '81 ', '84 ', '88 ']),
    ('UK', ['83 ', '87 ', '88 ', '90 ', '91 ', '93 ', '94 ', '95 ', '96 ']),
    ('XK', [': ', ': ', ': ', ': ', ': ', ': ', '89 ', '93 ', '93 ']),
]
```

`': '` means that we don't have data for that year.

Instructions: For the implementation it is required:

- function that prepares the dataset

  This function will take the raw dataset as an argument This function will return a dict with the following structure

```
{
    'Romania': [
        {
            'year': '2019',
            'coverage': 84
        }, {
            'year': '2018',
            'coverage': 67
        },
        ...,
        {
            'year': '2011',
            'coverage': 72
        }
    ],
    'Germany': [
        {
            'year': '2019',
            'coverage': 94
        }, {
            'year': '2018',
            'coverage': 87
        },
        ...,
        {
            'year': '2011',
            'coverage': 82
        }
    ]
}
```

- function to retrieve data for each year

  This function will take the dataset and year as an argument.

  This function will return any type that you choose.

- function to retrieve data for each country

This function will take the dataset and country as an argument.

This function will return any type that you choose.

- function to perform average from an iterable(of year data or country data)

This function will take an iterable as an argument.

This function will return a float.

Example of usage

```
get_year_data(dataset, "2019")

>>> {'2019': [('Romania', 84), ('Germany', 95), ..., ('Latvia', 85)]}

get_country_data(dataset, "Romania")

>>> {'Romania': [('2019', 84), ('2018', 86), ..., ('2011', 72)]}

country_data = get_country_data(dataset, "Romania")

perform_average(country_data['Romania'])

>>> 79.4
```