Python30

Release 0.1

Esteban Solórzano

Aug 20, 2020

Python30 is an inspiration of JavaScript30 and 100DaysOfCode created to help those who are starting with Python or who want to improve their skills in this language.

It is a small course with some challenges which I invite you to take.

Find the challenges and solutions to these in the Python30 repository.

CHAPTER 1

The Rules of #Python30 Challenge

- Create a github repository
- Add your solution for every challenge and tweet a link using the #Python30 hashtag
- Help others, give feedback and share your knowledge
- Keep learning

CHAPTER 2

Getting Started

Some basic features of Python let us work functionally, these characteristics are found in other programming languages. However, other features that in my knowledge are only in Python, allows us to give a better performance to the code.

2.1 Day 01 - *args and **kwargs

When defining functions, we can use of two kinds of arguments, ***positional arguments*** and ***named arguments or keyword arguments***.

`Keyword arguments` are normally used to declare arguments with a default value, which are then used in case we don't pass it in when we make the call.

2.1.1 Positional Arguments

```
def multiply(a, b):
    print(a * b)

multiply(5, 4)

# Output: 20
```

2.1.2 Keyword Arguments

```python
def keyword_multiply(a=2, b=9):
    print(a * b)

keyword_multiply()
keyword_multiply(5, 4)
keyword_multiply(b=4, a=4)
keyword_multiply(b=2)

# Output: 18
# Output: 20
# Output: 16
# Output: 4
```

Before explaining what are the `*args and **kwargs` we must emphasize that these variable names are only a **convention** and we can change their names when we need to, the important thing in this case are the * (asterisks)

2.1.3 Use of *args

We use *args for positional arguments, i.e. those that do not have a default value in the definition of the function

```python
def fun(a, b, c):
    print(a, b, c)

fun(1, 2, 3)
```

```python
def fun(*args):
    print(args)

fun(1, 2, 3)

# Output: (1, 2, 3)
```

When using *args, all positional arguments are treated as a **tuple**.

2.1.4 Use of **kwargs

On the other hand we use the **kwargs for those arguments that have a default value in the definition of the function.

```python
def fun(a=0, b=0, c=0):
    print(a, b, c)

fun(a=1, b=2, c=3)
```

```python
def fun(**kwargs):
    print(kwargs)

fun(a=1, b=2, c=3)

# Output: {"a": 1,"b": 2,"c": 3}
```

When we use double asterisk (**), the keyword arguments are treated as a **dict**.

This functionality allows us to pass an indefinite list of arguments, either positional or named, and treat them as 1 or 2 variables in the function, this is because just as we can combine both types of arguments when defining a function, we can make use of both variables in the same function.

Note that when working with both types of arguments, we must first define the positional arguments and then those that are named.

```python
def fun(*args, **kwargs):
    print(args)
    print(kwargs)

fun(1, 2, a=3, b=6)

# Output:
# (1, 2)
# {"a": 3, "b": 6}
```

2.1.5 Use of *args and **kwargs when calling a function

In the same way that we can pass multiple arguments and receive them in a single variable, we can make use of the asterisks (*) to do the process in the opposite way, in other words, we have a variable (tuple, list) with `n` values and pass it to a function with multiple arguments.

```python
def func(a, b, c):
    print(f"a = {a}")
    print(f"b = {b}")
    print(f"c = {c}")
```

Using a single asterisk

```
numbers = [5, 4, 2]
func(*numbers)

# Output:
# a = 5
# b = 4
# c = 2
```

Using double asterisk

```
numbers = {"c": 4, "a": 6, "b": 9}
func(**numbers)

# Output:
# a = 6
# b = 9
# c = 4
```

In this case we could use the double asterisk although the function has no arguments with default values, since positional arguments can be treated in both ways.

2.1.6 Python Example

If you see the Python documentation, the print function makes use of *args, in its case and as I indicated at the beginning, it is not necessary that it is called this way.

`print(*objects, sep=' ', end='\n', file=sys.stdout, flush=False)`

We can print multiple values by passing them to the function each separated by a comma or even using the *args

```
print("Hello", "World")

values = ["Hello", "World"]
print(*values)

# Output:
# Hello World
# Hello World
```

Go to the Challenge

Go to the Solution

2.2 Day 02 - Range

Range is a built-in funtion that generates a sequence of numbers from the given values.

The syntax is as follows:

```
range([start,] stop [, step])
```

As you can see, the function receives up to 3 parameters, however only one is required

- `start`: (Optional) The initial value in the sequence, by default is 0
- `stop`: (required) End of the sequence, without including its value.
- `step`: (optional) The size of steps in the sequence the default value is 1

2.2.1 Let's Try

We'll print out what the function **range** returns if we pass it number 5. Remember that it has a parameter that is required (stop).

So, In this case the sequence will start at 0 and end at 5 (not including it).

`start` and `step` will take their default values.

```
print(range(5))
```

We'll see that when printing it returns something similar to **range(0, 5)**, but what we want is to see the complete sequence, so to see the values that compose it, for this we must cast this range as a list.

```
sequence = list(range(5))
print(sequence)
```

The output is now more understandable:

```
[0, 1, 2, 3, 4]
```

Two parameters

When we call **range** passing two parameters, with the first value we define where the sequence starts and with the second one where it ends.

Remember that the value where it ends is not included in the sequence

```
print(list(range(2, 9)))

# Output:
# [2, 3, 4, 5, 6, 7, 8]
```

Three parameters

Finally, when calling the function with all its arguments, we will define where it starts, where it ends, and how many steps the sequence will be.

```
print(list(range(-5, 5, 2)))

# Output:
# [-5, -3, -1, 1, 3]
```

You can see that the sequence starts at -5 and ends at 5, but where's number 4?

This is due to the steps we are indicating, which are 2, this way it will omit 1 value before returning the next number.

2.2.2 Notes

- All values must be integers
- Values can be positive or negative

Try this

```
print(list(range(5, -5, -1)))
```

2.2.3 How can I use float numbers with range?

The answer, you can't… if you need to create a sequence of numbers with floating values, you could take a look to numpy.arange

Go to the Challenge

Go to the Solution

2.3 Day 03 - List Slices

Before explaining how list slices work we must emphasize a couple of things.

1. Lists or arrays in other programming languages always start at index 0, i.e. if we have a list [a, b, c] the equivalent indexes are [0, 1, 2].
2. To move forward a little faster in this challenge you should know that the same rules apply as when working with the function range.

```
my_list[start:stop:step]
```

- `start`: (required) The initial value in the slice.
- `stop`: (required) End of the slice, without including its value.
- `step`: (optional) The size of steps in the sequence the default value is 1

As we can see it is very similar to the function range, however there are some differences that we will detail below.

- Although I indicate that **start** and **stop** are required we can omit them and they will be taken from the index that begins and the index that ends, however we can define only one (either one), but this will be explained better with some examples.
- It cannot take negative values since as it works on the indexes of a list, these are only positive integers (0…). Stop can take negative values… ?

2.3.1 Examples

```
my_list = ["H", "E", "L", "L", "O"]
print(my_list[:])
```

If you haven't tried, what do you think is the output?

Going back to what I said before "start and stop are required", actually they are not, but as we are talking about slices, taking a part of the list, we must define the colon (:), this way we indicate that we are going to take from the first index (since before the colon no value was defined) to the last index (since no value was defined after the colon), this way we are generating a new list with this range that would be exactly the same.

Now how about we try to generate a list from the second index onwards? We don't need to define where it ends but where it begins.

```
my_list = ["H", "E", "L", "L", "O"]
print(my_list[2:])
```

Let's remember what the indices would be like:

```
["H", "E", "L", "L", "O"]
[ 0 ,  1 ,  2 ,  3 ,  4 ]
```

As we are generating a slice from position 2 onwards we will get a list like the following

```
["L", "L", "O"]
```

Now what we need is to obtain a similar list but up to the second last index, How do we do it? the initial index cannot contain negative indexes but the index where it ends does, indicating how many indexes it will exclude from the end.

So…

```
my_list = ["H", "E", "L", "L", "O"]
print(my_list[:-1])
```

It will take from the beginning to the second last element in the list:

```
["H", "E", "L", "L"]
```

2.3.2 Step

The step works in the same way as in the function range, its default value is 1, but we can modify it if we want to go faster or in reverse.

How do we create a new list but increase it from 2 items?

```
my_list = ["H", "E", "L", "L", "O"]
print(my_list[::2])

# Output:
# ['H', 'L', 'O']
```

Remember that if we want to take from the beginning to the end of the list we don't need to define any value but it must be blank

And What if we want the same thing but in reverse?

```
my_list = ["H", "E", "L", "L", "O"]
print(my_list[::-1])

# Output:
# ['O', 'L', 'L', 'E', 'H']
```

You can practice defining different values, combining them, changing the length of the list and its order.

Do you like it? Why don't you try:

```
hello = "Hello World"
print(hello[::-1])
```

You can use this slices not only with lists

Go to the Challenge

Go to the Solution

2.4 Day 04 - Map

Python comes with some higher order functions which can be very useful when working in a functional way.

In this first case we'll look at the `map` function. The map function receives two arguments in the following order:

- A function
- At least a sequence (list, tuple, iterator, etc)

2.4.1 How does it work?

Each of the elements of the sequence will be delivered to the function, this process is done 1 by 1 and what it will do depends on what we define in the function.

At the end we will obtain a new sequence which will contain each of the results of the calls to the function.

```
def multiply(element):
    return element * 2

numbers = [2, 4, 6, 8]
print(map(multiply, numbers))
```

In this example each number in the list will be given to the function and this number will be multiplied by 2, we return its result which at the end we will have a similar iterator as follows:

```
<map object at 0x7f98f2b52150>
```

How can we see all the results?

Just like when we work with range to be able to see the numbers, we have to cast the result as a list.

```
def multiply(element):
    return element * 2

```

(continues on next page)

(continued from previous page)

```
numbers = [2, 4, 6, 8]
print(list(map(multiply, numbers)))
```

We can see the same list of numbers but after we've gone through the function. We can do this same process for any sequence of dict, strings, floats, etc

```
users = [
    {
        "name": "Esteban",
        "age": 26
    },
    {
        "name": "Jose",
        "age": 23
    },
    {
        "name": "Jaime",
        "age": 40
    }
]

def get_age(user):
    return user["age"]

print(list(map(get_age, users)))

# Output:
# [26, 23, 40]
```

2.4.2 Multiple Sequences

We can make use of the function map by passing multiple sequences, each of the sequences will give a value to the function at the same time

```
def user(name, age, country):
    return {
        "name": name,
        "age": age,
        "country": country
    }


names = ["Esteban", "Jose", "Jaime"]
ages = [26, 23, 40]
```

(continues on next page)

(continued from previous page)

```
countries = ["Colombia", "España", "Argentina"]

print(list(map(user, names, ages, countries)))

# Output
# [
#   {'name': 'Esteban', 'age': 26, 'country': 'Colombia'},
#   {'name': 'Jose', 'age': 23, 'country': 'España'},
#   {'name': 'Jaime', 'age': 40, 'country': 'Argentina'}
# ]
```

The way the map function works depends on whether we need to perform the same action for each of the elements in a sequence.

Go to the Challenge

Go to the Solution

2.5 Day 05 - Filter

Another higher order function is `filter`, the function filter allows as its name indicates to filter a sequence and return only the values that fulfill a condition.

Like the Map function, it receives 2 arguments which are:

- A function.
- A sequence that is going to be filtered.

2.5.1 How does it work?

Each of the elements in the sequence will be delivered to the function, if the function returns **True** the value that is delivered will be kept, otherwise it will be discarded.

```
def keep_hello(element):
    return element == "hello"

strings = ["don't", "say", "hello"]
print(list(filter(keep_hello, strings)))
```

Each of the strings will be validated in the **keep_hello** function, in this case if the string is equal to **hello**, the value will be kept. The final list will be:

```
["hello"]
```

In the following example we will see how to validate which users are of legal age, remember that if they are not, we will discard them and keep only those who fulfill the condition.

```
users = [
    {
        "name": "Esteban",
        "age": 26
    },
    {
        "name": "Jose",
        "age": 15
    },
    {
        "name": "Jaime",
        "age": 18
    }
]

def validate_age(user):
    return user["age"] >= 18

print(list(filter(validate_age, users)))

# Output:
# [
#   {'name': 'Esteban', 'age': 26},
#   {'name': 'Jaime', 'age': 18}
# ]
```

Go to the Challenge

Go to the Solution

2.6 Day 06 - Reduce

This is the last higher order function we'll see and like the previous ones it gets one function and one sequence.

2.6.1 Differences from the others:

- The function takes two elements of the sequence at once.

- It doesn't return a list, it returns a single value.
- Must be imported.

  `from functools import reduce`

2.6.2 How does it work?

Suppose we have the following list:

```
[2, 6, 9, 1]
```

The **reduce** function takes 2 values and performs an action, the result of this continues as the first parameter in the next execution and the same action is performed.

For this case we will add the values:

```
from functools import reduce

def add(num_a, num_b):
    return num_a + num_b

numbers = [5, 9, 4, 1]
print(reduce(add, numbers))

# Output:
# 19
```

2.6.3 Steps

First of all

- `num_a`: 5
- `num_b`: 9
- returned value: 14

Then

- `num_a`: 14 (previous returned)
- `num_b`: 4 (next in sequence)
- returned value: 18

Finally

- `num_a`: 18

- num_b: 1
- Final returned value: 19

It's not the best way to do it, but it's a simple way to understand it.

Let's join the following list and return a string:

```
["My", "name", "is", "Esteban"]
```

```
from functools import reduce

def join_string(word_a, word_b):
    string = f"{word_a} {word_b}"
    return string

words = ["My", "name", "is", "Esteban"]
print(reduce(join_string, words))

# Output:
# My name is Esteban
```

Go to the Challenge

Go to the Solution

2.7 Day 07 - Lambdas

Lambda functions are also known as anonymous functions, they can, as a normal function, take different arguments and return a single value, but lambdas are one-line functions.

The syntax is as follows:

```
lambda arguments: expresion
```

2.7.1 Example

```
def normal_multiply(num_a, num_b):
    return num_a * num_b

lambda_multiply = lambda num_a, num_b: num_a * num_b

print(normal_multiply(5, 2))
```

(continues on next page)

(continued from previous page)

```
print(lambda_multiply(5, 2))

# Output:
# 10
# 10
```

As you can see both `normal_multiply` and `lambda_multiply` functions, another difference you can notice is that **lambda** functions return a value without explicitly using the `return` keyword.

2.7.2 Why use Lambda functions?

We can create functions that perform a simple actions or operations using **lambda functions**, so that we can make our code more compact and simple.

Let's try using the lambda functions with one of the previous examples.

Filter example

```
def multiple(number):
    return number % 2 == 0

numbers = range(1, 101)
filter_numbers = filter(multiple, numbers)
print(list(filter_numbers))
```

Using lambda function

```
numbers = range(1, 101)
filter_numbers = filter(lambda number: number % 2 == 0, numbers)
print(list(filter_numbers))
```

When you understand the syntax of these functions, you will see that they are really easy to use and can be very useful.

Go to the Challenge

Go to the Solution

2.8 Day 08 - Comprehensions

The Python Comprehensions allow us in a clear and simple way, create sequences from other sequences, these work in a similar way to the **filter** and **map** functions.

Types of compressions:

- list comprehensions
- dictionary comprehensions
- set comprehensions

2.8.1 List Comprehensions

We can create lists in a clear way, the list comprehension structure is:

```
new_list = [output for i in sequence if condition]
```

Almost all the comprehensions work in a similar way depending on the type of structure we need to obtain.

- `new_list`: The variable which will contain the list.
- `output`: The output of every element in the list.
- `i`: Value in the sequence.
- `condition`: (optional) Comprehensions can have conditions to decide which value it will keep.

Common Example

```
totals = []
for number in range(1, 6):
    totals.append(number * 2)

print(totals)
```

We can perform this same action with the **map** function as we saw before, however this time we will use comprehension:

```
totals = [number * 2 for number in range(1, 6)]
print(totals)

# Output:
# [2, 4, 6, 8, 10]
```

It works exactly the same, now how about filtering out just odd numbers?

Remember that we can add a conditional, if it is satisfied, the value will be kept in the new sequence

```
totals = [number * 2 for number in range(1, 6) if number % 2 != 0]
print(totals)

# Output:
# [2, 6, 10] -> 1, 3, 5 (odd numbers)
```

2.8.2 Dict Comprehensions

They work in a similar way, we just have to take into account, the type of braces that are used and the format of the dictionaries `key: value`

```
new_dict = {output_key: output_value for i in sequence if condition}
```

We will use a list of numbers and create a dictionary in which its key will be the real number and its value will be the number multiplied by itself.

```
new_dict = {}
numbers = [5, 8, 2, 6, 3]
for number in numbers:
    new_dict[number] = number * number

print(new_dict)

# Output:
# {5: 25, 8: 64, 2: 4, 6: 36, 3: 9}
```

Comprehension

```
numbers = [5, 8, 2, 6, 3]
new_dict = {number: number * number for number in numbers}
print(new_dict)

# Output:
# {5: 25, 8: 64, 2: 4, 6: 36, 3: 9}
```

In this example we can also use conditionals, the same way we can work with other types of data such as sets or generators.

Go to the Challenge

Go to the Solution

2.9 Day 09 - Enumerate

Enumerate is a built-in Python function to which we can provide an iterator and return each of its elements in a tuple with a counter.

The syntax is as follows:

```
enumerate(sequence, start=0)
```

The default start value is 0 since, as in other programming languages, the indexes start at 0.

```
languages = ["Go", "Python", "Java"]
print(list(enumerate(languages)))

# Output:
# [(0, 'Go'), (1, 'Python'), (2, 'Java')]
```

The enumerate function returns an iterator, to be able to visualize it we must parse it or iterate it.

```
languages = ["Go", "Python", "Java"]
for counter, element in enumerate(languages):
    print(counter, element)

# Output:
# 0 Go
# 1 Python
# 2 Java
```

The optional start argument can be manipulated with the argument called `start`.

```
languages = ["Go", "Python", "Java"]
for counter, element in enumerate(languages, start=1):
    print(counter, element)

# Output:
# 1 Go
# 2 Python
# 3 Java
```

Go to the Challenge

Go to the Solution

2.10 Day 10 - Error Handling

Python provides us with a very useful structure that allows us to handle the errors that can be caused in our projects.

`try/except` allows us to capture errors and manipulate them without stopping the workflow.

The `try` block contains the code that could cause an error and we're going to capture it in `except`.

```
try:
  int("Hello")
except:
  print("An exception occurred")
```

The code above is intended to capture any error generated within the try structure, this one is because we can't cast as a number the string "Hello".

2.10.1 Catching specific exceptions

Although in the previous code we were able to capture the error, it is not a good practice to do so since we have no knowledge of what is wrong.

If our code generates any exceptions of which we are not aware, it could affect us instead of helping us. Python comes with some exceptions, however each library can have its own exceptions as well as we can create our own.

```
int("Hello")

# Output:
# Traceback (most recent call last):
# File "<stdin>", line 1, in <module>
# ValueError: invalid literal for int() with base 10: 'Hello'
```

If we run only the casting we'll see the error that python generates, specifically **ValueError**, this will be the error that we'll capture.

```
try:
  int("Hello")
except ValueError as err:
  print("An exception occurred:", err)

# Output: An exception occurred: invalid literal for int() with base 10: 'Hello'
```

2.10.2 Multiple Exceptions

We could capture more than one exception to perform different actions, we can leave them separately to manipulate them separately or in the same exception for the same process.

```python
try:
  int("Hello")
except (ValueError, TypeError) as err:
  # Actions to take if the code generates ValueError or TypeError
  pass
except ZeroDivisionError:
  # ZeroDivisionError captures another action
```

2.10.3 Raising Exceptions

These errors that occur in our code were executed at some point, so we can do it ourselves using the `raise` keyword.

```python
try:
  number = "Hello"
  if not number.isnumeric():
    raise ValueError(f"{number} is not numeric")
except ValueError as err:
  print(err)

# Output:
# Hello is not numeric
```

2.10.4 Else

Usually when we hear about the `else` clause we relate it to conditionals, however in python we can use it for different structures, one of them is this.

The `else` clause allows us to execute the code if no exception has been generated.

```python
try:
  number = int("12")
except ValueError as err:
  print(err)
else:
  print("No exceptions occurred")
  print("Number:", number)

# Output:
```

(continues on next page)

(continued from previous page)

```
# No exceptions occurred
# Number: 12
```

2.10.5 Finally

Another clause we can use with try/except is `finally`, it allows us to execute a block of code whether or not an exception is generated. **It will be executed at the end of the process** and after the else clause in case we have one.

```
try:
  number = int("Hello")
except ValueError as err:
  print(err)
else:
  print("No exceptions occurred")
finally:
  print("This text will be printed whether or not an exception is generated")

# Output:
# invalid literal for int() with base 10: 'Hello'
# This text will be printed whether or not an exception is generated
```

Go to the Challenge

Go to the Solution

CHAPTER 3

Object-Oriented Programming

Object-oriented programming is an important part of any programming language and Python is no exception, however, we won't see how to create a class or an instance, instead, we'll talk about how to make object-oriented programming to another level.

3.1 Day 11 - Classmethod and Staticmethod

We haven't talked about decorators yet, however this time we will use two which are provided by Python, these are `staticmethod` and `classmethod`.

In order to use the decorators we must know that these are put over the definition of the method or function with an `@` as a prefix.

The methods are a kind of function which are defined in a class and can be of three types:

- *Instance Methods*
- *Class Methods*
- *Static Methods*

3.1.1 Instance Methods

Instance methods are the ones we usually create when working with classes, these methods must have an argument that is always the first one and as a convention, it is called **self**, these methods are part of the instance we create.

```python
class MyClass:
    def instance_method(self):
        print("Instance Method Called")

instance = MyClass()
instance.instance_method()

# Output:
# Instance Method Called
```

We defined a method called `instance_method` which as we indicated must have `self` as the first parameter which refers to the instance created and we can use it to read or create attributes or use other methods in this instance of the class.

3.1.2 Classmethod

In order to define a class method, we must use the `classmethod` decorator. When working with instance methods, the method must have an argument which is self, in this case, we will not define `self` because we don't use anything related to the instance, however, we must use an argument which by convention is `cls`.

```python
class MyClass:
    attribute = "hello, world"

    @classmethod
    def class_method(cls):
        """ Class Method """
        print(cls.attribute)

MyClass.class_method()

# Output:
# hello, world
```

cls refers to the class itself and the attributes defined in it.

3.1.3 Staticmethod

Static methods are defined the same as the class methods but this time the decorator `staticmethod` is used. Unlike the other methods, these do not receive any mandatory argument (besides that our logic requires).

So these methods can not access attributes or other methods of the class or instance.

```
class MyClass:
    def __init__(self):
        self.static_method(50)

    @staticmethod
    def static_method(number):
        print("Static Method Called")
        print(number * number)

MyClass.static_method(30)

# Output:
# Static Method Called
# 900

instance = MyClass()

# Output:
# Static Method Called
# 2500
```

Static methods can be called from both instance and class methods.

3.1.4 What's it for?

We may never have used them, but we probably needed them.

The static methods are useful to separate the logic that doesn't have to make use of the instance or the class, and it's also good practice to separate this kind of logic.

Let's see an example using both decorators:

```
class FileManager:
    def __init__(self, json_content):
        self.content = json_content
        print(json_content)

    @classmethod
    def from_csv(cls, filename):
        file_content = cls.read_csv_content(filename)
        return cls(file_content)

    @staticmethod
    def read_csv_content(filename):
        # Read file
        # Convert to json
```

(continues on next page)

(continued from previous page)

```
        return content

instance_from_json = FileManager(json_content)
instance_from_csv = FileManager.from_csv("path/to/the/file")
```

Static method are used for features that do not require the instance or the class, the class method allows us to create a new instance of a different format than the one the class receives when it is instantiated which is json.

Go to the Challenge

Go to the Solution

3.2 Day 12 - Property

In object-oriented programming, the getter and setter methods help us access private attributes of a class, allowing the code to be well encapsulated and to avoid accessing them directly.

Although in Python we can't define methods or attributes as private, we can use a convention that indicates to other developers the level of accessibility of these.

- Protected: For protected attributes or methods, we must set an underscore before the definition. e.g. `self._name = "My name"`
- Private: Private attributes or methods have two underscores. e.g. `def __private_method(self):`

3.2.1 Getter / Setter

```
class User:
    def __init__(self, name, age):
        self.__name = name
        self.__age = age

    def get_name(self):
        return self.__name

    def set_name(self, name):
        self.__name = name

user = User("Esteban", 26)
print(f"Name: {user.get_name()}")
```

(continues on next page)

(continued from previous page)

```
user.set_name("Cristian")
print(f"Name: {user.get_name()}")

# Output:
# Name: Esteban
# Name: Cristian
```

The traditional way or as it is normally used in other programming languages is to use normal methods.

This preamble is intended to give you an understanding of each of these levels and to help you understand how **Property** can help you create getters and setters in a Pythonic way.

3.2.2 Property

`Property` is a built-in decorator that allows us to define the getter and setter.

Let's see with the same code as above how to use the property decorator:

```
class User:
    def __init__(self, name, age):
        self.__name = name
        self.__age = age

    @property
    def name(self):
        print("Getter called")
        return self.__name

    @name.setter
    def name(self, name):
        print("Setter called")
        self.__name = name

user = User("Esteban", 26)
print(f"Name: {user.name}")
user.name = "Cristian"
print(f"Name: {user.name}")

# Output:
# Getter called
# Name: Esteban
# Setter called
```

(continues on next page)

(continued from previous page)

```
# Getter called
# Name: Cristian
```

As you can see, the `property` decorator did not change much in the way we do the getter/setter, however this allows it to be more intuitive and easy to read, also like the methods defined above, allow us to control how we access the attributes of the class.

The property decorator is defined in a method that will return the value of the attribute, to define the setter we must use the property created, `@<property>.setter`.

We don't have to define each of these methods for every property in our classes, as we only do this for those we're going to access, or if they can be read-only attributes we'll only need the getters.

Remember that we can implement these methods as we need them so that we won't affect the implementation of the code.

```python
class User:
    def __init__(self, name, age):
        self.__name = name
        self.__age = age

    @property
    def age(self):
        return self.__age

    @age.setter
    def age(self, age):
        if not isinstance(age, int) or age < 0:
            raise ValueError("Invalid age")

        self.__age = age

user = User("Esteban", 26)
user.age = "26"

# Output:
# Traceback (most recent call last):
#   File "<stdin>", line 1, in <module>
#   File "<stdin>", line 13, in age
# ValueError: Invalid age
```

Go to the Challenge

Go to the Solution

3.3 Day 13 - Dunder Methods

“Everything in Python is an object.” You’ve probably heard this before, that’s because it is, every data type we use is an object that has been enriched to increase its performance.

Dunder methods, or magic methods, are methods we can define in our classes to give them more functionality than they have. These methods allow us to simulate the behavior of different data types from those we can find in the language.

The name dunder methods come from double underscores as a prefix and suffix of the method, e.g. `__init__`.

```python
class User:
    def __init__(self, name):
        self.__name = name

new_user = User("Christopher")
print(new_user)

# Output:
# <__main__.User object at 0x7fafdc06cdd0>
```

I’ve created a basic class `User` which receives the name at the moment of its instantiation. When printing this instance, we can see the address in memory although now this is not what we need, how about improving this?

3.3.1 Object Representation

`__repr__` is a method that allows us to define what information is going to represent our class, in this case, to see clear information of the created instance.

```python
class User:
    def __init__(self, name):
        self.__name = name

    def __repr__(self):
        return f"Class User: {self.__name}"

new_user = User("Christopher")
print(new_user)

# Output:
# Class User: Christopher
```

3.3.2 Count

You have probably used the `len()` function more than once, but did you know we can also implement this in our objects?

```python
class User:
    def __init__(self, name):
        self.__name = name

    def __repr__(self):
        return f"Class User: {self.__name}"

    def __len__(self):
        return len(self.__name)

new_user = User("Christopher")
print(new_user)
print(len(new_user))

# Output:
# Class User: Christopher
# 11
```

The `len()` function allows us to know the number of characters in a string, however, we can use the magic method `__len__` to know from our object the number of characters that our `User` represents.

3.3.3 More

There are quite a few methods that we probably won't need, however, we can practice with each of them to enrich our code like never before. Dunder Methods

Go to the Challenge

Go to the Solution

3.4 Day 14 - Operator Overloading

Every day we write multiple lines of code, comparing values or performing calculations similar to this one:

```python
print(15 > 20)
print(36 - 23)

```

(continues on next page)

(continued from previous page)

```
# Output:
# False
# 13
```

Usually, we think about comparing or making numerical calculations since it is something basic that we know since we were children, but this type of comparison can be performed with any class using some magic methods.

For now we will create a `Product` class with some of its basic attributes:

```
class Product:
    def __init__(self, name, price):
        self.__name = name
        self.__price = price

    def __repr__(self):
        return f"Product: {self.__name}"

pizza = Product("pizza", 5)
hotdog = Product("hotdog", 1)

print(pizza)
print(hotdog)

# Output:
# Product: pizza
# Product: hotdog
```

Two instances of the Product class were created, what about trying to compare them.

```
print(pizza > hotdog)
print(pizza < hotdog)
print(pizza == hotdog)

# Output:
# Traceback (most recent call last):
#   File "<stdin>", line 1, in <module>
# TypeError: '>' not supported between instances of 'Product' and 'Product'
```

At this point it is not possible to compare these instances, in fact it is not known what we are comparing, to fix this we will add some magic methods.

```
class Product:
    def __init__(self, name, price):
        self.__name = name
        self.__price = price
```

(continues on next page)

(continued from previous page)

```
    def __repr__(self):
        return f"Product: {self.__name}"

    def __lt__(self, other_instance):
        return self.__price < other_instance.__price

    def __gt__(self, other_instance):
        return self.__price > other_instance.__price

    def __eq__(self, other_instance):
        return self.__price == other_instance.__price

pizza = Product("pizza", 5)
hotdog = Product("hotdog", 1)
```

Let's try again to make the comparisons.

```
print(pizza > hotdog)
print(pizza < hotdog)
print(pizza == hotdog)

# Output:
# True
# False
# False
```

These magic methods allow us to compare our items, in my case I had their prices compared, however, you can define these methods as you need them.

These methods receive as an argument another one, which is related to the value with which we are going to make the comparison.

3.4.1 Operations

It is possible to perform operations between different objects, what about knowing the total of an order? why not add up each of these products and get the total.

More info

```
class Product:
    def __init__(self, name, price):
        self.__name = name
        self.__price = price
```

(continues on next page)

(continued from previous page)

```
    def __repr__(self):
        return f"Product: {self.__name}"

    def __add__(self, other_instance):
        return self.__price + other_instance.__price

pizza = Product("pizza", 5)
hotdog = Product("hotdog", 1)

total = pizza + hotdog
print(total)

# Output:
# 6
```

Go to the Challenge

Go to the Solution

CHAPTER 4

Advanced Concepts

Different features are not always explained, when I started with Python I used some of them without really knowing how they worked, that's why part of this section will focus on some of these that are widely used and we should understand how they work and how can we create them.

4.1 Day 15 - Else

As you saw in ***Error Handling***, we can use the `else` clause not only with conditionals, in this case we'll see that we can use this clause with other statements.

4.1.1 For/Else

Before we explain how we can include the else, we'll create a loop like the one below:

```
numbers = range(10, 21)
for number in numbers:
    print(number)
```

The previous loop will print the numbers from 10 to 20.

If you are familiar with loops, you will know that the **break** sentence will interrupt the execution and continue outside of it.

```python
numbers = range(10, 21)
for number in numbers:
    if number == 17:
        break
    print(number)
```

The previous loop will print the numbers from 10 to 16, when the number 17 continues, the conditional will validate it and the loop will be interrupted, so the execution will continue.

However, there are cases when we are going through some kind of data we will realize that the loop ended without finding a **break**, in these cases is when we can make use of the `else` clause.

```python
numbers = range(10, 21)
for number in numbers:
    if number == 60:
        break
else:
    print("Number 60 was not found")
```

The else clause is executed when a loop is not interrupted, and will only execute the block of code when the loop is over.

4.1.2 While/Else

The while statement also allows the use of the `else` clause, and like the for loop, it will only be executed if no break is found in the while block.

```python
while conditional:
    # Do some stuff with possibilities of break
else:
    # Do some stuff if no break executed
```

Go to the Challenge

Go to the Solution

4.2 Day 16 - Recursion

Recursive functions are created when a function calls itself, this process will repeat indefinitely if we don't stop it at some point, and only when it concludes, it will return a value to the place where it started the process.

We can turn any loop into a recursion.

```
total = 0
for num in range(6):
    total += num

print(f"The sum of the numbers from 1 to 5 is equals to {total}")

# Output
# The sum of the numbers from 1 to 5 is equals to 15
```

We can convert this loop to a recursive function in the following way:

```
def recursive_function(num):
    if num == 1:
        return num
    return num + recursive_function(num - 1)

times = 5
total = recursive_function(times)

print(f"The sum of the numbers from 1 to {times} is equals to {total}")

# Output
# The sum of the numbers from 1 to 5 is equals to 15
```

The above function is a basic example of what we can do, it starts by receiving the number 5 as an argument.

The first condition is used to end the recursion once the value of the argument is 1 and it will go in a chain by returning the total and adding it to the previous number. In other words:

```
# First execution
>>> total = recursive_function(5)
>>> return 5 + recursive_function(4)
    # Second execution
    >>> return 4 + recursive_function(3)
        # Third execution
        >>> return 3 + recursive_function(2)
            # Fourth execution
            >>> return 2 + recursive_function(1)
                # Fifth execution
                >>> return 1
            >>> return 2 + 1 # 1 of the fifth execution
        >>> return 3 + 3 # 3 of the fourth execution
    >>> return 4 + 6 # 6 of the third execution
```

(continues on next page)

(continued from previous page)

```
>>> return 5 + 10 # 10 of the second execution

# The sum of the numbers from 1 to 5 is equals to 15
```

When we call the function for the first time, a value is passed to it and it calls itself but passing a lower value and so on until the end (when the value is 1 and therefore it stops making more calls). At that point they start to return. The returned values are added to the original parameter in each one of the iterations until arriving at the beginning of everything in which the first call returns the final value.

Go to the Challenge

Go to the Solution

4.3 Day 17 - Memoization

Memoization is a caching technique which stores the result of complex operations to be used in case the same call is made again.

In this way we can optimize not only the execution time but also the use of memory of our logic based on previous processes.

```
def fibonacci(n):
    if n == 0:
        return 0
    elif n == 1:
        return 1
    return fibonacci(n - 1) + fibonacci(n - 2)
```

This function is a bit complex depending on the number we send to calculate the fibonacci, to see how long the execution could take we will use **timeit**.

```
from timeit import timeit

def fibonacci(n):
    if n == 0:
        return 0
    elif n == 1:
        return 1
    return fibonacci(n - 1) + fibonacci(n - 2)

timeit("fibonacci(35)", setup="from __main__ import fibonacci", number=1)

# Output
# 3.4363581580000755
```

This process was performed once and took about 4 seconds, if you can see it is a recursive function which is called itself several times per execution and sometimes performs the same operation for the same arguments.

This process can be optimized because we will always get the same result if we pass the same argument, in other words, it is a **pure function**.

```python
from timeit import timeit

history = {}

def fibonacci(n):
    if n in history:
        return history[n]

    if n == 0:
        return 0
    elif n == 1:
        return 1

    fibonacci_result = fibonacci(n - 1) + fibonacci(n - 2)
    history[n] = fibonacci_result

    return fibonacci_result

timeit("fibonacci(35)", setup="from __main__ import fibonacci", number=1)

# Output
# 0.00014547999853675721
```

The execution time and resources are quite low thanks to a history, as you can see, when we know the fibonacci of a number x this is stored and before calculating it the only thing we have to do is check if we have calculated it before so we don't have to do it again.

Go to the Challenge

4.4 Day 18 - Decorators

In Python, ***decorators*** are very useful and powerful once we understand how they work and how we can create them.

Following the good practices of Python, we can use the decorators to have a cleaner and easier to understand code, without having to duplicate the code in different parts to express the same thing.

4.4.1 What are decorators?

The most common definition is the following:

> Decorators are functions that take another function and extends the behavior of the latter function without explicitly modifying it.

Decorators are **high order functions**, they allow us to add functionality to an existing code without affecting its behavior, each decorator must be preceded by a `@`.

```python
def decorator(fun, arg_a, arg_b):
    total = fun(arg_a, arg_b)
    print(f"The result of {arg_a} and {arg_b} is {total}")

def add(num_a, num_b):
    return num_a + num_b

decorator(add, 6, 2)

# Output:
# The result of 6 and 2 is 8
```

In this example, the "decorator" takes the function and the arguments as parameters and performs a new operation, different from that of the main function.

This same process as a decorator would be:

```python
def decorator(fun):
    def wrapper(arg_a, arg_b):
        total = fun(arg_a, arg_b)
        print(f"The result of {arg_a} and {arg_b} is {total}")

    return wrapper

@decorator
def add(num_a, num_b):
    return num_a + num_b

add(6, 2)

# Output
# The result of 6 and 2 is 8
```

4.4.2 Types of decorators

- *Functions with arguments*.

- *Decorators with arguments*.

Functions with arguments

The structure of decorators is made by nesting functions and returning each one of them depending on the values we receive (function/arguments). When a function to be decorated receives arguments, the decorator changes and a new function is nested which receives the arguments that the main function receives.

In order to make this clear, let's look at the following example:

```
def operation(function):
    print("Function as an argument")
    def wrapper(*args, **kwargs):
        print("All function parameters as arguments (positiona & named)")
        return function(*args, **kwargs)

    return wrapper

@operation
def add(num_a, num_b):
    print("Decorated function")
    return num_a + num_b

print(add(2, 4))

# Output
# Function as an argument
# All function parameters as arguments (positiona & named)
# Decorated function
# 6
```

The decorator keeps returning a function, in this case the wrapper function that is in charge of receiving as parameters each one of the arguments that were given to the `add` function and internally the execution of the main function is done.

Decorators with arguments

Decorators can also receive arguments to define their behavior, as mentioned above, nested functions must be created for each behavior, in this case as they are decorators' arguments, they go at the beginning of the decorator's definition.

```
def operation(print_result=False):
    def _operation(function):
        # Create wrapper to use arguments
```

(continues on next page)

(continued from previous page)

```
    return _operation

@operation(True)
def add(num_a, num_b):
    pass
```

The way we assign a decorator to a function changes, and we can send arguments to our needs, in the previous case a variable to indicate whether or not we want to print the result before returning to the place where the function was called.

- Decorators can also be created using **classes**.
- You can use multiple Decorators in the same function.

In python it is very important to know about decorators, the majority of libraries use them and as you saw before `@classmethod` and `@staticmethod` are decorators that python provides us to change the behavior of the methods of their classes.

Go to the Challenge

Go to the Solution

4.5 Day 19 - Generators

Before explaining what they are and how we can create our own generators, let's understand what iterators are.

An iterator is an object that allows to go through one by one the elements in a data structure, the iterators have a `next()` function which when called, returns the next element in the sequence, when there are no more elements, it throws an exception (StopIteration) which generates that the iteration stops.

Generators are basically iterators, the difference is that their elements are not stored in memory so we can only iterate over them once. The generators can be created using functions, however these functions do not use the keyword `return`, instead we use the keyword `yield`.

```
def generator():
    counter = 0
    while counter < 10:
        yield counter
        counter += 1

my_gen = generator()
```

(continues on next page)

(continued from previous page)

```
print(next(my_gen))
print(next(my_gen))
print(next(my_gen))
print(next(my_gen))

# Output
# 0
# 1
# 2
# 3
```

Initially we have a function which has a loop, however as you can see it has the keyword `yield`, which allows in each iteration to deliver a value and wait at that point until it is called again.

In other words, we have a counter with a value of 0 at the beginning, when we request an element from the iterator with the `next` function, the generator will reach the **yield** and deliver a value, it will increase the variable by one and return to the position of the **yield** waiting for the next call.

> Once the function yields, the function is paused and the control is transferred to the caller.

Finally, when the iterator ends, in this case when the counter reaches 10, an exception will be launched which will indicate that the iteration is over.

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

We can use `for` loops to iterate over each of its elements, this will print each of the elements and once it is done, instead of launching the exception, it will finish the iteration:

```
for num in generator():
    print(num)
```

4.5.1 send

The generators have a `send()` method which allows us to send values to the generators to the point where they are waiting to continue, this in turn returns the next element of the iteration:

```python
def generator():
    counter = 0
    while counter < 10:
        new_value = yield counter
        if new_value:
            counter = new_value
        else:
            counter += 1

my_gen = generator()

print(next(my_gen))
print(my_gen.send(7))
print(next(my_gen))
print(next(my_gen))

# Output
# 0
# 7
# 8
# 9
```

In this case we must validate whether or not we have received a value, since in each iteration he receives a null value or `None`.

4.5.2 throw

Also they have a `throw` method which is useful when we need to pass an exception to the generator, we must take into account that the generator will not stop if we can correctly validate it, and like the method *send*, it will return the next value in iteration.

```python
def generator():
    counter = 0
    while counter < 10:
        try:
            yield counter
        except ValueError:
            print("Raised Exception")
            counter += 2
        else:
            counter += 1

my_gen = generator()

print(next(my_gen))
```

(continues on next page)

(continued from previous page)

```
print(my_gen.throw(ValueError))
print(next(my_gen))

# Output
# 0
# Raised Exception
# 2
# 3
```

Generators allow us as their name suggests to **generate** data at runtime, that's why they are very useful when we work with a larger volume of data.

Go to the Challenge

Go to the Solution

4.6 Day 20 - Context Managers

When working with any type of resource in any programming language, we must release them once we no longer use them, this must be done because leaving resources open and not making use of them can slow down the system or even cause it to fail.

Context managers are a Python feature that allows us to manage these resources in a simple way leaving aside the need to always assign and release resources manually.

Let's suppose that we need to make some operation with the handling of plain files, so we must open a file, read its content and close it once we are done with it:

```
file = open('path/to/the/file')
content = file.read()
file.close()
```

This resource must be released after used, in this case it is simply to close the file. However this process of opening and closing files is not the best and for some reason we could forget to add it.

When using context managers, we use the keyword `with`, the previous example would look like this:

```
with open('path/to/the/file') as file:
    content = file.read()
```

It starts by opening a file for processing, read the content, and close it before leaving this block.

4.6.1 Implementation

- *Context Managers with classes*
- *Context Managers with functions*

Context Managers with classes

In order to create context managers using classes, we will be using the ***Dunder Methods***, to be more precise the `__enter__` and `__exit__`.

```
class ContextManager:
    def __init__(self):
        pass

    def __enter__(self):
        pass

    def __exit__(self, type, value, traceback):
        pass
```

- `__init__`: Creates the object.
- `__enter__`: Return the resource we are going to use.
- `__exit__`: Release the resource when we finish using the context manager instance.

This last one, as you could see, receives three arguments that in case an exception occurs, it will pass the **type**, **value** and **traceback** of the exception so we can decide how to proceed at the moment of releasing the resource.

```
class OpenFile:
    def __init__(self, filename):
        self.file = open(filename)

    def __enter__(self):
        return self.file

    def __exit__(self, type, value, traceback):
        self.file.close()
```

In this way we can try to simulate the behaviour when opening files, once we have defined our methods, we can make use of the `with` keyword.

```
with OpenFile("path/to/the/file") as file:
    content = file.read()
```

Context Managers with functions

We can also implement our context managers using functions, however to create them we must use ***Decorators*** and ***Generators***.

```python
from contextlib import contextmanager

@contextmanager
def open_file(filename):
    file = open(filename)

    try:
        yield file
    except:
        # Exceptions that may occur
    finally:
        file.close()
```

We use the decorator `@contextmanager` and the keyword `yield` to convert a simple function into a context manager, and in the same way here we can capture the exceptions that occur and manage them according to our needs.

The use of this context manager is the same as the one we did with classes, the only thing that would change is the name of the class/function we are using.

Go to the Challenge

Go to the Solution

CHAPTER 5

Built-in Modules

Some of our needs may have already been solved by someone else. The topics seen here should include some of the most common modules that are integrated into the language.

5.1 Day 21 - Datetime

Python lets us manipulate dates by using the `datetime` module that comes in the standard Python library, so we don't have to install it. When we work with dates we must understand that what we really modify are datetime objects and not strings.

There are four main objects that we use to handle dates:

- *date*
- *time*
- *datetime*
- *timedelta*

Before explaining each of these concepts, it is important that you understand that `datetime` as a module includes a class called the same way (`datetime`).

```
import datetime

datetime.date
datetime.time
```

(continues on next page)

(continued from previous page)

```
datetime.datetime
datetime.timedelta
```

5.1.1 date

The `date` class allows us to work only with date (year, month, day). In order to create an instance of the object we use the date class in the following way:

```
from datetime import date

birthday = date(1993, 12, 29)
print(birthday)

# Output
# 1993-12-29
```

Although we see the given date, what we have assigned is a `datetime.date` type object.

```
print(type(birthday))
```

today

The date class has a method that allows us to obtain the current day in this format.

```
from datetime import date

today = date.today()
print(today)
```

5.1.2 time

The time class in the same way allows us to create times from given data, however these values are not mandatory and by default the time obtained is 00:00:00.

In case of indicating a specific time, the values must go in the following order in case of not using `keyword arguments`:

```
time(hour, minute, second)
```

5.1.3 datetime

`datetime` contains the information of both objects mentioned above `date` and `time`, and the values must be added in the same order (year, month, day, hours, minutes, seconds), and like the `time` class, the last 3 values are not mandatory.

```
from datetime import datetime

my_date = datetime(2020, 8, 5, 15, 29)
print(my_date)

# Output
# 2020-08-05 15:29:00
```

now

Just like with the `date` class, datetime lets us get the exact date using the `now` method.

```
from datetime import datetime

now = datetime.now()
print(now)
```

5.1.4 Attributes

Each of the classes explained above has attributes that we can use to obtain certain information in a specific way.

```
from datetime import datetime

now = datetime.now()

print(f"Day: {now.day}")
print(f"Month: {now.month}")
print(f"Year: {now.year}")
print(f"Hour: {now.hour}")
print(f"Minute: {now.minute}")
print(f"Second: {now.second}")
```

These attributes can be used in the classes mentioned above to which they are related, i.e. we cannot get the year from a `time` type object.

5.1.5 Methods

Some of the methods we can use are:

- `weekday`: Weekday starting Monday as 0 and ending Sunday as 6.

```
from datetime import date

now = date(2020, 8, 5)
print(now.weekday())

# Output
# 2 (Miercoles)
```

- `isoweekday`: Day of the week starting on Monday as 1 and ending on Sunday as 7.

```
from datetime import date

now = date(2020, 8, 5)
print(now.isoweekday())

# Output
# 3 (Miercoles)
```

- `strftime`: Used to represent as a string objects `date`, `time` and `datetime`.

```
from datetime import datetime

now = datetime.now()
str_date = now.strftime("%b %dth, %Y %H:%M")

print(str_date)
print(type(str_date))

# Output
# Aug 05th, 2020 17:59
# <class 'str'>
```

There are a series of **format codes** that you should use to indicate in which format you want to represent your date.

- `%b`: Month as locale's abbreviated name. (Aug)
- `%d`: Day of the month as a zero-padded decimal number. (05)
- `%Y`: Year with century as a decimal number. (2020)
- `%H`: Hour (24-hour clock) as a zero-padded decimal number. (17)
- `%M`: Minute as a zero-padded decimal number. (59)

See the Format Codes you can use.

5.1.6 timedelta

We use the timedelta class when we want to get the difference between two dates or times. All the classes of the module `datetime` use some `magical methods` that allow us to make operations and comparisons between dates.

Some of the parameters that we can provide are:

- days
- seconds
- minutes
- hours

```
from datetime import timedelta

three_days = timedelta(days=3)
print(three_days)

# Output
# 3 days, 0:00:00
```

The timedelta object is representing a duration of 3 days, using this we can add or subtract this time to a given date.

```
from datetime import timedelta

now = datetime.now()
three_days = timedelta(days=3)

now_plus_three_days = now + three_days

print(now)
print(now_plus_three_days)

# Output
# 2020-08-05 18:40:06.691392
# 2020-08-08 18:40:06.691392
```

In the same way we can do these operations between different datetime objects:

```
now = date.today()
birthday = date(1993, 12, 29)
```

(continues on next page)

(continued from previous page)

```
delta_difference = now - birthday

print(delta_difference.days)

# Output
# 9716
```

5.1.7 strptime

This method of the `datetime` class allows us to obtain a date as a string according to a defined format, the format must be represented using the **Format Codes** mentioned above.

In case the date does not correspond with the format, an exception will be launched.

```
from datetime import datetime

str_date = '2020-12-01 10:30PM'
format = '%Y-%m-%d %H:%M%p'

obj_date = datetime.strptime(str_date, format)
print(obj_date)
print(type(obj_date))

# Output
# 2020-12-01 10:30:00
# <class 'datetime.datetime'>
```

ValueError

```
str_date = '2020-12-01'
format = '%Y/%m/%d'

obj_date = datetime.strptime(str_date, format)

# Output
# ValueError: time data '2020-12-01' does not match format '%Y/%m/%d'
```

5.1.8 Conclusion

This module has some limitations, such as

- Date difference every N years. (timedelta does not accept years or months)
- Obtain a datetime instance of a string without knowing its format.

These and other features that `datetime` may not have you can check out at a library called dateutil.

> It is important to emphasize that what is mentioned here is not all we can do with the module, but it is an introduction and what I think are some of the main functions that we should know how to use.

Go to the Challenge

Go to the Solution

5.2 Day 22 - Regex

Coming soon

5.3 Day 23 - Loggers

Coming soon

5.4 Day 24 - Collections

Coming soon

5.5 Day 25 - Itertools

Coming soon

5.6 Day 26 - CSV

Coming soon

5.7 Day 27 - Json

Coming soon

5.8 Day 28 - LRU Cache

Coming soon

5.9 Day 29 - Multiprocessing

Coming soon

5.10 Day 30 - Doctest

Coming soon