Bootcamp
Python

Day01
Basics 2

Bootcamp Python

Day01 - Basics 2

The goal of the day is to get familiar with object-oriented programming and much more.

Notions of the day

Objects, cast, class, inheritance, built-in functions, magic methods, generator, constructor, iterator, ...

General rules

- The version of Python to use is 3.7, you can check the version of Python with the following command: `python -V`
- The norm: during this bootcamp you will follow the PEP 8 standards. You can install pycodestyle which is a tool to check your Python code.
- The function eval is never allowed.
- The exercises are ordered from the easiest to the hardest.
- Your exercises are going to be evaluated by someone else, so make sure that your variable names and function names are appropriate and civil.
- Your manual is the internet.
- You can also ask questions in the dedicated channel in the 42 AI Slack: 42-ai.slack.com.
- If you find any issue or mistakes in the subject please create an issue on our dedicated repository on Github.

Helper

Ensure that you have the right Python version.

```
> which python
/goinfre/miniconda/bin/python
> python -V
Python 3.7.*
> which pip
/goinfre/miniconda/bin/pip
```

Exercise 00 - The Book

Exercise 01 - Family tree

Exercise 02 - The Vector

Exercise 03 - The Matrix

Exercise 04 - Generator!

Exercise 05 - Working with lists

Exercise 06 - Bank account

Exercise 00 - The Book

Turn-in directory:	ex00
Files to turn in:	book.py, recipe.py, test.py
Forbidden functions:	None
Remarks:	n/a

You will provide a `test.py` file to test your classes and prove that they are working the right way.
You can import all the classes into your test.py file by adding these lines at the top of the `test.py` file:

```
from book import Book
from recipe import Recipe
```

You will have to make a class `Book` and a class `Recipe`

Let's describe the `Recipe` class. It has some attributes:

- `name` (str)
- `cooking_lvl` (int) : range 1 to 5
- `cooking_time` (int) : in minutes (no negative numbers)
- `ingredients` (list) : list of all ingredients each represented by a string
- `description` (str) : description of the recipe
- `recipe_type` (str) : can be "starter", "lunch" or "dessert".

You have to initialize the object `Recipe` and check all its values, only the description can be empty.
In case of input errors, you should print what they are and exit properly.

You will have to implement the built-in method `__str__`.
It's the method called when you execute this code:

```
tourte = Recipe(...)
to_print = str(tourte)
print(to_print)
```

It's implemented this way:

```
def __str__(self):
    """Return the string to print with the recipe info"""
    txt = ""
    """Your code goes here"""
    return txt
```

The `Book` class also has some attributes:

- `name` (str)
- `last_update` (datetime)
- `creation_date` (datetime)
- `recipes_list` (dict) : a dictionnary why 3 keys: "starter", "lunch", "dessert".

You will have to implement some methods in `Book`:

```
def get_recipe_by_name(self, name):
    """Print a recipe with the name `name` and return the instance"""
    pass

def get_recipes_by_types(self, recipe_type):
    """Get all recipe names for a given recipe_type """
    pass
```

```
def add_recipe(self, recipe):
    """Add a recipe to the book and update last_update"""
    pass
```

You will have to handle the error if the arg passed in add_recipe is not a `Recipe`.

Exercise 01 - Family tree

Turn-in directory:	ex01
Files to turn in:	game.py
Forbidden functions:	None
Remarks:	n/a

You will have to make a class and its children.

Create a `GotCharacter` class and initialize it with the following attributes:

- `first_name`
- `is_alive` (by default is `True`)

Pick up a GoT House (e.g., Stark, Lannister...). Create a child class that inherits from `GotCharacter` and define the following attributes:

- `family_name` (by default should be the same as the Class)
- `house_words` (e.g., the House words for the Stark House is: "Winter is Coming")

Example:

```
class Stark(GotCharacter):
    def __init__(self, first_name=None, is_alive=True):
        super().__init__(first_name=first_name, is_alive=is_alive)
        self.family_name = "Stark"
        self.house_words = "Winter is Coming"
```

Add two methods to your child class:

- `print_house_words`: prints to screen the House words
- `die`: changes the value of `is_alive` to `False`

Running commands in the Python console, an example of what you should get:

```
> python
>>> from game import GotCharacter, Stark
>>> arya = Stark("Arya")
>>> print(arya.__dict__)
{'first_name': 'Arya', 'is_alive': True, 'family_name': 'Stark', 'house_words': 'Winter is
↪  Coming'}
>>> arya.print_house_words()
Winter is Coming
>>> print(arya.is_alive)
True
>>> arya.die()
>>> print(arya.is_alive)
False
```

You can add any attribute or method you need to your class and format the docstring the way you want to. Feel free to create other children of `GotCharacter`.

```
>>> print(arya.__doc__)
A class representing the Stark family. Or when bad things happen to good people.
```

Exercise 02 - The Vector

Turn-in directory:	ex02
Files to turn in:	vector.py, test.py
Forbidden functions:	None
Forbidden libraries:	NumPy
Remarks:	n/a

You will provide a testing file to prove that your class works as expected.
You will have to create a helpful class, with more options and providing enhanced ease of use for the user.

In this exercise, you have to create a `Vector` class. The goal is to have vectors and be able to perform mathematical operations with them.

```
>> v1 = Vector([0.0, 1.0, 2.0, 3.0])
>> v2 = v1 * 5
>> print(v2)
(Vector [0.0, 5.0, 10.0, 15.0])
```

It has 2 attributes:

- `values` : a list of floats
- `size` : the size of the vector -> `Vector([0.0, 1.0, 2.0, 3.0]).size == 4`

You should be able to initialize the object with either:

- a list of floats: `Vector([0.0, 1.0, 2.0, 3.0])` -> then the size of the vector will be `4`
- a size: `Vector(3)` -> the vector will be created with default values starting from `0.0`: `[0.0, 1.0, 2.0]`
- a range (min, max): `Vector((10,16))` -> the vector will be created with values in the given range: `[10.0, 11.0, 12.0, 13.0, 14.0, 15.0]`

You will implement all the following built-in functions (called 'magic methods') for your `Vector` class:

```
__add__
__radd__
# add : vectors and scalars, can have errors with vectors.
__sub__
__rsub__
# sub : vectors and scalars, can have errors with vectors.
__truediv__
__rtruediv__
# div : scalars only.
__mul__
__rmul__
# mul : vectors and scalars, can have errors with vectors.
# two vectors can be multiplied using the Dot product, return a scalar.
__str__
__repr__
```

Vector - Scalar authorized operations are:

- Addition between one vector (m * 1) and one scalar (1 * 1)

$$x + a = \begin{bmatrix} x_1 \\ \vdots \\ x_m \end{bmatrix} + a = \begin{bmatrix} x_1 + a \\ \vdots \\ x_m + a \end{bmatrix}$$

- Substraction between one vector (m * 1) and one scalar (1 * 1)

$$x - a = \begin{bmatrix} x_1 \\ \vdots \\ x_m \end{bmatrix} - a = \begin{bmatrix} x_1 - a \\ \vdots \\ x_m - a \end{bmatrix}$$

- Multiplication and division between one vector (m * 1) and one scalar (1 * 1)

$$x \cdot a = \begin{bmatrix} x_1 \\ \vdots \\ x_m \end{bmatrix} \cdot a = \begin{bmatrix} x_1 \cdot a \\ \vdots \\ x_m \cdot a \end{bmatrix}$$

Vector - Vector authorized operations are:

- Addition between two vectors of same dimension (m * 1)

$$x + y = \begin{bmatrix} x_1 \\ \vdots \\ x_m \end{bmatrix} + \begin{bmatrix} y_1 \\ \vdots \\ y_m \end{bmatrix} = \begin{bmatrix} x_1 + y_1 \\ \vdots \\ x_m + y_m \end{bmatrix}$$

- Substraction between two vectors of same dimension (m * 1)

$$x - y = \begin{bmatrix} x_1 \\ \vdots \\ x_m \end{bmatrix} - \begin{bmatrix} y_1 \\ \vdots \\ y_m \end{bmatrix} = \begin{bmatrix} x_1 - y_1 \\ \vdots \\ x_m - y_m \end{bmatrix}$$

- Compute the dot product between two vectors of same dimensons (m * 1)

$$x \cdot y = \begin{bmatrix} x_1 \\ \vdots \\ x_m \end{bmatrix} \cdot \begin{bmatrix} y_1 \\ \vdots \\ y_m \end{bmatrix} = \sum_{i=1}^{m} x_i \cdot y_i = x_1 \cdot y_1 + \cdots + x_m \cdot y_m$$

Don't forget to handle all kind of errors properly!

Exercise 03 - The Matrix

Turn-in directory:	ex03
Files to turn in:	matrix.py, test.py
Forbidden functions:	None
Forbidden libraries:	NumPy
Remarks:	n/a

You will provide a testing file to prove that your class works as expected.
You will have to create a helpful class, with more options and providing enhanced ease of use for the user.

In this exercise, you have to create a `Matrix` class. The goal is to have matrices and be able to perform both matrix-matrix operation and matrix-vector operations with them.

```
>> m1 = Matrix([[0.0, 1.0, 2.0, 3.0],
                [0.0, 2.0, 4.0, 6.0]])

>> m2 = Matrix([[0.0, 1.0],
                [2.0, 3.0],
                [4.0, 5.0],
                [6.0, 7.0]])
>> print(m1 * m2)
(Matrix [[28., 34.], [56., 68.]])
```

It has 2 attributes:

- `data` : list of lists -> the elements stored in the matrix
- `shape` : by shape we mean the dimensions of the matrix as a tuple (rows, columns) -> `Matrix([[0.0, 1.0], [2.0, 3.0], [4.0, 5.0]]).shape == (3, 2)`

You should be able to initialize the object with either:

- the elements of the matrix as a list of lists: `Matrix([[0.0, 1.0, 2.0, 3.0], [4.0, 5.0, 6.0, 7.0]])` -> the dimensions of this matrix are then (2, 4)
- a shape (rows, columns): `Matrix((3, 3))` -> by default the matrix will be filled with zeroes
- the expected elements and shape: `Matrix([[0.0, 1.0, 2.0], [3.0, 4.0, 5.0], [6.0, 7.0, 8.0]], (3, 3))`

You will implement all the following built-in functions (called 'magic methods') for your `Matrix` class:

```
    __add__
    __radd__
    # add : vectors and matrices, can have errors with vectors and matrices.
    __sub__
    __rsub__
    # sub : vectors and matrices, can have errors with vectors and matrices.
    __truediv__
    __rtruediv__
    # div : only scalars.
    __mul__
    __rmul__
    # mul : scalars, vectors and matrices , can have errors with vectors and matrices.
    # if we perform Matrix * Vector (dot product), return a Vector.
    __str__
    __repr__
```

Matrix - Vector authorized operations are:

- Multiplication between a (m * n) matrix and a (n * 1) vector

$$X \cdot y = \begin{bmatrix} x_1^{(1)} & \dots & x_n^{(1)} \\ \vdots & \ddots & \vdots \\ x_1^{(m)} & \dots & x_n^{(m)} \end{bmatrix} \cdot \begin{bmatrix} y_1 \\ \vdots \\ y_n \end{bmatrix} = \begin{bmatrix} x^{(1)} \cdot y \\ \vdots \\ x^{(m)} \cdot y \end{bmatrix}$$

In other words:

$$X \cdot y = \begin{bmatrix} \sum_{i=1}^{n} x_i^{(1)} \cdot y_i \\ \vdots \\ \sum_{i=1}^{n} x_i^{(m)} \cdot y_i \end{bmatrix}$$

Matrix - Matrix authorized operations are:

- Addition between two matrices of same dimension (m * n)

$$X + Y = \begin{bmatrix} x_1^{(1)} & \dots & x_n^{(1)} \\ \vdots & \ddots & \vdots \\ x_1^{(m)} & \dots & x_n^{(m)} \end{bmatrix} + \begin{bmatrix} y_1^{(1)} & \dots & y_n^{(1)} \\ \vdots & \ddots & \vdots \\ y_1^{(m)} & \dots & y_n^{(m)} \end{bmatrix} = \begin{bmatrix} x_1^{(1)} + y_1^{(1)} & \dots & x_n^{(1)} + y_n^{(1)} \\ \vdots & \ddots & \vdots \\ x_1^{(m)} + y_1^{(m)} & \dots & x_n^{(m)} + y_n^{(m)} \end{bmatrix}$$

- Substraction between two matrices of same dimension (m * n)

$$X - Y = \begin{bmatrix} x_1^{(1)} & \dots & x_n^{(1)} \\ \vdots & \ddots & \vdots \\ x_1^{(m)} & \dots & x_n^{(m)} \end{bmatrix} - \begin{bmatrix} y_1^{(1)} & \dots & y_n^{(1)} \\ \vdots & \ddots & \vdots \\ y_1^{(m)} & \dots & y_n^{(m)} \end{bmatrix} = \begin{bmatrix} x_1^{(1)} - y_1^{(1)} & \dots & x_n^{(1)} - y_n^{(1)} \\ \vdots & \ddots & \vdots \\ x_1^{(m)} - y_1^{(m)} & \dots & x_n^{(m)} - y_n^{(m)} \end{bmatrix}$$

- Multiplication or division between one matrix (m * n) and one scalar (1 * 1)

$$Xa = \begin{bmatrix} x_1^{(1)} & \dots & x_n^{(1)} \\ \vdots & \ddots & \vdots \\ x_1^{(m)} & \dots & x_n^{(m)} \end{bmatrix} \cdot a = \begin{bmatrix} x_1^{(1)}a & \dots & x_n^{(1)}a \\ \vdots & \ddots & \vdots \\ x_1^{(m)}a & \dots & x_n^{(m)}a \end{bmatrix}$$

- Mutiplication between two matrices of compatible dimension: (m * n) and (n * p)

$$XY = \begin{bmatrix} x_1^{(1)} & \dots & x_n^{(1)} \\ \vdots & \ddots & \vdots \\ x_1^{(m)} & \dots & x_n^{(m)} \end{bmatrix} \begin{bmatrix} y_1^{(1)} & \dots & y_p^{(1)} \\ \vdots & \ddots & \vdots \\ y_1^{(n)} & \dots & y_p^{(n)} \end{bmatrix} = \begin{bmatrix} x^{(1)} \cdot y_1 & \dots & x^{(1)} \cdot y_p \\ \vdots & \ddots & \vdots \\ x^{(m)} \cdot y_1 & \dots & x^{(m)} \cdot y_p \end{bmatrix}$$

In other words:

$$X \cdot Y = \begin{bmatrix} \sum_{i=1}^{n} x_i^{(1)} \cdot y_1^{(i)} & \dots & \sum_{i=1}^{n} x_i^{(1)} \cdot y_p^{(i)} \\ \vdots & \ddots & \vdots \\ \sum_{i=1}^{n} x_i^{(m)} \cdot y_1^{(i)} & \dots & \sum_{i=1}^{n} x_i^{(m)} \cdot y_p^{(i)} \end{bmatrix}$$

Don't forget to handle all kind of errors properly!

Exercise 04 - Generator!

Turn-in directory:	ex04
Files to turn in:	generator.py
Forbidden functions:	random.shuffle
Authorized functions:	random.randint
Remarks:	n/a

Code a function called `generator` that takes a text as input, uses the string `sep` as a splitting parameter, and `yield`s the resulting substrings.

The function can take an optional argument.
The options are:

- `shuffle`: shuffles the list of words.
- `unique`: returns a list where each word appears only once.
- `ordered`: alphabetically sorts the words.

```
# function prototype
def generator(text, sep=" ", option=None):
    '''Option is an optional arg, sep is mandatory'''
```

You can only call one option at a time.

Example:

```
>> text = "Le Lorem Ipsum est simplement du faux texte."
>> for word in generator(text, sep=" "):
...     print(word)
...
Le
Lorem
Ipsum
est
simplement
du
faux
texte.
>> for word in generator(text, sep=" ", option="shuffle"):
...     print(word)
...
simplement
texte.
est
faux
Le
Lorem
Ipsum
du
>> for word in generator(text, sep=" ", option="ordered"):
...     print(word)
...
Ipsum
Le
Lorem
du
est
faux
```

```
simplement
texte.
```

```
>> text = "Lorem Ipsum Lorem Ipsum"
>> for word in generator(text, sep=" ", option="unique"):
...     print(word)
...
Lorem
Ipsum
```

The function should return “ERROR” one time if the `text` argument is not a string, or if the `option` argument is not valid.

```
>> text = 1.0
>> for word in generator(text, sep="."):
...     print(word)
...
ERROR
```

Exercise 05 - Working with lists

Turn-in directory:	ex05
Files to turn in:	eval.py
Forbidden functions	while
Remarks:	use zip & enumerate

Code a class `Evaluator`, that has two static functions named `zip_evaluate` and `enumerate_evaluate`.

The goal of these 2 functions is to compute the sum of the lengths of every `words` of a given list weighted by a list a `coefs`.

The lists `coefs` and `words` have to be the same length. If this is not the case, the function should return -1.

You have to obtain the desired result using `zip` in the `zip_evaluate` function, and with `enumerate` in the `enumerate_evaluate` function.

Example:

```
>> from eval import Evaluator
>>
>> words = ["Le", "Lorem", "Ipsum", "est", "simple"]
>> coefs = [1.0, 2.0, 1.0, 4.0, 0.5]
>> Evaluator.zip_evaluate(coefs, words)
32.0
>> words = ["Le", "Lorem", "Ipsum", "n'", "est", "pas", "simple"]
>> coefs = [0.0, -1.0, 1.0, -12.0, 0.0, 42.42]
>> Evaluator.enumerate_evaluate(coefs, words)
-1
```

Exercise 06 - Bank Account

Turn-in directory:	ex06
Files to turn in:	the_bank.py
Forbidden functions:	None
Remarks:	n/a

It's all about security.
Have a look at the class named `Account` in the snippet of code below.

```
# in the_bank.py
class Account(object):

    ID_COUNT = 1

    def __init__(self, name, **kwargs):
        self.id = self.ID_COUNT
        self.name = name
        self.__dict__.update(kwargs)
        if hasattr(self, 'value'):
            self.value = 0
        Account.ID_COUNT += 1

    def transfer(self, amount):
        self.value += amount
```

Now, it is your turn to code a class named `Bank`!
Its purpose will be to handle the security part of each transfer attempt.

Security means checking if the `Account` is:

- the right object
- not corrupted
- and stores enough money to complete the transfer.

How do we define if a bank account is corrupted? A corrupted bank account has:

- an even number of attributes
- an attribute starting with `b`
- no attribute starting with `zip` or `addr`
- no attribute `name`, `id` and `value`

A transaction is invalid if `amount < 0` or if the amount is larger than the available funds of the sending account.

```
# in the_bank.py
class Bank(object):
    """The bank"""
    def __init__(self):
        self.account = []

    def add(self, account):
        self.account.append(account)

    def transfer(self, origin, dest, amount):
        """
            @origin:  int(id) or str(name) of the first account
            @dest:    int(id) or str(name) of the destination account
            @amount:  float(amount) amount to transfer
```

```
            @return         True if success, False if an error occured
        """

    def fix_account(self, account):
        """
            fix the corrupted account
            @account: int(id) or str(name) of the account
            @return         True if success, False if an error occured
        """
```

Check out the `dir` function.

WARNING: YOU WILL HAVE TO MODIFY THE INSTANCES' ATTRIBUTES IN ORDER TO FIX THEM.