D02 - Python-Django Training

Python Basics 2

Summary: Today, you're going to conquer Silicon Valley with your newly acquired skills in POO with Python!

Contents

Chapter I

Preamble

Here are the lyric of the "Free Software Song":

Join us now and share the software;
You'll be free, hackers, you'll be free.
Join us now and share the software;
You'll be free, hackers, you'll be free.

Hoarders can get piles of money,
That is true, hackers, that is true.
But they cannot help their neighbors;
That's not good, hackers, that's not good.

When we have enough free software
At our call, hackers, at our call,
We'll kick out those dirty licenses
Ever more, hackers, ever more.

Join us now and share the software;
You'll be free, hackers, you'll be free.
Join us now and share the software;
You'll be free, hackers, you'll be free.

Chapter II

Ocaml piscine, general rules

- Every output goes to the standard output, and will be ended by a newline, unless specified otherwise.
- The imposed filenames must be followed to the letter, as well as class names, function names and method names, etc.
- Unless otherwise explicitly stated, the keywords `open`, `for` and `while` are forbidden. Their use will be flagged as cheating, no questions asked.
- Turn-in directories are `ex00/`, `ex01/`, . . . , `exn/`.
- You must read the examples thoroughly. They can contain requirements that are not obvious in the exercise's description.
- Since you are allowed to use the `OCaml` syntaxes you learned about since the beginning of the piscine, you are not allowed to use any additional syntaxes, modules and libraries unless explicitly stated otherwise.
- The exercices must be done in order. The graduation will stop at the first failed exercice. Yes, the old school way.
- Read each exercise FULLY before starting it! Really, do it.
- The compiler to use is `ocamlopt`. When you are required to turn in a function, you must also include anything necessary to compile a full executable. That executable should display some tests that prove that you've done the exercise correctly.
- Remember that the special token `";;"` is only used to end an expression in the interpreter. Thus, it must never appear in any file you turn in. Regardless, the interpreter is a powerfull ally, learn to use it at its best as soon as possible!
- The subject can be modified up to 4 hours before the final turn-in time.
- In case you're wondering, no coding style is enforced during the `OCaml` piscine. You can use any style you like, no restrictions. But remember that a code your peer-evaluator can't read is a code he or she can't grade. As usual, big functions are a weak style.
- You will NOT be graded by a program, unless explictly stated in the subject. Therefore, you are given a certain amount of freedom in how you choose to do the

exercises. However, some piscine day might explicitly cancel this rule, and you will have to respect directions and outputs perfectly.

- Only the requested files must be turned in and thus present on the repository during the peer-evaluation.
- Even if the subject of an exercise is short, it's worth spending some time on it to be absolutely sure you understand what's expected of you, and that you did it in the best possible way.
- By Odin, by Thor! Use your brain!!!

Chapter III

Today's specific rules

- No code in the global scope. We want functions!
- Unless otherwise specified, every file written in Python will have to end with a block

```
if __name__ == '__main__':
    # Your tests and your error handling
```

- Any uncaught exception will negate the work, even in case of an error you were required to test.
- Any import is prohibited, except for the ones specified in the 'Authorized functions' section of each exercise's header.

Chapter IV

Exercise 00

	Exercise 00
Exercise 00 : Conquering Silicon Valley!	
Turn-in directory : *ex*00/	
Files to turn in : `render.py, myCV.template, settings.py`	
Allowed functions : `import sys, os, re`	

You've just achieved your amazing developer training course and new perspectives are about to change your life for ever. You arrive in the Silicon Valley with one goal in mind: develop your revolutionary resume generator with a groundbreaking technology and become the Bill Gates of the job-seeking industry!

Now all that's left to do is...craft your technology.

Create a `render.py` program that will take a file with a `.template` extension as a parameter. This program will have to read the file's content, replace some patterns with defined values in a `settings.py` file (a `if __name__ == '__main__':` block will not be necessary for this file) and write the result in a file with the `.html` extension.

You *will have* to be able to replicate the following example with your program.

```
$> cat settings.py
name = "duoquadragintian"
$> cat file.template
<p>"-Who are you?
-A {name}!"</p>
$> python3 render.py file.template
$> cat file.html
<p>"-Who are you?
-A duoquadragintian!"</p>
```

Errors, including wrong file extension, a non-existing file or a wrong number of arguments will have to be managed.

You will have to turn-in a `myCV.template` file which, once converted in HTML, will have to include at least the complete structure of a (`doctype`, `head` and `body` page, the page's title, name and surname of the resume's owner, their age and profession. Of course, these informations will not directly appear in the `.template` file.

`help(globals)`, `keyword expansion...`

Chapter V

Exercise 01

	Exercise 01
Exercise 01; Innovating start-up looking for intern. 10 years exp. required.	
Turn-in directory : *ex*01/	
Files to turn in : `intern.py`	
Allowed functions :	

You cannot start this journey alone. You choose to recruit someone to ~~make coffee~~, assist you, an intern would be better (they're cheaper).

Create the `Intern` class containing the following functionalities:

A `builder` taking a character chain as a parameter, assigning its value to a `Name` attribute. `"My name? I'm nobody, an intern, I have no name."` will be implemented as default value.

A method `__str__()` that will return `Name` attribute of the instance.

A `Coffee` class with a simple `__str__()` method that will return the character chain `"This is the worst coffee you ever tasted."`.

A `work()` method that will raise only one exception (use the basic (Exception) type) with the text `"I'm just an intern, I can't do that..."`.

A `make_coffee()` method that will return an instance of the `Coffee` class that you will have implemented in `Intern` class.

In your tests, you will have to instantiate twice the `Intern` class. Once without a name, the second time with the name "Mark".

Display the name of each instance. Ask mark to make you a coffee and display the result. Ask the other intern to work. You **will have** to manage the exception in your test.

Chapter VI

Exercise 02

	Exercise 02
Exercise 02: 5 classes 1 cup.	
Turn-in directory : *ex*02/	
Files to turn in : `beverages.py`	
Allowed functions :	

Coffee's good. Choosing your drink is better! Create a `HotBeverage` class with the following functionalities:

A `price` attribute with a value of `0.30`.

Aname attribute with the `"hot beverage"` value.

A `description()` method returning an instance description. The description value will be `"Just some hot water in a cup."`.

A `__str__()` method returning an instance description in this form:

```
name : <name attribute>
price : <price attribute limited to two decimal points>
description : <instance's description>
```

for instance, a HotBeverage instance display would look like this:

```
name : hot beverage
price : 0.30
description : Just some hot water in a cup.
```

Then create the following derived classes `HotBeverage`:

`Coffee :`

`name` `: "coffee"`

`price` `: 0.40`

`description` `: "A coffee, to stay awake."`

```
Tea :

    name : "tea"
    price : 0.30
    description : "Just some hot water in a cup."

Chocolate :

    name : "chocolate"
    price : 0.50
    description : "Chocolate, sweet chocolate..."

Cappuccino :

    name : "cappuccino"
    price : 0.45
    description : "Un po' di Italia nella sua tazza!"
```

You must ONLY redefine what's necessary, what you need to change, to redefine... (cf. DRY).

In your tests, instantiate each class among: `HotBeverage`, `Coffee`, `Tea`, `Chocolate` and `Cappuccino` and display them.

Chapter VII

Exercise 03

Exercise 03
Exercise 03: Glorious coffee machine!
Turn-in directory : *ex*03/
Files to turn in : `machine.py, beverages.py`
Allowed functions : `import random`

There you are! Your company is up and running! Your first fund raising offered you premises. You have an intern for the coffee and a level 10 green plant at the building entrance to keep everything in order.

Yet, everything is not perfect: your intern makes a god awful coffee and half a minimum wage is expensive for this mud. Time has come to invest in new equipment that will lead to your personal success!

Create the `CoffeeMachine` class containing:

- A builder.
- An `EmptyCup` class inheriting `HotBeverage`, with the name `"empty cup"`, a `0.90` price and the description `"An empty cup?!  Gimme my money back!"`.

  Copy the `beverages.py` file frome the previous exercise in this exercise folder to use the classes it contains.
- A `BrokenMachineException` class inheriting the `Exception` with the text `"This coffee machine has to be repaired."`. This text must be defined in the exception's builder.
- A `repair()` method that fixes the machine so it can serve hot drinks again.
- A `serve()` method that will have the following specifications:

  Parameters : A unique parameter (other than `self`) that will be a class derived from `HotBeverage`.

Return: Alternatively (randomly), the method returns an instance of the class set in parameter and alternatively, an `EmptyCup` instance.

Obsolescence: The machine is cheap and breaks down after serving 10 drinks.

When out of order: the call for the `serve()` method must raise a `CoffeeMachine.BrokenMachi` type exception until the `repair()` method is called.

Fixing: After calling the `repair()` method, the `serve()` method can work again without raising the exception before it breaks down again after serving 10 drinks.

In your tests, instantiate the `CoffeeMachine` class. Ask for various drinks from the `beverages.py` file and display the drink you're served until the machine breaks down (you will then manage the raised exception). Fix the machine and start again until the machine breaks down again (manage the exception again).

Chapter VIII

Exercise 04

	Exercise 04
Exercise 04: A basic class ft. RMS.	
Turn-in directory : *ex*04/	
Files to turn in : `elem.py`	
Allowed functions :	

It's now time to work on your WEB visibility. You'd like to use your newly acquired Python skills to efficiently design your HTML content but you'd like to get the advice from a superior being to learn how to do so. You choose to sacrifice your intern for the Gods of programming.

Now that you have a coffee machine, he became pretty useless... Just torch him...

Saint IGNUcius appears before you to speak a precious revelation:

"HTML elements share almost the same structure (tag, content, attributes). It would be wise to create a class that can assemble all those shared behaviors and specifications to use the legacy force in Python to simply and easily derive this class without having to rewrite the whole thing"

Only then does St. IGNUcius see the Mac you're working on. Scared like hell, he flees without further ado, leaving behind just a tests file and an incomplete class. In a rush, you complete the `Elem` class (the gaps needing filling are indicated with *[...]*) with the following specifications:

- A builder taking in parameter the element's name, `HTML` attributes and type (simple or double tags).
- A `__str__()` method returning the element's `HTML` code.
- A `add_content()` method allowing to add the elements at the end of the content.
- An *Exception* sub-class within it.

If the work is done well, you will be able to represent any `HTML` element and its content with your `Elem` class. Now for the final stretch:

- The `tests.py` file provided in the tarbal in the appendix of the subject **must** work properly (no assertion error, the test output explicitly stating its success). Of course, we're not cruel enough to test the functionalities that are not explicitly required in this exercise. Hahaha... We're not... really.
- You also must replicate and display the following structure with the help of your `Elem` class:

```
<html>
  <head>
    <title>
      "Hello ground!"
    </title>
  </head>
  <body>
    <h1>
      "Oh no, not again!"
    </h1>
    <img src="http://i.imgur.com/pfp3T.jpg" />
  </body>
</html>
```

Chapter IX

Exercise 05

	Exercise 05
Exercise 05: Create your own elements!	
Turn-in directory : *ex*05/	
Files to turn in : `elem.py, elements.py`	
Allowed functions :	

Congratulations! Now, you can generate any HTML element and its content. However, generating each element specifying each attribute with each instantiation is a little boring. Here is the opportunity to use the legacy to create small classes that will be easier to use. Create the following classes derived from the `Elem` class created in the previous exercise:

- `html`, `head`, `body`
- `title`
- `meta`
- `img`
- `table`, `th`, `tr`, `td`
- `ul`, `ol`, `li`
- `h1`
- `h2`
- `p`
- `div`
- `span`
- `hr`
- `br`

Each class's builder will be able to take the content as the first argument, and:

```
print( Html( [Head(), Body()] ) )
```

will display:

```
<html>
  <head></head>
  <body></body>
</html>
```

Be smart, reuse the functionalities you've coded in the Elem. class. You **must** use the inheritance.

Demonstrate how these classes work with a number of tests - your choice - that will cover all the functionalities. After coding these classes, you won't need to specify the name or type of a tag, which is very handy. You will never have to instantiate directly `Elem` class again. Actually, from now on, this is prohibited.

In order to understand the benefits of the `d'Elem` derived classes compared to the direct use of `d'Elem`, let's take the `HTML` document structure from the previous exercise. You must replicate it using your new classes.

```
<html>
  <head>
    <title>
      "Hello ground!"
    </title>
  </head>
  <body>
    <h1>
      "Oh no, not again!"
    </h1>
    <img src="http://i.imgur.com/pfp3T.jpg" />
  </body>
</html>
```

Way simpler, isn't it? :)

Chapter X

Exercise 06

	Exercise 06
Exercise 06: Validation.	
Turn-in directory : *ex*06/	
Files to turn in : `Page.py, elem.py, elements.py`	
Allowed functions :	

Though you've made amazing progress, your work still needs a little cleaning. A little more conform. You're like this: you love constraints and challenges. So why not imposing a norm to the structure of your `HTML` documents? Start copying the classes of both previous exercises in this exercise's folder.

Create a `Page` class which builder will take in parameter an instance of a class inheriting `Elem`. Your `Page` class must implement a `is_valid()` method that must send `True` if all the following rules are observed, and otherwise, si `False`:

- If, on the tree path, a node has not one of the following types: `html`, `head`, `body`, `title`, `meta`, `img`, `table`, `th`, `tr`, `td` , `ul`, `ol`, `li`, `h1`, `h2`, `p`, `div`, `span`, `hr`, `br` or `Text`, the tree is invalid.
- `Html` must strictly contain a `Head`, **then** a `Body`.
- `Head` must only contain one `Title` and only one `Title`.
- `Body` and `Div` must only contain the following type of elements: `H1`, `H2`, `Div`, `Table`, `Ul`, `Ol`, `Span`, or `Text`.
- `Title`, `H1`, `H2`, `Li`, `Th`, `Td` must only contain one `Text` and only this `Text`.
- `P` ne doit contenir que des `Text`.
- `Span` must only contain `Text` or some `P`.
- `Ul` and `Ol`] must contain at least one `Li` and only some `Li`.

- `Tr` must contain at least one `Th` or `Td` and only some `Th` or `Td`. The `Th` and the `Td` must be mutually exclusive.
- `Table`: must only contain `Tr` and only some `Tr`.

Your `Page` class must also be able to:

- Display its `HTML` code when we `print` an instance. Beware: the `HTML` code displays must be preceded by a doctype if and if only the root element type is `Html`.
- Write your `HTML` code in a file thanks to a `write_to_file` method that takes the file's name as a parameter. Beware: `HTML` code written in the file must be preceded by a doctype if and if only the root element's type is `Html`.

Demonstrate how your `Page` class works with a number of tests - you will choose - that will cover all the functionalites.