Exercises related to Python CLI programming

- When you write Python programs, it is important that you configure your program editor so that it does not put tabulator characters into the source program file. When you do these exercises, the settings of you program editor should be such that it puts three space charcters into the program file when the tabulator key is pressed.
- Because program blocks are formed through indentation in Python, tabulator characters can be harmful if tabulation step is not correct. The best way to get rid from all problems related to the tabulator character, is to not use that character in your souce program file. This way your program looks the same regardless of the editor or browser that is used to view the program.

Kari Laitinen
http://www.naturalprogramming.com
2022-12-28 File created.

EXERCISES WITH PROGRAM GuessAWord.py

You can find a program named **GuessAWord.py** in the folder

`http://www.naturalprogramming.com/pythonprograms/pythonfilesextra/`

This program is a simple computer game in which the player has to try to guess the characters of a word that is 'known' by the game. Study the program and play the game in order to find out how the game has been programmed.

Exercise 1:

Improve the Guess-A-Word game so that the word to be guessed is randomly taken from a list of words. A list of words can be created with a statement such as

```
words_to_be_guessed = [ "VIENNA", "HELSINKI", "COPENHAGEN",
                        "LONDON", "BERLIN", "AMSTERDAM" ]
```

A random index for a list such as the one above can be created with the `random.random()` method in an expression like

```
int ( ( random.random() * len( words_to_be_guessed ) ) )
```

The `random.random()` method returns a `double` value in the range 0.0 .... 1.0 so that the value 1.0 is never returned. The above expression thus calculates a suitable random index. When `double` values are converted to `int` values, they are always rounded 'downwards' to the smaller integer value. To use the `random.random()` method, you must have a suitable `import` statement in your program. (For advice, see the **MathDemo.py** program.)

Exercise 2:

Now the program is such that it terminates when the game is finished. Modify the program so that the game can be played several times during a single run of the program. In the above-mentioned folder there is a program named **RepeatableGame.py** which should be a helpful example.

Exercise 3:

Improve the program so that it counts how many guesses the player makes during a game. After a game is played, and before a new game starts, the program should print how many guesses were made. The following variable could be usedful in this task

```
number_of_guesses = 0
```

Exercise 4:

Improve the program so that it prints game statistics before the program terminates. This means that the program shows which words were being guessed and how many guesses were made for each word. The game statistics could look like the following.

```
PLAYED WORD       GUESSES

COPENHAGEN          7
LONDON              6
COPENHAGEN          4
BERLIN              5
HELSINKI            4
```

As the ’played words’ will be randomly selected from an array, it is possible that the same word is played several times.

You can use the following kind of data items to store data of games:

```
games_played = 0
played_words = []
guesses_in_games = []
```

A variable can be used to count how many games are played, and you can use lists to store the played words and the number of guesses made. A new data item can be added to the end of a list with a method named `append()`. New data should be put to the lists after each game is played, and the data should be displayed on the screen in the end when the user no longer wants to play new games.

EXERCISES WITH PROGRAM Animals.py

Exercise 1:

Write a new method named make_stomach_empty() to class Animal in Animals.py. The stomach of an Animal object can be emptied by writing an empty string "" as the stomach contents. The new method could be called

```
cat_object.make_stomach_empty()
dog_object.make_stomach_empty()
```

To test this new method, you should use method make_speak().

Exercise 2:

Modify the constructor of class Animal so that its parameter gets a default value. After this modification an Animal object can be created without supplying any parameters, for example, in the following way:

```
default_animal  =  Animal()
```

The data field (instance attribute) species_name can be given the value "default animal".

By studying program Windows.py you can find out how constructor parameters can be given default values. Also in this exercise you should use method make_speak() to check that your modifications are correct.

Exercise 3:

The data members or data fields of Python objects are frequently called attributes or, more precisely, instance attributes. In Python classes, you do not need to declare the data fields. Instead, the data fields become existent when you write inside a method

```
self.data_field_name = ...
```

Add a new data field (instance attribute) to class Animal by writing the statement

```
self.animal_name = ...
```

to the constructor. Here the aim is that an Animal object can be created, for example, with the statement

```
named_cat  =  Animal( "cat", "Arnold" )
```

Copying of an object must be modified so that the new data field will be copied as well. Method make_speak() must be modified so that it produces an output that looks like

```
Hello, I am a cat named Arnold.
I have eaten: ...
```

The new data field can be given the default value "nameless".

Exercise 4:

Modify method make_speak() so that it prints

```
Hello, I am a ... named ...
My stomach is empty.
```

in that case when stomach_contents references an empty string. The stomach of an Animal object is empty as long as method feed() has not been called. You can use standard function len() to check whether the stomach is empty. Function len() can be called, for example, in the following way.

```
if len( self.stomach_contents )  ==  0 :

   # stomach_contents references an empty string.
   ...
```

If the stomach is not empty, the program should produce the original output.

Exercise 5:

Modify method feed() so that it will be possible to feed another animal to an Animal object. Method feed() can operate in the same way as the constructor of the class: it can check the type of the received parameter, and then decide what to do. If a parameter of type Animal is given, the animal will be eaten.

Another animal can be "eaten", for example, so that the data field animal_name of the "eatable" animal will be copied to the stomach of the "eater". In the new feed() method, you can refer to the data field animal_name of the given parameter in the same way as is done in the constructor.

The data field animal_name of the "eaten" animal can be given the value "Eaten animal". When the new feed() method is written, the statements

```
tiger_object  =  Animal( "tiger", "Richard" )
cow_object    =  Animal( "cow", "Bertha" )

tiger_object.feed( cow_object )
tiger_object.make_speak()
```

should produce the output

```
Hello, I am a tiger named Richard
I have eaten: Bertha,
```

Exercise 6:

Modify class Animal so that the data field stomach_contents stores a list of strings. In the constructor, the new kind of stomach can be created in the following way

```
self.stomach_contents  =  []
```

The intention here is that when the feed() method is called, the given food (a string) is added to the end of this list. A new object can be added to a list with the append() method.

This modification requires that the constructor and methods of the class operate in a slightly different way. Method parameters, as well as the "main program" will remain unchanged.

When stomach contents is printed to the screen, the eaten strings should be read from the list. Strings stored in a list can be printed with a loop such as

```
list_of_strings  =  [ "first", "second", "third" ]

for string_to_print in list_of_strings :

   print string_to_print
```

EXERCISES WITH CLASS ISODate

The file ISODate.py contains a class named ISODate that can be used to make various calculations related to dates. This class is used in the programs Columbus.py, Birthdays.py, and Friday13.py.

The ISODate class handles date information in the so called ISO format, which means that dates are printed in format YYYY-MM-DD. (ISO is an abbreviation for International Organization for Standardization.)

Exercise 1:

Write a program that calculates your current age in years, months, and days. You can accomplish this when you first create ISODate objects in the following way

```
my_birhtday =  ISODate( 1977, 7, 14 )
date_now    =  ISODate()
```

By studying program Columbus.py, you'll find out how the time difference between two ISODate objects can be calculated. You can also easily find out on which day of the week you were born.

You can do this exercise so that you modify program Columbus.py. You should, however, change the names in the program so that they reflect what you are calculating. You can use the names given above.

Exercise 2:

Improve your program so that it prints a list of your most important birthdays and tells on which day of week those birthdays occur. You should study program Birthdays.py to find out how to do this.

Exercise 3:

In this exercise we'll study how a class can inherit another class in Python. Programs BankPolymorphic.py and Windows.py are examples in which inheritance is used.

Derive another class named AnotherDate from class ISODate. You can write the new AnotherDate class before the "main program" after the import statements. The AnotherDate class should be the same as class ISODate with the exception that it has an additional method that begins as follows

```
def to_anti_iso_format( self )  :
    # Here begin the internal statements of the method
```

This method should return a string that contains the date in anti-ISO format which is DD.MM.YYYY. You can create this method quite easily if you make of copy of method __str__(self), and change its name and internal statements. With this method you should be able to print dates in the following way

```
print "\n\n   " +  test_date.to_anti_iso_format()
```

Note that when you derive a new class from an existing class in Python, the constructor

method __init__(self) is also inherited. So you do not always need to write a constructor to a derived class.

Exercise 4:

Improve your program so that it prints dates when you are 10000 and 20000 days old. The age of a person is 10000 days, when his/her "conventional" age is approximately 27 years and 4 1/2 months. With this feature in your program, you'll get new days for partying. This feature can be programmed when you increment a day counter and an ISODate object inside a loop, for example, in the following way.

```
day_counter         =  0
date_to_increment   =  ISODate( my_birthday )

while  ... :

   day_counter += 1
   date_to_increment.increment()

   if ...
```

Exercise 5:

Improve your program so that it tells when you are 1000000000 seconds old. Also this feature can be programmed so that you count days starting from your birthday. Each day has 24 * 60 * 60 seconds. 1000000000 seconds will be reached some time after you are 31 years old. Your program should print your age in years, months, and days on the day when you are 1000000000 seconds old. Again you'll have one more day to celebrate!