CSCI 235, Lab Exercise 6 (Lists, Dictionaries, Classes, MyPy)

Deadline: 21.10.2022 at 11PM

Solutions must be submitted through Moodle as a single text file, called **lab06.py**. Do not use an archiver.
Goal of this exercise is that will see dictionaries, class definitions in Python, and type hints in action. Additional goal is that you gain some more experience with Python programming.
Make sure that you are using **Python** version 3.9 or higher, and that you have access to `mypy`. Download the starter code **lab06.py**. It contains the following definitions:

- An exception class that will be thrown (or more politely 'raised') when the user tries to construct an ill-formed Date.

```
class WrongDate( Exception ) :
   def __init__( self, reason ) :
      self. __reason = reason

   def __repr__( self ) :
      return self.__reason
```

- A class definition for Date:

```
class Date :
   year : int
   month : int
   day : int

   # In monthnames, the first name is the 'preferred name', which will be used
   # when printing. Any further names are optional names.
   # One can also add different languages.

   monthnames : Tuple[ List[ Union[ str, int ]], ... ] = (
       [ 'january', 'jan', 1, '1' ], [ 'february', 'feb', '2', 2 ],
       [ 'march', 3, '3' ],
```

```
    [ 'april', 4, '4' ],  [ 'may', 5, '5' ], [ 'june', 6, '6' ],
    [ 'july', 7, '7' ],  [ 'august', 8, '8' ],
    [ 'september', 'sept', 9, '9' ], [ 'october', 'oct', 10, '10' ],
    [ 'november', 'nov', 11, '11' ],
    [ 'december', 'dec', 12, '12' ] )

monthindex : Dict[ Union[ str, int ], int ] = { name : ind
        for ind, names in enumerate( monthnames ) for name in names  }

normalyear = ( 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 )
leapyear =   ( 31, 29, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 )

weekdays = ( 'sunday', 'monday', 'tuesday', 'wednesday',
             'thursday', 'friday', 'saturday' )
```

The first four lines appear to declare three fields `year, month, day`, but in fact they are only *type hints*. They have no effect at all when the program is run, because the interpreter merrily ignores them. This applies to local variables, function signatures, and class variables.

Type checking in Python is done by a separate program called `mypy`. We expect you to always use mypy, and annotate variables in your program as much as possible. Type errors are annoying, but runtime errors are much more annoying.

The field `monthnames` is a tuple of 12 lists. Each of these lists describes the possible names that a month can have, for example for January, it is `[ 'january', 'jan', 1, '1' ]`. If you want, you can add other languages, for example 'qantar'. The first entry is always the preferred name, which will be used when the month is printed.

The field `monthindex` contains a dictionary that maps month names to indices. It is automatically constructed from `monthnames`. So we have `monthindex[ 'january' ]`, `monthindex[ 'jan' ]`, `monthindex[ 1 ]`, and `monthindex[ '1' ]` all equal to 0.

Field `normalyear` contains the days in each month in a normal year. Field `leapyear` contains the days for each month in a leap year. The only difference between them is that `normalyear[1]` = 28, and `leapyear[1]` = 29.

Tasks

1. Create a static class method

   ```
   @staticmethod
   def isleapyear( y : int ) -> bool :
   ```

   that returns `True` if `y` is a leap year, and `False` otherwise. The rule for determining when a year is a leap year can be easily found on the internet.

Test your implementation carefully, because the remaining tasks rely on it.

2. Next create a constructor for `Date`:

```
@classmethod
def __init__( self, year : int, month : Union[ int, str ], day : int ) :
```

Note that in Python, constructors are always named `__init__` instead of the class name. (But when you call the constructor, it is still `Date`.) Another difference is that the `this` argument, which is implicit in C^{++} and Java, must be included in the parameter list. It must be called `self`. This applies to all non-static methods of a class. If you want to refer to a field of a class inside a class method, you also have to write `self.` in front of it.

Make sure to use the right order in the remaining parameters: `year, month, day` (from big to small.)

The constructor must check that `year`, `month`, `day` are a valid date between 1900 and 2100. If not, it must raise a `WrongDate` containing an informative message. In order to raise an `WrongDate` exception, write for example

```
raise WrongDate( "year {} is not an integer". format( year ))
```

Your constructor must check the following:

- `year` must have type `int` (use `isinstance` to check this.)
- `year` must lie between 1900 and 2100.
- `month` must be in the domain of `monthindex`.
- `day` must have type `int`
- `day` must lie between 1 and the number of days in `month`, taking into account the possibility that `year` is a leap year.

Even though the parameters have type declarations, there is nothing that prevents the user from calling the constructor with ill-typed arguments. Hence, types of parameters must still be checked at runtime.

In order to print a `Date`, add:

```
@classmethod
def __repr__( self ) -> str :
   return "( {}, {}, {} )". format( self. year, self. month, self. day )
```

Examples:

```
Date( 'a', 4, 5 )
date.WrongDate: year a is not an integer
Date( 1899, 'jan', 25 )
   date.WrongDate: year 1899 is before 1900
Date( 1910, 5, 25 )
   (1910, 5, 25)   # this is a successful construction
Date( 1967, 'wrongember', 12 )
   date.WrongDate: unknown month wrongember
Date( 1910, 'oct', 25 )
   (1910, 10, 25)
Date( 1910, 'jan', 25.1 )
   date.WrongDate: day 25.1 is not an integer
Date( 1919, 'feb', 29 )
   date.WrongDate: month february does not have 29 in year 1919
Date( 1920, 'feb', 29 )
   (1920, 2, 29)  # successful construction
```

We recommend that you add all these cases to the test function below.

3. Python distinguishes between pretty printing (for the user) and ugly printing (for the developer). Ugly printing is done with `__repr__( self )`, which we already defined.

   In order to enable pretty-printing, implement a class method `def __str__( self ) -> str :` It must print the month by its name (not a number). Use the first entries in field `monthnames` for this.

   ```
   print( Date( 1989, 'jan', 12 ))
   12 january 1989
   ```

4. Create a method `def weekday( self ) -> str :` that returns the day of the week of the given date.

   This can be done by counting the days since January 1st 1900 (it was a Monday), take this number modulo 7, and index in weekdays.

   ```
   d = Date( 1991, 'dec', 16 )
   d. weekday( )
   'monday'
   ```

5. To start your tests, we provided a testing function `tester( )`. Study the code, and learn to write such functions by yourself. Careful testing is as important as programming itself.

   The function `tester( )` consists of two parts. The first part tests the constructor, that only correct Dates can be constructed, and that appropriate exceptions are raised when the arguments do not represent a valid Date.

The second part tests `weekday( )`, using a mixture of lucky and unlucky dates. Make sure that all returned weekdays are correct.

Add your own tests, in particular you must add tests for `leapyear( )`.