Introduction to Programming Homework 8 Solutions

Exercise 1 (Moving files around)

Write a module called `file_sorter.py`

- **a.** Generate a file with at least 50 lines of random text using your `generate_random_text` from the previous homework. Write a function called `make_files(path_to_random_text)` which
  - creates a directory called `sorting_dir`, if it doesn't already exist
  - takes the path to your random text file and creates a blank file in `sorting_dir` for **every word in the file of random text**
  - the names of the blank files should be `<word>.ext`, for example if your random text file contains a line

    ```
    M TiRmMn weyfZVKT WNftXrUrjuLmECV
    ```

    you should generate files with names `M.ext`, `TiRmMn.ext`, `weyfZVKT.ext`, and `WNftXrUrjuLmECV.ext` for that line. Make sure to generate files for **all** lines in your random text file.
  - you can either just `open` and `close` a file to make a bank file or you can use the function

    ```
    import os
    def touch(file_name):
    """ Updates file access and modify times or creates file."
    ""
    try :
       os.utime(file_name)
    except :
       open(file_name, 'a').close()
    ```

In [3]:

```
import os

def touch(file_name):
    """ Updates file access and modify times or creates file. """
    try :
        os.utime(file_name)
    except :
        open(file_name, 'a').close()

rand_files_dir = 'sorting_dir'
ext = 'ext'

def make_files(path_to_random_text) :
    if not os.path.isdir(rand_files_dir) :
        os.mkdir(rand_files_dir)
    with open(path_to_random_text, 'r') as fp :
        for line in fp :
            basenames = line.split()
            for name in basenames :
                new_file = os.path.join(rand_files_dir,
                                        name + '.' + ext)
                touch(new_file)

def first_letter_sort(sorting_dir) :
    for root, dirs, files in os.walk(sorting_dir) :
        orig_cwd = os.getcwd()
        os.chdir(root)
        for path in files :
            dir_name = path[0].upper()
            # make sure our dir_name is a letter
            if not dir_name.isalpha() : continue
            if not os.path.isdir(dir_name) :
                os.mkdir(dir_name)
            shutil.move(path, dir_name)
        os.chdir(orig_cwd)
        break # only touch files in the top dir
```

- **b.** Write a function called `first_letter_sort(sorting_dir)`, which takes a directory and **moves/sorts all the files** in `sorting_dir` into subdirectories by the first letter of the file name. The subdirectories should just be named by the **uppercase** first letter they represent (and they must be created if they don't already exist).
    - For example, the files `M.ext`, `TiRmMn.ext`, `weyfZVKT.ext`, and `WNftXrUrjuLmECV.ext` would be sorted into directories `M/`, `T/` and `W/` based on the first letter. Note that `weyfZVKT.ext` and `WNftXrUrjuLmECV` would both go into `W/`.

In [4]:

```
import os
import shutil

def first_letter_sort(sorting_dir) :
    for root, dirs, files in os.walk(sorting_dir) :
        orig_cwd = os.getcwd()
        os.chdir(root)
        for path in files :
            dir_name = path[0].upper()
            # make sure our dir_name is a letter
            if not dir_name.isalpha() : continue
            if not os.path.isdir(dir_name) :
                os.mkdir(dir_name)
            shutil.move(path, dir_name)
        os.chdir(orig_cwd)
        break # only touch files in the top dir
```

Exercise 2 (Ring of Dual Numbers)

Create a module called `dual_numbers.py`.

- **a.** Create a `class` called `DualNumber` which should represent an element of the ring $\mathbb{R}[\epsilon]/(\epsilon^2)$. Here, $\mathbb{R}$ will just be `floats`.
  - your `__init__` method should build a dual number $z = a + b\epsilon$
  - define **readolny** property attributes `.real` and `.dual` that return a and b, respectively, for `DualNumber(a,b)`.
    - this means you **don't** need to write setters.
  - define == for `DualNumber`.
    - for example, `` `DualNumber(1,2) == DualNumber(1,2)`` should be `True`.
  - make it so that `str(DualNumber(1.67,1.4)) == '1.67 + 1.4 eps'` and `repr(DualNumber(1.7,1)) == 'DualNumber(1.7,1)'`
  - your `DualNumber` should support all ring operations, i.e. `+, -, *`.
    - for example, `DualNumber(1,2) + DualNumber(2,1) == DualNumber(3,3)` should be `True`.
  - define `**` such that `z**x` works for an `int x >= 0` and a `DualNumber z`.
  - define `/` for `DualNumber` but return `NotImplemented` when you cannot perform the division.
  - all of these methods should return **new** `DualNumber` instances.

Note : the idea here is that, once created, a `DualNumber` is immutable.

- **b.** If we want to do operations like `5 + DualNumber(2,1)`, we need to define some extra methods. Since `__add__` is always called on the **left** operand, `5 + DualNumber(2,1)` will fail as `int` doesn't know how to add a `DualNumber`. In the case of failure, python with try to call the `__radd__` method on the **right** operand. Your tasks are :
  - allow for adding `int` and `float` to a `DualNumber`
    - make sure that `__add__(self, other)` works when `other` is a `DualNumber`, `float`, or `int` and define a method `__radd__(self,`

`other)` to just return `self + other`.
  - allow for subtracting a `DualNumber` from `int` and `float` (you'll need to define `__rsub__` and `__neg__`)
  - allow for multiplying a `DualNumber` by `int` and `float` (you'll need to define `__rmul__`)
  - allow for `/` with `int` and `float` (you'll need to define `__rtruediv__`)

  At the end, calls like `5 + DualNumber(2,1)`, `5.4*DualNumber(2,1)`, and `1.2/DualNumber(2.7,1)` should all work.

- **c.** define a global function `derivative(f, a)`, which takes a **rational function** `f(x)` defined with only the `+,-,*,/,` and `**` operations performed on `x`, and returns `f'(a)`.
  - for example, let

    ```
    def f(x) :
      return x**2 + 2
    ```

    then `derivative(f, 1.5)` should be `3.` (or some very close float).
  - hint : use dual numbers!

In [5]:

```
def is_num(x) :
    """ Returns True if x is an int, float, or complex. """
    return isinstance(x, (int, float, complex))

class DualNumber :
    """ Implements dual numbers of int, float, or complex. """
    def __init__(self, real = 0, dual = 0) :
        assert is_num(real) and is_num(dual)
        self._real = real
        self._dual = dual

    @property
    def real(self) :
        return self._real

    @property
    def dual(self) :
        return self._dual

    def __eq__(self, other) :
        if is_num(other) :
            return self.real == other.real and \
                   self.dual == 0
        if isinstance(other, DualNumber) :
            return self.real == other.real and \
                   self.dual == other.dual
        return NotImplemented

    def __str__(self) :
        sgn = '-' if self.dual < 0 else '+'
        if isinstance(self.dual, complex) :
            dual = self.dual
        else :
```

```
        dual = abs(self.dual)
    return '{} {} {} eps'.format(self.real, sgn, dual)

def __repr__(self) :
    return 'DualNumber({},{})'.format(self.real, self.dual)

def __add__(self, other) :
    if is_num(other) :
        return DualNumber(self.real + other, self.dual)
    if isinstance(other, DualNumber) :
        return DualNumber(self.real + other.real,
                          self.dual + other.dual)
    return NotImplemented

def __sub__(self, other) :
    if is_num(other) :
        return DualNumber(self.real - other, self.dual)
    if isinstance(other, DualNumber) :
        return DualNumber(self.real - other.real,
                          self.dual - other.dual)
    return NotImplemented

def __mul__(self, other) :
    if is_num(other) :
        return DualNumber(self.real * other, self.dual * other)
    if isinstance(other, DualNumber) :
        return DualNumber(self.real * other.real,
                          self.real * other.dual +
                          self.dual * other.real )
    return NotImplemented

def __pow__(self, power, *modulo) :
    """ Computes self ** power. If modulo is given or
    power is not an int, returns NotImplemented. """
    if modulo or not isinstance(power, int) :
        return NotImplemented
    result = DualNumber(1,0)
    double = self
    recip = False
    if power < 0 :
        recip = True
        power = -power
    # standard power accumulation algo
    while power > 0 :
        if power % 2 != 0 :
            result = result * double
        double = double * double
        power //=2

    if recip :
        return 1 / result
    return result

def __truediv__(self, other) :
    if is_num(other) and other != 0 :
        return DualNumber(self.real / other, self.dual / other)
    if isinstance(other, DualNumber) and other.real != 0 :
        new_real = self.real / other.real
```

```
            new_dual = (self.dual * other.real -
                        self.real * other.dual) / other.real**2
            return DualNumber(new_real, new_dual)
        return NotImplemented

    def __neg__(self) :
        return DualNumber(-self.real, -self.dual)

    def __radd__(self, other) :
        return self + other

    def __rsub__(self, other) :
        return -self + other

    def __rmul__(self, other) :
        return self * other

    def __rtruediv__(self, other) :
        if is_num(other) :
            return DualNumber(other) / self
        return NotImplemented

def derivative(f,a) :
    """ Given a function f defined using +,-,*,/, and integer pow,
    returns the numeric value f'(a). Computed using dual numbers. """
    return f(DualNumber(a,1)).dual
```