Exercise 7: Spatial data

Kristian Urup Olesen Larsen[1]

[1]kuol@econ.ku.dk

March 28, 2020

Note: the problem numbering is wrong in the exercise set. I hope this doesn't cause confusion.

The warmup

This weeks warmup is difficult, and for that reason i will give some hints here that will help you solve it. This is not a solution guide for the warmup, but it should contain the information about decorators that you need to solve it. First of all some terminology. A *decorator* in python is a special kind of function that takes as input another function, and as output returns a third function. Decorators takes in functions, transform them and returns the transformed function. The name *decorator* is a reference to the special python syntax that is used to apply them (more on that later). Before we move on, I will define a simple function that we can use for testing our decorators.

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

With our test subject defined, let us look at the simplest example of a decorator possible

```
def simple_decorator(fnc):
    return fnc
```

This decorator takes in the function fnc and immediately gives it back untouched. If we try defining a new variable g = simple_decorator(f), g becomes callable and g(2) will return 4.
This should get your thoughts started - we have just opened up a way to inject code between defining g and the functionality it inherits from f. So what kinds of useful code can we put in the definition of simple_decorator?

It turns out that a very useful pattern is to define another function inside simple_decorator. The code block below does exactly the same as simple_decorator, but this time instead of returning fnc directly, it returns the inner function _wrapped(x), which in turns applied fnc to x.

```
def complicated_decorator(fnc):
    def _wrapped(x):
        return fnc(x)
    return _wrapped
```

We could apply this decorator exactly as we did before, i.e. by

```
g = complicated_decorator(f)
```

Very often we would just overwrite f here, and this pattern of calling a decorator on one function, to overwrite it with another is so common that python has special syntax for doing it in definition of the initial function. So let us redefine f using this special syntax:

```
@complicated_decorator
def f(x):
    return x**2
```

Of course complicated_decorator have no functionality right now, so f(2) is still 4. Notice that this is where the name *decorator* comes from; we decorate the definition of f by adding @complicated_decorator above it.

Now we have an even larger opening for injecting code in the function definition step. We could write anything we want in _wrapped before or after calling fnc! In fact let us define a decorator that modifies f to not return x^2, but $(x-1)^2$. Let us also make it very clear that we are doing this by printing a message whenever the function gets called:

```
def sub_one(fnc):
    def _wrapped(x):
        print(f'Calling f({x}) with substitution y={x}-1={x-1}')
        y = x - 1
        return fnc(y)
    return _wrapped
```

Let us redefine f to apply this decorator

```
@sub_one
def f(x):
    return x**2
```

Now f(2)=1, f(3)=4 and so on. You should try this out in a notebook and check that it does indeed also print the message whenever f is called. Do you have any prediction about the output values of f if we redefine it like i have done below? Check in the notebook to find out if your guess was correct.

```
@sub_one
@sub_one
def f(x):
    return x**2
```

We need one last layer of abstraction to get the full benefit from our decorators. What we really need is the ability to pass variables to the decorators we write. Say for instance that we really wanted a decorator to compute $f(x-k)$ for varying values of k. We would like to be able to write code like

```
@sub_k(k=3)
def f(x):
    return x**2
```

In this way we could write flexible decorators that could be adjusted to the specific functions they apply to. So how do we do this? By thinking a bit about the problem we might realize that what we really need is a function that returns a decorator when it is called. In this way @sub_k(3) would evaluate the parentheses and return a decorator (let's call it _decorator(fnc)), this decorator would then evaluate on f and we would have a working argument-receiving decorator.

Let us try this. To begin with i will simply implement _decorator as the very simple example you saw in the beginning that passed fnc right through without changes.

```
def sub_k(k):
    def _decorator(fnc):
        return fnc
    return _decorator
```

Now let us expand _decorator to include a _wrapped function just like we did above. I will implement the subtraction of k straight away, instead of subtracting 1 as we did above. I will also take out the printed message. If you want to play around with this decorator you can plug it back into _wrapped

```
def sub_k(k):
    def _decorator(fnc):
        def _wrapped(x):
            y = x - k
            return fnc(y)
        return _wrapped
    return _decorator
```

So there you have it, a decorator that takes a variable as input, and which we can use to quickly redefine any function of x in terms of $x - k$. Notice that `k` is accepted in the outer function, and is thus available to all of the inner functions. The opposite is not the case; function arguments only travel inwards. (this exact same phenomenon is why functions can access variables defined outside them, but not the other way around).

Finally let us talk about some technical stuff. Right now `_wrapped` is a function of `x`. Typically we dont care to much about the arguments that goes into `_wrapped`, because we control which ones we expect to be there when we define `f`. For that reason it is typically better to write `def _wrapped(*arg, **kwargs)` to simply accept all arguments. Of course this would also imply rewriting the line `return f(x)` into `return f(*args, **kwargs)`. In this way our decorator becomes input-agnostic and thus more general.

Problem 7.1.1 - Polygons

The first problem gives some very basic intuition for polygons. A polygon is defined as the interior of a sequence of points that are connected with straight lines. `shapely.geometry.Polygon` requires a list of such "corner coordinates" to define a polygon, so we can make a square by

```
square = Polygon([(2,2), (2,5), (5,5), (5,2)])
```

and a triangle is equivalently defined by

```
triangle = Polygon([(0,0), (4,0), (2,4)])
```

Problem 7.1.2 - Polygon operations

We can subtract different shapes by their difference

```
square.difference(triangle)
```

Or join them by their union

```
square.union(triangle)
```

Problem 7.1.3 - GeoSeries

Geopandas geoseries are to geographical data what `pandas.Series` is to columns of data. Typically geopandas is imported as `gpd`.

```
gpd.GeoSeries([square, triangle]).plot()
```

you will notice that geopandas is build on pandas, and therefore has a very similar interface.

Problem 7.2.1 - Loading the data

In this problem you are asked to load a geojson containing a map of the Danish municipalities. For this we use gpd.read_file. This is one of the methods that makes geopandas different from pandas. In geopandas there are a bunch of special functions designed specifically to handle the often complex dataformats that come with geographical data.

```
url = "https://raw.githubusercontent.com/ok-dk/dagi/master/geojson/kommuner.geojson"
kommuner = gpd.read_file(url)
```

If you look at the data you will see that it has a special column *geometry* which is populated by polygons. To view the CRS of the data use kommuner.crs, we can change the CRS by calling

```
kommuner.to_crs({'init':'epsg:25832'}, inplace=True)
```

Be aware that this CRS-changing is notorious for causing problems. If this line does not run on your PC, you will need to do some googling to search for posts with other people sharing your problems. Assuming you can change the CRS we can plot the map using

```
import matplotlib.pyplot as plt

kommuner.plot(figsize=(14,8))
plt.axis('off')
```

You should end up with a map looking like this Finally we need to identify the three largest municipalities.

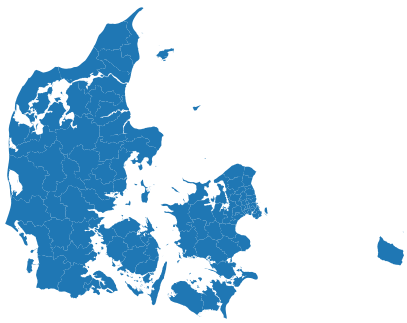

Fig 1. Map of the municipalities in Denmark.

Here you need to be careful because many municipalities are disjoint. Most of the time becaue they cover multiple islands. In this case each island will have it's own polygon, and thus its own row in the data. We can join them together by using the Polygon.unary_union method. In a GeoDataFrame this looks like

```
sizes = kommuner.groupby('KOMNAVN').geometry.apply(lambda d: d.unary_union.area)
```

Getting the three largests is then as simple as calling sizes.nlargest(3).