Python defaultdict with data

If you ever downloaded data from the internet where the source is unstable - which means you are not using any API but you are scraping some URL and parsing it's content no matter if it's HTML or more friendly JSON data you always face the issue if the data will be there next week. In such cases your code contains a lot of `if`/`else` statements because you need to check every param if it's present and has the right datatype (at least).

Building up a structure

Everyone (I hope) knows defaultdict which helps us to build up a dictionery with no-yet-initialized keys.

```
from collections import defaultdict

data = defaultdict(dict)
data["key"]["key"] = "value"
```

You can even make it nested/recursive:

```
from collections import defaultdict

tree = lambda: defaultdict(tree)
data = tree()
data["key"]["key"]["key"]["key"]["key"] = "value"
```

All this is good if you need to build up data structrure from scratch were you own data values.

Converting an existing structure

Another story is when you have an existing structure - like just downloaded JSON data - and you want to apply the same convenient acceess to the key where you just ask and don't have to worry about `KeyError` exception in case the key mistically disaapears from the data for no reason and you don't wanna flood your code with `if`/`else` everywhere. This cannot be done by `defaultdict` - you rather want your current data wrap up with "something" that implements similar logic like `default-duct` does.

In this case we can extend Python dict and add up a tiny logic which gives us exactly what we are looking for:

```
class Data(dict):
    """
```

```
    Dict substitution which recursivelly handles
    non-existing keys.
    """

    def __getitem__(self, key):

        try:
            data = super().__getitem__(key)

            # If the data is dict we need to wrap it with
            # this class so it will carry this logic.
            if type(data) == dict:
                return self.__class__(data)

            # Data is not a dict so we return what we found.
            return data
        except:

            # In case of non existing key we return empty self
            # which makes sure another direct key demand will
            # copy this logic.
            return self.__class__()
```

This wrapper can be used like:

```
data = Data(json_data)
data["existing_key"]["non_existing_key"] or ""
data["existing_key"]["non_existing_key"]["another_non_existing_key"] or
 ↪  False
```

The class returns the key it finds or empty dict (wrapped in self) so you can chain non-existing keys. So the “not-found” value is always {}.