vec

Coding the Matrix, Summer 2013

Please fill out the stencil file named "`vec.py`". While we encourage you to complete the Ungraded Problems, they do not require any entry into your stencil file.

We have said that, mathematically speaking, a vector $\boldsymbol{v}$ is a function $\boldsymbol{v} : D \to C$, where D and C are the domain and codomain, and C is a field. We will implement vectors as a class `Vec`. Our class `Vec` will have two data members (also known as fields): a set `D`, the domain, and a dictionary `f`, the possibly sparse representation of the underlying function (i.e., items for which the value is zero need not be represented).

We have provided a skeleton Python file `vec.py` with procedure definitions and a class built on these definitions. The file defines some procedures using the Python statement `pass`, which does nothing. The file also defines the class `Vec` in such a way that expressions such as `u+v` and `u*v` and `-v` and `alpha*v` and `v[d]` are legal. However, currently the operations do nothing.

Your job is to complete the definitions of the procedures, replacing each occurence of the `pass` statement with appropriate code. The specification of each procedure is described in the procedure's documentation string. **Be brief!** You need only write about nine lines of code in total. Most procedures require you to write only one line of code.

You don't need to edit the definition of the class itself.

Assertions: For most of the procedures to be written, the first statement is an *assertion*. Executing an assertion verifies that the condition is true, and raises an error if not. The assertions are there to detect errors in the use of the procedures. For example, an assertion prevents you from trying to add two vectors with different domains. Take a look at the assertions to make sure you understand them. Please keep the assertions in the code.

Arbitrary set as domain: Our vector implementation allows the domain to be, for example, a set of strings. Do not make the mistake of assuming that the domain consists of integers. If your code includes `len` or `range`, you're doing it wrong.

Sparse representation: Your procedures should be able to cope with our sparse representation, i.e. an element in the domain `v.D` that is not a key of the dictionary `v.f`. For example, `getitem(v, k)` should return a value for every domain element even if `k` is not a key of `v.f`. However, your procedures should *not* make any effort to retain sparsity when adding two vectors. That is, for two instances `u` and `v` of `Vec`, it is okay if every element of `u.D` is represented explicitly in the dictionary of the instance `u+v`.

Several other procedures need to be written with the sparsity convention in mind. For example, two vectors can be equal even if their `.f` fields are not equal: one vector's `.f` field can contain a key-value pair in which the value is zero, and the other vector's `.f` field can omit this particular key. For this reason, the `equal(u, v)` procedure needs to be written with care.

Testing: Because much of what is coming is based on the class `Vec`, it is very important that your implementation be correct. We have provided a file `test_vec.py` that shows (in one big string) examples of correct use of the `Vec` class and the correct Python values of expressions. Working in the Python REPL, import your Vec class using the command

```
>>> from vec import Vec
```

and try out the examples. You can copy and paste the examples into a Python dialogue to see what your implementation does. Please work on your implementation until its behavior matches that in the examples (aside from order in sets and dictionaries, of course).

There is a simple way to test your `vec` module against all the tests in `test_vec`. From a console (not running the Python REPL), type the command

```
python3 -m doctest test_vec.py
```

This will run the tests given in `test_vec.py`, including importing your `vec` module, and will print messages about any discrepancies that arise. If your code passes the tests, nothing will be printed.
Note: Simply importing `test_vec` from within Python will *not* carry out any tests.

Using `Vec`: After you have completed your `Vec` implementation, you should get used to using it in the proper way. In code outside `vec.py`, you should not call the named procedures. Instead, when operating on Vecs you should import only the `Vec` class itself, and use the operators [ ], +, *, -, /, and so on. Here's a table with the syntax for working with vectors.

operation	syntax
vector addition	`u+v`
vector negation	`-v`
vector subtraction	`u-v`
scalar-vector multiplication	`alpha*v`
division of a vector by a scalar	`v/alpha`
dot-product	`u*v`
getting value of an entry	`v[d]`
setting value of an entry	`v[d] = ...`
testing vector equality	`u == v`
pretty-printing a vector	`print(v)`
copying a vector	`v.copy()`