# vec

Coding the Matrix, 2015

`For auto-graded problems, edit the file vec.py to include your solution.`

**Problem 1:** You will edit the stencil file `vec.py`. The file has two parts. The first part defines seven procedures. The second part defines the class `Vec` using the seven procedures in defining the class `Vec`. This class definition allows you to use operators such as * and + and [] to operate on instances of `Vec`. Currently, however, these operators do nothing because the seven procedures are defined to do nothing: the body of each of them consists of the statement `pass` that is Pythonese for "do nothing". Your job is to replace each occurence of the `pass` statement with appropriate code. Your code for a procedure can include calls to others of the seven. You should make no changes to the class definition.

**Docstrings** At the beginning of each procedure body is a multi-line string (deliminated by triple quote marks). This is called a documentation string (*docstring*). It specifies what the procedure should do.

**Doctests** The documentation string we provide for a procedure also includes examples of the functionality that procedure is supposed to provide to `Vec`s. The examples show an interaction with Python: statements and expressions are evaluated by Python, and Python's responses are shown. These examples are provided to you as tests (called *doctests*). You should make sure that your procedure is written in such a way that the behavior of your `Vec` implementation matches that in the examples. If not, your implementation is incorrect.[a]

Python provides convenient ways to test whether a module such as `vec` passes all its doctests. You don't even need to be in a Python session. From a console, make sure your current working directory is the one containing `vec.py`, and type

```
python3 -m doctest vec.py
```

to the console, where `python3` is the name of your Python executable. If your implementation passes all the tests, this command will print nothing. Otherwise, the command prints information on which tests were failed.

You can also test a module's doctest from within a Python session:

```
>>> import doctest
>>> doctest.testfile("vec.py")
```

**Assertions** For most of the procedures to be written, the first statement after the docstring 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. Take a look at the assertions to make sure you understand them. You can take them out, but you do so at your own risk.

**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 need *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.

[a] The examples provided for each procedure are supposed to test that procedure; however, note that, since equality is used in tests for procedures other than `equal(u,v)`, a bug in your definition for `equal(u,v)` could cause another procedure's test to fail.