Dictionary-based representations of vectors

- A vector is a function from some domain D to a field
- Can represent such a function in Python by a *dictionary*.
- It's convenient to define a Python class `Vec` with two instance variables (fields):
  - `f`, the function, represented by a Python dictionary, and
  - `D`, the domain of the function, represented by a Python set.
- We adopt the convention in which entries with value zero may be omitted from the dictionary `f`

(Simplified) class definition:

```
class Vec:
    def __init__(self, labels, function):
        self.D = labels
        self.f = function
```

Dictionary-based representations of vectors

(Simplified) class definition:

```
class Vec:
    def __init__(self, labels, function):
        self.D = labels
        self.f = function
```

Can then create an instance:

```
>>> Vec({'A','B','C'}, {'A':1})
```

- First argument is assigned to `D` field.
- Second argument is assigned to `f` field.

Dictionary-based representations of vectors

Can assign an instance to a variable:

```
>>> v=Vec({'A','B','C'}, {'A':1.})
```

and subsequently access the two fields of v, e.g.:

```
>>> for d in v.D:
...   if d in v.f:
...    print(v.f[d])
...
1.0
```

Dictionary-based representations of vectors

Quiz: Write a procedure `zero_vec(D)` with the following spec:

- *input:* a set `D`
- *output:* an instance of `Vec` representing a `D`-vector all of whose entries have value zero

Dictionary-based representations of vectors

Quiz: Write a procedure `zero_vec(D)` with the following spec:

- *input:* a set `D`
- *output:* an instance of `Vec` representing a `D`-vector all of whose entries have value zero

```
def zero_vec(D): return Vec(D, {})
```

or

```
def zero_vec(D): return Vec(D, {d:0 for d in D})
```

Dictionary-based representations of vectors: Setter and getter

Setter:

```
def setitem(v, d, val):  v.f[d] = val
```

- Second argument should be member of `v.D`.
- Third argument should be an element of the field.

Example:

```
>>> setitem(v, 'B', 2.)
```

Dictionary-based representations of vectors: Setter and getter

Quiz: Write a procedure `getitem(v, d)` with the following spec:

- *input:* an instance `v` of `Vec`, and an element `d` of the set `v.D`
- *output:* the value of entry `d` of `v`

Dictionary-based representations of vectors: Setter and getter

Quiz: Write a procedure `getitem(v, d)` with the following spec:

- *input:* an instance `v` of `Vec`, and an element `d` of the set `v.D`
- *output:* the value of entry `d` of `v`

Answer:

```
def getitem(v,d): return v.f[d] if d in v.f else 0
```

Another answer:

```
def getitem(v,d):
    if d in v.f:
        return v.f[d]
    else:
        return 0
```

Why is `def getitem(v,d): return v.f[d]` not enough?
Sparsity convention

Vec class

We gave the definition of a rudimentary Python class for vectors:

```
class Vec:
    def __init__(self,
       labels, function):
        self.D = labels
        self.f = function
```

The more elaborate class definition allows for more concise vector code, e.g.

```
>>> v['a'] = 1.0
>>> b = b - (b*v)*v
>>> print(b)
```

More elaborate version of this class definition allows *operator overloading* for element access, scalar-vector multiplication, vector addition, dot-product, etc.

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()`

You will code this class starting from a template we provide. (See quizzes for help.)

Using Vec

You will write the bodies of named procedures such as `setitem(v, d, val)` and `add(u,v)` and `scalar_mul(v, alpha)`.

However, in actually using Vecs in other code, you must use operators instead of named procedures, e.g.

```
>>> v['a'] = 1.0
>>> b = b - (b*v)*v
```

instead of

```
>>> setitem(v, 'a', 1.0)
>>> b = add(b, neg(scalar_mul(v, dot(b,v))))
```

In fact, in code outside the `vec` module that uses `Vec`, you will import just `Vec` from the `vec` module:

```
from vec import Vec
```

so the named procedures will not be imported into the namespace. Those named procedures in the `vec` module are intended to be used *only* inside the `vec` module itself.

In short: Use the operators [ ], +, *, -, / when working with Vecs

Assertions in `Vec`

For each procedure you write, we will provide the stub of the procedure, e.g. for `add(u,v)`, we provide the stub

```
def add(u,v):
    "Returns the sum of the two vectors"
    assert u.D == v.D
    pass
```

The first line in the body is a documentation string, basically a comment.

The second line is an assertion. It asserts that the two arguments `u` and `v` must have equal domains. If the procedure is called with arguments that violate this, Python reports an error.

The assertion is there to remind us that two vectors can be added only if they have the same domain.

Please keep the assertions in your `vec` code while using it for this course.

Testing Vec

Because you will use `Vec` a lot, making sure your implementation is correct will save you from lots of pain later.

We have provided a file `test_vec.py` with lots of examples to test against.

You can test each of these examples while running Python in interactive mode by importing `Vec` from the module `vec` and then copying the example from `test_vec.py` and pasting:

```
>>> from vec import Vec
>>> getitem(Vec({'a','b','c', 'd'},{'a':2,'c':1,'d':3}),'d')
3
```

You can also run all the tests at once from the console (outside the Python interpreter) using the following 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.

list2vec

The `Vec` class is useful for representing vectors
but is not the only useful representation.

We sometimes represent vectors by lists.

A list L can be viewed as a function from $\{0, 1, 2, \ldots, \text{len}(L) - 1\}$, so it is easy to convert between list-based and dictionary-based representations.

Quiz: Write a procedure `list2vec(L)` with the following spec:

- *input:* a list L of field elements
- *output:* an instance $\mathbf{v}$ of `Vec` with domain $\{0, 1, 2, \ldots, \text{len}(L) - 1\}$ such that $\mathbf{v}[i] = L[i]$ for each integer i in the domain

list2vec

The `Vec` class is useful for representing vectors
but is not the only useful representation.

We sometimes represent vectors by lists.

A list L can be viewed as a function from $\{0, 1, 2, \ldots, \text{len}(L) - 1\}$, so it is easy to convert between list-based and dictionary-based representations.

Quiz: Write a procedure `list2vec(L)` with the following spec:

- *input:* a list L of field elements
- *output:* an instance $\mathbf{v}$ of `Vec` with domain $\{0, 1, 2, \ldots, \text{len}(L) - 1\}$ such that $\mathbf{v}[i] = L[i]$ for each integer i in the domain

Answer:

```
def list2vec(L):
  return Vec(set(range(len(L))), {k:x for k,x in enumerate(L)})
```

or

```
def list2vec(L):
  return Vec(set(range(len(L))), {k:L[k] for k in range(len(L))})
```

The `vecutil` module

The procedures `zero_vec(D)` and `list2vec(L)` are defined in the file `vecutil.py`, which you can download.