Subject: Re: Is there a less clunky syntax for retrieving data from an array of hashes?
Posted by kagoldberg on Thu, 29 Aug 2013 18:01:10 GMT
View Forum Message <> Reply to Message

David,
That's a good point. With hashes vs. objects, the lack of rigidity leads to a potential reduction in transparency. One can always look at an object's __define method to see what fields are there (and what other classes we inherit from.) I think it's more of a matter of transferring some of the programming burden elsewhere, but the benefits for pure data handling are there.

For graphics I use objects built from IDLgr... pieces in the conventional way, and the predictable structure of fields and methods makes a great framework. By comparison, hashes don't have classes or methods, so we're only focusing on the data--accessing and updating the data freely.

Here's an example of where I appreciate the flexibility of hashes. Say I have a folder containing 1000 images, and a file or files containing their metadata (100 pieces of metadata per image, where each field may be a scalar, text, or a small array.) In addition to the stored metadata, I want to add-on a few fields that are specific to the way I'm processing the stored data, to give me continuity from one session to the next.

For me, the old way of approaching this is with a structure array. Create a structure to contain the metadata from each individual image, and contain all of it those structures in an array, indexed 0,1,2,... One field in the structure is the image file name.

Now every time I access the metadata, I use where() to find the array index of the structure I want, then I take that index and get at the structure fields. If some images were missing some metadata that others have, I need to rely on flag or default values to realize that (adding steps to the processing). Adding fields to the structure for ad-hoc information requires me to add it to everyone's metadata structure (it's a class thing). OK, that system works, and it's not too bad. It worked for me for years.

Now consider my approach with hashes. YMMV.

1. Each image gets its own metadata hash, analogous to the structure, except that it's perfectly acceptable for some data to be missing, or added to one image's metadata and not all of them. The hash will not complain that it doesn't match the others. I can add fields related to the image processing by just declaring hash['newField'] = 'easy'. The hash fields can handle what would otherwise require pointers. Like this

```
hash['a'] = 15.
hash['a'] = [15., 16., 17.]
hash['a'] = !null   ; (Structures hate this one.)
```

2. Now we need to bundle up all of these individual hashes, like the structure array. One way is with a list(). Lists are nice because you don't have to know their count in advance. But with lists you have to somehow remember which list index corresponds to which image, or you'll be stuck searching every time. Here's where hashes shine. Imagine a hash of metadata hashes, using the filenames as the keys.

So with h1 as the metadata hash for 'image_001.tif' (containing whatever), then we assign

```
masterHash = hash()
masterHash['image_001.tif'] = h1
masterHash['image_001.tif'] = h2
...
```

Accessing metadata from an image now looks like this (notice, no searching)

```
xy = masterHash['image_001.tif', 'xy']
  which is a short way of writing
xy = (masterHash['image_001.tif'])['xy']
  (i.e. Give me the 'xy' field in the hash keyed as 'image_001.tif')
```

If you want to be careful, you can test for existence first. A full-paranoid example would be like this

```
file = 'image_001.tif'
xy = !null  ;--- test for this in the end.
if masterHash.haskey(file) $
  then if masterHash[file].hasKey('xy') $
    then xy = masterHash[file, 'xy']
```

In the above example, we ask if the masterHash knows about the image called 'image_001.tif'. Then once that's established, we ask if the metadata contains an 'xy' field.

With structures, it might go something like this, where we know for sure that xy is a field.

```
file = 'image_001.tif'
xy = !null
w = where(sArray.filename EQ file, count)
if count GT 0 $
  then xy = sArray[w[0]].xy
```

Adding new images is less painful in the hash, we just

```
masterHash[newFile] = hNew
```

In the structure case we might append a new structure element to the array.

```
sArray = [sArray, newStruct]
```

It's nice to be able to pass the hash around as an argument to functions. Since it's object-like, it is always a reference to the hash--no memory copying, no multiple-copies.

TL;DR Hashes are great for image metadata because they're flexible, you can key them by filename in a hash of hashes so there's no WHERE searching for your data, you can pass around the reference without copying memory, you can add new, custom fields or images on the fly. Give them a try.
