Errors and Boolean Logic

Errors and Boolean Logic

Usually, in a programming language, error conditions and boolean logic are unrelated topics.

In the offsider framework, however, they are related, as we shall show.

We deal with errors first.

Errors

You can generate an error by using the `Error' method. An example is

```
Dictionary create titanic
titanic Error HIT_ICEBURG Abandon ship! Wealthy women and
children first
```

The generic form is

```
Error errorKey arguments
```

where *`arguments`* is optional, and depends on the *`errorKey`*. The arguments are inserted into the message designated for that *`errorKey`*, according to the message template for that *`errorKey`*.

The details of how this is implemented is not important here, what is important is what this method actually does.

The Error method does three things, and they all depend on the error key:

1. outputs a message to stderr

2. outputs an exit code to stdout

3. exits with the same exit code.

It has been designed this way to work well with the UNIX shell.

Typical usage, which you will see all through the framework source code is:

```
if [ some condition ]
then
exit `Dictionary Error SOME_CONDITION blah blah blah`
fi
```

If the condition is false, then the method exits with a message AND an exit code.

Also,

```
if [ "`Dictionary Error KEY`" ]
```

and

```
if [ `Dictionary Error KEY` ]
```

will both yield TRUE for any value of *KEY*.

Also,

```
Dictionary Error $key && echo I never happen
Dictionary Error $key || echo I always happen
```

Also,

```
exitcode=`Dictionary Error $key`
echo "$? always equals $exitcode"
```

Also, even if you redirect the stderr, like so:

```
Dictionary Error $key 2>/dev/null
```

all of the above examples still work.

In summary, `Dictionary Error` will ALWAYS exit with an error, and write a non-zero integer to `stdout`. In addition, it will write a text message to `stderr`. *(Actually, it returns an integer if the error template has been set up correctly. Currently, it is possible that it could return an arbitrary string if the programmer has been careless).*

Boolean logic.

The Offsider framework is NOT a programming language, so it doesn't actually have any boolean logic built in. (It doesn't actually have a requirement for it).

Nevertheless, it has methods that act like boolean values when used from the UNIX shell.

For example, `hasKey`.

`hasKey` is used to determine whether a particular offsider has a particular key.

If true (the key exists), then it returns a non-empty string, and exits with a zero exit code.

If false (the key doesn't exist), then it returns nothing on stdout, and exits with a non-zero exit code.

Therefore, the following shell code will always work as expected:

```
if [ "`Dictionary hasKey $key`" ]
then
echo has the key
else
echo does not have the key
fi
if [ `Dictionary hasKey $key` ]
then
echo has the key
```

```
else
echo does not have the key
fi
Dictionary hasKey $key || echo does not have the key
Dictionary hasKey $key && echo has the key
```

The reason this works it that `hasKey` makes use of `Error` to generate appropriate error exit codes. This is just a convention, but the convention is used throughout the Offsider framework.

not

There is a convenience method called `not`. It basically changes an empty string to a non-empty string and vice-versa. It also adjusts the exit code appropriately.

The syntax is:

```
not message
```

where *`message`* is any valid message.

For example:

```
Dictionary not hasKey $key
```

The following two lines are equivalent in the UNIX shell:

```
if [ -z "`Dictionary $message`" ]
if [ `Dictionary not $message` ]
```

If you have a multi-part message (a message within a message), then you can often place the `not` in various places within the message. You will need to think about how the message is parsed, and the meaning of the result. Remember, it is not really a boolean logic operator, it is a method that dispatches a message, and then manipulates the resulting string.

For example:

These both work as expected, and are equivalent:

```
Weave not oldest hasKey foo
Weave oldest not hasKey foo
```

This will not work as expected (ie, will work as expected, not.):

```
Weave oldest hasKey not foo
```

The reason this will not work is that the method `hasKey` doesn't parse for `not`. Ie, in this situation, `not` isn't even recognised as a method.

This will work, but what does it mean?

```
Weave oldest after not after hasKey foo
```

(answer, it is equivalent to saying either of the following:

```
Weave not oldest after after hasKey foo
Weave oldest after after not hasKey foo
```

)