chores Documentation

Release 0.6.3

Jonathan Eunice

Mar 09, 2017

Contents

Just about all programs process “items” of one sort or another. That’s what loops are for, right?

But with the exception of the current loop value or index, programming languages don’t help track how processing is going. How many items have been *successfully* processed? How many *errors* are there? How far along the total job are we right now? Which items had problems that need to be looked at later?

Even though these bookkeeping tasks are essential to just about every program, they’re “left to the reader.” “Here are some basic loops. Have fun!” So developers “reinvent the wheel,” tracking status with *ad hoc* containers, counters, and status flags for every new program. Not so high-level after all, huh?

`chores` fights this needless complexity, errors, and effort by providing a simple, repeatable pattern for processing items and tracking their status.

CHAPTER 1

Usage

```
from chores import Chores

chores = Chores('Jones able baker charlie 8348 Smith Brown Davis'.split())

for c in chores:
    status = 'name' if c.istitle() else 'other'
    chores.mark(c, status)

print chores.count('name'), "names,", \
      chores.count('^name'), "others"
```

Yields:

```
4 names, 4 others
```

Or if you decide you actually want more information, change just the output statements:

```
print todos.count('name'),  "names:", todos.marked('name')
print todos.count('^name'), "others:", todos.marked('^name')
```

Now you get:

```
4 names: ['Jones', 'Smith', 'Brown', 'Davis']
4 others: ['able', 'baker', 'charlie', '8348']
```

CHAPTER 2

Discussion

Many programs track the status of items being processed with various lists, dictionaries, sets, counters, and status flags. `chores` might not seem a great advance at first, since it has the same kind of initialization and looping.

But it gets more interesting at the end of the processing loop, where the summary or report of what was processed, the disposition of each item worked on, what items yielded errors or other conditions, and what special cases were handled is produced.

In the examples above, we never had to keep a counter of how many names were found, or how many non-names. When we decided we wanted to change the output from summary counts to a full listing, we didn't have go back and collect different information. We just differently displayed information already at at hand. Also note that the order of the results is nicely maintained. When we're reviewing reports about "what transpired," we don't have to work very hard to correlate the results with the inputs; unlike when using `dict` and `set` structures, items are reported on in the same order they arrived.

Typically a developer will start with only a little thought about various dispositions for each item being processed. Over time, she'll start to realize: "I need to count those cases, so I can report on them!" Or, "I kept an error counter, but I really should have been keeping a list of which items broke, because I now have to tell the user not just how many went wrong, but which ones in particular." Or "I need to keep track of which ones failed the main processing so that I can do more intensive processing on just those special cases." Then she'll go back and add counters, collection lists, and so on–adding a fair amount of *ad hoc* code that must be built, tested, and debugged.

This is especially tricky for data that needs to move through multiple stages or phases of work. The developer then has to add structures to communicate from earlier processing steps to later ones.

With `chores`, there's no need for such custom work. It takes over tracking which items led to which outcomes. It's always ready to render quality information, either for reporting or for managing subsequent processing. Bookkeeping information is readily available in a tidy, logical format, with no additional development effort.

`chores` especially shows its virtues as processing code becomes more intricate and as program needs evolve over time.

CHAPTER 3

Dropping Down

A `Chores` is a specialized form of `OrderedDict` used to both loop over your items and to remember things about them at the same time.

Each item must have a unique identifier. Ideally this would be a human-readable name, but it can be as simple as an integer index. The second most important attribute of each item is its "status." Every item starts with a default status of `"new"` (though this is configurable with the `status` keyword argument when the `Chores` is created).

It is not strictly necessary to loop over the `Chores`. It will "just" keep track of items regardless of what collection is being looped over; it's just often convenient to not have a separate collection. A typical use case might be:

```
from chores import Chores

records = get_records_from_database()    # external source
chores = Chores(rec.rowid for rec in records)

for c in chores:
    try:
        process_item(c)
        chores.mark(c, 'done')
    except Exception:
        chores.mark(c, 'error')

tally = chores.tally()

print tally.done, "items completed,", tally.error, "errors"
if tally.error > 0:
    print "ERRORS:", chores.marked('error')
```

Here the `tally` method returns a counter (similar to `collections.Counter`) that counts how many of each status were seen.

CHAPTER 4

Item Ids

Every item to be processed needs a unique identifier. Identifiers must be hashable, so that they can serve as the key of a Python `dict`. Strings such as file paths or file names work well, though item numbers or tuples of strings and numbers can also work. Good ids will be short and easily understood by those running the program and analyzing its output.

For situations where you want to use titles or file names/paths as your keys, slugifying modules such as slugger, slugify, unicode-slugify, and python-slugify can help turn messy strings into tidy item ids.

In theory, the ids can be anything that makes a good dictionary key, but in practice, any strings that include punctuation used by `chores` as status-selection meta-charactes (e.g. `|` and `^`) is a bad idea; including commas (`,`) in your keys also not recommended.

CHAPTER 5

Associated Data

If you need to carry data along with each item, `todos[chore_id]` returns an attribute-exposed dictionary that lets you fold information related to each item into the status tracking. For example:

```
for t in todos:
    todos[t].upcase = t.upper()
```

Or recast with a more dictionary idiom, dealing with full `Chore` objects not just a key:

```
for todo in todos.values():
    todo.upcase = todo.id.upper()
```

In complex processing, there are always places you want to associate "extra" information with each work item. Here's an easy way to do that. Tying supporting data directly to each work item reduces the need to create or manage auxiliary or "supporting" data structures. (Just mind that you don't overload the `id` or `status` attributes, which are used directly by `chores`; using `data` is also suspect.)

One example of where this might come in handy is error processing. For items that complete successfully, you might not need additional information. But for error conditions, you'll want to know later *why* it failed. So:

```
for todo in todos:
    try:
        process_item(todo)
        todo.mark('done')
    except Exception as e:
        todo.mark('error')
        todo.errmsg = repr(e)
```

Now the error message is quickly appended for later inspection.

CHAPTER 6

Selecting Items

`chores` provides multiple mechanisms to select items. The most important is the `marked` method:

```
todos.marked('done')
```

Returns all items marked done. `todo.count('done')` returns a simple count of such items. This "marked X" mechanism is the most common. It's a bit more general that it might first appear:

```
todos.marked(['done', 'partial'])
```

for example returns items marked **either** of those options. This is an inclusive or.

There is also an `exclude` keyword argument. So:

```
todos.marked(exclude='error')
```

Gives everything not marked as an error (which might be `'done'`, `'partial'`, and `'other'`). The `exclude` kwarg can also take an iterable, to exclude multiple status tags. If no positive inclusion set is provided, the exclusion is against the set of all possible markings.

The combination of include and exclude sets gives a very powerful selection mechanism.

There is also a simplified, less verbose form that depends on a string specification. In this, the vertical bar (`|`) stands for alternation. E.g. The following are identical:

```
todos.marked('done|partial')
todos.marked(['done', 'partial'])
```

Exclusions can also be defined with the caret (`^`), meaning not:

```
todos.marked('^done')
todos.marked(exclude='done')
```

Are identical, and return any items with statuses other than `'done'`. Alternation can be used in either the inclusion or exclusion spec, with two caveats: 1. The negation caret must be the first symbol, if present; if used, all the alternatives are excluded. 2. String specifications can be used in both the default selector and the `exclude` kwarg, but if the negation character is present in an `exclude` argument, it only adds up to single negation; it is not a fancy double-negative.

CHAPTER 7

Other Methods

The `keys`, `values`, and `items` methods work as though each `Chores` is a dictionary (which it more or less is). The “keys” are the individual chore ids. The values are `Chore` objects, which is also an attribute-accessible kind of `dict`, including the status of each item and any user-defined values added to the tracker.

One can loop over a `Chores` collection, with each iteration getting an item id as the loop value. You can use `enumerate` with these loops as well:

```
for i, t in enumerate(todos, start=1):
    print i, t
```

One divergence from standard Python loops is that items can be added to a `Chores` while looping over it.

todos = Chores(range(4)) for i, t in enumerate(todos, start=1):

if t % 2 == 1 and t < 10: todos.add(len(todos), status=’dynamic’)

print i, t

Items are added at the end of the collection, and will be processed at the end of the loop. This is an important feature, for example in tasks like directory and web crawling where some work items discover further work items that need to be done later. You should *never* remove a work item from the collection. Mark it as `'dead'`, `'junk'`, `'invalid'` or some other dustbin status, but work items should never be removed.

`todos.statuses()` returns a list of all the current statuses (marks).

`todos.tally()` returns a counter indicating how many of each kind of status there are.

`todos.bystatus()` returns a mapping of status to a list of the keys / ids associated with that status. Or, `todos.bystatus(justkeys=False)` returns a similar mapping but with the values being full `Chore` objects, not just their keys/ids.

CHAPTER 8

Performance

`chores` clearly adds some performance overhead to loops, because it's doing some additional work for every item processed. It's therefore probably not a good choice for the inner loops of performance-critical code or numerical routines. But inner computational loops are not really what it's designed for.

`chores` is intended first and foremost for the macro loops of utility programs and applications. Here, the small additional overhead is inconsequential. The real performance "cost" lies in the processing of each element, not in a tiny bit of extra housekeeping.

The other cost–and the one `chores` is most aimed at reducing–is programming and debugging time. There is a typical assumption that the housekeeping associated with application loops is "extra." But that's a false assumption; most programs have to do at least some housekeeping already.

So in many cases, any `chores` performance overhead is nominal, and well-compensated by the additional ease and correctness of high-function program construction.

CHAPTER 9

Notes

- I've successfully used `chores` in my own projects, and it has a real test suite. But realistically it should be considered "early beta" and/or "still experimental" code. Its API and mode of use will evolve.
- In the future, it may be possible to assign multiple tags to each chore, rather than just a single status indicator. Currently, one status per item is it. A `Stage` class is also under development to create a reporting framework for multi-stage processing.
- Automated multi-version testing managed with the wonderful pytest and tox. Successfully packaged for, and tested against, all late-model versions of Python: 2.6, 2.7, 3.2, 3.3, and 3.4, as well as PyPy 2.6.0 (based on 2.7.9) and PyPy3 2.4.0 (based on 3.2.5). Should run fine on Python 3.5, though py.test is broken on its pre-release iterations.
- The author, Jonathan Eunice or @jeunice on Twitter welcomes your comments and suggestions.

CHAPTER 10

Installation

To install the latest version:

```
pip install -U chores
```

To `easy_install` under a specific Python version (3.3 in this example):

```
python3.3 -m easy_install --upgrade chores
```

(You may need to prefix these with "sudo " to authorize installation.)