Pycon Israel 2016 – Keynote Documentation

Release 1.0

Raymond Hettinger

May 05, 2016

CONTENTS

CHAPTER

ONE

RAYMOND HETTINGER KEYNOTE

My Mission Train thousands of Python Programmers

Contact Info raymond dot hettinger at gmail dot com

Twitter Account @raymondh

Date May 2, 2016

Contents:

1.1 Introducing the TransformDict

Dictionaries are a fabulous general purpose tool. Guido's thought is that if you could have only two data structures, they should be dicts and lists which can be used to model almost any other mutable data structure.

Over the years, there have been many dozens of dict variants in use. Some have made it into the standard library: sets, ordered dictionaries, chained dictionaries, counters, and the defaultdict.

A core developer working on a case-folding-case-preserving dictionary had an inspiration: Why not factor-out the case-folding transformation and make a general pure key-transforming-key-preserving dictionary.

1.1.1 Docstring

```
>>> print(d.__doc__)
Dictionary that calls a transformation function when looking
up keys, but preserves the original keys.

>>> d = TransformDict(str.lower)
>>> d['Foo'] = 5
>>> d['foo'] == d['FOO'] == d['Foo'] == 5
True
>>> set(d.keys())
{'Foo'}
```

1.1.2 Non-underscore Methods

```
>>> print([name for name in dir(d) if not name.startswith('_')])
['clear', 'copy', 'get', 'getitem', 'items', 'keys', 'pop',
'popitem', 'setdefault', 'transform_func', 'update', 'values']
```

1.1.3 Other Capability

In addition to its case-folding capabilities, the *TransformDict* also had the ability to lookup initial key and the currently stored value:

```
>>> d.getitem('foo')
('Foo', 5)
```

1.1.4 Internal Design

In short, it created a single, dict-like tool that combined:

1. A transformation function such as `str.casefold`
2. A first dictionary: `transformed key --> value`
3. A second dictionary: `transformed key --> original_untransformed_key`

1.1.5 External Design

Various API choices were made:

1. The transformation function was stored in a read-only attribute
2. The two internal dicts are not exposed
3. The combined dict modeled: `untransformed_key -> value`
4. The `items` method modeled: `[(original_untransformed_key, current_value), ...]`
5. The new `getitem` method modeled: `untransformed key --> (original_untransformed_key, value)`
6. The `__missing__` method was not supported.

1.1.6 Test Your Understanding by Making Predictions

```
########### Instrument calls to casefold ###################

count = 0

def reset():
    global count
    count = 0

def show():
    print('{0} calls'.format(count))

def casefold(s):
    global count
    count += 1
    return str.casefold(s)

########### Now, make predictions ##########################

from transform import TransformDict
```

```
d = TransformDict(casefold)
d['RAYMOND'] = 'red';               show()
d['RACHEL'] = 'blue';               show()
d['Rachel'] = 'azure';              show()
print(d['RAChel']);                 show()

print(d.items())                    # case of raymond and rachel
print(d.getitem('racHEL'))          # ? How to get to "rachel"

########### Make more predictions #########################

e = TransformDict(int)
e['12'] = 'twelve'
e['13'] = 'thirteen'
for num in [12, 12.0, b'12', 8, '12.0', 13j, 'hello']:
    print('Looking up: {0!r}'.format(num))
    try:
        print('--> {0}'.format(e[num]))
    except KeyError:
        print('--> number is not in the dictionary')

print(e[12])
print(e[12.0])
print(e[b"12"])
print(e['12.0'])
```

1.1.7 Source Code

```
#########################################################################
###   TransformDict
#########################################################################

from collections import MutableMapping

_sentinel = object()

class TransformDict(MutableMapping):
    '''Dictionary that calls a transformation function when looking
    up keys, but preserves the original keys.

    >>> d = TransformDict(str.lower)
    >>> d['Foo'] = 5
    >>> d['foo'] == d['FOO'] == d['Foo'] == 5
    True
    >>> set(d.keys())
    {'Foo'}
    '''

    __slots__ = ('_transform', '_original', '_data')

    def __init__(self, transform, init_dict=None, **kwargs):
        '''Create a new TransformDict with the given *transform* function.
        *init_dict* and *kwargs* are optional initializers, as in the
        dict constructor.
        '''
        if not callable(transform):
            raise TypeError("expected a callable, got %r" % transform.__class__)
        self._transform = transform
        # transformed => original
        self._original = {}
        self._data = {}
        if init_dict:
            self.update(init_dict)
        if kwargs:
            self.update(kwargs)

    def getitem(self, key):
        'D.getitem(key) -> (stored key, value)'
        transformed = self._transform(key)
        original = self._original[transformed]
        value = self._data[transformed]
        return original, value

    @property
    def transform_func(self):
        "This TransformDict's transformation function"
        return self._transform

    # Minimum set of methods required for MutableMapping

    def __len__(self):
        return len(self._data)

    def __iter__(self):
        return iter(self._original.values())

    def __getitem__(self, key):
        return self._data[self._transform(key)]

    def __setitem__(self, key, value):
        transformed = self._transform(key)
        self._data[transformed] = value
        self._original.setdefault(transformed, key)

    def __delitem__(self, key):
        transformed = self._transform(key)
        del self._data[transformed]
        del self._original[transformed]

    # Methods overriden to mitigate the performance overhead.

    def clear(self):
        'D.clear() -> None.  Remove all items from D.'
        self._data.clear()
        self._original.clear()

    def __contains__(self, key):
        return self._transform(key) in self._data

    def get(self, key, default=None):
        'D.get(k[,d]) -> D[k] if k in D, else d.  d defaults to None.'
        return self._data.get(self._transform(key), default)

    def pop(self, key, default=_sentinel):
        '''D.pop(k[,d]) -> v, remove specified key and return the corresponding value.
```

```
          If key is not found, d is returned if given, otherwise KeyError is raised.
        '''
        transformed = self._transform(key)
        if default is _sentinel:
            del self._original[transformed]
            return self._data.pop(transformed)
        else:
            self._original.pop(transformed, None)
            return self._data.pop(transformed, default)

    def popitem(self):
        '''D.popitem() -> (k, v), remove and return some (key, value) pair
           as a 2-tuple; but raise KeyError if D is empty.
        '''
        transformed, value = self._data.popitem()
        return self._original.pop(transformed), value

    # Other methods

    def copy(self):
        'D.copy() -> a shallow copy of D'
        other = self.__class__(self._transform)
        other._original = self._original.copy()
        other._data = self._data.copy()
        return other

    __copy__ = copy

    def __getstate__(self):
        return (self._transform, self._data, self._original)

    def __setstate__(self, state):
        self._transform, self._data, self._original = state

    def __repr__(self):
        try:
            equiv = dict(self)
        except TypeError:
            # Some keys are unhashable, fall back on .items()
            equiv = list(self.items())
        return '%s(%r, %s)' % (self.__class__.__name__,
                               self._transform, repr(equiv))
```

1.2 Concepts Arising During Design Reviews

1.2.1 Orthogonality

Orthogonality is a strategy for achieving loose coupling between parts so that they remain flexible and one can change without the other.

Combining a hammer with a nail remover is reasonable, but combining a hammer and saw into one tool is unreasonable.

"The basic idea of orthogonality is that things that are not related conceptually should not be related in the system. Parts of the architecture that really have nothing to do with the other, such as the database and the UI, should not need to be changed together. A change to one should not cause a change to the other. " – Andy Hunt

Automobile Example: Orthogonal controls for stopping, starting, turning.

Helicopter Controls Example: Helicopters have four basic controls. The cyclic is the stick you hold in your right hand. Move it, and the helicopter moves in the corresponding direction. Your left hand holds the collective pitch lever. Pull up on this and you increase the pitch on all the blades, generating lift. At the end of the pitch lever is the throttle . Finally you have two foot pedals, which vary the amount of tail rotor thrust and so help turn the helicopter.

Transform Dict:

1. A transformation function such as `str.casefold`
2. A first dictionary: `transformed key --> value`
3. A second dictionary: `transformed key --> original_untransformed_key`

1.2.2 Implied Warranties

In U.S. commercial law, there is a notion that all sales of products carry implied warranties including two that have implications for design reviews.

Warranty of Merchantability

The warranty of merchantability is implied, unless expressly disclaimed by name, or the sale is identified with the phrase “as is” or “with all faults.” To be “merchantable”, the **goods must reasonably conform to an ordinary buyer’s expectations, i.e., they are what they say they are.** For example, a fruit that looks and smells good but has hidden defects would violate the implied warranty of merchantability if its quality does not meet the standards for such fruit “as passes ordinarily in the trade”.

TransformDict: Results of potentially expensive computations go into the data structure with legal way of getting them back out.

Warranty of Fitness for a Particular Purpose

The warranty of fitness for a particular purpose is implied **when a buyer relies upon the seller to select the goods to fit a specific request**. For example, this warranty is violated when a buyer asks a mechanic to provide snow tires and receives tires that are unsafe to use in snow. This implied warranty can also be expressly disclaimed by name, thereby shifting the risk of unfitness back to the buyer.

Vehicle Example: You need a light family pickup truck for small loads and are sold a heavy dump truck that works but is to heavy duty for such tasks.

TransformDict: Always includes a dict tracking original keys whether the application needs it or not (most don’t). It locks in a choice of keeping only the first stored key whether or not the application needs it (existing case-insensitive dicts choose the last stored key and other applications want the set of stored keys or no keys at all).

1.2.3 Requirements versus Realities

Chris Alexander: So it became clear that the free functioning of the system did not purely depend on meeting a set of requirements. It had to do, rather, with the system coming to terms with itself and being in balance with the forces that were generated internal to the system, not in accordance with some arbitrary set of requirements we stated.... What bothered me was that the correct analysis of the ticket booth could not be based purely on one’s goals, that **there were realities emerging from the center of the system itself and that whether you**

succeeded or not had to do with whether you created a configuration that was stable with respect to these realities.

Chris Alexander paraphrased: Correct analysis depends on reconciling your requirements with realities emerging from the system itself.

TransformDict: Was designed to requirements (factor the transform function out of a case-preserving-case-insensitive dict) rather than being designed to user stories and informed by actual use (it was a mistake to propose its direct addition to the standard library without being vetted by real-world users – more of a 0.1 release, rather than a 1.0 release).

1.2.4 Generalization versus HyperGeneralization

1. Factoring is all about extracting redundancy, separating the variable from the invariant, improving reusability and learnabilty by creating a single general purpose tool instead of multiple special purpose tools.
2. Hyper-generalization is when generalizations have gone beyond the point of diminishing returns. Over-engineering adds more complexity than it removes.

The FizzBuzz Enterprise Edition is a stellar example of hyper-generalization.

Another example is closer to home is provided by the `str.startswith` and `str.endswith` methods. Why do we have the weird double parentheses?

```
>>> routes = ['homepage.css', 'index.html', 'funstuff.html', 'weather.xml', 'analytics.php']
>>> [route for route in routes if route.endswith(('.html', '.xml'))]
['index.html', 'funstuff.html', 'weather.xml']
```

The method signatures are:

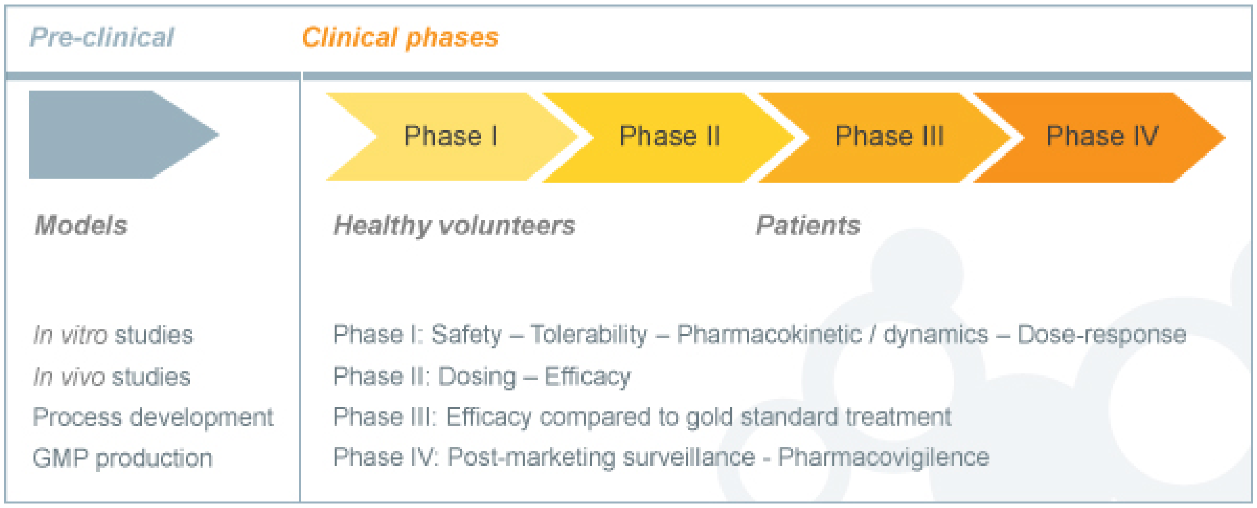
```
S.startswith(prefix[, start[, end]]) -> bool
S.endswith(suffix[, start[, end]]) -> bool
```

The root cause was hyper-generalization of the signature for other string methods:

```
S.count(sub[, start[, end]]) -> int
S.find(sub[, start[, end]]) -> int
S.index(sub[, start[, end]]) -> int
```

1.3 Drug Trials

Some of the concepts from clinical drug trials are loosely related to design reviews.


Pre-clinical
Clinical phases
Phase I
Phase II
Phase III
Phase IV
Models
Healthy volunteers
Patients
In vitro studies
In vivo studies
Process development
GMP production
Phase I: Safety – Tolerability – Pharmacokinetic / dynamics – Dose-response
Phase II: Dosing – Efficacy
Phase III: Efficacy compared to gold standard treatment
Phase IV: Post-marketing surveillance - Pharmacovigilence

1.3.1 Phase I (first studies with humans)

1. find safest dose
2. find most effective way to deliver the drug
3. identify side-effects

TransformDict: Side-effects include loss of access to computed value of function, space and speed overhead, complicated stack traces, and a hard to find method for finding the original key. Some dict invariants are broken and the `__missing__` method is not supported.

1.3.2 Phase II (more patients with specific types of disease)

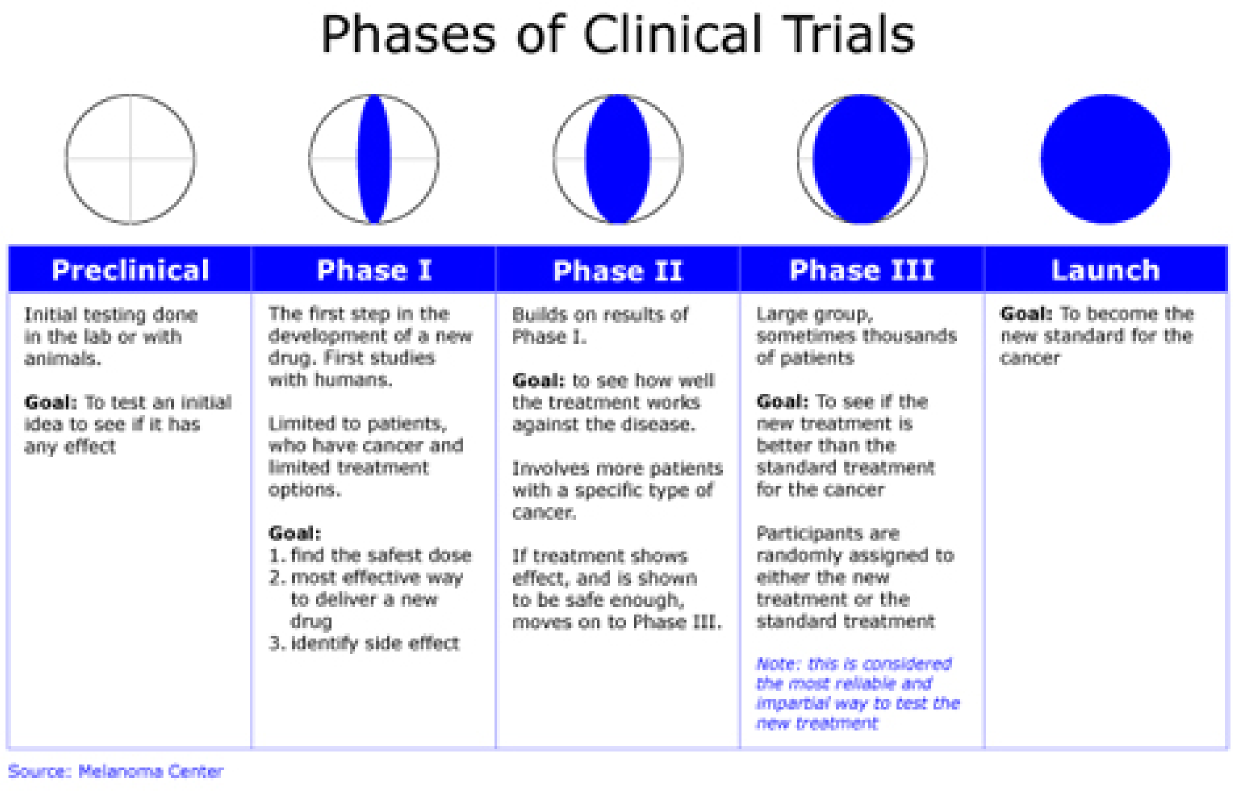
- See how well the treatment works against the disease

TransformDict: It can be made to work for many suggested use cases except where the computed value is needed or where an alternative dictionary type is wanted (counter, defaultdict, shelve, chaindict, etc).

1.3.3 Phase III (random, larger group of patients)

- Determine whether the new treatment is better than the standard treatment

TransformDict: In most of the cases discussed, it was almost always better to use plain dicts and regular functions rather than a more tightly coupled tool.



Phases of Clinical Trials

Preclinical	Phase I	Phase II	Phase III	Launch
Initial testing done in the lab or with animals. **Goal:** To test an initial idea to see if it has any effect	The first step in the development of a new drug. First studies with humans. Limited to patients, who have cancer and limited treatment options. **Goal:** 1. find the safest dose 2. most effective way to deliver a new drug 3. identify side effect	Builds on results of Phase I. **Goal:** to see how well the treatment works against the disease. Involves more patients with a specific type of cancer. If treatment shows effect, and is shown to be safe enough, moves on to Phase III.	Large group, sometimes thousands of patients **Goal:** To see if the new treatment is better than the standard treatment for the cancer Participants are randomly assigned to either the new treatment or the standard treatment *Note: this is considered the most reliable and impartial way to test the new treatment*	**Goal:** To become the new standard for the cancer

Source: Melanoma Center

1.4 Comparative Results

1.4.1 Synonym Example

The goal is to provide synonyms for various keys:

```
alias = dict(
    flee='flee', surrender='flee', giveup='flee',
    fight='fight', attack='fight', shoot='fight',
    talk='talk', negotiate='talk', bargain='talk',
)
```

Is the TransformDict approach fit for this task?

```
from transform import TransformDict

outcome = TransformDict(alias.__getitem__,
    flee = 'Lose all weapons and points but live to fight another day',
    fight = 'Die in a blaze of glory and songs will be sung in your honor',
    talk = 'You are captured while negotiating and never seen again',
)

print('You are faced with three Nossigans who have wronged you,')
print('but you are in the right!  They challenge you to defend your honor')
```

```
action = input('What do you do?  ')
print(outcome[action])
```

Is the TransformDict better than existing solutions?

```
outcome = dict(
    flee = 'Lose all weapons and points but live to fight another day',
    fight = 'Die in a blaze of glory and songs will be sung in your honor',
    talk = 'You are captured while negotiating and never seen again',
)

print('You are faced with three Nossigans who have wronged you,')
print('but you are in the right!  They challenge you to defend your honor')
action = input('What do you do?  ')
print(outcome[alias[action]])
```

The differences here are:

1. The TransformDict is a little more awkward to create with `TransformDict(alias.__getitem__, ...)` compared to the simple dict `{}`.
2. The plain dict is a little more awkward during lookup with `outcome[alias[action]]` versus the TransformDict with `outcome[action]`.
3. The "original key" feature of the TransformDict is unused so the runtime performance cost and double storage requirements are wasted.
4. Tracebacks in the baseline approach are much easier to read.
5. The plain dict approach is much easier to follow through PDB.

1.4.2 Scrubbed Postal Address Example

We have a REST API to a paid service that converts malformed postal addresses into canonical form according to postal regulations.

Lookups are slow (anything over the wire is slower than disk lookups) and each lookup incurs a monetary cost .

Accordingly we want to limit the number of calls to the address scrubber.

Assume we are a police department with a large dictionary mapping canonical 2 million street addresses to the name of the occupant:

```
from internal.mail_utils import scrub_address

occupant = {
    '516 Mansion Ct Unit 1204\nSanta Clara CA 95054-4331\n': 'Raymond Hettinger',
    '1628 Clover Cir\nMountain View CA 94043-1028\n': 'Terry Prachett',
    '781 Maple St Unit 318\nSunnyvale CA 94671-2217\n': 'Luther Blissett',
}
```

Now, the police have obtained a suspicious location:

```
suspect_address = '''\
      516 Mannsion Court
      Apartment 1204
      Santaclara Calif, 95054
'''
```

Is the TransformDict fit for this task?

```python
from transform import TransformDict

who_is_at = TransformDict(scrub_address, occupant)

suspect = who_is_at[suspect_address]
print('Suspect is:', suspect)
print('First reported alias address is:', who_is_at.getitem(suspect_address)[0])
```

Is the TransformDict better than existing solutions?

```python
suspect_address = scrub_address(suspect_address)
suspect = occupant[suspect_address]
print('Suspect is:', suspect)
print('Scrubbed address is:', suspect_address)
```

The TransformDict has some immediately evident issues:

1. We need to create a second dict name when the first dict is already has the preferred name.
2. Creating the TransformDict requires calling the REST API once for *every* address even though the inputs are already in canonical form. This is expensive and slow (two million unnecessary calls to the REST API just to instantiate the TransformDict). The API precludes use of previously computed transformations.
3. The TransformDict can only report the "first reported alias address" which is not an interesting value. Storing that uninteresting data doubles the size of the application.
4. The TransformDict has no way of reporting the scrubbed address which is problematic because it is the interesting value and because we paid for that address and may have uses for it elsewhere.

1.5 Summary

One design reviewer summarized her thoughts like this: "There is a learning curve to be climbed to figure out what it does, how to use it, and what the applications [are]. But, the [working out the same] examples with plain dicts requires only basic knowledge." – Patricia

1.5.1 Emergent Issues

Biggest issue: Code without the TD generally clearer, faster, more flexible, more intuitive, and more debuggable. It was only slightly less succinct.

Essential data access is walled-off. There is no way to load previously computed transformations, nor is there a way access the transformed values once they have been computed.

User does not have a choice of dict style: chainmap, counter, defaultdict, shelve, or OrderedDict

User not given a choice for storage of original keys: no storage, first key only, last key only, set of all keys

User has only one accessor method to original key: no get(), keys(), items(), etc

There is no accessor for the transformed value which is problematic if it is expensive to compute.

Unintuitive invariant loss: "k in td" does not imply that "k in list(td.keys())"

Tracebacks and debugger sessions became more complicated with the TransformDict

In most of the use cases that were explored by the design reviewers, the TransformDict proved to be a distractor from better solutions without the TransformDict.

1.5.2 Formal Rejection of PEP 455 on Python-Dev

See the rationale at https://mail.python.org/pipermail/python-dev/2015-May/140003.html and for a earlier partial review, see https://mail.python.org/pipermail/python-dev/2013-October/129937.html .