Fun with Python's Newer Tools

by Raymond Hettinger
@raymondh

Collections.Counter

Tool for making rapid tallies or counts

Modeled after:

- Multisets in C++
- Bags in Smalltalk and Objective C

Very flexible, unrestricted implementation as a dictionary

You can put anything in it:

- Count with positive and negative numbers
- Count with decimals, floats, or fractions
- It is just a dictionary

Counter is a dictionary

Simple design as a dict subclass

With __missing__() that returns zero

```
c[x] += 1          # easy to tally
```

Has __delitem__() to match __missing__()

```
del c[x]           # easy to delete
```

Counter – Basic Implementation

```
class Counter(dict):

    def __missing__(self, key):
        'The count of elements not in the Counter is zero.'
        return 0

    def __delitem__(self, elem):
        'Like dict.__delitem__() but does not raise KeyError for missing values.'
        if elem in self:
            super().__delitem__(elem)
```

```
>>> c = Counter()
>>> c['x'] += 1
>>> c['y'] += 2
>>> c['z'] = 10
>>> c
{'y': 2, 'x': 1, 'z': 10}
>>> del c['w']
```

Example: Unittest Result Counter

```
results = Counter()
for test in test_suite:
    if test():
        results['succeeded'] += 1
    results['attempted'] += 1
```

Example: Word Count

```
with open('sometext.txt') as f:
    text = f.read().lower()

words = re.findall('\w+', text)

print( Counter(words).most_common(50) )
```

Convenience Methods

most_common([n])

- Returns sorted list of the n highest counts
- Handles the common use-case of finding most popular entries
- Reduces the code to a simple one-liner
- Implemented using either sorted() or heapq.nlargest()

elements()

- Lists all of the contents individually
- If an element has a count of three, it is emitted three times
- Works like a Bag in Smalltalk
- Differs from __iter__ which returns pairs: (elem, count)

One Kind of Counter Math

Regular math (non-saturating)

- Used when counts are allowed to go negative
- cnt.update(d) # add counters
- cnt.substract(d) # difference two counters

```
>>> x = Counter(a=1, b=1)
>>> x.subtract(a=1, c=1)
>>> x
Counter({'b': 1, 'a': 0, 'c': -1})
```

Another Kind of Counter Math

Multiset use case:

- With multisets, the counts are always positive
- Math operations with omit zero or negative counts from the result
- Operations are: `+ - & |`
- The subtraction operation is said to be saturating
- When the counts are all one, works just like regular sets:

```
>>> {'a', 'b'} - {'a', 'c'}
{'b'}
>>> Counter(a=1, b=1) - Counter(a=1, c=1)
Counter({'b': 1})
```

collections.namedtuple()

Works just like a regular tuple

And let's you assign names to each field

Can profoundly improve your code base:

- Makes the code self-documenting
- Makes the printed __repr__ intelligible
- Let's you change tuple order without affecting client code

One of the single best changes you can make to existing code

Example from Doctest

```
TestResults = namedtuple('TestResults', 'failed attempted')


 >>> print(doctest.testmod)
TestResults(failed=0, attempted=7)
```

Simple Implementation

```
class TestResults(tuple):
    'TestResults(failed, attempted)'

    __slots__ = ()

    _fields = ('failed', 'attempted')

    def __new__(_cls, failed, attempted):
        'Create new instance of TestResults(failed, attempted)'
        return _tuple.__new__(_cls, (failed, attempted))

    def __repr__(self):
        'Return a nicely formatted representation string'
        return self.__class__.__name__ + '(failed=%r, attempted=%r)' % self

    failed = _property(_itemgetter(0), doc='Alias for field number 0')
    attempted = _property(_itemgetter(1), doc='Alias for field number 1')
```

Convenience Methods

_asdict()

- Turns a named tuple into a dictionary

```
{'failed': 0, 'attempted': 7}
```

- Principle: key/value pairs should be convertible to dicts

_replace()

- Creates a new named tuple with altered values

```
>>> result._replace(failed = 2)
TestResult(failed=2, attempted=7)
```

- Much cleaner than:

```
(2,) + result[1]
```

Pro Tip: Using the Field Structure

```
LabeledResult = namedtuple(
                'LabeledResult,
                 TestResult._fields + ('test_name',))
```

Pro Tip: Instance Prototypes

The _replace() method can be used to modify prototype instances:

```
prototype = Cell(color='red', size=10, border=False)

intro = prototype._replace(size=20)
lead = prototype._replace(color='blue', border=True)
summary = prototype._replace(size=20, border=True)
```

Pro Tip: subclass a named tuple

```
>>> class Point(namedtuple('Point', 'x y')):
...     __slots__ = ()
...     @property
...     def hypot(self):
...         return (self.x ** 2 + self.y ** 2) ** 0.5
...     def __str__(self):
...         return 'Point: x=%6.3f  y=%6.3f  hypot=%6.3f'
```

```
>>> for p in Point(3, 4), Point(14, 5/7):
...     print(p)
Point: x= 3.000  y= 4.000  hypot= 5.000
Point: x=14.000  y= 0.714  hypot=14.018
```

Pro Tip: How to make an Enum type

```
Color = namedtuple('Color',
          'red orange yellow green blue indigo') \
          ._make(range(6))


>>> Color.red
0
>>> Color.green
3
```

Caching

Simple unbounded cache:

```
def f(*args, cache={})
    if args in cache:
        return cache[args]
    result = big_computation(*args)
    cache[args] = result
    return result
```

But, that would grow without bound

LRU Cache

To limit its size, we need to make room for new entries

One strategy is to remove the least-recently used entry

Provided in the standard library as a decorator:

```
from functools import lru_cache

@lru_cache(maxsize=100)
def big_computation(*args):
    ...
```

Dynamic Programming with a Cache

```
@lru_cache()
def fibonacci(n):
    if n <= 1:
        return n
    return fibonacci(n-1) + fibonacci(n-2)

print(fibonacci(100))
```

New String Formatting Syntax

```
>>> 'Page {0}, Line {1}'.format(10, 20)
'Page 10, Line 20’
```

```
>>> cite = dict(page=20, line=10)
>>> 'Page {page}, Line {line}'.format(**cite)
'Page 20, Line 10'
```

Fill, Align, and Width

- Specify width after a colon
- Alignment: < ^ >
- Optional fill charcter before alignment

```
>>> '{page:8}'.format(**d)
'      12'
>>> '{page:<8}'.format(**d)
'12      '
>>> '{page:^8}'.format(**d)
'   12   '
>>> '{page:#^8}'.format(**d)
'###12###'
```

Sign, Zero, Comma, Width, Precision

```
>>> from math import pi

>>> '{0:+08.2f}'.format(pi)
'+0003.14’

>>> '{0:,.2f}'.format(pi*1E6)
'3,141,592.65'
```

Questions and Answers