defaultdict and dict.__missing__

New in Python 2.5

Author: Jason Kirtland
Date: January 13th, 2009

defaults can be handy

```
strings = ('puppy', 'kitten', 'puppy', 'puppy',
           'weasel', 'puppy', 'kitten', 'puppy')
counts = {}

for kw in strings:
    counts[kw] += 1
assert counts['weasel'] == 1
```

missing keys

```
>>> counts = dict()
>>> counts
{}
>>> counts['puppy'] += 1
```

missing keys

```
>>> counts = dict()
>>> counts
{}
>>> counts['puppy'] += 1
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'puppy'
```

guard

```
strings = ('puppy', 'kitten', 'puppy', 'puppy',
           'weasel', 'puppy', 'kitten', 'puppy')
counts = {}

for kw in strings:
    if kw not in counts:
        counts[kw] = 1
    else:
        counts[kw] += 1
assert counts['weasel'] == 1
```

setdefault

```
strings = ('puppy', 'kitten', 'puppy', 'puppy',
           'weasel', 'puppy', 'kitten', 'puppy')
counts = {}

for kw in strings:
    counts.setdefault(kw, 0)
    counts[s] += 1
assert counts['weasel'] == 1
```

collections.defaultdict

`defaultdict` is like a `dict`, but is instantiated with a type

```
>>> from collections import defaultdict
>>> dd = defaultdict(list)
>>> dd
defaultdict(<type 'list'>, {})
```

collections.defaultdict

instances of type fill in for missing keys

```
>>> from collections import defaultdict
>>> dd = defaultdict(list)
>>> dd
defaultdict(<type 'list'>, {})
>>> dd['foo']
[]
>>> dd
defaultdict(<type 'list'>, {'foo': []})
>>> dd['bar'].append('quux')
>>> dd
defaultdict(<type 'list'>, {'foo': [], 'bar': ['quux']})
```

collections.defaultdict

but only for `dictionary[key]`

```
>>> from collections import defaultdict
>>> dd = defaultdict(list)
>>> 'something' in dd
False
>>> dd.pop('something')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'pop(): dictionary is empty'
>>> dd.get('something')
>>> dd['something']
[]
```

collections.defaultdict

`defaultdict` will actually take any no-arg callable

```
>>> from collections import defaultdict
>>> def zero():
...     return 0
...
>>> dd = defaultdict(zero)
>>> dd
defaultdict(<function zero at 0xb7ed2684>, {})
>>> dd['foo']
0
>>> dd
defaultdict(<function zero at 0xb7ed2684>, {'foo': 0})
```

incrementing with defaultdict

```
from collections import defaultdict

strings = ('puppy', 'kitten', 'puppy', 'puppy',
           'weasel', 'puppy', 'kitten', 'puppy')
counts = defaultdict(lambda: 0)

for s in strings:
    counts[s] += 1
assert counts['weasel'] == 1
```

defaultdict uses __missing__

```
>>> from collections import defaultdict
>>> print defaultdict.__missing__.__doc__
__missing__(key) # Called by __getitem__ for missing key; pseudo-code:
  if self.default_factory is None: raise KeyError(key)
  self[key] = value = self.default_factory()
  return value
```

The extended version of `__getitem__` that checks for `__missing__` is actually in the base class, `dict`. The functionality is available to all dictionaries.

dict.__missing__

`__missing__` is supported by `dict`, but

```
>>> print dict.__missing__.__doc__
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: type object 'dict' has no attribute '__missing__'
```

dict.__missing__

`__missing__` is supported by `dict`, but

```
>>> print dict.__missing__.__doc__
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: type object 'dict' has no attribute '__missing__'
>>> {}.__missing__
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'dict' object has no attribute '__missing__'
```

The method is supported but not present in the base class.

Sub-classing dict

There's no docstring, so let's experiment.

```
>>> class Missing(dict):
...     def __missing__(self, key):
...         return 'missing'
...
>>> d = Missing()
>>> d
{}
>>> d['foo']
'missing'
>>> d
{}
```

Sub-classing dict

For `defaultdict`-like behavior, set a value.

```
>>> class Defaulting(dict):
...     def __missing__(self, key):
...         self[key] = 'default'
...         return 'default'
...
>>> d = Defaulting()
>>> d
{}
>>> d['foo']
'default'
>>> d
{'foo': 'default'}
```

defaultdict in older Pythons

Duplicating `defaultdict` in older Pythons is easy. It won't be as fast as Python 2.5's, but the functionality will be the same.

defaultdict in older Pythons

First, `__getitem__` needs to consult `__missing__` on a miss.

```
class defaultdict(dict):
    def __getitem__(self, key):
        try:
            return dict.__getitem__(self, key)
        except KeyError:
            return self.__missing__(key)
```

defaultdict in older Pythons

Second, need a `__missing__` that sets default values.

```
class defaultdict(dict):
    def __getitem__(self, key):
        try:
            return dict.__getitem__(self, key)
        except KeyError:
            return self.__missing__(key)
    def __missing__(self, key):
        self[key] = value = self.default_factory()
        return value
```

defaultdict in older Pythons

Third, `__init__` needs to take a type or callable.

```
class defaultdict(dict):
    def __init__(self, default_factory=None, *a, **kw):
        dict.__init__(self, *a, **kw)
        self.default_factory = default_factory
    def __getitem__(self, key):
        try:
            return dict.__getitem__(self, key)
        except KeyError:
            return self.__missing__(key)
    def __missing__(self, key):
        self[key] = value = self.default_factory()
        return value
```

defaultdict in older Pythons

```
try:
    from collections import defaultdict
except ImportError:
    class defaultdict(dict):
      def __init__(self, default_factory=None, *a, **kw):
          dict.__init__(self, *a, **kw)
          self.default_factory = default_factory
      def __getitem__(self, key):
          try:
              return dict.__getitem__(self, key)
          except KeyError:
              return self.__missing__(key)
      def __missing__(self, key):
          self[key] = value = self.default_factory()
          return value
```

Thanks!

A complete emulation with error checking and minutia is available in the Python Cookbook:

http://aspn.activestate.com/ASPN/Cookbook/Python/Recipe/523034