Module 09 – Collections

Agenda

- ChainMap
- Counter
- deque
- defaultdict
- namedtuple()
- OrderedDict
- UserDict
- UserList
- UserString

ChainMap objects

- A **ChainMap** class is provided for quickly linking a number of mappings so they can be treated as a single unit. It is often much faster than creating a new dictionary and running multiple update() calls.
- The class can be used to simulate nested scopes and is useful in templating.
- class **collections.ChainMap(*maps)**
  - A **ChainMap** groups multiple dicts or other mappings together to create a single, updateable view. If no maps are specified, a single empty dictionary is provided so that a new chain always has at least one mapping.
  - The underlying mappings are stored in a list. That list is public and can be accessed or updated using the maps attribute. There is no other state.
  - Lookups search the underlying mappings successively until a key is found. In contrast, writes, updates, and deletions only operate on the first mapping.
  - A **ChainMap** incorporates the underlying mappings by reference. So, if one of the underlying mappings gets updated, those changes will be reflected in **ChainMap**.
  - All of the usual dictionary methods are supported. In addition, there is a maps attribute, a method for creating new **subcontexts**, and a property for accessing all but the first mapping:

`maps`

- A user updateable list of mappings.
- The list is ordered from first-searched to last-searched.
- It is the only stored state and can be modified to change which mappings are searched.
- The list should always contain at least one mapping.

new_child(*m=None, **kwargs*)

- Returns a new **ChainMap** containing a new map followed by all of the maps in the current instance.

- If **m** is specified, it becomes the new map at the front of the list of mappings; if not specified, an empty dict is used, so that a call to **d.new_child()** is equivalent to: **ChainMap({}, *d.maps)**.

- If any keyword arguments are specified, they update passed map or new empty dict.
- This method is used for creating **subcontexts** that can be updated without altering values in any of the parent mappings.

`parents`

- Property returning a new **ChainMap** containing all of the maps in the current instance except the first one. This is useful for skipping the first map in the search.
- Use cases are similar to those for the nonlocal keyword used in nested scopes.
- The use cases also parallel those for the built-in super() function. A reference to **d.parents** is equivalent to: **ChainMap(*d.maps[1:]).**

ChainMap Examples and Recipes

Demo

Example of simulating Python's internal lookup chain:

```
import builtins
pylookup = ChainMap(locals(), globals(), vars(builtins))
```

Example of letting user specified command-line arguments take precedence over environment variables which in turn take precedence over default values:

```
import os, argparse

defaults = {'color': 'red', 'user': 'guest'}

parser = argparse.ArgumentParser()
parser.add_argument('-u', '--user')
parser.add_argument('-c', '--color')
namespace = parser.parse_args()
command_line_args = {k: v for k, v in vars(namespace).items() if v is not None}

combined = ChainMap(command_line_args, os.environ, defaults)
print(combined['color'])
print(combined['user'])
```

Example patterns for using the ChainMap class to simulate nested contexts:

```
c = ChainMap()        # Create root context
d = c.new_child()     # Create nested child context
e = c.new_child()     # Child of c, independent from d
e.maps[0]             # Current context dictionary -- like Python's locals()
e.maps[-1]            # Root context -- like Python's globals()
e.parents             # Enclosing context chain -- like Python's nonlocals

d['x'] = 1            # Set value in current context
d['x']                # Get first key in the chain of contexts
del d['x']            # Delete from current context
list(d)               # All nested values
k in d                # Check all nested values
len(d)                # Number of nested values
d.items()             # All nested items
dict(d)               # Flatten into a regular dictionary
```

The **ChainMap** class only makes updates (writes and deletions) to the first mapping in the chain while lookups will search the full chain. However, if deep writes and deletions are desired, it is easy to make a subclass that updates keys found deeper in the chain:

```
class DeepChainMap(ChainMap):
    'Variant of ChainMap that allows direct updates to inner scopes'

    def __setitem__(self, key, value):
        for mapping in self.maps:
            if key in mapping:
                mapping[key] = value
                return
        self.maps[0][key] = value

    def __delitem__(self, key):
        for mapping in self.maps:
            if key in mapping:
                del mapping[key]
                return
        raise KeyError(key)

>>> d = DeepChainMap({'zebra': 'black'}, {'elephant': 'blue'}, {'lion': 'yellow'})
>>> d['lion'] = 'orange'         # update an existing key two levels down
>>> d['snake'] = 'red'           # new keys get added to the topmost dict
>>> del d['elephant']            # remove an existing key one level down
>>> d                            # display result
DeepChainMap({'zebra': 'black', 'snake': 'red'}, {}, {'lion': 'orange'})
```

Counter objects

class **collections.Counter([iterable-or-mapping])**

- A Counter is a dict subclass for counting hashable objects. It is a collection where elements are stored as dictionary keys and their counts are stored as dictionary values. Counts are allowed to be any integer value including zero or negative counts. The **Counter** class is similar to bags or multisets in other languages.

- Changed in version 3.7: As a dict subclass, Counter inherited the capability to remember insertion order. Math operations on Counter objects also preserve order. Results are ordered according to when an element is first encountered in the left operand and then by the order encountered in the right operand.

- Counter objects support additional methods beyond those available for all dictionaries:

elements()

Return an iterator over elements repeating each as many times as its count. Elements are returned in the order first encountered. If an element’s count is less than one, elements() will ignore it.

```
c = Counter(a=4, b=2, c=0, d=-2)
sorted(c.elements())
['a', 'a', 'a', 'a', 'b', 'b']
```

`most_common([n])`

Return a list of the n most common elements and their counts from the most common to the least. If n is omitted or None, most_common() returns all elements in the counter. Elements with equal counts are ordered in the order first encountered:

```
Counter('abracadabra').most_common(3)
[('a', 5), ('b', 2), ('r', 2)]
```

subtract([*iterable-or-mapping*])

Elements are subtracted from an iterable or from another mapping (or counter). Like dict.update() but subtracts counts instead of replacing them. Both inputs and outputs may be zero or negative.

```
c = Counter(a=4, b=2, c=0, d=-2)
d = Counter(a=1, b=2, c=3, d=4)
c.subtract(d)
c
Counter({'a': 3, 'b': 0, 'c': -3, 'd': -6})
```

- **`total`**`()`

Compute the sum of the counts.

```
c = Counter(a=10, b=5, c=0)
c.total()
15
```

- **fromkeys(iterable)**

  This class method is not implemented for Counter objects.

- **update([iterable-or-mapping])**

  Elements are counted from an iterable or added-in from another mapping (or counter). Like dict.update() but adds counts instead of replacing them. Also, the iterable is expected to be a sequence of elements, not a sequence of (key, value) pairs.

Common patterns for working with Counter objects:

```
c.total()                       # total of all counts
c.clear()                       # reset all counts
list(c)                         # list unique elements
set(c)                          # convert to a set
dict(c)                         # convert to a regular dictionary
c.items()                       # convert to a list of (elem, cnt) pairs
Counter(dict(list_of_pairs))    # convert from a list of (elem, cnt) pairs
c.most_common()[:-n-1:-1]       # n least common elements
+c                              # remove zero and negative counts
```

Counter objects – cont’d

Counters were primarily designed to work with positive integers to represent running counts; however, care was taken to not unnecessarily preclude use cases needing other types or negative values. To help with those use cases, this section documents the minimum range and type restrictions.

- The Counter class itself is a dictionary subclass with no restrictions on its keys and values. The values are intended to be numbers representing counts, but you could store anything in the value field.

- The **most_common()** method requires only that the values be orderable.

- For in-place operations such as c[key] += 1, the value type need only support addition and subtraction. So fractions, floats, and decimals would work and negative values are supported. The same is also true for update() and subtract() which allow negative and zero values for both inputs and outputs.

- The multiset methods are designed only for use cases with positive values. The inputs may be negative or zero, but only outputs with positive values are created. There are no type restrictions, but the value type needs to support addition, subtraction, and comparison.

- The elements() method requires integer counts. It ignores zero and negative counts.

deque objects

class collections.deque([iterable[, maxlen]])

- Returns a new deque object initialized left-to-right (using append()) with data from iterable. If iterable is not specified, the new deque is empty.

- Deques are a generalization of stacks and queues (the name is pronounced “deck” and is short for “double-ended queue”). Deques support thread-safe, memory efficient appends and pops from either side of the deque with approximately the same O(1) performance in either direction.

- Though list objects support similar operations, they are optimized for fast fixed-length operations and incur O(n) memory movement costs for pop(0) and insert(0, v) operations which change both the size and position of the underlying data representation.

deque objects – cont’

- If maxlen is not specified or is None, deques may grow to an arbitrary length. Otherwise, the deque is bounded to the specified maximum length. Once a bounded length deque is full, when new items are added, a corresponding number of items are discarded from the opposite end.

- Bounded length deques provide functionality similar to the tail filter in Unix. They are also useful for tracking transactions and other pools of data where only the most recent activity is of interest.

- Deque objects support the following methods:

deque objects – cont’d

- **append(x)**
  Add x to the right side of the deque.

- **appendleft(x)**
  Add x to the left side of the deque.

- **clear()**
  Remove all elements from the deque leaving it with length 0.

- **copy()**
  Create a shallow copy of the deque.

- **count(x)**
  Count the number of deque elements equal to x.

- **extend(iterable)**
  Extend the right side of the deque by appending elements from the iterable argument.

- **extendleft(iterable)**
  Extend the left side of the deque by appending elements from iterable. Note, the series of left appends results in reversing the order of elements in the iterable argument.

- **index(x[, start[, stop]])**
  Return the position of x in the deque (at or after index start and before index stop). Returns the first match or raises ValueError if not found.

deque objects – cont’d

- **insert(i, x)**
  Insert x into the deque at position i. If the insertion would cause a bounded deque to grow beyond maxlen, an IndexError is raised.

- **pop()**
  Remove and return an element from the right side of the deque. If no elements are present, raises an IndexError.

- **popleft()**
  Remove and return an element from the left side of the deque. If no elements are present, raises an IndexError.

- **remove(value)**
  Remove the first occurrence of value. If not found, raises a ValueError.

- **reverse()**
  Reverse the elements of the deque in-place and then return None.

- **rotate(n=1)**
  Rotate the deque n steps to the right. If n is negative, rotate to the left.

- **maxlen**
  Maximum size of a deque or None if unbounded.

deque objects Examples and Recipes

Demo

Bounded length deques provide functionality similar to the tail filter in Unix:

```
def tail(filename, n=10):
    'Return the last n lines of a file'
    with open(filename) as f:
        return deque(f, n)
```

Another approach to using deques is to maintain a sequence of recently added elements by appending to the right and popping to the left:

```
def moving_average(iterable, n=3):
    # moving_average([40, 30, 50, 46, 39, 44]) --> 40.0 42.0 45.0 43.0
    # https://en.wikipedia.org/wiki/Moving_average
    it = iter(iterable)
    d = deque(itertools.islice(it, n-1))
    d.appendleft(0)
    s = sum(d)
    for elem in it:
        s += elem - d.popleft()
        d.append(elem)
        yield s / n
```

A round-robin scheduler can be implemented with input iterators stored in a deque. Values are yielded from the active iterator in position zero. If that iterator is exhausted, it can be removed with popleft(); otherwise, it can be cycled back to the end with the rotate() method:

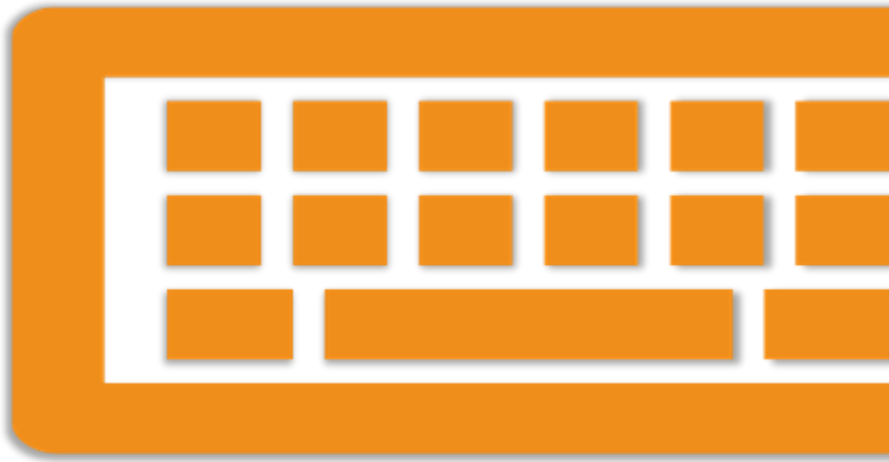
```
def roundrobin(*iterables):
    "roundrobin('ABC', 'D', 'EF') --> A D E B F C"
    iterators = deque(map(iter, iterables))
    while iterators:
        try:
            while True:
                yield next(iterators[0])
                iterators.rotate(-1)
        except StopIteration:
            # Remove an exhausted iterator.
            iterators.popleft()
```

The rotate() method provides a way to implement deque slicing and deletion. For example, a pure Python implementation of del d[n] relies on the rotate() method to position elements to be popped:

```
def delete_nth(d, n):
    d.rotate(-n)
    d.popleft()
    d.rotate(n)
```

Lab 01

Lab



defaultdict objects

class collections.defaultdict(default_factory=None, /[, ...])

- Return a new dictionary-like object. defaultdict is a subclass of the built-in dict class. It overrides one method and adds one writable instance variable. The remaining functionality is the same as for the dict class and is not documented here.

- The first argument provides the initial value for the default_factory attribute; it defaults to None. All remaining arguments are treated the same as if they were passed to the dict constructor, including keyword arguments.

- defaultdict objects support the following method in addition to the standard dict operations:

defaultdict objects— cont’d

__missing__(key)

- If the **default_factory** attribute is None, this raises a **KeyError** exception with the key as argument.
- If **default_factory** is not None, it is called without arguments to provide a default value for the given key, this value is inserted in the dictionary for the key, and returned.
- If calling **default_factory** raises an exception this exception is propagated unchanged.
- This method is called by the **__getitem__()** method of the **dict** class when the requested key is not found; whatever it returns or raises is then returned or raised by **__getitem__().**
- Note that __missing__() is not called for any operations besides **__getitem__().** This means that get() will, like normal dictionaries, return None as a default rather than using **default_factory.**

defaultdict objects support the following instance variable:

default_factory

- This attribute is used by the __missing__() method; it is initialized from the first argument to the constructor, if present, or to None, if absent.

defaultdict Examples

Demo

Using list as the default_factory, it is easy to group a sequence of key-value pairs into a dictionary of lists:

```
s = [('yellow', 1), ('blue', 2), ('yellow', 3), ('blue', 4), ('red', 1)]
d = defaultdict(list)
for k, v in s:
    d[k].append(v)

sorted(d.items())
```

Setting the default_factory to int makes the defaultdict useful for counting (like a bag or multiset in other languages):

```
s = 'mississippi'
d = defaultdict(int)
for k in s:
    d[k] += 1

sorted(d.items())
```

The function int() which always returns zero is just a special case of constant functions. A faster and more flexible way to create constant functions is to use a lambda function which can supply any constant value (not just zero):

```
def constant_factory(value):
    return lambda: value
d = defaultdict(constant_factory('<missing>'))
d.update(name='John', action='ran')
'%(name)s %(action)s to %(object)s' % d
```

Setting the default_factory to set makes the defaultdict useful for building a dictionary of sets:

```
s = [('red', 1), ('blue', 2), ('red', 3), ('blue', 4), ('red', 1), ('blue', 4)]
d = defaultdict(set)
for k, v in s:
    d[k].add(v)

sorted(d.items())
```

namedtuple()

The **namedtuple()** function in the collections module is a factory function used to create tuple subclasses with named fields. It allows you to define tuples where each position has a specific meaning and can be accessed by name instead of index.

Here is a summary of the important points about namedtuple():

- Syntax: **collections.namedtuple(typename, field_names, *, rename=False, defaults=None, module=None)**
- The function returns a new tuple subclass named typename. This subclass can be used to create tuple-like objects with named fields.
- **field_names** is a sequence of strings that define the names of the fields in the tuple. It can also be a single string with field names separated by whitespace and/or commas.
- Field names must be valid Python identifiers and cannot start with an underscore. They should consist of letters, digits, and underscores, but not start with a digit or underscore, and should not be a Python keyword.

namedtuple() – cont’d

- If **rename** is set to **True**, invalid field names are automatically replaced with positional names. This is useful to avoid conflicts with keywords or duplicate field names.
- **defaults** can be **None** or an iterable of default values. Default values are assigned to the rightmost parameters. For example, if the field names are ['x', 'y', 'z'] and the defaults are (1, 2), then x will be a required argument, y will default to 1, and z will default to 2.
- The **module** parameter allows you to specify the value for the __module__ attribute of the named tuple. This can be useful for pickling and other purposes.
- Named tuple instances do not have per-instance dictionaries, making them lightweight and memory-efficient.
- The named tuple class should be assigned to a variable with the same name as typename to support pickling.
- The behavior of **namedtuple()** has evolved in different Python versions. Changes include the addition of **rename**, **defaults**, and **module** parameters, as well as the removal of **verbose** and **_source** attributes. Please refer to the specific Python version documentation for more details.
- Named tuples are a convenient way to create structured data containers that improve code readability and self-documentation. They provide a balance between the readability of dictionaries and the memory efficiency of tuples.

namedtuple() basic example

Demo

namedtuple() – basic example

```
Point = namedtuple('Point', ['x', 'y'])
p = Point(11, y=22)     # instantiate with positional or keyword arguments
p[0] + p[1]             # indexable like the plain tuple (11, 22)

x, y = p                # unpack like a regular tuple
x, y

p.x + p.y               # fields also accessible by name

p                       # readable __repr__ with a name=value style
```

namedtuple() – cont’d

In addition to the methods inherited from tuples, named tuples support three additional methods and two attributes. To prevent conflicts with field names, the method and attribute names start with an underscore.

- **classmethod somenamedtuple._make(iterable)**
  - Class method that makes a new instance from an existing sequence or iterable.

```
t = [11, 22]
Point._make(t)
```

- somenamedtuple._asdict()
  - Return a new dict which maps field names to their corresponding values:

```
p = Point(x=11, y=22)
p._asdict()
```

namedtuple() – cont'd

In addition to the methods inherited from tuples, named tuples support three additional methods and two attributes. To prevent conflicts with field names, the method and attribute names start with an underscore.

- somenamedtuple._replace(**kwargs)
  - Return a new instance of the named tuple replacing specified fields with new values:

```
p = Point(x=11, y=22)
p._replace(x=33)


for partnum, record in inventory.items():
    inventory[partnum] = record._replace(price=newprices[partnum], timestamp=time.now())
```

- somenamedtuple._fields
  - Tuple of strings listing the field names. Useful for introspection and for creating new named tuple types from existing named tuples.

```
p._fields           # view the field names


Color = namedtuple('Color', 'red green blue')
Pixel = namedtuple('Pixel', Point._fields + Color._fields)
Pixel(11, 22, 128, 255, 0)
```

namedtuple() – cont’d

- **somenamedtuple._field_defaults**
  - Dictionary mapping field names to default values.

```
Account = namedtuple('Account', ['type', 'balance'], defaults=[0])
Account._field_defaults

Account('premium')
```

- To retrieve a field whose name is stored in a string, use the getattr() function:

```
getattr(p, 'x')
```

- To convert a dictionary to a named tuple, use the double-star-operator (as described in Unpacking Argument Lists):

```
d = {'x': 11, 'y': 22}
Point(**d)
```

OrderedDict objects

Ordered dictionaries are similar to regular dictionaries but have additional capabilities related to ordering operations. They were more significant before Python 3.7 when the built-in dict class gained the ability to remember insertion order.

Here are some key points about ordered dictionaries:

- Regular dictionaries prioritize mapping operations, while tracking insertion order was secondary.
- **OrderedDict** is designed to handle reordering operations efficiently, with less emphasis on space efficiency, iteration speed, and update operation performance.
- The **OrderedDict** algorithm is well-suited for implementing LRU (Least Recently Used) caches and can handle frequent reordering operations better than regular dictionaries.
- The equality operation for **OrderedDict** considers matching order, while a regular dict can emulate order-sensitive equality by checking if p == q and all(k1 == k2 for k1, k2 in zip(p, q)).
- The **popitem()** method of **OrderedDict** has a different signature, allowing an optional argument to specify which item should be popped.

OrderedDict objects – cont’d

- Regular dictionaries can emulate OrderedDict's **od.popitem(last=True)** with **d.popitem(),** which is guaranteed to pop the rightmost (last) item.
- Regular dictionaries can emulate OrderedDict's **od.popitem(last=False)** with **(k := next(iter(d)), d.pop(k)),** which returns and removes the leftmost (first) item if it exists.
- OrderedDict provides a **move_to_end()** method to efficiently reposition an element to an endpoint.
- Regular dictionaries can emulate OrderedDict's **od.move_to_end(k, last=True)** with **d[k] = d.pop(k),** which moves the key and its associated value to the rightmost (last) position.
- Regular dictionaries do not have an efficient equivalent for OrderedDict's **od.move_to_end(k, last=False),** which moves the key and its associated value to the leftmost (first) position.
- Until Python 3.8, regular dictionaries lacked a **__reversed__()** method, which is available in OrderedDict.
- In summary, ordered dictionaries provide specific capabilities for reordering operations and maintaining the order of items. Although the built-in **dict** class gained similar capabilities in Python 3.7, OrderedDict remains useful in scenarios that require frequent reordering or order-sensitive operations.

OrderedDict objects – cont’d

class collections.OrderedDict([items])

Return an instance of a dict subclass that has methods specialized for rearranging dictionary order.

- **popitem(last=True)**
  - The popitem() method for ordered dictionaries returns and removes a (key, value) pair. The pairs are returned in LIFO order if last is true or FIFO order if false.
- **move_to_end(key, last=True)**
  - Move an existing key to either end of an ordered dictionary. The item is moved to the right end if last is true (the default) or to the beginning if last is false. Raises KeyError if the key does not exist:

```
d = OrderedDict.fromkeys('abcde')
d.move_to_end('b')
''.join(d)

d.move_to_end('b', last=False)
''.join(d)
```

OrderedDict Examples and Recipes

Demo

OrderedDict Examples and Recipes

It is straightforward to create an ordered dictionary variant that remembers the order the keys were last inserted. If a new entry overwrites an existing entry, the original insertion position is changed and moved to the end:

```
class LastUpdatedOrderedDict(OrderedDict):
    'Store items in the order the keys were last added'

    def __setitem__(self, key, value):
        super().__setitem__(key, value)
        self.move_to_end(key)
```

An OrderedDict would also be useful for implementing variants of functools.lru_cache():

OrderedDict Examples and Recipes – cont’d

```
from time import time

class TimeBoundedLRU:
    "LRU Cache that invalidates and refreshes old entries."

    def __init__(self, func, maxsize=128, maxage=30):
        self.cache = OrderedDict()      # { args : (timestamp, result)}
        self.func = func
        self.maxsize = maxsize
        self.maxage = maxage

    def __call__(self, *args):
        if args in self.cache:
            self.cache.move_to_end(args)
            timestamp, result = self.cache[args]
            if time() - timestamp <= self.maxage:
                return result
        result = self.func(*args)
        self.cache[args] = time(), result
        if len(self.cache) > self.maxsize:
            self.cache.popitem(0)
        return result
```

OrderedDict Examples and Recipes – cont'd

```
class MultiHitLRUCache:
    """ LRU cache that defers caching a result until
        it has been requested multiple times.

        To avoid flushing the LRU cache with one-time requests,
        we don't cache until a request has been made more than once.

    """

    def __init__(self, func, maxsize=128, maxrequests=4096, cache_after=1):
        self.requests = OrderedDict()   # { uncached_key : request_count }
        self.cache = OrderedDict()      # { cached_key : function_result }
        self.func = func
        self.maxrequests = maxrequests  # max number of uncached requests
        self.maxsize = maxsize          # max number of stored return values
        self.cache_after = cache_after

    def __call__(self, *args):
        if args in self.cache:
            self.cache.move_to_end(args)
            return self.cache[args]
        result = self.func(*args)
        self.requests[args] = self.requests.get(args, 0) + 1
        if self.requests[args] <= self.cache_after:
            self.requests.move_to_end(args)
            if len(self.requests) > self.maxrequests:
                self.requests.popitem(0)
        else:
            self.requests.pop(args, None)
            self.cache[args] = result
            if len(self.cache) > self.maxsize:
                self.cache.popitem(0)
        return result
```

UserDict objects

The class, **UserDict** acts as a wrapper around dictionary objects. The need for this class has been partially supplanted by the ability to subclass directly from **dict**; however, this class can be easier to work with because the underlying dictionary is accessible as an attribute.

class collections.UserDict([initialdata])

Class that simulates a dictionary. The instance's contents are kept in a regular dictionary, which is accessible via the data attribute of **UserDict** instances. If initialdata is provided, data is initialized with its contents; note that a reference to initialdata will not be kept, allowing it to be used for other purposes.

In addition to supporting the methods and operations of mappings, **UserDict** instances provide the following attribute:

data

A real dictionary used to store the contents of the **UserDict** class.

UserList objects

This class acts as a wrapper around list objects. It is a useful base class for your own list-like classes which can inherit from them and override existing methods or add new ones. In this way, one can add new behaviors to lists.

The need for this class has been partially supplanted by the ability to subclass directly from list; however, this class can be easier to work with because the underlying list is accessible as an attribute.

class collections.UserList([list])

Class that simulates a list. The instance's contents are kept in a regular list, which is accessible via the data attribute of UserList instances. The instance's contents are initially set to a copy of list, defaulting to the empty list []. list can be any iterable, for example a real Python list or a UserList object.

In addition to supporting the methods and operations of mutable sequences, UserList instances provide the following attribute:

data

A real list object used to store the contents of the UserList class.

UserList objects – cont’d

- Subclassing requirements: Subclasses of UserList are expected to offer a constructor which can be called with either no arguments or one argument. List operations which return a new sequence attempt to create an instance of the actual implementation class. To do so, it assumes that the constructor can be called with a single parameter, which is a sequence object used as a data source.

- If a derived class does not wish to comply with this requirement, all of the special methods supported by this class will need to be overridden; please consult the sources for information about the methods which need to be provided in that case.

UserString objects

The class, **UserString** acts as a wrapper around string objects. The need for this class has been partially supplanted by the ability to subclass directly from str; however, this class can be easier to work with because the underlying string is accessible as an attribute.

class collections.UserString(seq)

Class that simulates a string object. The instance's content is kept in a regular string object, which is accessible via the data attribute of **UserString** instances. The instance's contents are initially set to a copy of seq. The seq argument can be any object which can be converted into a string using the built-in **str()** function.

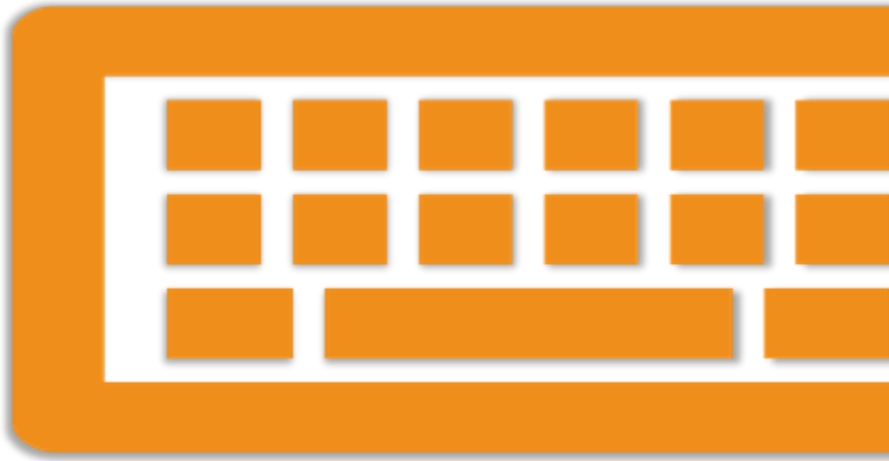
In addition to supporting the methods and operations of strings, **UserString** instances provide the following attribute:

data

A real str object used to store the contents of the **UserString** class.

Lab 02

Lab



Questions