05. Advanced Data Structures

October 15, 2025

The `collections` module in Python provides specialized container datatypes that offer alternatives to Python's general-purpose built-in containers like lists, tuples, dictionaries, and sets. These specialized containers are designed to provide high performance and more features for specific tasks.

0.1 1. `namedtuple` - improving code readabiltity

Python's `namedtuple()` is a factory function that lets you create tuple subclasses with named fields, allowing direct access to values using dot notation (e.g., `obj.attr`). This feature improves code readability and reduces errors compared to accessing values by index in a regular tuple, especially when the tuple has many items and is used far from where it was created.

Consider `divmod` builting function, which returns a tuple of `quotient` and `remainder`:

[1]:
```
divmod(15, 2)
```

[1]:
```
(7, 1)
```

In order to use the result, you would have to use indexing on the tuple, which decreases readability:

[2]:
```
result = divmod(15, 2)
```

[3]:
```
result[0]
```

[3]:
```
7
```

[4]:
```
result[1]
```

[4]:
```
1
```

Using `namedtuple` can improve readability. Instead of a tuple where items can be accessed only by position, we will get an enhanced tuple that also has the items accessible as attributes:

[5]:
```
from collections import namedtuple

DivMod = namedtuple("DivMod", ["quotient", "remainder"])
result = DivMod(*result)
```

[6]:
```
result
```

[6]:
```
DivMod(quotient=7, remainder=1)
```

[7]:
```
result.quotient
```

[7]:
```
7
```

[8]:
```
result.remainder
```

[8]:
```
1
```

Creating a namedtuple:

```
TupleSubclass = namedtuple(typename, field_names)
```

- `typename` is the name of the new class you're creating.
- `field_names` is the list of field names you'll use to access the items in the resulting tuple. It can be:
  - An iterable of strings, such as `["field1", "field2", ..., "fieldN"]`
  - A string with whitespace-separated field names, such as `"field1 field2 ... fieldN"`
  - A string with comma-separated field names, such as `"field1, field2, ..., fieldN"`

[9]:
```
Point = namedtuple("Point", "x y")
```

Creating an instance: `Point` is now a new class; it must receive values for `field_names` at instantiation:

[10]:
```
p = Point(11, 22)
```

Accessing Elements:

- Using field names:

[11]:
```
p.x
```

[11]:
```
11
```

[12]:
```
p.y
```

[12]:
```
22
```

- Using indices (like a regular tuple):

[13]:
```
p[0]
```

[13]:
```
11
```

[14]:
```
p[1]
```

[14]:
```
22
```

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

- **_fields**: A tuple containing the field names.

[15]:
```
p._fields
```

[15]:
```
('x', 'y')
```

- **_asdict()**: A method that returns a `dict` representation of the namedtuple.

[16]:
```
p._asdict()
```

[16]:
```
{'x': 11, 'y': 22}
```

- **_replace()**: A method to create a new instance with some fields replaced.

[17]:
```
p2 = p._replace(x=33)
```

[18]:
```
p2
```

[18]:
```
Point(x=33, y=22)
```

- **_make()**: A class method to create a new instance from an iterable.

[19]:
```
p3 = Point._make([44, 55])
```

[20]:
```
p3
```

[20]:
```
Point(x=44, y=55)
```

0.1.1 Use Cases:

- **Data Storage**: When you need a simple class for storing data without behavior (methods).
- **Readability**: When you want to make tuples more readable by using field names instead of indices.
- **Lightweight**: When you need a lightweight object type for data encapsulation.

0.1.2 Exercises 1

1. Create a namedtuple `Book` with the following fields: title, author, genre, pages, publisher. Create a `Book` object and try out some of the `namedtuple` operations presented above on it.
2. Use the namedtuple to read rows from books.csv into `Book` records. Iterate on all books and display the name and author of books in *computer_science* genre.

0.2 2. `deque` - building queues and stacks

Deques are a generalization of stacks and queues (the name is pronounced “deck” and is short for “double-ended queue”).

In Python, insert and pop operations on the beginning or left side of `list` objects are inefficient, with *O(n)* time complexity. These operations are especially expensive if you're working with large lists because Python has to move all the items to the right to insert new items at the beginning of the list.

On the other hand, append and pop operations on the right side of a list are normally efficient (*O(1)*) except for those cases in which Python needs to reallocate memory to grow the underlying list for accepting new items.

Python's `deque` was created to overcome this problem. Append and pop operations on both sides of a `deque` object are stable and equally efficient because deques are implemented as a doubly linked list. That's why deques are particularly useful for creating stacks and queues.

0.2.1 Key Features:

1. **Double-Ended**: Elements can be added or removed from both the front and the rear of the deque.
2. **Optimized Performance**: O(1) time complexity for append and pop operations at both ends.
3. **Thread-Safe**: Supports thread-safe, memory-efficient appends and pops from both ends without the need for locks.

0.2.2 Creating a deque

`deque(iterable=None, maxlen=None)`

- `iterable` holds an iterable that serves as an initializer.
- `maxlen` holds an integer number that specifies the maximum length of the deque.

[21]:
```
from collections import deque
d = deque()  # empty deque
```

[22]:
```
d = deque([1, 2, 3])  # initialize deque from iterable
```

0.2.3 Operations:

1. **Appending Elements**:
   - `append(x)`: Adds `x` to the right end.

[23]:
```
d.append(4)
d
```

[23]:
```
deque([1, 2, 3, 4])
```

   - `appendleft(x)`: Adds `x` to the left end.

[24]:
```
d.appendleft(0)
d
```

[24]:
```
deque([0, 1, 2, 3, 4])
```

2. **Popping Elements**:

- `pop()`: Removes and returns the rightmost element.

[25]:
```
d.pop()
```

[25]:
```
4
```

[26]:
```
d
```

[26]:
```
deque([0, 1, 2, 3])
```

- `popleft()`: Removes and returns the leftmost element.

[27]:
```
d.popleft()
```

[27]:
```
0
```

[28]:
```
d
```

[28]:
```
deque([1, 2, 3])
```

3. **Extending Deques**:

- `extend(iterable)`: Extends the right end by appending elements from the iterable.

[29]:
```
d.extend([4, 5, 6])
```

- `extendleft(iterable)`: Extends the left end by appending elements from the iterable (note: elements are added in reverse order).

[30]:
```
d.extendleft([0, -1, -2])
d
```

[30]:
```
deque([-2, -1, 0, 1, 2, 3, 4, 5, 6])
```

4. **Other Useful Methods**:

- `clear()`: Removes all elements from the deque.

[31]:
```
d.clear()
d
```

[31]:
```
deque([])
```

- `rotate(n)`: Rotates the deque `n` steps to the right (if `n` is negative, to the left).

[32]:
```
d = deque([1, 2, 3, 4, 5])
d.rotate(2)
d
```

[32]:
```
deque([4, 5, 1, 2, 3])
```

- `reverse()`: Reverses the elements of the deque in place.

[33]:
```
d.reverse()
d
```

[33]:
```
deque([3, 2, 1, 5, 4])
```

5. **Accessing Elements**:

- Similar to lists, you can access elements using indexing.

[34]:
```
d[0]
```

[34]:
```
3
```

[35]:
```
d[-1]
```

[35]:
```
4
```

0.2.4 Use Cases:

- **Queue Implementation**: `deque` is ideal for implementing queues with efficient append and pop operations from both ends.
- **Stack Implementation**: Can be used as a stack with `append` and `pop` methods.
- **Sliding Window**: Useful for maintaining a sliding window of elements for algorithms requiring a fixed-size buffer.
- **Breadth-First Search (BFS)**: Often used in BFS implementations where elements need to be added and removed from both ends.

0.2.5 Exercises 2

1. Use `deque` to implement a simple queue for a movie theater. Run the following changes and print the queue after each of them:
   - Add customers to the queue in the order they arrive: Alice, Bob, and Charlie.
   - Alice can't decide which movie she wants to see, so the clerk sends her back to the end of the queue.
   - Bob buys his ticket and leaves the queue.
   - Another customer, David, joins the queue.
2. Write a generator function that receives two required parameters: an iterable of integers and an integer `size`. The function should yield the maximum value in each sliding window of size `size` in the iterable. E.g. for `iterable = [3, 6, 8, 2, 6, 2, 3, 5, 7], size=3` should yield `8, 8, 8, 6, 6, 5, 7`.

0.3 3. `defaultdict` - handling missing keys

`defaultdict` is a subclass of built-in `dict` class. It is part of the `collections` module and provides all the standard dictionary methods but also includes a default factory function to supply default values for missing keys. This feature simplifies the handling of missing keys and avoids key errors.

0.3.1 Key Features:

1. **Default Factory**: The `defaultdict` constructor accepts a function (`default_factory`) that provides default values for missing keys.
2. **Automatic Initialization**: When you access or assign a value to a missing key, `defaultdict` automatically initializes the key with a value returned by the `default_factory`.
3. **Ease of Use**: Reduces the need for explicit checks for key existence, making the code cleaner and more readable.

0.3.2 Creating a default dict:

```
d = defaultdict(default_factory)
```

- `default_factory` is a callable object (can be a function or a class) which is called without arguments to provide values for missing keys.

0.3.3 Example Usage:

1. **Creating a defaultdict with int as the default factory**:

[36]:
```
from collections import defaultdict
d = defaultdict(int)
```

[37]:
```
d["a"]  # Accessing a missing key
```

[37]:
```
0
```

[38]:
```
d["a"] += 1
```

[39]:
```
d["a"]
```

[39]:
```
1
```

2. **Using a lambda function as the default factory**:

[40]:
```
d = defaultdict(lambda: [0, 0, 0])
```

[41]:
```
d["a"]
```

[41]:
```
[0, 0, 0]
```

[42]:
```
d["a"][1] = 5
```

[43]:
```
d["a"]
```

[43]:
```
[0, 5, 0]
```

0.3.4 Exercises 3

1. Given a list of `(student, grade)` tuples, group the students by their grades. E.g. `students = [("Alice", 9), ("Charlie", 7), ("Jane", 9), ("David", 10),`

("Eve", 10)] should return {9: ["Alice", "Jane"], 7: ["Charlie], 10: ["David", "Eve"]}

2. Given a list of (product, category, price) tuples, compute totals per category. E.g.

```
items = [
    ("bread", "food", 2),
    ("milk", "food", 1.5),
    ("chair", "furniture", 45),
    ("charger", "electronics", 50),
    ("eggs", "food", 3.8),
    ("desk", "furniture", 180),
]
```

0.4 4. Counter - counting objects

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.

0.4.1 Key Features:

1. **Element Counting**: Easily count the occurrences of each element in an iterable.
2. **Dictionary-Like Structure**: Elements are stored as dictionary keys, and their counts are stored as dictionary values.
3. **Methods for Arithmetic Operations**: Support for addition, subtraction, intersection, and union of counters.
4. **Common Elements**: Methods to find the most common elements.

Creating a Counter

```
c = Counter(iterable_or_mapping)
```

[2]:
```
from collections import Counter
c = Counter("hello world")
```

[3]:
```
c
```

[3]:
```
Counter({'l': 3, 'o': 2, 'h': 1, 'e': 1, ' ': 1, 'w': 1, 'r': 1, 'd': 1})
```

Besides dict operations, Counter objects support some extra methods and operators:

- **most_common(n)** - return a list of the n most common elements and their counts from the most common to the least.

[4]:
```
c.most_common(3)
```

[4]:
```
[('l', 3), ('o', 2), ('h', 1)]
```

- **total()** - Compute the sum of the counts.

[5]:
```
c.total()
```

[5]:
```
11
```

- **Arithmetic Operations**:

[6]:
```
counter1 = Counter("hello")
counter2 = Counter("wooorld")
```

[7]:
```
counter1 + counter2  # Addition
```

[7]:
```
Counter({'o': 4, 'l': 3, 'h': 1, 'e': 1, 'w': 1, 'r': 1, 'd': 1})
```

[8]:
```
counter1 - counter2  # Subtraction
```

[8]:
```
Counter({'h': 1, 'e': 1, 'l': 1})
```

[9]:
```
counter1 & counter2  # Intersection
```

[9]:
```
Counter({'l': 1, 'o': 1})
```

[10]:
```
counter1 | counter2  # Union
```

[10]:
```
Counter({'o': 3, 'l': 2, 'h': 1, 'e': 1, 'w': 1, 'r': 1, 'd': 1})
```

0.4.2 Exercises 4

1. Use a Counter to count the frequency of each word in a sentence.

0.5 5. `OrderedDict` - keeping dictionaries ordered

`OrderedDict` is a `dict` subclass that has methods specialized for rearranging dictionary order.

Sometimes you need your dictionaries to remember the order in which key-value pairs are inserted. Python's regular dictionaries were unordered data structures for years. In the meantime (since Python 3.7) dictionaries keep their items in the same order they're first inserted.

There are some features of OrderedDict that still make it valuable:

- **Intent communication**: With `OrderedDict`, your code will make it clear that the order of items in the dictionary is important. You're clearly communicating that your code needs or relies on the order of items in the underlying dictionary.
- **Control over the order of items**: With `OrderedDict`, you have access to `.move_to_end()`, which is a method that allows you to manipulate the order of items in your dictionary. You'll also have an enhanced variation of `.popitem()` that allows removing items from either end of the underlying dictionary.
- **Equality test behavior**: With `OrderedDict`, equality tests between dictionaries take the order of items into account. So, if you have two ordered dictionaries with the same group of items but in a different order, then your dictionaries will be considered non-equal.

[53]:
```
from collections import OrderedDict
d = OrderedDict.fromkeys('abcde')
```

[54]:
```
d
```

[54]:
```
OrderedDict([('a', None), ('b', None), ('c', None), ('d', None), ('e', None)])
```

[55]:
```
d.move_to_end('b')
```

[56]:
```
d
```

[56]:
```
OrderedDict([('a', None), ('c', None), ('d', None), ('e', None), ('b', None)])
```

[57]:
```
d.move_to_end('b', last=False)
```

[58]:
```
d
```

[58]:
```
OrderedDict([('b', None), ('a', None), ('c', None), ('d', None), ('e', None)])
```

[59]:
```
d.popitem()
```

[59]:
```
('e', None)
```

[60]:
```
d.popitem(last=False)
```

[60]:
```
('b', None)
```

0.5.1 Exercises 5

1. Implement a simple Least Recently Used (LRU) cache using `OrderedDict` (most recent items will be at the end).
   - Create an `OrderedDict` to act as an LRU cache; consider its maximum size to be 3
   - Create a helper function to add an item to the LRU cache:
     - if the item is already in the cache, move it to the end of the cache
     - update the value for the item
     - if maximum size is exceeded, remove oldest item
   - Add multiple values to the cache and inspect its value between operations.