A Look at the Python Collections Module

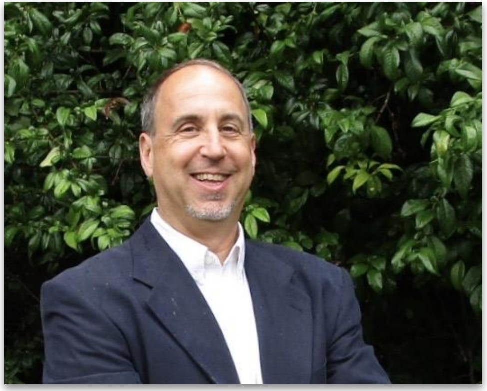
Roger Hurwitz
March 30, 2023
Boston Python User Group

Special thanks to:

- Emily Charles
- Ned Batchelder

Who am I?

- Senior Python software director/developer at Intel
- Web-based server system apps/services focus
- Pre-Python, I programmed in C/C++ for years
- A few of my favorite things about Python:
    - The language: intuitive and elegant (to me)
    - The ecosystem: “batteries included”
    - The web: best in class (backend) framework support
    - The people: “I came for the language and stayed for the community” (Brett Cannon)



https://www.linkedin.com/in/rogerhurwitz/

Collections Overview

- Collections module is ~18 years old, first showing up in Python 2.4
- The module “implements specialized container datatypes providing alternatives to Python’s general purpose built-in containers, dict, list, set, and tuple”
- Hot Take: While many datatypes in the module are still quite useful, others are showing their age
- Further Reading:
  - https://docs.python.org/3/library/collections.html
  - http://pymotw.com/2/collections/
  - https://realpython.com/python-collections-module/

`namedtuple()`	factory function for creating tuple subclasses with named fields
`deque`	list-like container with fast appends and pops on either end
`ChainMap`	dict-like class for creating a single view of multiple mappings
`Counter`	dict subclass for counting hashable objects
`OrderedDict`	dict subclass that remembers the order entries were added
`defaultdict`	dict subclass that calls a factory function to supply missing values
`UserDict`	wrapper around dictionary objects for easier dict subclassing
`UserList`	wrapper around list objects for easier list subclassing
`UserString`	wrapper around string objects for easier string subclassing

Ones I use frequently

Collections in a Nutshell

Datatype	Utility	Key Feature	Alternatives	Hot Take
`namedtuple`	Med	Adds named fields to `tuple` but still interchangeable w/ `tuple`	`dataclass, tuple, typing.NamedTuple`	Great readability substitute for `tuple`, but `dataclass` better for other cases
`deque`	Med	Much higher performance than `list` for element insert/remove intensive use cases	`list`	`deque` features make `it` well suited for queue, stack, and buffer operations
`ChainMap`	Med/Low	Single virtual `dict` composed of multiple actual `dict` objects	`dict.update`	Situational, but very useful when data structured with nested scopes/contexts
`Counter`	Med	Has multiple methods in support of counting operations on iterables	`pandas, dict, defaultdict,`	`Counter` is viable for bare bones data analysis but lacks scalability/flexibility
`OrderedDict`	Low	As of Python 3.7, less about key ordering and more about order-based methods	`dict`	Situational, but equality operator and reordering performance can matter
`defaultdict`	Med/High	Caller specified default factory function automatically invoked when adding keys	`dict.setdefault`	Subtle: unlike `setdefault`, its object factory only invoked on key creation
`User: Dict, List, String`	Med/Low	Specifically designed to be better than base types for subclassing	`dict, str, list`	With Python duck typing (protocols), less needed than in other languages

Examples

namedtuple

Enhance readability

```python
from collections import namedtuple

DivMod = namedtuple("DivMod", ["quotient", "remainder"])

def readable_divmod(x, y):
    """Return built-in divmod result as namedtuple."""
    dm = divmod(x, y)
    return DivMod(quotient=dm[0], remainder=dm[1])
```

```python
result = readable_divmod(10, 3)
print(f"{result = }")
print(f"{result.quotient = }")
print(f"{result.remainder = }")
```

```
result = DivMod(quotient=3, remainder=1)
result.quotient = 3
result.remainder = 1
```

… without losing the benefits of `tuple`

```python
result = readable_divmod(10, 3)
print(f"{isinstance(result, tuple) = }")
print(f"{result[0] = }, {result[1] = }")
quotient, remainder = result
print(f"{quotient = }, {remainder = }")
```

```
isinstance(result, tuple) = True
result[0] = 3, result[1] = 1
quotient = 3, remainder = 1
```

deque

The name of the game is "speed"

```python
from collections import deque
```

```python
%%time
dq = deque()
for x in range(10_000_000):
    dq.append(x)
    dq.pop()
```

```
CPU times: total: 1.55 s
Wall time: 1.54 s
```

```python
%%time
a = list()
for x in range(10_000_000):
    a.append(x)
    a.pop()
```

```
CPU times: total: 4.36 s
Wall time: 4.38 s
```

...but there's more

```python
# maxlen makes circular buffers easy
dq = deque(maxlen=10)
for x in range(100):
    dq.append(x)
print(f"{dq = }")
```

```
dq = deque([90, 91, 92, 93, 94, 95, 96, 97, 98, 99], maxlen=10)
```

```python
# more readable queue (FIFO) syntax
dq = deque()
dq.appendleft(1)  # versus a.insert(0, 1)
dq.appendleft(2)  # versus a.insert(0, 2)
while dq:
    print(dq.pop(), end=" ")
```

```
1 2
```

Be careful who you trust…

How is that better than using append on a fixed length deque? Doesn't appending to a fixed length deque push the oldest element out?

You're right, appending to a fixed length `deque` object will indeed push out the oldest element. However, using the `rotate` method has a couple of advantages over simply appending to a fixed length deque:

1

1. **Efficiency:** When you append an element to a `deque` object, it has to shift all the elements over by one position to make room for the new element. This can be an expensive operation, especially if the `deque` is large. In contrast, the `rotate` method simply rotates the elements of the `deque` in place, which can be a more efficient operation, especially if the number of positions to rotate is small.
2. **Flexibility:** Using the `rotate` method allows you to implement more complex circular buffer behaviors than simply appending to a fixed length `deque`. For example, you might want to rotate the buffer by a different number of positions depending on some condition, or you might want to rotate the buffer in both directions. The `rotate` method gives you this flexibility.

That being said, if you only need to implement a simple circular buffer and efficiency is not a concern, using `deque.append` to overwrite the oldest element will work just fine.

2

I don't agree with your efficiency assertion. deque, unlike list, uses links to connect elements therefore it should not require any element shifting

You are correct. I apologize for the mistake in my previous response. `deque` does indeed use a doubly-linked list to connect its elements, so appending an element to a fixed-length `deque` should not require any element shifting.

ChainMap

Scope-Based Lookup

```python
from collections import ChainMap

member_specials = {
    "Chicken": 6.79,
    "Wine": 3.64,
}
monthly_specials = {
    "Butter": 1.49,
    "Wine": 4.64,
}
regular_prices = {
    "Butter": 1.99,
    "Chicken": 8.79,
    "Wine": 5.64,
    "Bread": 2.99,
}

member_prices = ChainMap(member_specials, monthly_specials, regular_prices)

print(f"{member_prices['Wine'] = }")
print(f"{member_prices['Butter'] = }")
print(f"{member_prices['Bread'] = }")
```

```
member_prices['Wine'] = 3.64
member_prices['Butter'] = 1.49
member_prices['Bread'] = 2.99
```

Shortcut for Skipping First Mapping

```python
nonmember_prices = member_prices.parents

print(f"{nonmember_prices['Wine'] = }")
print(f"{nonmember_prices['Butter'] = }")
print(f"{nonmember_prices['Bread'] = }")
```

```
nonmember_prices['Wine'] = 4.64
nonmember_prices['Butter'] = 1.49
nonmember_prices['Bread'] = 2.99
```

Counter

Base Case: Count the Elements in an Iterable

```python
from collections import Counter
import random

# create a list of random coin flip results
coinflip_outcomes = ["heads", "tails"]
coinflips = random.choices(coinflip_outcomes, k=100)


coinflip_cnt = Counter(coinflips)
print(f"{coinflip_cnt = }")
```

```
coinflip_cnt = Counter({'tails': 59, 'heads': 41})
```

Next Level: Slice and Dice the Results

```python
print(f"{coinflip_cnt.most_common(1) = }")
print(f"{coinflip_cnt.total() = }")
print(f"{[elem for elem in coinflip_cnt.elements()] = }")
print(f"{coinflip_cnt['edges'] = }")
```

```
coinflip_cnt.most_common(1) = [('tails', 59)]
coinflip_cnt.total() = 100
[elem for elem in coinflip_cnt.elements()] = ['tails', 'tails', 'tails', 'tails', 'tai
ls', 'tails', 'tails', 'tails', 'tails', 'tails', 'tails', 'tails', 'tails', 'tails', '
'tails', 'tails', 'tails', 'tails', 'tails', 'tails', 'tails', 'tails', 'tails', 'tail
s', 'heads', 'heads', 'heads', 'heads', 'heads', 'heads', 'heads', 'heads', 'heads', 'h
'heads', 'heads', 'heads', 'heads', 'heads', 'heads', 'heads']
coinflip_cnt['edges'] = 0
```

OrderedDict

In Python 3.6 `dict` Found Order

```python
from collections import OrderedDict

ordered_dict = OrderedDict.fromkeys("abcdef")
regular_dict = dict.fromkeys("abcdef")

for keys in zip(ordered_dict, regular_dict):
    print(keys, end=" ")
```

```
('a', 'a') ('b', 'b') ('c', 'c') ('d', 'd') ('e', 'e') ('f', 'f')
```

Equality: `OrderedDict` vs `dict`

```python
ordered_dict_a = OrderedDict.fromkeys("abcdef")
ordered_dict_b = OrderedDict.fromkeys("bacdef")

print(f"{ordered_dict_a == ordered_dict_b = }")
```

```
ordered_dict_a == ordered_dict_b = False
```

```python
regular_dict_a = dict.fromkeys("abcdef")
regular_dict_b = dict.fromkeys("bacdef")

print(f"{regular_dict_a == regular_dict_b = }")
```

```
regular_dict_a == regular_dict_b = True
```

Reordering Flexibility/Efficiency

```python
ordered_dict = OrderedDict.fromkeys("abcdef")

ordered_dict.move_to_end("f", last=False)
for key in ordered_dict:
    print(key, end=" ")
```

```
f a b c d e
```

defaultdict

```python
from collections import defaultdict

pets = [
    ("cat", "Luna"), ("cat", "Lily"), ("bird", "Bella"), ("cat", "Lucy"),
    ("cat", "Nala"), ("gerbil", "Callie"), ("cat", "Kitty"), ("cat", "Cleo"),
    ("bird", "Willow"), ("cat", "Chloe"), ("dog", "Max"), ("gerbil", "Charlie"),
    ("dog", "Cooper"), ("bird", "Milo"), ("dog", "Buddy"), ("bird", "Rocky"),
    ("dog", "Bear"), ("dog", "Teddy"), ("dog", "Duke"), ("gerbil", "Leo"),
]
```

Typical Use: Initialize New Keys w/ List

```python
pet_names_by_type = defaultdict(list)

for pet_type, pet_name in pets:
    pet_names_by_type[pet_type].append(pet_name)

print(f"{pet_names_by_type['dog'] = }")
```

```
pet_names_by_type['dog'] = ['Max', 'Cooper', 'Buddy', 'Bear', 'Teddy', 'Duke']
```

`dict` Alternative: Less Readable/Efficient

```python
pet_names_by_type = {}

for pet_type, pet_name in pets:
    pet_names_by_type.setdefault(pet_type, []).append(pet_name)

print(f"{pet_names_by_type['dog'] = }")
```

```
pet_names_by_type['dog'] = ['Max', 'Cooper', 'Buddy', 'Bear', 'Teddy', 'Duke']
```

Exercises

Customers.csv

https://github.com/rogerhurwitz/pythonCollectionsPresentation

	customer_id	gender	age	annual_income	spending_score	profession	work_experience	family_size
1	1	Male	19	15000	39	Healthcare	1	4
2	2	Male	21	35000	81	Engineer	3	3
3	3	Female	20	86000	6	Engineer	1	1
4	4	Female	23	59000	77	Lawyer	0	2
5	5	Female	31	38000	40	Entertainment	2	6
6	6	Female	22	58000	76	Artist	0	2
7	7	Female	35	31000	6	Healthcare	1	3
8	8	Female	23	84000	94	Healthcare	1	3
9	9	Male	64	97000	3	Engineer	0	3
10	10	Female	30	98000	72	Artist	1	4
11	11	Male	67	7000	14	Engineer	1	3
12	12	Female	35	93000	99	Healthcare	4	4
13	13	Female	58	80000	15	Executive	0	5
14	14	Female	24	91000	77	Lawyer	1	1
15	15	Male	37	19000	13	Doctor	0	1
16	16	Male	22	51000	79	Healthcare	1	2
17	17	Female	35	29000	35	Homemaker	9	5
18	18	Male	20	89000	66	Healthcare	1	6
19	19	Male	52	20000	29	Entertainment	1	4
20	20	Female	35	62000	98	Artist	0	1
21	21	Male	35	96000	35	Homemaker	12	1
22	22	Male	25	4000	73	Healthcare	3	4
23	23	Female	46	42000	5	Artist	13	2
24	24	Male	31	71000	73	Artist	5	2
25	25	Female	54	67000	14	Executive	1	3
26	26	Male	29	52000	82	Artist	1	3
27	27	Female	45	68000	32	Healthcare	9	8

Exercise 0: Execute this on your computer

```python
import csv

with open("Customers.csv", newline="") as csvfile:
    customer_reader = csv.reader(csvfile)
    headers = next(customer_reader)
    customers = [
        tuple(row)
        for row in customer_reader
    ]

print(f"{customers[0] = }")
```

```
customers[0] = ('1', 'Male', '19', '15000', '39', 'Healthcare', '1', '4')
```

Exercise 1: Replace use of `tuple` with `namedtuple` ¶

```python
from collections import namedtuple
import csv

with open("Customers.csv", newline="") as csvfile:
    customer_reader = csv.reader(csvfile)
    headers = next(customer_reader)

    Customer = namedtuple("Customer", headers)

    customers = [
        Customer(*row)
        for row in customer_reader
    ]

print(f"{customers[0] = }")
```

```
customers[0] = Customer(customer_id='1', gender='Male', age='19', annual_income='15000', spending_
```

Exercise 2: Use `Counter` and show the three most common customer professions

```python
from collections import namedtuple, Counter
import csv

with open("Customers.csv", newline="") as csvfile:
    customer_reader = csv.reader(csvfile)
    headers = next(customer_reader)

    Customer = namedtuple("Customer", headers)

    profession_cnt = Counter([
        Customer(*row).profession
        for row in customer_reader
    ])

profession_cnt.most_common(3)
```

```
[('Artist', 612), ('Healthcare', 339), ('Entertainment', 234)]
```

Exercise 3: Clean this code using collections module

```
from collections import namedtuple
import csv

with open("Customers.csv", newline="") as csvfile:
    customer_reader = csv.reader(csvfile)
    headers = next(customer_reader)

    Customer = namedtuple("Customer", headers)

    genders_by_profession = {}

    for row in customer_reader:
        customer = Customer(*row)

        if genders_by_profession.get(customer.profession) is None:
            genders_by_profession[customer.profession] = {}

        if genders_by_profession[customer.profession].get(customer.gender) is None:
            genders_by_profession[customer.profession][customer.gender] = 1
        else:
            genders_by_profession[customer.profession][customer.gender] += 1

print(genders_by_profession)
```

```
{'Healthcare': {'Male': 143, 'Female': 196}, 'Engineer': {'Male': 76, 'Female': 103}, 'Lawyer': {'Female': 86, 'Ma
2}, 'Executive': {'Female': 87, 'Male': 66}, 'Doctor': {'Male': 72, 'Female': 89}, 'Homemaker': {'Female': 39, 'Ma
```

```python
from collections import namedtuple, defaultdict, Counter
import csv

with open("Customers.csv", newline="") as csvfile:
    customer_reader = csv.reader(csvfile)
    headers = next(customer_reader)

    Customer = namedtuple("Customer", headers)

    genders_by_profession = defaultdict(Counter)

    for row in customer_reader:
        customer = Customer(*row)
        genders_by_profession[customer.profession][customer.gender] += 1

print(genders_by_profession)
```

```
defaultdict(<class 'collections.Counter'>, {'Healthcare': Counter({'Female': 196, 'Male': 143}), 'Engine
t': Counter({'Female': 133, 'Male': 101}), 'Artist': Counter({'Female': 380, 'Male': 232}), 'Executive':
({'Female': 39, 'Male': 21}), 'Marketing': Counter({'Female': 53, 'Male': 32}), '': Counter({'Female': 2
```