# Custom Collection Types

# Python's Collection Types

Python has excellent built-in data types for containing collections of data: `dict`, `list`, `tuple`, `set`, etc. You use them all the time (or you should).

```
atomics_d = {"gold": 79, "neon": 10, "zinc": 30}
primes_l = [17, 163, 277, 479]
data_t = (42, "potato")
characters_s = {"Bart", "Homer", "Marge"}
```

# Accessing Items

Most let you access elements with square brackets.

```
>>> atomics_d["gold"]
79
>>> primes_l[3]
479
>>> data_t[0]
42
```

How can you use `[...]` in your own classes?

# List of Uniques

Let's explore by creating a `UniqueList` data type.

```
class UniqueList:
    def __init__(self, items):
        self.items = []
        for item in items:
            self.append(item)
    def append(self, item):
        if item not in self.items:
            self.items.append(item)
```

```
>>> u = UniqueList([3,7,2,9,3,4,2])
>>> u.items
[3, 7, 2, 9, 4]
```

# [Access]

We can access elements by index, via `u.items`. But that is not the ideal.

```
>>> # We can do this to get the 4th element...
... u.items[3]
9
>>> # ... but really, it's better to do this:
... u[3]
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
TypeError: 'UniqueList' object does not support indexing
```

So how can we make `u[3]` work?

# __getitem__

```
class UniqueList:
    # __init__() and append() as above
    def __getitem__(self, index):
        return self.items[index]
```

```
>>> u = UniqueList([3,7,2,9,3,4,2])
>>> u[3]
9
```

It works!

# How `__getitem__` works

In short: Python translates `foo[index]` into `foo.__getitem__(index)`.

This lets you make your custom objects quack like a list.

Considerations:

- Raising `IndexError` appropriately
- Negative indexes
- Slices

# Follow `lists`'s lead

The whole point is to provide a familiar interface with **correct** intuitions.

```
>>> # Negative indices ought to return from the end of the list.
... u[-1]
4
>>> # An out-of-range index ought to raise IndexError.
... u[42]
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
  File "<stdin>", line 10, in __getitem__
IndexError: list index out of range
```

`UniqueList` follows these intuitions correctly.

# Recreating the interface

Sometimes we have to do more work, if the sequence isn't represented internally by a list. Maybe it involves a database lookup or API call, for example.

In that case, you may need to manually `raise IndexError` in the body of `__getitem__`, for example.

# Dictionary Key Access

How does this work for dictionary-like classes?

Let's create a class that stores multiple values for each key.

When we look up a key via `get()`, we get a list of 1 or more values.

```
class MultiDict:
    def __init__(self):
        self.data = { }
    def insert(self, key, value):
        if key not in self.data:
            self.data[key] = []
        self.data[key].append(value)
    def get_values(self, key):
        return self.data[key]
    def get(self, key):
        return self.get_values(key)[-1]
```

# MultiDict usage

```
>>> m = MultiDict()
>>> m.insert("x", 3)
>>> m.insert("y", 1)
>>> m.insert("x", 7)
>>> m.get_values("x")
[3, 7]
>>> m.get("x")
7
>>> m.get_values("y")
[1]
>>> m.get("y")
1
```

That implements the functionality. But how can we do `m["x"]` instead of `m.get("x")`? Or `m["y"] = 1` rather than `m.insert("y", 1)`, for that matter?

# `__getitem__` for dict

`__getitem__` works for dict-like objects much as it does for list-like objects. The main difference is that it must raise `KeyError` for a missing key intead of `IndexError`.

In short: just replace `def get` with `def __getitem__`.

```
class MultiDict:
    # replacing get()...
    def __getitem__(self, key):
        return self.get_values(key)[-1]
    # ...
```

```
>>> m = MultiDict()
>>> m.insert("x", 3)
>>> m["x"]
3
```

# Dictionary Setting

Python translates "`foo[bar] = baz`" to "`foo.__setitem__(bar, baz)`".

In `MultiDict`, we can just rename `insert` to be `__setitem__`.

```
class MultiDict:
    def __setitem__(self, key, value):
        if key not in self.data:
            self.data[key] = []
        self.data[key].append(value)
    # And keeping __init__, __getitem__  and get_values() as above.
```

```
>>> m = MultiDict()
>>> m["x"] = 3
>>> m["x"]
3
>>> m["x"] = 7
>>> m["x"]
7
```

# Final MultiDict Class

```
class MultiDict:
    def __init__(self):
        self.data = { }
    def __setitem__(self, key, value):
        if key not in self.data:
            self.data[key] = []
        self.data[key].append(value)
    def get_values(self, key):
        return self.data[key]
    def __getitem__(self, key):
        return self.get_values(key)[-1]
```

```
>>> m = MultiDict()
>>> m["x"] = 3
>>> m["x"]
3
>>> m["x"] = 7
>>> m["x"]
7
>>> m.get_values("x")
[3, 7]
```

# Iterable Collection Types

Python's built-in data types can be used in a `for` loop:

```
>>> primes_1 = [17, 163, 277, 479]
>>> for prime in primes_1:
...     print("{} is not prime".format(prime * 2))
34 is not prime
326 is not prime
554 is not prime
958 is not prime
```

How can your own types be iterable as well?

# Sequence Protocol

Define a `__getitem__` method which

- accepts 0, 1, 2, ...
- and raises `IndexError` once past the last index.

That's it. Your object then automatically works in a `for` loop.

Guess what. Our `UniqueList` class already does this!

```
>>> nums = UniqueList([3,7,3,4,3])
>>> for num in nums:
...     print(num)
...
3
7
4
```

# Iterator Protocol

Your collection class can produce an arbitrary sequence, using the *iterator protocol*.

```
import random
class RandomNumbers:
    'Produces QUANTITY random numbers from 0 to 9.'
    def __init__(self, quantity):
        self.quantity = quantity
    def __iter__(self):
        return self
    def __next__(self):
        self.quantity -= 1
        if self.quantity < 0:
            raise StopIteration
        return random.randrange(10)

for n in RandomNumbers(20): print(n)
```

# When do you use it?

If your needs can be met by defining `__getitem__` and acting like a list, that's often a better choice.

If that will not suffice, use the iterator protocol. But it's better to do so via generator functions.

For an extensive treatment of this rich and valuable topic, sign up for "Scaling Python with Generators."