[Index | Exercise 2.2 | Exercise 2.4]

Exercise 2.3

Objectives:

- Iterate like a pro

Files Modified: None.

Iteration is an essential Python skill. In this exercise, we look at a number of common iteration idioms.

Start the exercise by grabbing some rows of data from a CSV file.

```
>>> import csv
>>> f = open('Data/portfolio.csv')
>>> f_csv = csv.reader(f)
>>> headers = next(f_csv)
>>> headers
['name', 'shares', 'price']
>>> rows = list(f_csv)
>>> from pprint import pprint
>>> pprint(rows)
[['AA', '100', '32.20'],
 ['IBM', '50', '91.10'],
 ['CAT', '150', '83.44'],
 ['MSFT', '200', '51.23'],
 ['GE', '95', '40.37'],
 ['MSFT', '50', '65.10'],
 ['IBM', '100', '70.44']]
>>>
```

(a) Basic Iteration and Unpacking

The `for` statement iterates over any sequence of data. For example:

```
>>> for row in rows:
        print(row)

['AA', '100', '32.20']
['IBM', '50', '91.10']
['CAT', '150', '83.44']
['MSFT', '200', '51.23']
['GE', '95', '40.37']
['MSFT', '50', '65.10']
['IBM', '100', '70.44']
>>>
```

Unpack the values into separate variables if you need to:

```
>>> for name, shares, price in rows:
        print(name, shares, price)

AA 100 32.20
IBM 50 91.10
CAT 150 83.44
MSFT 200 51.23
GE 95 40.37
MSFT 50 65.10
IBM 100 70.44
>>>
```

It's somewhat common to use `_` or `__` as a throw-away variable if you don't care about one or more of the values. For example:

```
>>> for name, _, price in rows:
        print(name, price)

AA 32.20
IBM 91.10
CAT 83.44
MSFT 51.23
GE 40.37
MSFT 65.10
IBM 70.44
>>>
```

If you don't know how many values are being unpacked, you can use `*` as a wildcard. Try this experiment in grouping the data by name:

```
>>> from collections import defaultdict
>>> byname = defaultdict(list)
>>> for name, *data in rows:
        byname[name].append(data)

>>> byname['IBM']
[['50', '91.10'], ['100', '70.44']]
>>> byname['CAT']
[['150', '83.44']]
>>> for shares, price in byname['IBM']:
        print(shares, price)

50 91.10
100 70.44
>>>
```

(b) Counting with enumerate()

`enumerate()` is a useful function if you ever need to keep a counter or index while iterating. For example, suppose you wanted an extra row number:

```
>>> for rowno, row in enumerate(rows):
        print(rowno, row)

0 ['AA', '100', '32.20']
1 ['IBM', '50', '91.10']
2 ['CAT', '150', '83.44']
3 ['MSFT', '200', '51.23']
4 ['GE', '95', '40.37']
5 ['MSFT', '50', '65.10']
6 ['IBM', '100', '70.44']
>>>
```

You can combine this with unpacking if you're careful about how you structure it:

```
>>> for rowno, (name, shares, price) in enumerate(rows):
        print(rowno, name, shares, price)

0 AA 100 32.20
1 IBM 50 91.10
2 CAT 150 83.44
3 MSFT 200 51.23
4 GE 95 40.37
5 MSFT 50 65.10
6 IBM 100 70.44
>>>
```

(c) Using the zip() function

The `zip()` function is most commonly used to pair data. For example, recall that you created a `headers` variable:

```
>>> headers
['name', 'shares', 'price']
>>>
```

This might be useful to combine with the other row data:

```
>>> row = rows[0]
>>> row
['AA', '100', '32.20']
```

```
>>> for col, val in zip(headers, row):
        print(col, val)

name AA
shares 100
price 32.20
>>>
```

Or maybe you can use it to make a dictionary:

```
>>> dict(zip(headers, row))
{'name': 'AA', 'shares': '100', 'price': '32.20'}
>>>
```

Or maybe a sequence of dictionaries:

```
>>> for row in rows:
        record = dict(zip(headers, row))
        print(record)

{'name': 'AA', 'shares': '100', 'price': '32.20'}
{'name': 'IBM', 'shares': '50', 'price': '91.10'}
{'name': 'CAT', 'shares': '150', 'price': '83.44'}
{'name': 'MSFT', 'shares': '200', 'price': '51.23'}
{'name': 'GE', 'shares': '95', 'price': '40.37'}
{'name': 'MSFT', 'shares': '50', 'price': '65.10'}
{'name': 'IBM', 'shares': '100', 'price': '70.44'}
>>>
```

(d) Generator Expressions

A generator expression is almost exactly the same as a list comprehension except that it does not create a list. Instead, it creates an object that produces the results incrementally--typically for consumption by iteration. Try a simple example:

```
>>> nums = [1,2,3,4,5]
>>> squares = (x*x for x in nums)
>>> squares
<generator object <genexpr> at 0x37caa8>
>>> for n in squares:
        print(n)

1
4
9
16
```

```
25
>>>
```

You will notice that a generator expression can only be used once. Watch what happens if you do the for-loop again:

```
>>> for n in squares:
        print(n)

>>>
```

You can manually get the results one-at-a-time if you use the `next()` function. Try this:

```
>>> squares = (x*x for x in nums)
>>> next(squares)
1
>>> next(squares)
4
>>> next(squares)
9
>>>
```

Keeping typing `next()` to see what happens when there is no more data.

If the task you are performing is more complicated, you can still take advantage of generators by writing a generator function and using the `yield` statement instead. For example:

```
>>> def squares(nums):
        for x in nums:
            yield x*x

>>> for n in squares(nums):
        print(n)

1
4
9
16
25
>>>
```

We'll return to generator functions a little later in the course--for now, just view such functions as having the interesting property of feeding values to the `for`-statement.

(e) Generator Expressions and Reduction Functions

Generator expressions are especially useful for feeding data into functions such as `sum()`, `min()`, `max()`, `any()`, etc. Try some examples using the portfolio data from earlier. Carefully observe that these examples are missing some extra square brackets ([]) that appeared when using list comprehensions.

```
>>> from readport import read_portfolio
>>> portfolio = read_portfolio('Data/portfolio.csv')
>>> sum(s['shares']*s['price'] for s in portfolio)
44671.15
>>> min(s['shares'] for s in portfolio)
50
>>> any(s['name'] == 'IBM' for s in portfolio)
True
>>> all(s['name'] == 'IBM' for s in portfolio)
False
>>> sum(s['shares'] for s in portfolio if s['name'] == 'IBM')
150
>>>
```

Here is a subtle use of a generator expression in making comma separated values:

```
>>> s = ('GOOG',100,490.10)
>>> ','.join(s)
... observe that it fails ...
>>> ','.join(str(x) for x in s)    # This works
'GOOG,100,490.1'
>>>
```

The syntax in the above examples takes some getting used to, but the critical point is that none of the operations ever create a fully populated list of results. This gives you a big memory savings. However, you do need to make sure you don't go overboard with the syntax.

(f) Saving a lot of memory

In Exercise 2.1 you wrote a function `read_rides_as_dicts()` that read the CTA bus data into a list of dictionaries. Using it requires a lot of memory. For example, let's find the day on which the route 22 bus had the greatest ridership:

```
>>> import tracemalloc
>>> tracemalloc.start()
>>> import readrides
>>> rows = readrides.read_rides_as_dicts('Data/ctabus.csv')
>>> rt22 = [row for row in rows if row['route'] == '22']
>>> max(rt22, key=lambda row: row['rides'])
{'date': '06/11/2008', 'route': '22', 'daytype': 'W', 'rides': 26896}
>>> tracemalloc.get_traced_memory()
... look at result. Should be around 220MB
>>>
```

Now, let's try an example involving generators. Restart Python and try this:

```
>>> # RESTART
>>> import tracemalloc
>>> tracemalloc.start()
>>> import csv
>>> f = open('Data/ctabus.csv')
>>> f_csv = csv.reader(f)
>>> headers = next(f_csv)
>>> rows = (dict(zip(headers,row)) for row in f_csv)
>>> rt22 = (row for row in rows if row['route'] == '22')
>>> max(rt22, key=lambda row: int(row['rides']))
{'date': '06/11/2008', 'route': '22', 'daytype': 'W', 'rides': 26896}
>>> tracemalloc.get_traced_memory()
... look at result. Should be a LOT smaller than before
>>>
```

Keep in mind that you just processed the entire dataset as if it was stored as a sequence of dictionaries. Yet, nowhere did you actually create and store a list of dictionaries. Not all problems can be structured in this way, but if you can work with data in an iterative manner, generator expressions can save a huge amount of memory.

[Solution | Index | Exercise 2.2 | Exercise 2.4]

>>> Advanced Python Mastery
. . . A course by dabeaz
. . . Copyright 2007-2023