Python Data Science Toolbox (P2)

Using iterators in PythonLand

1. iterable
    - list, strings, dictionaries, file connections
    - object with an associated iter() method
    - applying iter() to an iterable creates an iterator → for loop

    ```
    word = 'Da'
    it = iter(word)

    next(it)

    #Output:
    'D'

    next(it)

    #Output:
    'a'

    #iterating at once with *
    word = 'Data'
    it = iter(word)
    print(*it)

    #Output:
    D a t a
    ```

2. iterator
    - defined as an object that as an associated next() method that produces the consecutive values

3. enumerate()
    - allow the user to add counter to any iterable

    ```
    #Type 1
    avengers = ['hawkeye', 'iron man', 'thor', 'quicksilver']
    ```

```
for index, value in enumerate(avengers):
  print(index, value)

#Output:
0 hawkeye
1 iron man
2 thor
3 quicksilver

#Type 2
for index, value in enumerate(avengers, start=10):
  print(index, value)

#Output:
10 hawkeye
11 iron man
12 thor
13 quicksilver
```

4. zip
    - allow the user to stitch together an arbitrary number of iterables

```
#Method 1
avengers = ['hawkeye', 'iron man', 'thor', 'quicksilver']
names = ['barton', 'stark', 'odinson', 'maximoff']

for z1, z2 in zip(avengers, names):
  print(z1, z2)

#Output:
hawkeye barton
iron man stark
thor odinson
quicksilver maximoff

#Method 2:
avengers = ['hawkeye', 'iron man', 'thor', 'quicksilver']
names = ['barton', 'stark', 'odinson', 'maximoff']
z = zip(avengers, names)
print(*z)

#Output:
('hawkeye', 'barton') ('iron man', 'stark') ('thor', 'odinson') ('quicksilver', 'maximoff')
```

5. loading data from the file in chunks by using read_csv from pandas package

```
import pandas as pd
result = []
#specify the chunk size when loading the data
for chunk in pd.read_csv('data.csv', chunksize = 1000):
  result.append(sum(chunk['x']))
```

```
total = sum(result)
print(total)

#Output:
4252532
```

List Comprehension and generators

1. list comprehension

    - a simple version of for loop
    - collapse fo loop for building list in a single line
    - component required: → [[output expression] for iterator variable in iterable]

        1. iterable
        2. iterator variable (represent members of iterable)
        3. output expression

    ```
    nums = [12, 8, 21, 3, 16]
    new_nums = [nums + 1 for num in nums]
    print(new_nums)

    #Output:
    [13, 9, 22, 4, 17]
    ```

2. nested loop in list comprehension

    ```
    #Nested loop
    pairs_1 = []

    for num1 in range(0,2):
      for num2 in range(6,8):
        pairs_1.append(num1, num2)

    print(pairs_1)

    #Output:
    [(0, 6), (0, 7), (1, 6), (1, 7)]

    #Nested loop in list comprehension
    pairs_2 = [(num1, num2) for num1 in range(0, 2) for num2 in range(6, 8)]
    print(pairs_2)

    #Output:
    (0, 6), (0, 7), (1, 6), (1, 7)]
    ```

3. conditionals in comprehensions

    ```
    [num ** 2 for num in range(10) if num % 2 == 0]

    #Output:
    [0, 4, 16, 36, 64]
    ```

4. dict comprehension

    - used to create dictionaries from iterables
    - the key and value are separated by a colon in output expression
    - use curly braces {} instead of square brackets []

        ```
        pos_neg = {num: -num for num in range(9)}

        #Output:
        {0: 0, 1: -1, 2: -2, 3: -3, 4: -4, 5: -5, 6: -6, 7: -7, 8: -8}
        ```

5. generator expression

    - like a list comprehension which can iterate over the object but it do not store the list in the memory
    - will return a generator object
    - anything can be done on list comprehension also can be done on generators

    ```
    #print values from generators
    result = (num for num in range(6))
    for num in result:
      print(num)

    #conditionals in generators
    even_nums = (num for num in range(10) if num % 2 == 0)
    print(list(even_nums)

    #Output:
    [0, 2. 4, 6, 8]
    ```

    - generator functions
        - produces generator objects when called

- defined like a regular function by using def but it will yield a sequence of values instead of returning a single value
- use **yield** keyword

```
def num_sequence(n):
  """Generate values from 0 to n."""
  i = 0
  while i < n:
    yield i
    i += 1
```