Tutorial 6 CSC1002 Lab

A. Quick Review

Iterators

Part 1: Basics

Iteration is a general term for taking each item of something, one after another. Any time you use a loop, explicitly or implicitly, to go over a group of items, that is called iteration.

In **Python**,

(1) An **iterable** object is an object that implements `__iter__` (two underlines), which converts the iterable object to an **iterator** object.

(2) An **iterator** is an object that implements `__next__` (two underlines) method, which returns the next value from the iterator. If there is no more items to return then it should raise *StopIteration* exception.

Python has several built-in iterables such as, lists, tuples, strings, dictionaries and files.

Whenever you use a `for` loop, or `map`, or a list comprehension, etc. In **Python**, the `__next__` method is called automatically to get each item from the corresponding **iterator**, thus going through the process of **iteration**.

```
>>> L = [1, 2, 3]
>>> L2 = L.__iter__() #iter(L)
>>> L2
<list_iterator at 0x108b1cd68>
```

Please try the following experiments:

Experiment 1:

```
>>> L = iter([1, 2, 3])
>>> L.__next__() #L[0]
>>> L.__next__() #L[1]
>>> L.__next__() #L[2]
(Type one more, observe what happens)
>>> L.__next__()
```

```
Traceback (most recent call last):
  File "<pyshell#4>", line 1, in <module>
    L.__next__()
StopIteration
```

Experiment 2:

```
>>> L = [1, 2, 3]
>>> L2 = L.__iter__()

>>> for i in L: print(i, end =" ")

>>> for i in L2: print(i, end =" ")
```

Using the iterator in **for loop** example we saw, the following example tries to show the code behind the scenes.

```
>>> for x in [1, 2, 3]:
...     pass

>>> iterator = iter([1, 2, 3])
>>> while True:
...     try:
...         x = iterator.__next__()
...         print(x, end=' ')
...     except StopIteration as e:
...         break
```

Part 2: Properties

2.1 Here we create a custom object that implements the iterator protocol.

(without defining a `StopIteration` exception)

Remark: We add **2.1** here to make the interation part more complete. However, as **2.1** involve the knowledge of "class" module. We will skip it and you can learn by yourself after learning the "class" module in CSC1001.

```
class Sequence:
   def __init__(self):
      self.x = 0
   def __next__(self):
      self.x += 1

      #if self.x > 14:
         #raise StopIteration

      return self.x**self.x
   def __iter__(self):
      return self
```

Try the following code:

```
s = Sequence()
n = 0
for e in s:
    print(e)
    n += 1
    if n > 10:
        break
```

This is a sequence of numbers 1, 4, 27, 256, ... This demonstrates that with iterators, we can work with **infinite** sequences.

2.2 Make code cleaner

Before showing the example, we have to learn basic skill of **file opening**.

Before you can read or write a file, you have to open it using Python's built-in `open()` function. This function creates a file object, which would be utilized to call other support methods associated with it.

```
f = open('file_name', 'mode') #access mode can be 'r', 'w', 'a' for read, write, append
f = open('readme.txt')        #default mode is 'r'
```

mode	further description
r	(**default** mode) For reading only. The file pointer is placed at the **beginning** of the file.
w	For writing only. Overwrites the file if the file exists. If not, creates a new file for writing.
a	For appending. The file pointer is at the **end** of the file if the file exists. If not, it creates a new file for writing.
rb	For binary files reading only. Other same as mode 'r'.
wb	For binary files writing only. Other same as mode 'w'.

1. **Read file**

   First we have a text file including three lines, named 'test.txt'.

   ```
   one
   two
   three
   ```

```
>>>f = open('test.txt', 'r')   #using absolute path or relative path
>>>f.read()                    #read all characters by default
'one\ntwo\nthree'             #QUESTION: where is the pointer now???
>>>f.read()
''
>>>f.seek(0)                   #reset the pointer to the beginning
>>>f.read(6)                   #read next 6 characters from where the
pointer is
'one\ntw'
```

`read()` : extract a string that contains all characters in the file.

`seek()`: set the pointer to any position in the file, and `read()` will then read the remaining part of file.

You can read lines in a file with `readline()` or `readlines()`.

```
>>>f.readline()
'one'
>>>f.readline()                #read the next line every time
'two'
>>>f.seek(0)
>>>f.readlines()
['one\n', 'two\n', 'three']
```

2. **Write to a file**

   Similarly... we juse mode 'w', `write()` and `writelines()`.

```
>>>f = open('write.txt', 'w')
>>>f.write('one\n')
>>>f.writelines(['two\n', 'three'])
>>>f.close()                   #remember to close file after use.
```

```
one
two
three
```

   Notice: Whilst 'w' mode is activated, any existing files with the same name will be erased.

3. `with` **statement**

   It is an alternative way to open files.

   GOOD NEWS: file will be closed automatically on exit of the inner block logic.

```
with open('file_name', 'mode') as f:
```

*Then we come back to our **iterator**. Why it can make our code cleaner?*

See this example:

```
with open('data.txt', 'r') as f:
    while True:
        line = f.readline()
        if not line:
            break
            else:
                print(line)
```

This code prints the contents of the `data.txt` file. Instead of using a while loop, we can apply an **iterator**, which simplyfies our task.

```
with open('data.txt', 'r') as f:
    for line in f:
        print(line)
```

The `open()` function returns a file object, which is an iterator. We can use it in a for loop. With the usage of an iterator, the code is cleaner. Yeah!

Part3: A special type of iterator: generator

1. **Generator expression**

   It is similar to a list comprehension. But the difference is that a generator expression returns a generator, not a list.

   A generator expression is created with round brackets. Creating a list comprehension in this case would be very inefficient because the example would occupy a lot of memory unnecessarily. Thus, we create a generator expression, which generates values lazily on demand.

   (Here we generate 100 values with a generator without extensive usage of memory)

   ```
   n = (e for e in range(50000000) if not e % 3)
   i = 0
   for e in n:
       print(e)
       i += 1
       if i > 100:
           raise StopIteration
   ```

   Try the following experiment:

```
>>> L = [x * x for x in range(3)]
>>> L #0 1 4
>>> g = (x * x for x in range(3))
>>> g #<generator object <genexpr> at ...>
>>> next(g) #0
>>> next(g) #1
>>> next(g) #4

>>> next(g)
```

```
Traceback (most recent call last):
  File "<pyshell#19>", line 1, in <module>
    next(g)
StopIteration
```

```
>>> g = (x * x for x in range(10))
>>> for n in g:
...     print(n)
```

Therefore, after we create a generator, we usually do not prefer to call `next()`. Instead, we use a `for` loop to iterate it. Meanwhile, there is no need to pay attention to the rasing error about `StopIteration`.

Generator is really strong. If the algorithms are too intricate to implement by a for loop likewise a list comprehension, we can use functions。

2. **Generator**

   Generator is a 'function' that can be used to control the iteration behaviour of a loop. A generator is similar to a function returning an array. A generator can be used to generate a sequence of numbers. But unlike functions, which *return* a whole array, a generator *yields* one value at a time. This requires less memory. A generator returns an object on which you can call `next`, such that for every call it returns some value, until it raises a `StopIteration` exception.

   Generators in Python:

   - Are defined with the `def` keyword
   - Use the `yield` keyword
   - May use several `yield` keywords
   - Return an iterator

     For example:

```
def fib(max):
    n, a, b = 0, 0, 1
    while n < max:
        print(b)
        a, b = b, a + b
        n = n + 1
    return 'done'

>>> fib(10)

def fib_g(max):
    n, a, b = 0, 0, 1
    while n < max:
        yield b
        a, b = b, a + b
        n = n + 1
    return 'done'

>>> for n in fib_g(6):
...     print(n)
```

Part4: Anonymous Functions (lambda)

Other than list comprehensions, generators, another way to specify the key is to use lambda expressions.

Small anonymous functions can be created with the `lambda` keyword. Lambda functions can be used wherever function objects are required, especially when the function is used only once. They are syntactically restricted to a single expression.

The syntax is `function_name = lambda [parameter_names]: returned expression`.

```
#example 1: add
def add(a, b):
    return a + b
#is equivalent to:
add = lambda a, b: a + b

>>> fn = lambda x: x**2
>>> fn(3)
9
>>> (lambda x: x**2)(3)
9
>>> (lambda x: [x*_ for _ in range(5)])(2)
[0, 2, 4, 6, 8]
>>> (lambda x: x if x>3 else 3)(5)
5

# multiline lambda example
>>> (lambda x:
... True
... if x>0
... else
... False)(3)
True
```

Another use of lambda is to pass a small function as an argument, which can be used in specifying the key in sort() or sorted().

If you are not familiar with `sort` and `sorted`, look HERE! Otherwise, skip it!

sort() : a built-in function for list type in Python. list.sort() sorts the list **in-place**, mutating the list indices, and returns None.

```
>>>list1 = [1, 6, 4, 15, 3]
>>>list_new = list1.sort()
>>>print(list1)
[1, 3, 4, 6, 15]
>>>print(list_new)
None
```

sorted() : a built-in function in Python which can be used on any iterable, such as list, dictionary(sort by keys), tuples. sorted(list) returns a **new** sorted list, leaving the original list unaffected.

```
>>>list2 = sorted(list1)
>>>print(list2)
[1, 3, 4, 6, 15]
>>>print(list1)
[1, 6, 4, 15, 3]
```

The complete syntax for sort() and sorted() is `sort(key=None, reverse=False)` and `sorted(iterable, key=None, reverse=False)`. A custom key function can be supplied to customize the sort order. If you have a list of tuples, it gives you the ability to control on which element of the tuple the sorting must to run against. The default order is sort on 1st element first, if same then sort on 2nd element, ...

Now go back to see the magic of lambda!

```
>>> pairs = [(1, 'one'), (2, 'two'), (3, 'three'), (4, 'four')]
>>> pairs.sort(key = lambda pair: pair[1])
>>> pairs
[(4, 'four'), (1, 'one'), (3, 'three'), (2, 'two')]
#sorted with 2nd element of tuples (alphabetical order)
```

B. Exercises

Exercise 1

Part 1

Download the test file "the_best_programming_language.txt" from Moodle, write a program to complete the following tasks:

1. Replace all the words "Java" by "Python" in the file and save it to a new file named "output.txt"
2. Count all words frequency in this new file "output.txt", and save it to a file named "word_count.txt". This file should looks like:

```
to                    5
Python                5
and                   4
is                    3
easy                  3
advantages            2
languages             2
learn                 2
...
```

Part 2 Divide and Conquer

Based on your program in part 1, create follow functions for certain task.

1. One function named `read_file(p_filename)` for reading a text file and return the text of the file. (`p_filename` should be the name of the text file.)
2. One function named `word_count(P_string_list)` for count the word frequency of the input string, return a data structure which store the word with its frequency.
3. One function named `write_file(p_string, p_filename)` to create a new file named as `p_filename` and write `p_string` to the file.

Then try to create a main function `main()`, using those functions you created above to do the same work like in part 1.

Exercise 2

Download the test file "AreaCodes.txt" from Moodle. The file reads like:

```
201-New Jersey
202-District of Columbia
203-Connecticut
204-Manitoba
205-Alabama
206-Washington
207-Maine
208-Idaho
209-California
210-Texas
...
```

Each line is a combination of area code with the name of the state. A state can have multiple area code, like the state have 4 area codes: 303, 719, 720 and 970, but one area code must refer to one state name.

We want you to write a program to combine all state name with its area code then write it to a new file, like:

```
Alabama                    205
                           251
                           256
                           334
                           659
                           938

Alaska                     907
                           908

Alberta                    403
                           587
                           780

Arizona                    480
                           520
                           602
                           623
                           928
...
```

Think about your program structure before you start to write your program. Can you use the same function build in exercise 1? Can you update your function in exercise 1 for this exercise?

Optional (Answers will be later uploaded onto Moodle)

1. Modify your exercise 2 program to allow users to add new state names with their area codes.
2. Write a function ***fib(n)*** that returns Fn. For example, if *n* = 0, then *fib()* should return 0. If n = 1, then it should return 1. For n > 1, it should return Fn-1 + Fn-2.

   Try different solutions.

   Hint:

   1) Recursion,

   2) Dynamic Programming(you can search by yourself if interested),

   3) Space Optimized

   ...