Lecture 18 (2-25-26)

Python: Iterators, Generators

Motivating Questions:

1. What is exactly is an **iterator?**
2. How do we create and use **generators?**

Notes:

- remember a **generator** is not stored in memory
- so if you loop through a generator using a loop you can only access the items once
- if you try looping through again you won't get anything
- in most cases you do not convert generator to list

Idea of a generator:

- rather than aggregating the data into a big pile
- Instead it hands one task to someone else after finishing
- When you call a generator and the execution sees `yeild`, that means the execution is **suspended**
  - enables concurrency
- generators save you **memory**
- the key thing about generators is that they use concurency to avoid allocating memory
- `yeild from` yields the generator and internally loops though it

`review1.py` : An example of iterative programming

```
#!/usr/bin/env python3

# Imperative

numbers = range(0, 10)

doubled = []
```

```
for n in numbers:
    doubled.append(n * 2)

selected = []
for d in doubled:
    if not d % 4:
        selected.append(d)

for s in selected:
    print(s)
```

`review2.py` : Same task but functional programming

```
#!/usr/bin/env python3

# Functional

numbers = range(0, 10)

doubled = map(lambda n: n*2, numbers)

selected = filter(lambda d: not d % 4, doubled)

for s in selected:
    print(s)
```

`review3.py` : Same task but list comprehension

```
#!/usr/bin/env python3

# List Comprehension

numbers = range(0, 10)

doubled = [n*2 for n in numbers]

selected = [d for d in doubled if not d % 4]
```

```
for s in selected:
    print(s)
```

`iterators.py`

```
import os

#list
for element in [1,2,3,4]
     print(element)

#string
for letter in 'samuel':
     print(letter)

#dict
for var, value in os.environ.items():
     print(var, value)

#file
for line in open('/etc/passwd'):
     print(line.rstrip())
```

Example of how to get stuff from iterator:

```
i = iter(range(5))

next(i) #0
next(i) #1
next(i) #2
next(i) #3
next(i) #4

next(i) # throws StopIteration
```

```
##################

r = reversed(range(5))

next(r) = 4
next(r) = 3
next(r) = 2

#you cannot do the following (Because an iterator IS NOT a list):
len(r)
r[0]
```

`triples.py` Using a Generator

```
#!/usr/bin/env python3

# List

def triples(sequence):
    ts = []
    for i in sequence:
        ts.append(i * 3)
    return ts

# Generator
def triples_gr(sequence):
     for i in sequence:
          print('before')
          yield i * 3
          print('after')


# Main Execution

numbers = triples_gr(range(10))

for i in triples([1, 2, 3]):
    print(i)
```

---

example of generator comprehension

```
return (number for number in map(int, stream) if number % 2)
```

---

slist.py

```
#!/usr/bin/env python3

from typing      import Iterator, Optional
from collections import namedtuple
from dataclasses import dataclass

# Structures                                    # Review: singly linked list

'''
Node = namedtuple('Node', 'data next')          # Review: named tuple
'''

@dataclass
class Node:                                     # Discuss: Data class
    data: str = ''
    next: Optional['Node'] = None

# Functions (Iterative)

'''                                             # Discuss: Iterative (List)
def slist_data(head: Optional[Node]) -> list[str]:
    data = []
    curr = head
    while curr:
        data.append(curr.data)
        curr = curr.next
    return data
                                                # Discuss: Recursion (List)
def slist_reverse(head: Node) -> list[str]:
    return slist_reverse_r(head.next, head)
                                                # Discuss: Helper Function
def slist_reverse_r(curr: Optional[Node], prev: Node) -> list[str]:
    if curr is not None:                        # Recursive case
        data = slist_reverse_r(curr.next, curr)
    else:                                       # Base case
```

```
        data = []

    return data + [prev.data]                   # Discuss: Concatenating
lists
'''

# Functions (Generators)
                                                # Discuss: Iterative
(Generator)
def slist_data(head: Optional[Node]) -> Iterator[str]:
    curr = head
    while curr:
        yield curr.data
        curr = curr.next
                                                # Discuss: Recursion
(Generator)
def slist_reverse(head: Node) -> Iterator[str]:
    return slist_reverse_r(head.next, head)
                                                # Discuss: Helper Function
def slist_reverse_r(curr: Optional[Node], prev: Node) -> Iterator[str]:
    if curr is not None:                        # Discuss: yield from
        yield from slist_reverse_r(curr.next, curr)

    yield prev.data

# Main Execution

def main():
    slist = Node('a', Node('b', Node('c', Node('d', None))))

    for item in slist_data(slist):
        print(item)

    print()

    for item in slist_reverse(slist):
        print(item)

if __name__ == '__main__':
    main()
```