EECS 1210 Introduction to Programming in Python
Spring Semester 2024
Prof. Pai H. Chou

Self-Check 12

Answer the following questions to check your understanding of your material. Expect the same kind of questions to show up on your tests.

1. Definitions and Short Answers - functions

1. Given a for loop:

```
1 for i in L:
2     print(i)
```

Can L be the following? If so, what does the loop print? If not, why not?

   a. ['a', 'b', 'c'] a b c
   b. ('a', 'b', 'c') a b c
   c. 'abc'a b c
   d. {'a', 'b', 'c'}a c b
   e. {'a': 100, 'b': 200, 'c': 300}a c b
   f. 0xabcd TypeError: 'int' object is not iterable
   g. range(3) 0 1 2
   h. 23+4j TypeError: 'complex' object is not iterable

2. Given an iterable data structure L,
   a. How do you obtain an **iterator** r of L? [r = iter(L)]
   b. Once you have an iterator r, what can you do to get the next value? [next(r)]
   c. What happens when you call next(r) but your iterator r has finished iterating over all values of L? [you get a StopIteration exception]
   d. Is there a limit to the number of iterators that you can create on the same iterable? no
3. Assume you have

```
1 D = ['Sun', 'Mon', 'Tue', 'Wed', 'Thu', 'Fri', 'Sat']
2 r = iter(D)
3 L = [next(r) for i in range(3)]
4 s = iter(D)
5 M = [next(s) for i in range(2)]
```

after executing these five lines

   a. What is the value of L?
   b. What is the value of M?

c. What is the value of D?

```
>>> L
['Sun', 'Mon', 'Tue']
>>> M
['Sun', 'Mon']
>>> D
['Sun', 'Mon', 'Tue', 'Wed', 'Thu', 'Fri', 'Sat']
```

4. Recall the Vector class from the previous lecture,

```
1  import operator as op
2  class Vector:
3      def __init__(self, *v):
4          self._v = list(v) # covert tuple to list
5      def __repr__(self):
6          return __class__.__name__+repr(tuple(self._v))
```

Suppose a class defines an __iter__() special method, and v is an instance of Vector.

a. How does Python intend that v's __iter__() special method be invoked by the programme r? Hint: not v.__iter__()
iter(v)

b. What kind of object should the __iter__() method return?
[an iterator object.]

c. What is one simple way to implement Vector's __iter__() method, given that the iterator f or Vector would essentially be the same as the iterator for the list self._v ? [def __iter__(s elf): return iter(self._v)]

5. An alternative to part 4.(c) is to define a class for VectorIterator, and Vector's __iter__() method w ould instantiate and return it. The code is as follows:

```
1  class Vector:
2      def __iter__(self):
3          return Vector_Iterator(self):
4      ....
5  class VectorIterator:
6      def __init__(self, vec):
7          self._vec = vec
8          self._i = 0
9      def __next__(self):
10         if self._i >= len(self._vec):
11             raise StopIteration
12         val = self._vec[self._i]
13         self._i += 1
14         return val
```

a. In VectorIterator's constructor, what is the purpose of initializing _i = 0? 調用時會看i的值來決定取第幾個, 所以i初值設0便可從第一個開始取
b. Why does VectorIterator's constructor need to set its _vec attribute to the iterable? Why isn't it enough to just keep track of its position _i?
因為vector之值可能會變動? 助教: 不太確定
c. How does Python intend that the __next__() method of a VectorIterator instance vi be invoked? Hint: not vi.__next__()
[next(vi)]
d. How does __next__() special method indicate that it has finished iterating all elements?
except StopIteration:

6. Assume Vector is iterable, rewrite the following for-loop using a while loop and explicit iter() instantiation, next(), and catching StopIteration exception:

```
1 v = Vector(7, 1, 4, 3, 9, 6, 5)
2 for i in v:
3     print(i, end='')
```

```
v = Vector(...)
it = iter(v)
try:
    while True:
        print(next(it), end='')
except StopIteration:
    pass
```

7. Can any iterable object v be passed as arguments to
a. list(v) [yes]
b. max(v) [only if the elements can be compared]

8. For the Blackjack game example, Card is declared as a class:

```
1  class Class:
2      ACE, JACK, QUEEN, KING = 'A', 'J', 'Q', 'K'
3      FACES = (ACE,2,3,4,5,6,7,8,9,10, JACK, QUEEN, KING)
4      SUITS = tuple(map(chr, (9824, 9827, 9829, 9830)))
5      SPADE, CLUB, HEART, DIAMOND = SUITS # ♠ ♣ ♥ ♦
6      def __init__(self, suit, face):
7          self._suit = suit
8          self._face = face
9      def __int__(self):
10         if self._face in {Card.JACK,Card.QUEEN,Card.KING}:
11             return 10
12         return 1 if self._face == Card.ACE else self._face
13     def __str__(self):
14         return self._suit + str(self._face)
```

```
15    def __repr__(self):
16        return __class__.__name__ + \
17               repr((self._suit, self._face))
```

a. Why is it a good practice to declare class attributes such as SPADE, CLUB, HEART, and DIAMOND even though Python3 handles unicode character literals such as '♠' '♣' '♥' '◆'.
比較直觀,不用查ascii碼且將來若是想要新增花色也只要改class即可

b. What is the purpose of special method __int__()?
把不是數字的點數轉成21點規則對應的數字

c. Why declare a __str__() special method even though __repr__() also exists and can make a string that represents the card?
因為遊戲內只要知道花色跟數字,不需要給玩家知道別的資訊

d. Is Card class **iterable**? Should it be iterable?
不是,也不應該是

e. Is Card class for instantiating **iterators**?

Deck class use

```
>>> d = Deck()
>>> di = iter(d)
>>> next(di)
Card('♠', 'A')
>>> next(di)
Card('♠', 2)
>>> d.shuffle()
>>> list(map(str, d._deck))
['♠5', '♠4', '♥4', '♠8', '♦K', '♣3', '♥7', '♠9', '♥5', '♥9', '♣4',
'♦Q', '♣7', '♥K', '♣5', '♣8', '♦J', '♣K', '♠6', '♥10', '♣J', '♠Q',
'♦3', '♠3', '♦6', '♥A', '♠K', '♦7', '♦A', '♠2', '♦9', '♦10', '♣A',
'♣9', '♥2', '♦5', '♦2', '♦4', '♠J', '♥J', '♥3', '♥Q', '♥8', '♥6',
'♠A', '♣7', '♠10', '♣10', '♣2', '♣6', '♣Q', '♦8']
>>> di = iter(d)
>>> next(di), next(di), next(di), next(di)
(Card('♠', 5), Card('♠', 4), Card('♥', 4), Card('♠', 8))
```

d是iterable
di才是iterator
而透過next(di)方法得到的才是Card Class的Instance

9. Continuing with the BlackJack example, a separate class named Deck is also declared.

```
1  class Deck:
2      def __init__(self):
3          self._deck = [Card(suit, face) \
4              for suit in Card.SUITS for face in Card.FACES]
5      def shuffle(self):
6          import random
7          random.shuffle(self._deck)
```

```
8    def __iter__(self):
9        return iter(self._deck)
```

a. Is Deck an iterable? If so, is it required to implement the __getitem__() special method?
是Iterable 且因為提供了__iter__這個function 這樣就會以自已寫的iter為主不會再用原本的iter呼叫getitem

b. Explain how the Deck class is able to create iterators by simply returning iter(self._deck) from its __iter__() special method. Explain why this works.
self._deck is a list, and since a list is iterable, it can simply use the same iterator as list's iterator to return one card at a time when next() is called.

10. In Single-player BlackJack,

```
1  def BlackJack():
2      D = Deck()
3      D.shuffle()
4      total = 0
5      it = iter(D)
6      while True:
7          c = next(it)
8          total += int(c)
9          print(f'your card: {c}, total = {total}.', end='')
10         if total > 21:
11             print(f'you lose! total = {total}')
12             break
13         if total == 21:
14             print(f'you win! total = 21')
15             break
16         ans = input('More cards? [y/n] ')
17         if ans not in 'Yy':
18             c = next(it) # draw one more to test
19             print(f'next card {c}. You ' +\
20                 ('win' if total + c > 21 else 'lose'))
21             break
```

a. What kind of object is it as created on line 6?
iterator on the deck of cards

b. What kind of object is returned by a call to next(it) on line 7 or 18?
Card

c. Why doesn't this program have to handle the case where the iterator raises StopIteration exception when the deck is empty?
因為不可能把牌抽空,在那之前就爆點了

11. Is the following a **function** or a **generator**?

a.
```
def X(z):  # generator
    for i in range(20):
        yield i
```

b.
```
def Y(z):          # function
    for i in range(20):
        return i
```

c.
```
def K(z):   # generator
    for i in range(20):
        yield i
    return -1
```

12. if fib() is a generator for Fibonacci numbers, what is the syntax for

a. instantiating a generator, g=fib()

b. generate the initial number, init_num = 0

c. generate 10 more numbers after?

Fill in the blanks below. next(g)

```
g = ____  # instantiate generator
init_num = _________
print('initial number = ', init_num)
for i in range(10):
    num = _______
    print(num)
```

13. Assume

g = fib() is a generator for Fibonacci numbers, and

r = iter(deck) is an iterator where deck is an instance of iterable class Deck

Which of the following are allowed?

a. list(r) allowed

b. list(g) not allowed

c. list(fib()) not allowed

d. list(deck) allowed

e. [i for i in r] allowed

f. [i for i in g] not allowed

g. [i for i in fib()]not allowed

h. [i for i in iter(deck)] allowed

i. next(r) allowed

j. next(g) allowed

k. x, y, z = deck

ValueError: too many values to unpack (expected 3)

l. x, y, z = fib()

ValueError: too many values to unpack (expected 3)

m. x, y, z = g

ValueError: too many values to unpack (expected 3)

n. x, y, z = r

ValueError: too many values to unpack (expected 3)

2. Programming

1. (Difficulty: ★★☆☆☆) Define a generator function CharRange, which generates a range of characters with inclusive bounds. It takes two parameters for the starting and ending characters. It yields one character at a time whose unicode number is one closer to the ending. For example,

```
$ python3 -i charrange.py
>>> cr = CharRange('A', 'E')
>>> list(cr)
['A', 'B', 'C', 'D', 'E']
>>> dr = CharRange('E', 'A')
>>> list(dr)
['E', 'D', 'C', 'B', 'A']
>>>
```

Hint: the generator looks like

```
def CharRange(start, end):
    ...
```

It helps to convert between the character and the code using the ord() and chr() functions. To support stepping up or down, you need to check if the start is larger or smaller than the end. You may use range() to get one value at a time, but range() works for integers only; also, range's bound is exclusive, not inclusive, so you will need to make an adjustment for the bound's value. A generator uses yield instead of return to pass values back. After you finish yielding values, you don't have to do anything special, and your function will implicitly return None to mark the end of generation.

2. (Difficulty: ★★★☆☆) Define an iterable class named DaysInYear for iterating the days in a year. It takes the year as the argument to the constructor. Instead of implementing the __iter__ method to return the iterator object, it implements the __getitem__ method to return the ith value. The index to __getitem__ indicates the i^{th} day of the year, where i = 0 means January 1, i = 1 means January 2, etc.

```
$ python3 -i daysinyear.py
>>> y = DaysInYear(2019)
>>> y[5]
'2019.01.04'
```

```
>>> y[364]
'2019.12.31'
>>> y[31]
'2019.02.01'
>>> y[365]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "daysinyear.py", line 15, in __getitem__
    raise StopIteration
StopIteration
```

By defining the __getitem__ method, it makes the class iterable and you don't need to define the __iter__ method to return an iterator object -- the caller is responsible for tracking the iteration state. You do need to raise a StopIteration exception when the index is beyond the last day.
This allows you to convert it to a list, use in a for loop, etc.

3. (Difficulty: ★★★★☆) Define a class named CountingTuple. It works like a tuple except it also keeps track of the number of times each element is accessed. It should also be iterable but its iterator outputs elements in decreasing order of access count. An access is defined by a call to __getitem__, which may be an int or a slice.

```
>>> d = CountingTuple(('A', 'B', 'C', 'D', 'E'))
>>> d[0], d[2], d[2], d[4]   # these call __getitem__
('A', 'C', 'C', 'E')         # access counts = [1, 0, 2, 0, 1] >>> d[-1], d[-2], d[4]
('E', 'D', 'E')              # access counts = [1, 0, 3, 2, 4]
>>> for i in d:
...     print(d)
...
E
C
D
A
B
```

As you can probably figure out, you should define CountingTuple by subclassing from the built-in tuple class, like

```
class CountingTuple(tuple):
    def __init__(self, d = ()):
        super().__init__(d)
        # additional code here
```

```
  def __getitem__(self, i):
      # i is the index or slice.
      # (1) use the same i to increment the access count,
      #     your code here...
      # (2) return what the base class does, as below
      return super()[i]
  def __iter__(self):
      # This returns an iterator that outputs elements in
      # order of decreasing access count.
```

need to implement the following methods:

a. The constructor:
   It should first call the superclass's init to initialize the tuple data structure, and then define additional data structures to keep an access count of the elements. A good one to use is a list structure, which can be indexed using the same index as that for accessing the tuple. It contains the access count for the corresponding element in the tuple and should be initialized to zero.

b. The `__getitem__(self, i)` method:
   It needs intercept the accesses to each element by incrementing the corresponding count. Note that the type of `i` parameter can be either `int` or `slice`. In any case, this method needs to return the value, which can can be done by calling its base class's `__getitem__` using the same `i`.

c. The `__iter__(self)` method:
   It needs to return an iterator object but in order of decreasing access count. To do so, one way is to make a list whose elements are (access count, value) and sort in decreasing order, i.e., reverse=True. Then, you can return an iterator that iterates over the sorted value (but without the access count).