National University of Singapore
School of Computing
CS1010S: Programming Methodology
Semester II, 2019/2020

Solutions for Recitation 5
Working with Sequences

Python

1. Equality Testing
   - `==` returns `True` if two objects are equivalent.
   - `is` returns `True` if two objects are identical, i.e. they are the same object.
2. Membership Testing
   - x `in` y returns `True` if x is contained in the sequence type y. There are three basic sequence types in Python: tuples, lists, and range objects. We will discuss lists after the midterm break.
3. Range Type
   - The range type represents an immutable sequence of numbers and is commonly used for looping a specific number of times in for loops.
   - `range(start, stop[, step])`: Creates a sequence starting at `start`, at intervals of `step`, up to (but not including `stop`). If `step` is zero, `ValueError` will be raised.
   - `range(stop)` : Creates a sequence starting at 0, at intervals of 1, up to (but not including `stop`).
   - The advantage of the range type over a regular tuple is that a range object will always take the same (small) amount of memory, no matter the size of the range it represents (as it only stores the start, stop and step values, calculating individual items and subranges as needed), i.e. $O(1)$ space.

Problems

1. Evaluate the following expressions:

```
1 == 1    # True

1 is 1    # True
```

Recursive Solution:

```
def contains(obj, tup):
    if tup == ():
        return False
    elif tup[0] is obj:
        return True
    else:
        return contains(obj, tup[1:])
```

Iterative Solution:

```
def contains(obj, tup):
    for item in tup:
        if item is obj:
            return True
    return False
```

Write a function `deep_contains` that will check if an object is nested arbitrarily deep within a tuple. For example,

```
x = (1,2)
a = ((1,2), ((3,4), x), (5,6))

contains(x,a) => False
deep_contains(x,a) => True
```

Recursive Solution:

```
def deep_contains(obj, tup):
    if tup == ():
        return False
    elif tup[0] is obj:
        return True
    elif type(tup[0]) == tuple and deep_contains(obj, tup[0]):
        return True
    else:
        return deep_contains(obj, tup[1:])
```

Iterative Solution:

```
def deep_contains(obj, tup):
    for item in tup:
        if item is obj:
            return True
        elif type(item) == tuple and deep_contains(obj, item):
            return True
    return False
```

2. The `accumulate` procedure discussed in lecture is also known as `fold_right`, because it combines the first element of the sequence with the result of combining all the elements to the right. There is also a `fold_left`, which is similar to `fold_right`, except that it combines elements working in the opposite direction:

```
def accumulate(fn, initial, seq):
    if seq == ():
        return initial
    else:
        return fn(seq[0], accumulate(fn, initial, seq[1:]))

def fold_left(fn, initial, seq):
    if seq == ():
        return initial
    else:
        return fn(fold_left(fn, initial, seq[:-1]),seq[-1])
```

(a) Suppose we also define the following functions:

```
def pair(a, b):
    return (a, b)

def divide(a, b):
    return a/b
```

What are the values of

```
fold_right(divide, 1,(1, 2, 3))
=> 1.5

fold_left(divide,1,(1, 2, 3))
=> 0.16666666666666666

fold_right(pair,(),(1, 2, 3))
=> (1, (2, (3, ())))

fold_left(pair,(),(1, 2, 3))
=>((((), 1), 2), 3)
```

(b) Give a property that `fn` should satisfy to guarantee that `fold_right` (or `accumulate`) and `fold_left` will produce the same values for any sequence.

Where `fn` *is represented with* $\oplus$ *which needs to be both:*

i. Associative:

$$a \oplus (b \oplus c) = (a \oplus b) \oplus c$$

ii. Commutative:

$$a \oplus b = b \oplus a$$

3. A *queue* is a data structure that stores elements in order. Elements are enqueued onto the tail of the queue. Elements are dequeued from the head of the queue. Thus, the first element enqueued is also the first element dequeued (FIFO, first-in-first-out). The `qhead` operation is used to get the element at the head of the queue.

```
qhead(enqueue(5, empty_queue()))
# Value: 5

q = enqueue(4, enqueue(5, enqueue(6, empty_queue())))

qhead(q)
# Value: 6

qhead(dequeue(q))
# Value: 5
```

(a) Decide on an implementation for *queue*.

We will use a tuple `q` to represent the queue. The head will be `q[0]`. Note that it is often not sufficient to specify the underlying data structure. We also have to specify certain conditions or constraints, which in this case is the position of the head of the queue.

(b) Implement `empty_queue`

```
def empty_queue():
    return ()
```

Order of growth in time? $O(1)$ Space? $O(1)$

(c) Implement `enqueue`; a function that returns a new queue with the element `x` added to the tail of `q`.

```
def enqueue(x, q):
    return q + (x,)
```

Order of growth in time? $O(n)$ Space? $O(n)$

Note that the $O(n)$ is due to the fact that the tuple is an immutable data structure and so there is a lot of copying.

(d) Implement `dequeue`; a function that returns a new queue with the head element removed from `q`.

```
def dequeue(q):
    return q[1:]
```

Order of growth in time? $O(n)$ Space? $O(n)$

(e) Implement `qhead`; a function that returns the value of the head element of `q`.

```
def qhead(q):
    return q[0]
```

Order of growth in time? $O(1)$ Space? $O(1)$

4. **Homework:** Suppose `x` is bound to the tuple `(1, 2, 3, 4, 5, 6, 7)`. Using `map`, `filter`, `accumulate` and/or lambdas (as discussed in Lecture), write an expression involving `x` that returns:

(a) `(1, 4, 9, 16, 25, 36, 49)`

⇒ `map(lambda i:i**2, x)`

(b) `(1, 3, 5, 7)`

⇒ `filter(lambda i: i%2 != 0, x)`

(c) `((1, 1), (2, 2), (3, 3), (4, 4), (5, 5), (6, 6), (7, 7))`

⇒ `map(lambda i:(i,i), x)`

(d) `((2, 4), 6)`

```
⇒ fold_left(lambda a, b: b if a == () else (a,b),
            (),
            tuple(filter(lambda i: i%2 == 0, x)) )
```

(e) `(((2,), 4), 6)`

```
⇒ fold_left(lambda a, b: (b,) if a == () else ((a,b)),
                     (),
                     tuple(filter(lambda i: i%2 == 0, x)) )
```

(f) The maximum element of `x: 7`

```
⇒ accumulate(lambda a, b: a if a>= b else b, 1, x)
```

(g) The minimum element of `x: 1`

```
⇒ accumulate(lambda a, b: b if a>= b else a, 9999, x)
```

(h) The maximum squared even element of `x: 36`

```
⇒ accumulate(lambda a, b: a if a>= b else b,
          0,
          tuple(map(lambda i:i**2, filter(lambda i:i%2 == 0, x))) )
```

(i) The sum of the square of each value in `x: 140`

```
⇒ accumulate(lambda a,b: a+b, 0, tuple(map(lambda i:i**2, x)) )
```

You are encouraged to provide multiple solutions for the above questions.

Note that in the above, because `map` and `filter` return an iterable instead of tuple, you need to convert iterable into tuple using the function `tuple` before passing to `fold_left` or `accumulate`.