

1 Mutability

Questions

1.1 Name two data types that are mutable. What does it mean to be mutable?

1.2 Name at least two data types that are not mutable.

1.3 Will the following code error? If so, why?

```
a = 1
b = 2
dt = {a: 1, b: 2}
```

```
a = [1]
b = [2]
dt = {a: 1, b: 2}
```

1.4 Fill in the output and draw a box-and-pointer diagram for the following code. If an error occurs, write Error, but include all output displayed before the error.

```
a = [1, [2, 3], 4]
c = a[1]
c
```

```
a.append(c)
a
```

```
c[0] = 0
c
```

```
a
```

```
a.extend(c)
c[1] = 9
a
```

```
list1 = [1, 2, 3]
```

2 Mutability, ADTs, Trees, Iterators/Generators

```
list2 = [1, 2, 3]
list1 == list2

list1 is list2
```

1.5 Check your understanding:

1 What is the difference between the append function, extend function, and the '+' operator?

2 Given the below code, answer the following questions: a = [1, 2, [3, 4], 5]
b = a[:]
b[1] = 6
b[2][0] = 7

What does b evaluate to?

What does a evaluate to? Are a and b the same? Please explain your reasoning.

2 Data Abstraction

Questions

2.1 What are the two types of functions necessary to make an Abstract Data Type? What do they do?

2.2 Assume that **rational**, **numer**, **denom**, and **gcd** run without error and behave as described below. Can you identify where the abstraction barrier is broken? Come up with a scenario where this code runs without error and a scenario where this code would stop working.

```
def rational(num, den): # Returns a rational number ADT
    #implementation not shown
def numer(x): # Returns the numerator of the given rational
    #implementation not shown
def denom(x): # Returns the denominator of the given rational
    #implementation not shown
def gcd(a, b): # Returns the GCD of two numbers
    #implementation not shown

def simplify(f1): #Simplifies a rational number
    g = gcd(f1[0], f1[1])
    return rational(numer(f1) // g, denom(f1) // g)

def multiply(f1, f2): # Multiplies and simplifies two rational numbers
    r = rational(numer(f1) * numer(f2), denom(f1) * denom(f2))
    return simplify(r)

x = rational(1, 2)
y = rational(2, 3)
multiply(x, y)
```

2.3 Check your understanding

1 How do we know what we are breaking an abstraction barrier?

2 What are the benefits to Data Abstraction?

3 Trees

Questions

3.1 Fill in this implementation of the Tree ADT.

```
def tree(label, branches = []):
    for b in branches:
        assert is_tree(b), 'branches must be trees'
    return [label] + list(branches)

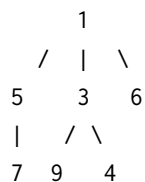
def is_tree(tree):
    if type(tree) != list or len(tree) < 1:
        return False
    for b in branches(tree):
        if not is_tree(b):
            return False
    return True

def label(tree):
    pass

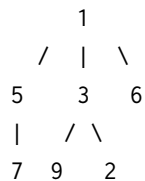
def branches(tree):
    pass

def is_leaf(tree):
    pass
```

3.2 A min-heap is a tree with the special property that every nodes value is less than or equal to the values of all of its children. For example, the following tree is a min-heap:



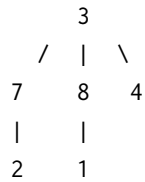
However, the following tree is not a min-heap because the node with value 3 has a value greater than one of its children:



Write a function **is_min_heap** that takes a tree and returns True if the tree is a min-heap and False otherwise.

```
def is_min_heap(t):
```

- 3.3 Write a function **largest_product_path** that finds the largest product path possible. A product path is defined as the product of all nodes between the root and a leaf. The function takes a tree as its parameter. Assume all nodes have a non-negative value.



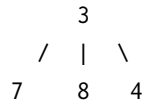
For example, calling **largest_product_path** on the above tree would return 42, since $3 * 7 * 2$ is the largest product path.

```
def largest_product_path(tree):
    """
    >>> largest_product_path(None)
    0
    >>> largest_product_path(tree(3))
    3
    >>> t = tree(3, [tree(7, [tree(2)]), tree(8, [tree(1)]), tree(4)])
    >>> largest_product_path(t)
    42
    """
```

- 3.4 Implement a function `max_tree`, which takes a tree `t`. It returns a new tree with the exact same structure as `t`; at each node in the new tree, the entry is the largest number that is contained in that node's subtrees or the corresponding node in `t`.

```
def max_tree(t):
>>> max_tree(tree(1, [tree(5, [tree(7)]),tree(3,[tree(9),tree(4)]),tree(6)])
tree(9, [tree(7, [tree(7)]),tree(9,[tree(9),tree(4)]),tree(6)])
    if ____:
        return ____
    else:
        new_branches= ____
        new_label = ____
        return ____
```

- 3.5 Challenge Question: The level-order traversal of a tree is defined as visiting the nodes in each level of a tree before moving onto the nodes in the next level. For example, the level order of the following tree is: 3 7 8 4



Write a function **level_order** that takes in a tree as the parameter and returns a list of the values of the nodes in level order.

```
def level_order(tree):
```

- 3.6 Challenge Question: Write a function **all_paths** which will return a list of lists of all the possible paths of an input tree, t. When the function is called on the same tree as the problem above, the function would return: [[3, 7], [3, 8], [3, 4]]

```
def all_paths(t):
```

```
    if _____:
```

```
        _____
```

```
    else:
```

```
        _____
```

```
        _____
```

```
            _____
```

```
                _____
```

```
        _____
```

4 Nonlocal Questions

4.1 Draw an environment diagram for the following code:

```
spiderman = 'peter parker'
def spider(man):
    def myster(io):
        nonlocal man
        man = spiderman
        spider = lambda stark: stark(man) + ' ' + io
        return spider
    return myster
truth = spider('quentin is')('the greatest superhero')(lambda x: x)
```


4.2 Draw an environment diagram for the following code:

```
fa = 0
```

```
def fi(fa):  
    def world(cup):  
        nonlocal fa  
        fa = lambda fi: world or fa or fi  
        world = 0  
        if (not cup) or fa:  
            fa(2022)  
            fa, cup = world + 2, fa  
            return cup(fa)  
        return fa(cup)  
    return world
```

```
won = lambda opponent, x: opponent(x)  
us = won(fi(fa), 2019)
```

- 4.3 Write **make_max_finder**, which takes in no arguments but returns a function which takes in a list. The function it returns should return the maximum value it's been called on so far, including the current list and any previous list. You can assume that any list this function takes in will be nonempty and contain only non-negative values.

```
def make_max_finder():  
    """  
    >>> m = make_max_finder()  
    >>> m([5, 6, 7])  
    7  
    >>> m([1, 2, 3])  
    7  
    >>> m([9])  
    9  
    >>> m2 = make_max_finder()  
    >>> m2([1])  
    1  
    """
```

4.4 Check your understanding:

```

x = 5
def f(x):
    def g(s):
        def h(h):
            nonlocal x
            x = x + h
            return x
        nonlocal x
        x = x + x
        return h
    print(x)
    return g
t = f(7)(8)(9)

```

- What is t after the code is executed?
- In the h frame, which x is being referenced? Which frame?
- In the g frame, is a new variable x being created?

5 Iterators and Generators

Questions

- 5.1 What is the definition of an iterable? What is the definition of an iterator? What is the definition of a generator? What built-in functions or keywords are associated with each. Give an example of each.

- 5.2 Evaluate if each line is valid? If not, state the error and how you would fix it.

```
>>> new_list = [2, 3, 6, 8, 8, 3]
>>> next(new_list)
```

```
>>> iter(new_list)[1]
```

```
>>> [x for x in iter(new_list)]
```

```
>>> for i in range(len(iter(new_list))):
...     new_list.append(2)
```

5.3 What is the difference between these two statements?

- a. **def** infinity1(start):
 while True:
 start = start + 1
 return start
- b. **def** infinity2(start):
 while True:
 start = start + 1
 yield start

What would python display?

```
>>> infinity1

>>> infinity2

>>> infinity1(2)

>>> infinity2(2)

>>> x = infinity1(2)

>>> next(x)

>>> y = infinity2(2)

>>> next(y)

>>> next(y)

>>> next(infinity2(2))
```

- 5.4 They can't stop all of us!!! Write a function **generate_constant** which, a generator function that repeatedly yields the same value forever.

```
def generate_constant(x):
    """A generator function that repeats the same value x forever.
    >>> area = generate_constant(51)
    >>> next(area)
    51
    >>> next(area)
    51
    >>> sum([next(area) for _ in range(100)])
    5100
    """
```

- 5.5 4.2 Now implement **black_hole**, a generator that yields items in seq until one of them matches trap, in which case that value should be repeatedly yielded forever. You may assume that **generate_constant** works. You may not index into or slice seq.

```
def black_hole(seq, trap):
    """A generator that yields items in SEQ until one of them matches TRAP, in which case that
    value should be repeatedly yielded forever.
    >>> trapped = black_hole([1, 2, 3], 2)
    >>> [next(trapped) for _ in range(6)]
    [1, 2, 2, 2, 2, 2]
    >>> list(black_hole(range(5), 7))
    [0, 1, 2, 3, 4]
    """
```

5.6 What Would Python Display?

```
>>> def weird_gen(x):
...     if x % 2 == 0:
...         yield x * 2
>>> wg = weird_gen(2)
>>> next(wg)
>>> next(weird_gen(2))
```

```
>>> next(wg)
```

```
>>> def greeter(x):
...     while x % 2 != 0:
...         print('hi')
...         yield x
...         print('bye')
>>> greeter(5)
```

```
>>> gen = greeter(5)
>>> g = next(gen)
```

```
>>> g = (g, next(gen))
>>> g
```

```
>>> next(gen)
```

```
>>> next(g)
```

An iterator _____ a generator
 A generator **is** a(n) _____ iterator

- 5.7 Write a generator function **gen_inf** that returns a generator which yields all the numbers in the provided list one by one in an infinite loop.

```
>>> t = gen_inf([3, 4, 5])
>>> next(t)
3
>>> next(t)
4
>>> next(t)
5
>>> next(t)
3
>>> next(t)
4
def gen_inf(lst):
```


- 5.8 Implement a generator function called `filter(iterable, fn)` that only yields elements of `iterable` for which `fn` returns `True`.

```
def naturals():
    i = 1
    while True:
        yield i
        i += 1

def filter(iterable, fn):
    """
    >>> is_even = lambda x: x % 2 == 0
    >>> list(filter(range(5), is_even))
    [0, 2, 4]
    >>> all_odd = (2*y-1 for y in range (5))
    >>> list(filter(all_odd, is_even))
    []
    >>> s = filter(naturals(), is_even)
    >>> next(s)
    2
    >>> next(s)
    4
    """
```

- 5.9 What could you use a generator for that you could not use a standard iterator paired with a function for?

- 5.10 Define **tree_sequence**, a generator that iterates through a tree by first yielding the root value and then yielding the values from each branch.

```
def tree_sequence(t):  
    """  
    >>> t = tree(1, [tree(2, [tree(5)]), tree(3, [tree(4)])])  
    >>> print(list(tree_sequence(t)))  
    [1, 2, 5, 3, 4]  
    """
```

- 5.11 Write a function **make_digit_getter** that, given a positive integer *n*, returns a new function that returns the digits in the integer one by one, starting from the rightmost digit.

Once all digits have been removed, subsequent calls to the function should return the sum of all the digits in the original integer.

```
def make_digit_getter(n):
    """ Returns a function that returns the next digit in n
    each time it is called, and the total value of all the integers
    once all the digits have been returned.
    >>> year = 8102
    >>> get_year_digit = make_digit_getter(year)
    >>> for _ in range(4):
    ... print(get_year_digit())
    2
    0
    1
    8
    >>> get_year_digit()
    11
    """
```

5.12 Sorry another environment diagram, but it's the last one I promise.

```
def iter(iterable):
    def iterator(msg):
        nonlocal iterable
        if msg == 'next':
            next = iterable[0]
            iterable = iterable[1:]
            return next
        elif msg == 'stop':
            raise StopIteration
    return iterator
def next(iterator):
    return iterator('next')
def stop(iterator):
    iterator('stop')

lst = [1, 2, 3]
iterator = iter(lst)
elem = next(iterator)
```