Discussion 7

DSC 20, Spring 2023

Midterm 2 Practice + Prep

Midterm Logistics - basically same as midterm 1

- Make sure to bring pen/pencil/eraser and your **student ID**
- Exam takes place Friday, May 19th in MANDE B-210 (in lecture)

Topics

- Iterators, Map, Filter,lambda
- Higher order function
- Argument passing (arg, *args, **kwargs)
- Time Complexity
- Recursion

note: Though not explicitly covered, you are still expected to be able to apply concepts from midterm 1.

Lambda Functions

- known as anonymous functions (their functions are so simple, they don't need a name)
- syntax: lambda (input): (operation)
- creating a lambda is O(1) (no operation is performed)

note: lambda functions can't include statements (ex. return, assert)

Map

Syntax: map(function, iterable)

- Map allows you to apply a function to all elements to an iterable input
- very common to use a lambda function as the function to apply
- returns an iterator through the iterable object, applying the function as it traverses

note: Without being called, creating a map is O(1).

Filter

Syntax: filter(function, iterable)

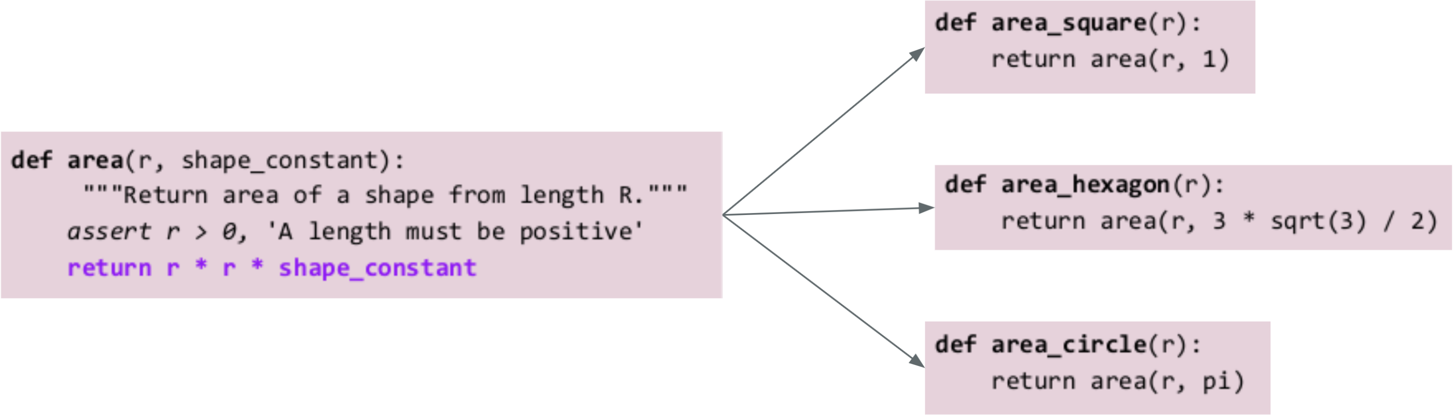
- Filter takes in a function that returns a boolean and only keeps elements that satisfy the function
- Very common to use a lambda function as the function. Keep in mind the function **must return a boolean**.
- Returns an iterator through the iterable object that only yields values that pass the function.

note: filters are a unique subset of maps. You **can** theoretically write filters as maps, but that's unnecessary complication.

note: Without being called, creating a filter is O(1).

Higher Order Function

- Design structure to minimize repetitive code.
- Returns another function thats built within the outer function.
- Prevents inner function from being exposed to operations in the global scope.



```python
def area(r, shape_constant):
     """Return area of a shape from length R."""
    assert r > 0, 'A length must be positive'
    return r * r * shape_constant
```

```python
def area_square(r):
    return area(r, 1)
```

```python
def area_hexagon(r):
    return area(r, 3 * sqrt(3) / 2)
```

```python
def area_circle(r):
    return area(r, pi)
```

*args

- Used when an unknown number of arguments will be passed into a function
- Denoted by * in the method header (IMPORTANT)
- processed in a similar manner to a list

In [93]:

```
def test(*names): # METHOD HEADER
    return
```

**kwargs

- Used when an unknown number of **keyworded** arguments will be passed into a function
- Denoted by ** in the method header (IMPORTANT)
- processed in a similar manner to a dictionary

In [94]:

```
def test(**grades): # METHOD HEADER
    return
```

default_arguments

- Basically normal arguments, but with a default value
- if no value is passed, default value is set
- if a value is passed, default value is overwritten

note: complex argument ordering: def func(normal_arguments, *args, default_args,* *kwargs)

on the exam if the order def func(normal_arguments, default_args, *args,* *kwargs) is presented, we will still accept it as a correct answer.

In [41]:

```
def test(exam, *names, questions = 18, **grades):
    return
```

In [44]:

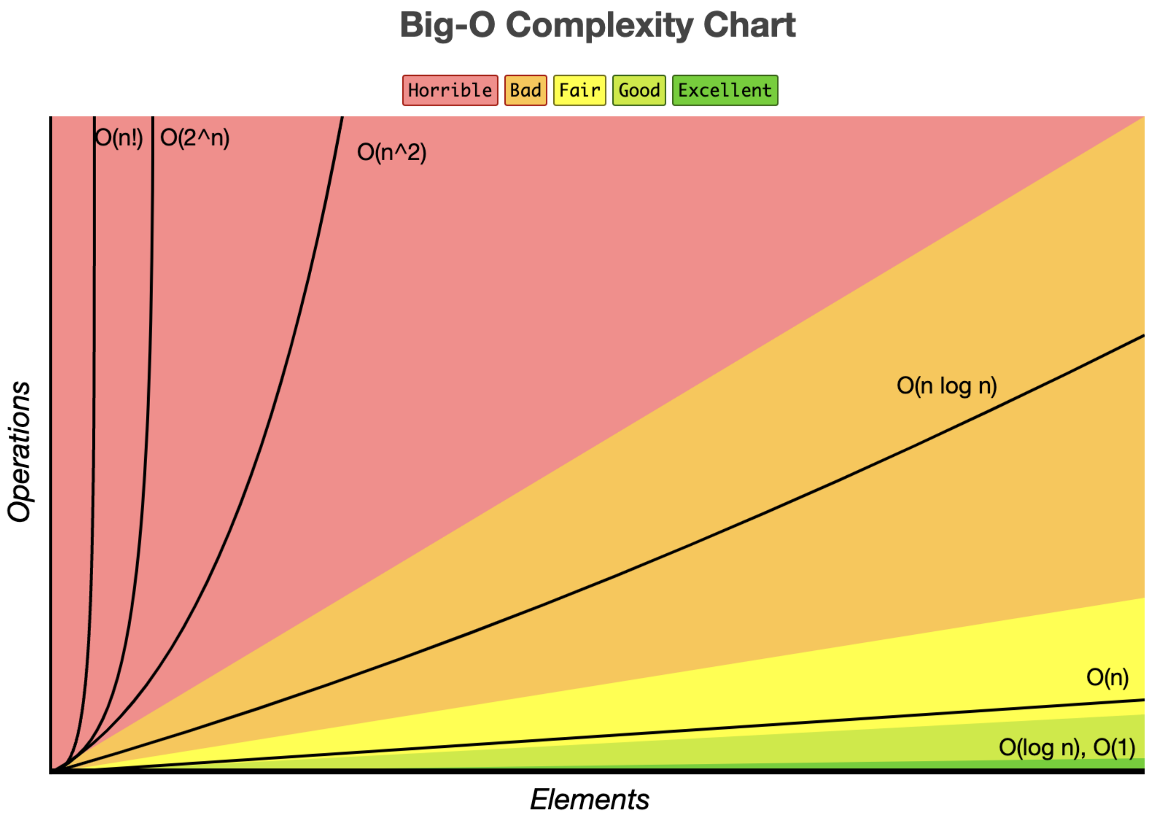
```
def test(exam, questions = 18, *names, **grades):
    return
```

Time Complexity

Utilize mathematical principles to classify code as runtime families to quantify their relative efficiency

tips:

1. Always look for "hallmark features" (ex. if I see i // 2, there's probably log involved)
2. If there are loops, check the iteration count AND the inside operations (don't assume n runtime)
3. Order of Growth follows mathematical principles. Only consider the largest term (without constants)


Big-O Complexity Chart
Horrible
Bad
Fair
Good
Excellent
O(n!)
O(2^n)
O(n^2)
O(n log n)
O(n)
O(log n), O(1)
Operations
Elements

source

Recursion

Recursion is a design method for code that features functions invoking itself.

- **base case** - determines the stop point for recursion and begins the argument passing up the "stack" of recursive calls. When writing recursion questions, always start with determining the base case.
- **recursive calls** - repeated invocation of function until the base case is reached. Keep in mind when writing recursive calls that every call needs to trend towards the base case.

Practice Questions

Question 1

For each equation, determine if the expression is True or False. If False, determine the actual time complexity.

1.

$$\sqrt{n} + n = O(n \log n)$$

2.

$$100n^3 + n^2 = O(n^3)$$

3.

$$\frac{1}{n} + \frac{n^2}{n} = O(n)$$

4.

$$n + n\log(n^3) + n\log(n^n) = O(n\log n)$$

5.

$$n + \log(n^n) = O(n\log(n))$$

Question 1 Solution

1. False -

$$O(n)$$

2. True

3. True

4. False -

$$O(n^2(\log n))$$

5. True

Question 2

For each piece of code, determine the time complexity.

In [12]:

```
def practice(x): # Question 1
    for i in range(10000000000):
        print(i)

def mod_10(n): # Question 2
    if n%10 == 0:
        return 10
    else:
        return mod_10(n-1)

def func(n): # Question 3
    j = 1
    curr_sum = 100
    while j <= n:
        for i in range(1, n//2):
            curr_sum += i*j
        j+=1
    return curr_sum

def foo(lst): # Question 4
    return filter(lambda x: x.lower() not in 'aeiou',lst)
```

Question 2 Solution

1. $O(1)$
2. $O(1)$
3. $O(n^2)$
4. $O(1)$

Question 3

What is the output?

In [ ]:

```
def trace(n):
    if n=='':
        return 'h'
    if n[0] == 'a':
        return 'y' + trace(n[1:])
    else:
        return trace(n[1:])
print(trace('apple'))
```

Question 3 Solution

In [15]:
```
print(trace('apple'))
```

```
yh
```

Since the first letter is 'a', the first recursive call is triggered, yielding a 'y' for now. Afterwards, each call is triggered, adding nothing until an empty string is passed, which results in 'h' being added.

Question 4

Write a function that mimics len() functionality using recursion

In [ ]:

```
def len_recursion(iterable):
    """
    >>> len_recursion([1,2,3,4])
    4
    """
    # Write your implementation here
```

Question 4 Solution

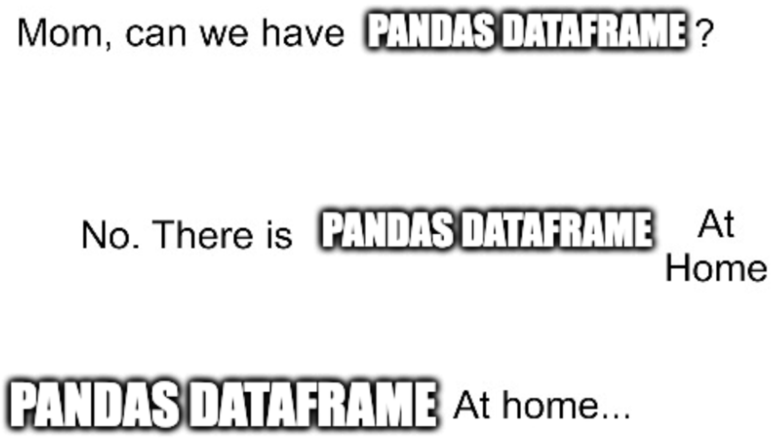
In [21]:

```
def len_recursion(iterable):
    if not iterable:
        return 0
    else:
        return 1 + len_recursion(iterable[1:])
len_recursion([1,2,3,4])
```

Out[21]:

```
4
```

Question 5

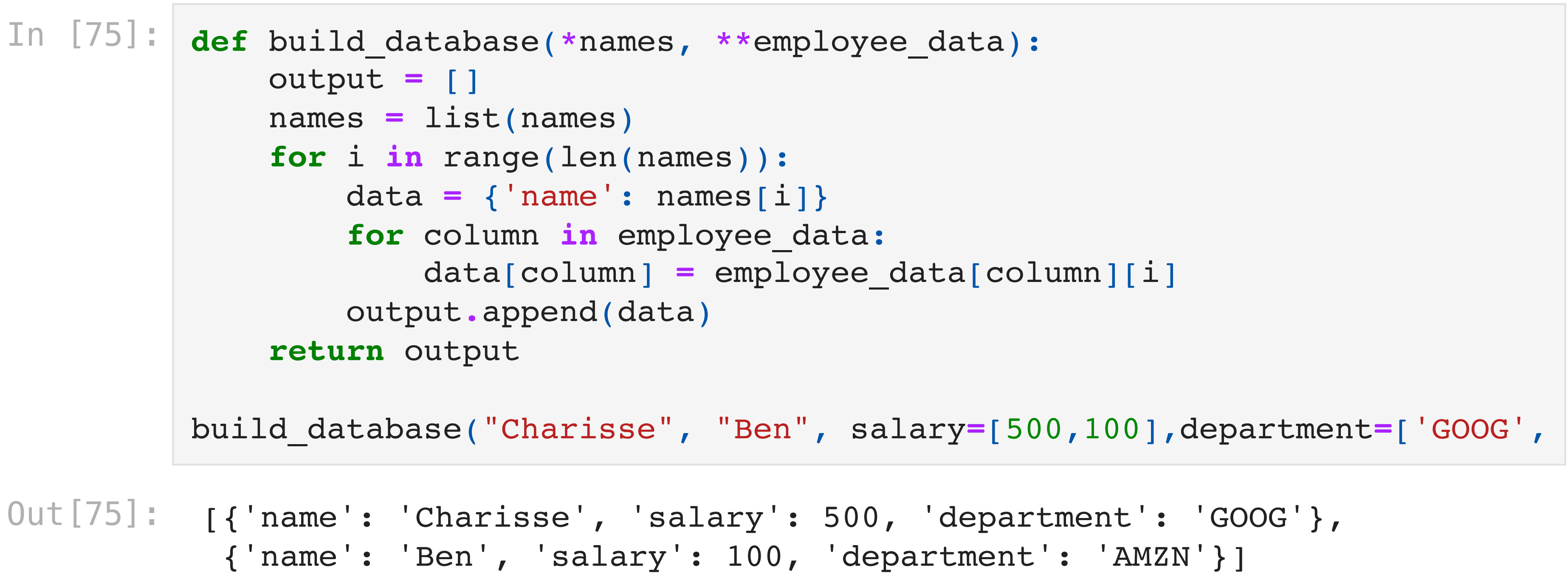

Mom, can we have PANDAS DATAFRAME?
No. There is PANDAS DATAFRAME At Home
PANDAS DATAFRAME At home...

Given an unknown number of names and unknown amount of keyworded information, write a function that creates a list of dictionaries to store the data. You may assume len(names) == len(employee_data).

In [114]:

```
def build_database(*names, **employee_data):
    """
    >>> build_database(Charisse, Ben,\
    salary=[500,100],department=['GOOG', 'AMZN'])
    [{"name": "Charisse", "salary":500, department": "GOOG"}, \
    {"name": "Ben", "salary":100, department": "AMZN"}]
    """
    # Write your implementation here
```

Question 5 Solution



In [75]:

```
def build_database(*names, **employee_data):
    output = []
    names = list(names)
    for i in range(len(names)):
        data = {'name': names[i]}
        for column in employee_data:
            data[column] = employee_data[column][i]
        output.append(data)
    return output

build_database("Charisse", "Ben", salary=[500,100],department=['GOOG',
```

Out[75]:

```
[{'name': 'Charisse', 'salary': 500, 'department': 'GOOG'},
 {'name': 'Ben', 'salary': 100, 'department': 'AMZN'}]
```

Question 6

Given a list of dictionaries representing employees of a company, create a new list containing only the names of employees who work in the "Sales" department and earn more than 50,000 per year using map, filter, and lambda.

In [110]:

```
employees = build_database("Suraj", "Eldridge", "Marina", "Babak", "Joh
salary=[60000,70000,75000,40000,45000], \
department=['Sales','IT','Sales','Sales','Marketing'])
employees
```

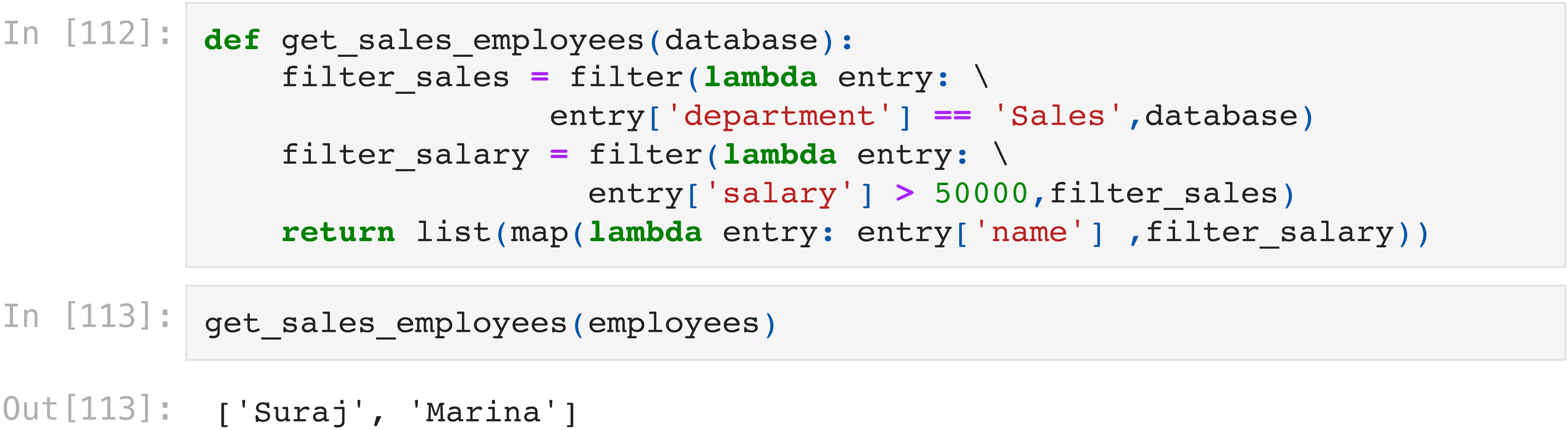
Out[110]:

```
[{'name': 'Suraj', 'salary': 60000, 'department': 'Sales'},
 {'name': 'Eldridge', 'salary': 70000, 'department': 'IT'},
 {'name': 'Marina', 'salary': 75000, 'department': 'Sales'},
 {'name': 'Babak', 'salary': 40000, 'department': 'Sales'},
 {'name': 'Johan', 'salary': 45000, 'department': 'Marketing'}]
```

In [111]:

```
def get_sales_employees(database):
    """
    >>> get_sales_employees(employees)
    ['Suraj', 'Marina']
    """
    # Write your implementation here
```

Question 6 Solution



In [112]:
```
def get_sales_employees(database):
    filter_sales = filter(lambda entry: \
                  entry['department'] == 'Sales',database)
    filter_salary = filter(lambda entry: \
                    entry['salary'] > 50000,filter_sales)
    return list(map(lambda entry: entry['name'] ,filter_salary))
```

In [113]:
```
get_sales_employees(employees)
```

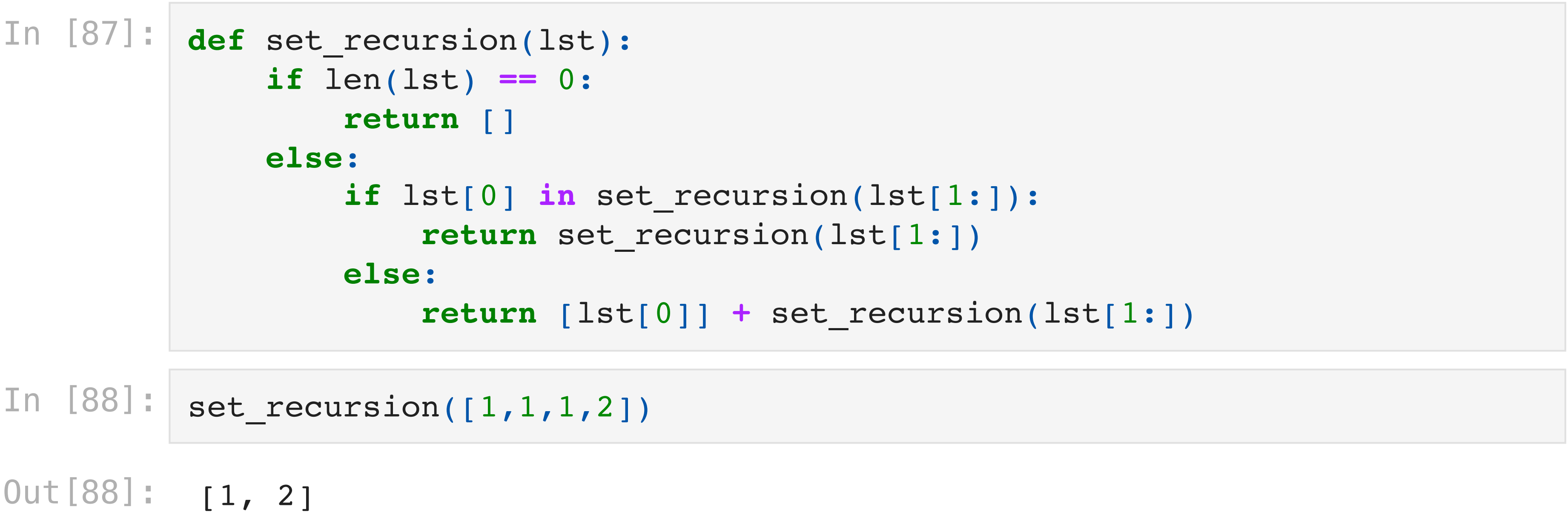
Out[113]:
```
['Suraj', 'Marina']
```

Question 7

Write a function that mimics set() cast functionality using recursion

```
In [86]: def set_recursion(lst):
             """
             >>> set_recursion([1,1,1,2])
             [1,2]
             """
             # Write your implementation here
```

Question 7 Solution



```
In [87]: def set_recursion(lst):
             if len(lst) == 0:
                 return []
             else:
                 if lst[0] in set_recursion(lst[1:]):
                     return set_recursion(lst[1:])
                 else:
                     return [lst[0]] + set_recursion(lst[1:])
```

```
In [88]: set_recursion([1,1,1,2])
```

```
Out[88]: [1, 2]
```

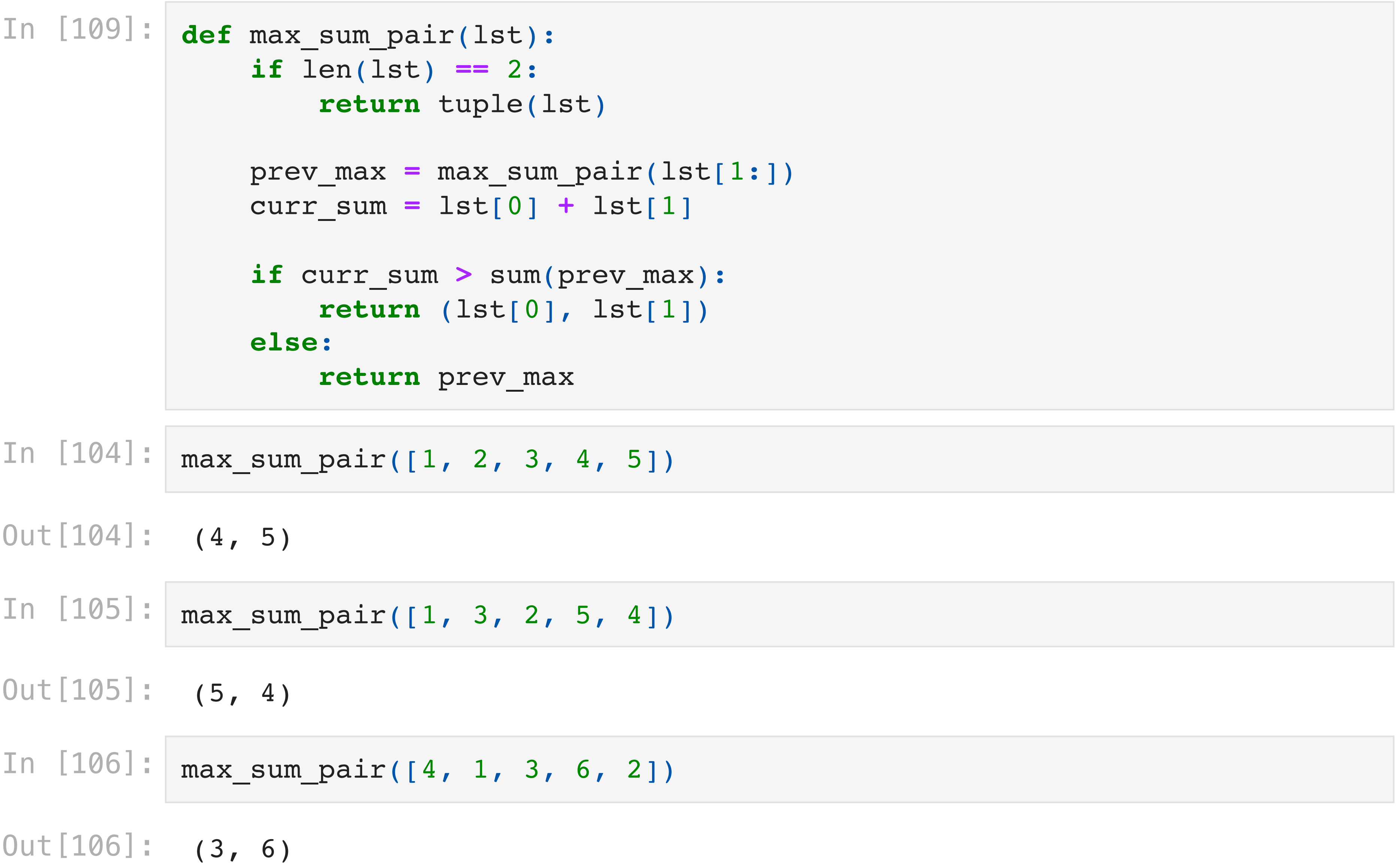
Question 8

Write a recursive function that finds the consecutive pair of numbers with the largest sum from a list.

In [107]:

```
def max_sum_pair(lst):
    """
    >>> max_sum_pair([1, 2, 3, 4, 5])
    (4, 5)
    >>> max_sum_pair([1, 3, 2, 5, 4])
    (5, 4)
    >>> max_sum_pair([4, 1, 3, 6, 2])
    (3, 6)
    """
    # Write your implementation here
```

Question 8 Solution



```
In [109]: def max_sum_pair(lst):
              if len(lst) == 2:
                  return tuple(lst)

              prev_max = max_sum_pair(lst[1:])
              curr_sum = lst[0] + lst[1]

              if curr_sum > sum(prev_max):
                  return (lst[0], lst[1])
              else:
                  return prev_max
```

```
In [104]: max_sum_pair([1, 2, 3, 4, 5])

Out[104]: (4, 5)
```

```
In [105]: max_sum_pair([1, 3, 2, 5, 4])

Out[105]: (5, 4)
```

```
In [106]: max_sum_pair([4, 1, 3, 6, 2])

Out[106]: (3, 6)
```

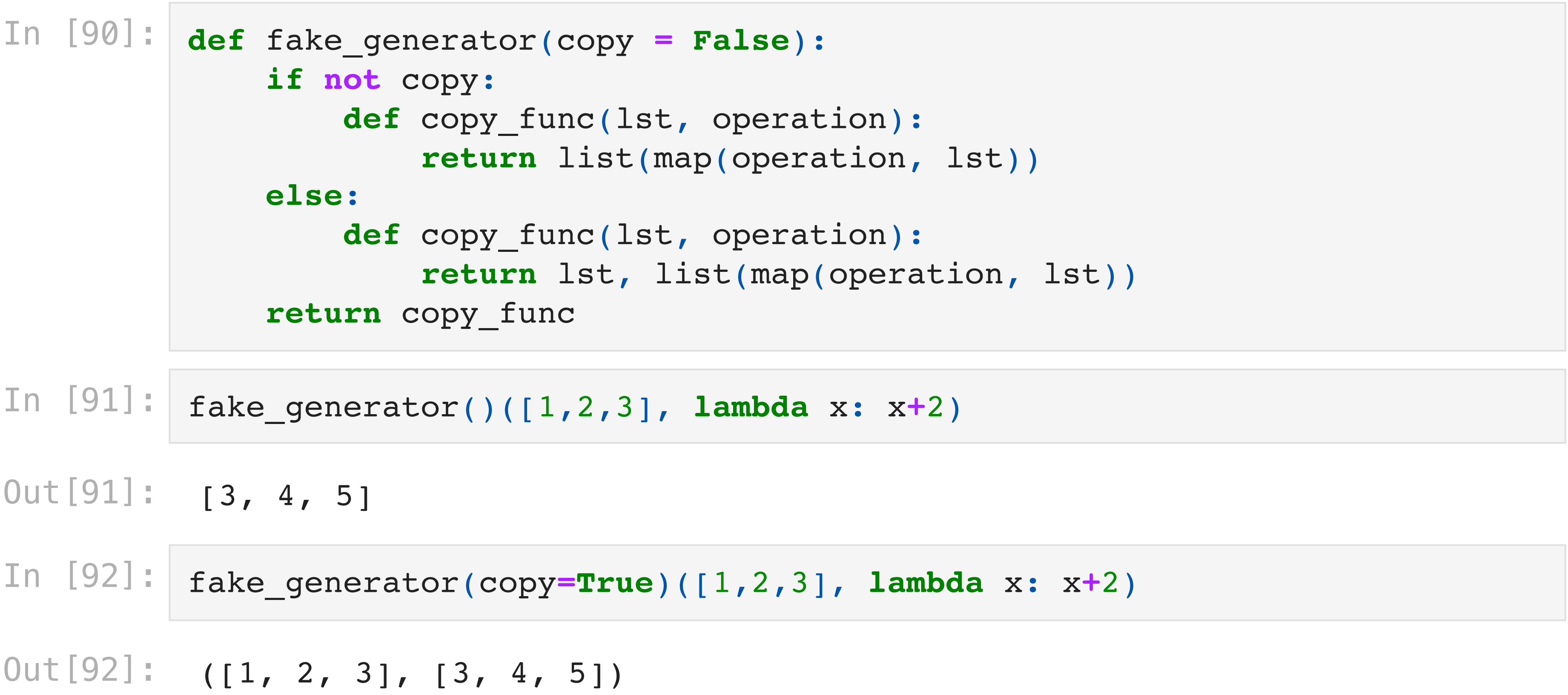
Question 9

Write a function that returns another function that maps a lambda to a list. Depending on the value of copy, include the original list in the output.

In [89]:

```
def fake_generator(lst, operation, copy = False):
    """
    >>> fake_generator([1,2,3], lambda x: x+2)
    [3,4,5]
    >>> fake_generator([1,2,3], lambda x: x+2, copy=True)
    ([1,2,3], [3,4,5])
    """
    # Write your implementation here
```

Question 9 Solution



In [90]:

```
def fake_generator(copy = False):
    if not copy:
        def copy_func(lst, operation):
            return list(map(operation, lst))
    else:
        def copy_func(lst, operation):
            return lst, list(map(operation, lst))
    return copy_func
```

In [91]:

```
fake_generator()([1,2,3], lambda x: x+2)
```

Out[91]:

```
[3, 4, 5]
```

In [92]:

```
fake_generator(copy=True)([1,2,3], lambda x: x+2)
```

Out[92]:

```
([1, 2, 3], [3, 4, 5])
```

Thanks for coming!

There's a discussion quiz on canvas!

Good luck on the midterm!