CS 5001: Intensive Foundations of CS

Final Synthesis

Xiaowen Sun

Dec 2022

Index

1. Chapter 0: Concept map

Recursion	In chapter 1 recursive math, 8 math functions are created in a recursion way, including factorial, square sum, Fibonacci, triple Fibonacci, power calculation, binary power calculation, traverse search, and binary search. Detailed in recursion_math.py
Error Handling	In chapter 2 Account register and login, in account_management.py, Error Handling is used in line 62 ~ 183, In admin_management.py, Error Handling is used in line 20 ~ 36. Try-except syntax is used to catch error and avoid crushing, an error is raised in such cases: *The email is not ended with "@northeastern.edu"* *Password does not have at least 8 characters* *Password is wrong* *The user does not exist when login* *The account has been registered when creating a new account*
Dictionaries and Sets	In chapter 2 Account register and login, the data structure of the dictionary and set are used to save email – password mapping in a dictionary, and save all emails in a set, allowing the user to delete, add, and change data by the functions. Detailed in account_management.py, line 4~29, 62~183, admin_management.py, line 4~36
Classes (Basic)	In chapter 2 Account register and login, I created Account_management class and Admin_management classes, the instance of two classes are used in the driver.py line 32~36, Two classes are in account_management.py and admin_management.py.

Stacks and Queues	In Chapter 3: Course waitlist & Cooking pancakes, a queue is used to represent a waitlist for a course in course_waitlist.py, including operations such as enqueue, dequeue, isempty, isfull. a stack is used to represent a plate for pancakes in cooking_pancakes.py, including operations such as pop, push, peek, isempty.
Efficiency (Very Basic)	In chapter 1 recursive math, 2.4 efficiency, the time and space complexity of list traverse, nested for loops, and binary search are given with explanation. The time efficiency are provided in all the functions in iteration_math.py and recursion_math.py.
Iteration	In chapter 1 recursive math, 7 math functions are created in an iteration way, including factorial, square sum, Fibonacci, triple Fibonacci, power calculation, binary power calculation, traverse search, and binary search. Detailed in iteration_math.py In chapter 2 Account register and login, in driver.py, a CLI based menu is created in a while loop.
Functions and Testing	In Chapter 3: Course waitlist & Cooking pancakes, a series of functions are created to manage complexity and reduce code, and use a combination of functions to compare the outcome and the supposed result to test correctness. detailed in in course_waitlist.py and cooking_pancakes.py.
Unit test(Basic)	In chapter 1 recursive math, 15 test cases are created for all recursive math and iteration math functions to verify the correctness. All tests are passed. Detailed in iteration_math_test.py and recursion_math_test.py

Binary Theory (basic)	In chapter 1 recursive math, the binary theory is used in the binary search function and binary power function(using the property $x^n = x^{n/2} * x^{n/2}$). Detailed in recursion_math.py line 95 ~ 123, 152 ~ 189 and iteration_math.py line 148 ~ 188

2. Chapter 1: Recursive math

Topic: recursion, iteration, efficiency, binary search, unittest, functions

The concept of recursion is the idea that a function calls itself, it is a top-down approach to solving a problem. In this assignment, we will practice recursive functions by incorporating them into some math calculations.

Except for recursion, you will also have a chance to practice its iteration form, analyze time efficiency and use unittest to verify the functions' correctness.

2.1. Recursion

Recursion has two parts, recursive call and base case.

The base case is the simplest version of the problem and is used to control when the recursion stops.

In recursive calls, we make each recursive step smaller than the previous one, until we stop at the base case.

Take the factorial of a number(n!, multiplication of a number and its all previous numbers) for example:

fact(1) = 1

fact(2) = 1 * 2 = fact(1) * 2

fact(3) = (1 * 2) * 3 = fact(2) * 3

fact(4) = ((1 * 2) * 3) * 4 = fact(3) * 4

fact(5) = (((1 * 2) * 3) * 4) * 5 = fact(4) * 5

…

From the above functions, it is easy to find that the function relies on the previous one, get the following pattern:

fact(n) = ((((1 * 2) * 3) * 4) * 5…* (n – 1)) * 5 = fact(n - 1) * n

Base case, n = 1, f(n) = 1

Recursive call: f(n) = f(n - 1) * n

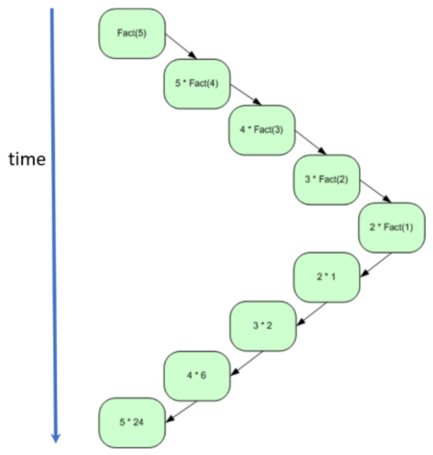
The recursion function is:

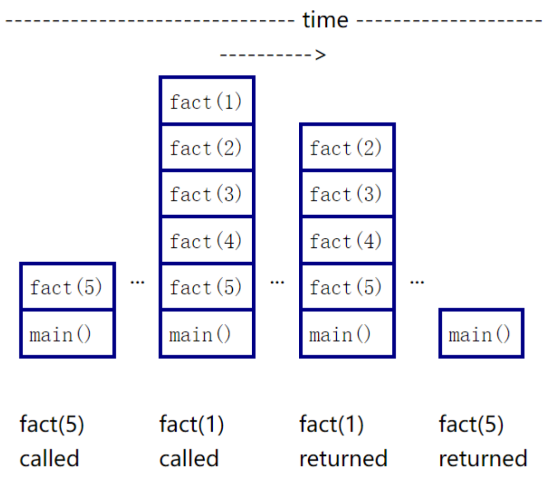
```
def  fact(n):
    #This is the base case
    if n == 1:
            return 1
    #recursive call
    return n * fact(n - 1)
```

2.2. Recursion and the stack

Each recursion call is saved in the stack until the base case is met, since the stack has a last in first out property, the recursion calls are popped out of the stack from smaller case to larger one by one until the stack is empty.

The following picture depicts the process in a factorial recursion process.





2.3. Iteration

Common iteration approaches are for loops and while loops, the recursive algorithm can be rewritten iteratively (with a loop) however it might require more complex solutions than just using a loop.

A while loop contains a loop condition and code block, while the condition is True, the code block will be executed again and again until the condition is False.

while loop syntax:

```
while condition:
        statments
```

A for loop can be used for iterating repeatedly over a range and iterator through a sequence, which allows us to run a series of instructions once for each element of a string, a list, a tuple, a set, or a range.

for loop syntax:

```
for iterator_variable in sequence:
        statments
```

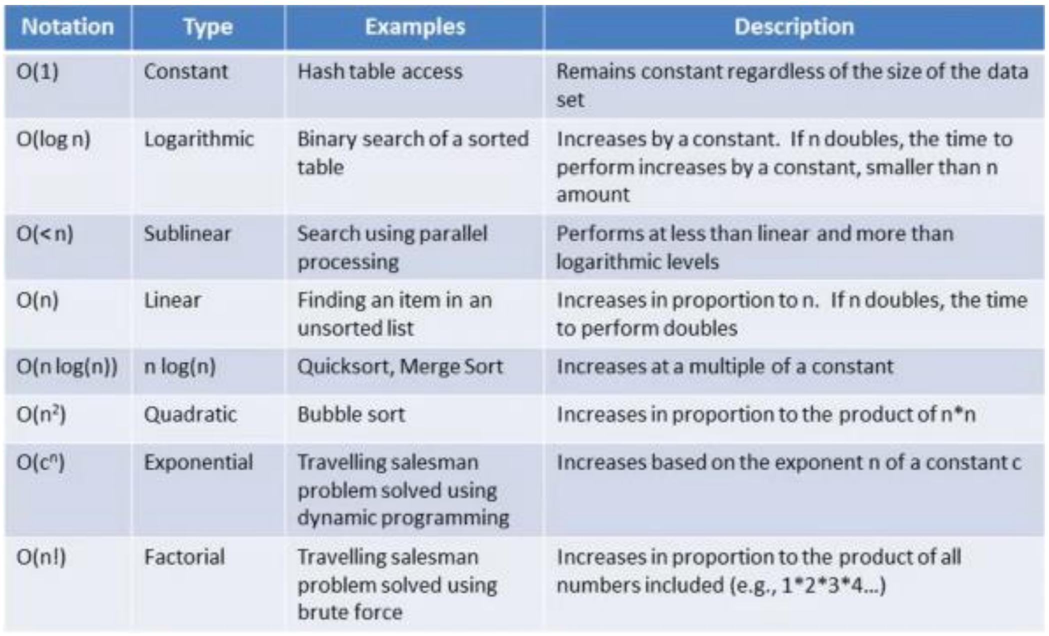
Example of for loop a range:

```
for i in range(0, 5):
        print(i)
```

2.4. Efficiency(Baisc)

Algorithms execute at varying speeds, mostly because certain algorithms simply require more steps to reach the desired result than others.

Typically we use big O notation to describe the efficiency, below is a table describing the efficiency of different data structures.



Notation	Type	Examples	Description
O(1)	Constant	Hash table access	Remains constant regardless of the size of the data set
O(log n)	Logarithmic	Binary search of a sorted table	Increases by a constant. If n doubles, the time to perform increases by a constant, smaller than n amount
O(< n)	Sublinear	Search using parallel processing	Performs at less than linear and more than logarithmic levels
O(n)	Linear	Finding an item in an unsorted list	Increases in proportion to n. If n doubles, the time to perform doubles
O(n log(n))	n log(n)	Quicksort, Merge Sort	Increases at a multiple of a constant
$O(n^2)$	Quadratic	Bubble sort	Increases in proportion to the product of n*n
$O(c^n)$	Exponential	Travelling salesman problem solved using dynamic programming	Increases based on the exponent n of a constant c
O(n!)	Factorial	Travelling salesman problem solved using brute force	Increases in proportion to the product of all numbers included (e.g., 1*2*3*4...)

Example 1: Traverse a list and get the sum

List = [1,2,3,4,….,n]

Sum = 0

```
list = [1,2,3,4,….,n]
sum = 0
for num in list:
    sum += num
return sum
```

The time efficiency is O(N) since the traverse takes N operations, which is the liner. The number of operations are the same as the size of the list.

The memory needed is constant so the memory efficiency is O(1).

Example 2: a nested for loop

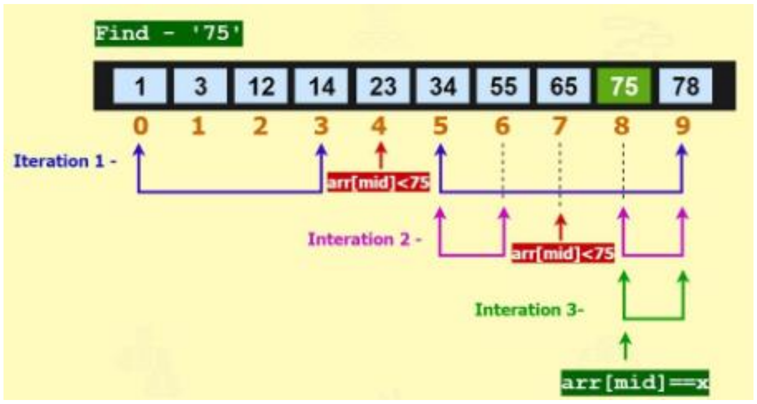
```
list = [1,2,3,4,….,n]
sum = 0
for num1 in list:
    for num2 in list:
        sum += num1 * num2
return sum
```

The time efficiency is O(N^2) since the traverse of the outer for loop takes N operations, and for each outer loop operation, there are N operations for the inner loop, which is quadradic. Times of operations are the same as the squire of the size of the list.

The memory needed is constant so the memory efficiency is O(1).

Example 3: binary search

A sorted array may be searched using the binary search method by continually halving the search interval, shown as the picture as follow.



In binary search, it is quicker than traversing the whole list. Like in the picture, it takes 3 comparations to find 75. It can be deduced that if the list is much larger, it would take about $\log_2 n$ comparations to get the target since each time we cut half of the search range.

In big-O notation, the time efficiency is O(logN), The memory needed is constant so the memory efficiency is O(1).

2.5. Unittest

Instead of using print statements to test each function, unittest is a better way for you to check correctness.

If we need to check a function “iteration_fact(n)” in iteration_math module, here is a simple syntax of creating a test case.

```
import unittest
import iteration_math

class IterationTest(unittest.TestCase):

    def test_iteration_fact(self):
        self.assertEqual(iteration_math.iteration_fact(1), 1)
        self.assertEqual(iteration_math.iteration_fact(3), 6)
…

unittest.main()
```

The assertequal function will compare the value produced by the function and the supposed value, if they are all the same in a test case, the test case will pass.

2.6. Assignment Goals

Understand the concept of recursion and stack

To explore base case confirmation and usage of recursion

To practice recursion and its iteration form

To explore different algorithms, and analyze time efficiency

Practice unittest for your math functions

2.7. Instructions

Objective 1: Create a function to calculate $1^2 + 2^2 + 3\,^2 + \ldots + (n - 1)^2 + n^2$ using recursion.

Hint1: use f(n) to express the sum

$f(1) = 1^2$

$f(2) = 1^2 + 2^2$

$f(3) = 1^2 + 2^2 + 3^2$

$f(n) = 1^2 + 2^2 + 3^2 + 4^2 + \dots + n^2$

Hint2: $f(n) = f(n - 1) + n^2$

Objective 2: Create a function to calculate the Fibonacci number using recursion.

The Fibonacci numbers, expressed as F(n), constitute a series known as the Fibonacci sequence, where each number, starting at 0 and 1, is the sum of the two before it.

F(0) = 0

F(1) = 1

F(2) = F(1) + F (0) = 1

F(3) = F(2) + F (1) = 2

F(4) = F(3) + F (2) = 3

F(5) = F(4) + F (3) = 5

…

F(n) = F(n - 1) + F (n - 2)

Hint1:

look at the function F(n) = F(n - 1) + F (n - 2), if we create a function to calculate F(n), can we call F(n - 1) and F(n - 2) inside function?

Hint2:

F(0) and F(1) is the smallest case and the result of them are certain, what is the base case according to this fact?

Objective 2 - 1: Using recursion to create a "triple" Fibonacci number as follows(maybe there is no such term, I just made it up).

F(0) = 0

F(1) = 1

F(2) = 2

…

F(n) = F(n - 1) + F (n - 2) + F (n - 3)

Objective 3: Given an increasing ordered list with integers, use recursion algorithm to create a function to find a target number's list index, if the target is not found, return -1 instead.

Hint1:

A plain idea is to traverse the list one by one, until the target number is found and return the number.

Hint2:

O(N) is not enough, according to the binary search method, the persudo code is:

```
Find middle number
Compare the middle number to the target
If the middle number = target, target find
If middle number > target, binary search left half of the list
If middle number < target, binary search right half of the list
```

Hint3:

Can you see the recursion in hint2?

Objective 4: Create a function to make power calculations using recursion.

Given an integer x and positive integer n, you should create a function to calculate x^n recursively.

Hint1:

Intuitive thought is that $x^n = x * x * x *…* x$, the plain way is to multiply x in n times.

Hint2:

If n is even, then $x^n = x^{n/2} * x^{n/2}$

If n is odd, then $x^n = x * x^{n/2} * x^{n/2}$

Hint3:

Since the function is to calculate x^n, you can call the function inside to calculate $x^{n/2}$, using a method like binary search!

Objective 5:Use an iteration way to rewrite all the recursion functions.

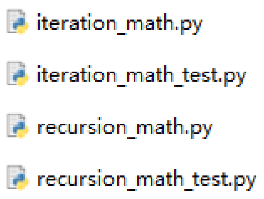
You do not have to write an iteration_quick_pow(x, n) function since it is much more complicated to use iteration than recursion.

Objective 6:Write unit tests to verify the correctness of each function.

Objective 7: Analyse the time efficiency(write in the comment of your function code) of each function and try to achieve better time efficiency.

2.8. Keys

Please refer to the keys in the attached files:



iteration_math.py

iteration_math_test.py

recursion_math.py

recursion_math_test.py

3. Chapter 2: Account management

Topic: class, dictionary, set, error handling, functions

In this assignment, The basic idea is that you are going to create a CLI-based user login menu, which would allow a user to register an account, login, change the password, and delete the account, also allow the admin to get all account information.

Raise an error if something is wrong, and protect your functions with error-handling skills.

You will approach some basic operations in class, such as the class declaration, constructor, attribute settings, create class methods, and instance creation.

You should use the dictionary and set data structures to save email – password mapping and all user emails.

3.1. Class

Class: To arrange code into units that correspond to objects in the actual world, we can use the concept of class.

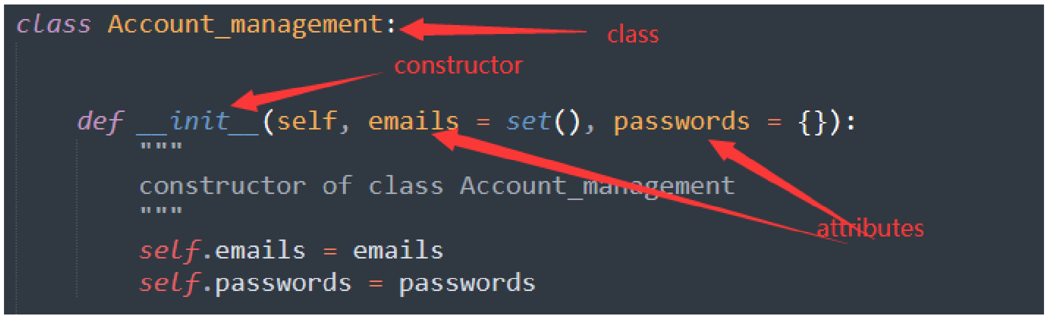
Examples:

Class: animal, Object: elephant

Class: employee, Object: software engineer

Constructor: the purpose of the __init__ function is to create a constructor function. The constructor allows the construction of an instance, to establish what the attributes are and give them initial values.

Attribute: one or more variables consisted in classes.

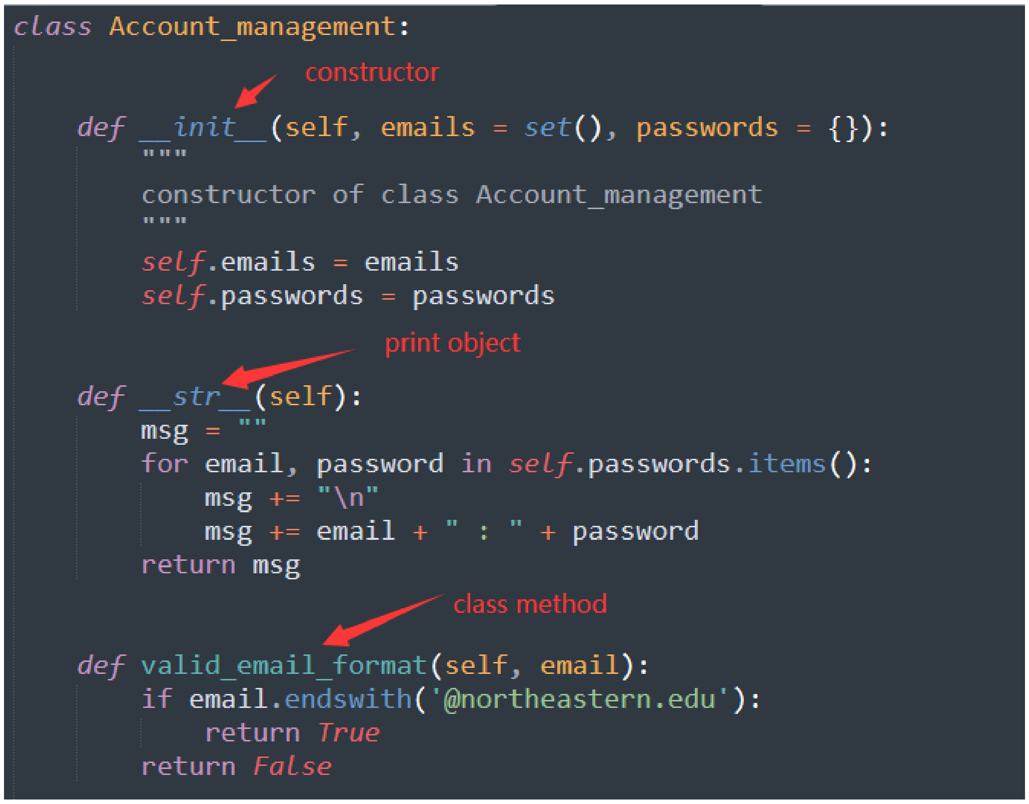


```python
class Account_management:

    def __init__(self, emails = set(), passwords = {}):
        """
        constructor of class Account_management
        """
        self.emails = emails
        self.passwords = passwords
```

Methods: one or more functions called methods consisted of classes. The method is defined inside a class and operates on the attributes of the class.

Use the "self." modifier when you need to make a current object passed to every method of a class.

Use the __str__ notion when you want to change the way of the object printing, a __str__ function will allow you to return the string representation of the object



```
class Account_management:

    def __init__(self, emails = set(), passwords = {}):
        """
        constructor of class Account_management
        """
        self.emails = emails
        self.passwords = passwords

    def __str__(self):
        msg = ""
        for email, password in self.passwords.items():
            msg += "\n"
            msg += email + " : " + password
        return msg

    def valid_email_format(self, email):
        if email.endswith('@northeastern.edu'):
            return True
        return False
```

3.2. Dictionary and Set

Dictionary: An associative array, where arbitrary keys are mapped to values.

In a dictionary, all keys must be unique and immutable but values can repeat and can be anything including lists and other dictionaries.

To create a dictionary:

variable = {} (empty dictionary)

variable = {key1:value1, key2:value2….}

variable = dict() (empty dictionary)

variable = dict(key1 = value1, key2=value2….)

Using a dictionary:

get() : Returns the value of the specified key

add a key, value mapping: dict[key] = value

remove(key) : remove a key, value mapping from dictionary

items(): return all dictionary key-value mapping

keys(): return all the keys in the dictionary

valus(): return all the values in the dictionary

Set: Unordered grouping of **unique** objects.

To create a set:

set1 = set()   (empty set)

set2 = {1, ‘hello’, [1,2,3], set1}

Using a set:

add(element): Adds a given element to a set

discard(element): Removes the element from the set, will not raise an error if the element does not exist in set

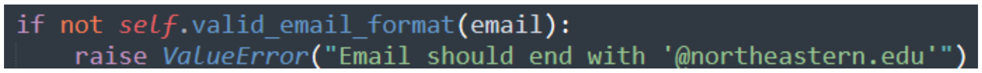
remove(element): Removes the element from the set, will raise an error if the element does not exist in set

clear(): Remove all elements from the set

3.3. Error-handling

The best approach to make sure that the values in your program are what you anticipate is to raise errors. It is advised to raise an error that is unique to the issue and has a helpful error message.

Ask permission by raising errors:

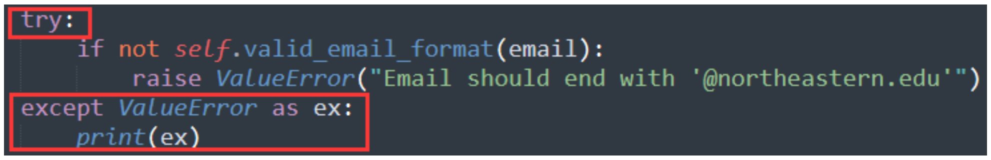


```
if not self.valid_email_format(email):
    raise ValueError("Email should end with '@northeastern.edu'")
```

Give forgiveness:

When a statement is simply assumed to work, an error will be triggered when the statement is attempted to be executed.

Using a try-except construct to catch the error, give an error message and give forgiveness.



```
try:
    if not self.valid_email_format(email):
        raise ValueError("Email should end with '@northeastern.edu'")
except ValueError as ex:
    print(ex)
```

3.4. Assignment Goals

Understand the general concept of class

Explore constructor function, instance creation, attribute, and class methods.

Practice error handling

Get familiar with Dictionary and Set data structures and their commonly used methods.

3.5. Instructions

Objective 1: Create a CLI menu in driver.py, ask a user to register an account, login, change the password, and delete the account; allowing the admin to get all account information, the menu should look like this.

```
Main menu
1. create account
2. login
3. change password
4. delete account
5. get all account information(admin)
6. quit
```

Objective 2: Create Account_management class and Admin_management class

Two classes are in separate files, in each class create a constructor function using the syntax： def __init__(self, attribute_1, …, attribute_k).

Objective 3: Set attributes in the constructors.

Set attributes in the constructor of Account_management class, should be a set saving all the user emails and a dictionary saving user email and password mapping. Set attributes in the constructor of Admin_management class, should be a set saving admin password.

Objective 4: Create functional methods for the two classes.

In Account_management class, you are suggested to have these functions:

__str__(self): print all email and password mapping, need to login as admin

create_account(self, email, password): let the user input email and password to create an account

login(self, email, password): let the user input email and password to login, return True if successfully login.

change_password(self, email, password, new_password): let the user input email, password, and new password to change the password.

delete_account(self, email, password): let the user input email, and password to delete the account

Admin_management class, you are suggested to have the function:

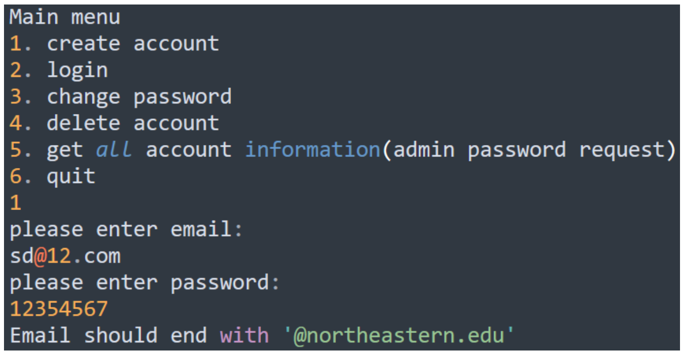
admin_login(self, admin_pass): let the user input the admin password to login as admin

Objective 5: Using the data structure of the dictionary and set, save email – password mapping in a dictionary, and save all emails in a set, allowing the user to delete, add, and change data by the functions.

Objective 6: Raise an error if something is wrong, use error-handling skills to protect your functions, give a nice error message and let the user try again.

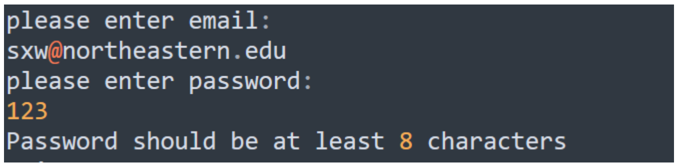
Consider the following conditions to raise an error:

The email is not ended with "@northeastern.edu"

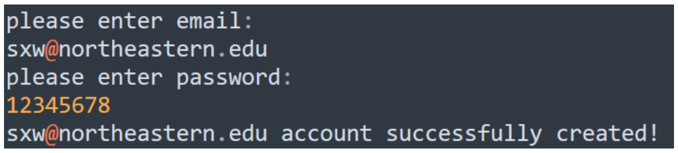


```
Main menu
1. create account
2. login
3. change password
4. delete account
5. get all account information(admin password request)
6. quit
1
please enter email:
sd@12.com
please enter password:
12354567
Email should end with '@northeastern.edu'
```

Password does not have at least 8 characters

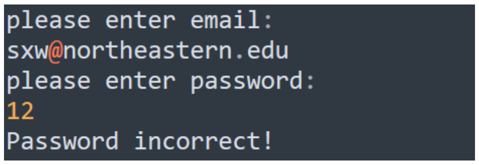


```
please enter email:
sxw@northeastern.edu
please enter password:
123
Password should be at least 8 characters
```

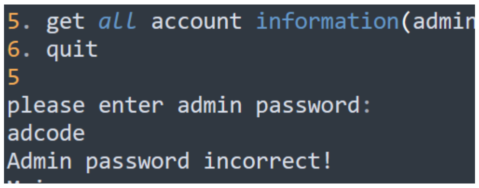


```
please enter email:
sxw@northeastern.edu
please enter password:
12345678
sxw@northeastern.edu account successfully created!
```

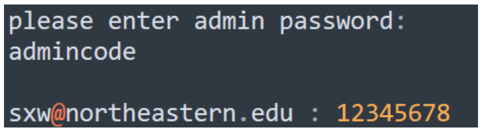
Password is wrong（either user or admin）



```
please enter email:
sxw@northeastern.edu
please enter password:
12
Password incorrect!
```

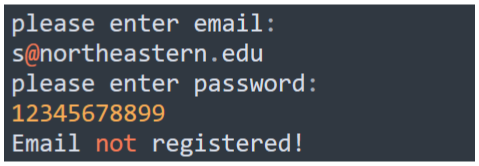


```
5. get all account information(admin
6. quit
5
please enter admin password:
adcode
Admin password incorrect!
```



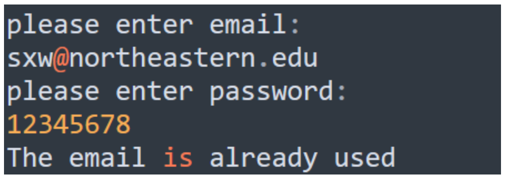
```
please enter admin password:
admincode

sxw@northeastern.edu : 12345678
```

The user does not exist when login



```
please enter email:
s@northeastern.edu
please enter password:
12345678899
Email not registered!
```

The account has been registered when creating a new account



```
please enter email:
sxw@northeastern.edu
please enter password:
12345678
The email is already used
```

Objective 7: Import two classes to utilize the class methods, create instances, and complete the menu-driven login interface in driver.py.

3.6. Keys

Please refer to the keys in the attached files:

account_management.py

admin_management.py

driver.py

4. Chapter 3: Course waitlist & Cooking pancakes

Topic: queue, stack, functions

In this assignment, we will utilize the data structure of queues and stacks to implement the simulation of the course register waitlist and pancake cooking.

Course register waitlist is a typical queue since the person who is first enqueued will dequeue first as well. This is the same as the FIFO(first in first out) property of the queue.

As for pancake cooking, we have a plate to put cooked pancakes, then cooked pancakes will be put on the top of the plate one by one, when we need to serve the pancakes, just take the pancakes from the top. The newly cooked pancakes will serve first, which is the same as the LIFO(last in first out) property of the stack.

You will write different functions for operations course register, drop, make pancakes, serve…and use a combination of functions to test the outcome and supposed result.

4.1. Function

A function is a code block that only executes when called, the code is related to achieving a specific task.

Function definition syntax

```
def function_name(parameter1…):
        statment1
        statment2
        …
        return …
```

Call a function syntax

```
function_name(parameter1…)
```

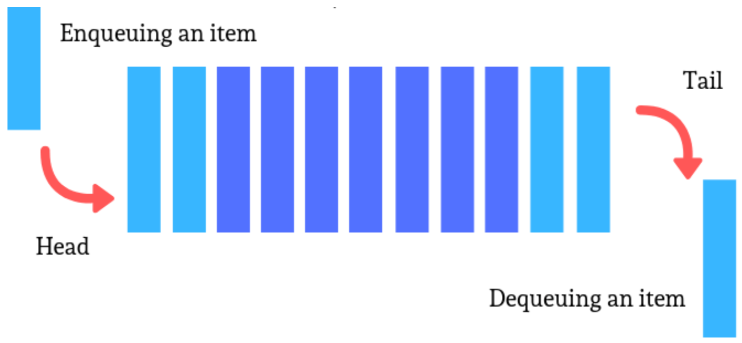
In python, the function should be called only after it is defined.

Functions can separate complex codes into small parts, and each function executes a specific task, which could make code clear, modular, and manageable.

Functions can be reused to reduce repetitive codes and increase teamwork efficiency.

4.2. Queue

Queues have the FIFO pattern, once you enqueue an item it is added at the head, and once you dequeue an item it is dequeued from the tail, as shown in the following picture.



Standard functions:

enqueue() - enqueue an item to the head

dequeue() - remove an item from the tail and return the item

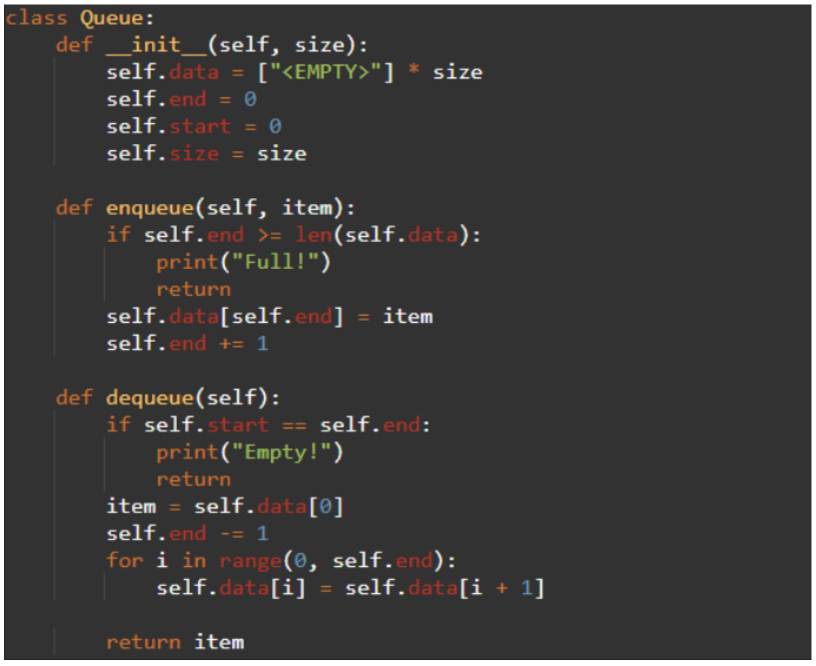
Extra functions:

is_empty() - return true when the stack is empty

is_full() - return true when the stack is full

peek() - return the top item(not removing)

Here is a simple implementation of a queue using a list in python.



```
class Queue:
    def __init__(self, size):
        self.data = ["<EMPTY>"] * size
        self.end = 0
        self.start = 0
        self.size = size

    def enqueue(self, item):
        if self.end >= len(self.data):
            print("Full!")
            return
        self.data[self.end] = item
        self.end += 1

    def dequeue(self):
        if self.start == self.end:
            print("Empty!")
            return
        item = self.data[0]
        self.end -= 1
        for i in range(0, self.end):
            self.data[i] = self.data[i + 1]

        return item
```

You can also use the built-in Queue module in python, here are some functions for the module:

Queue.put(): basically same as enqueue()

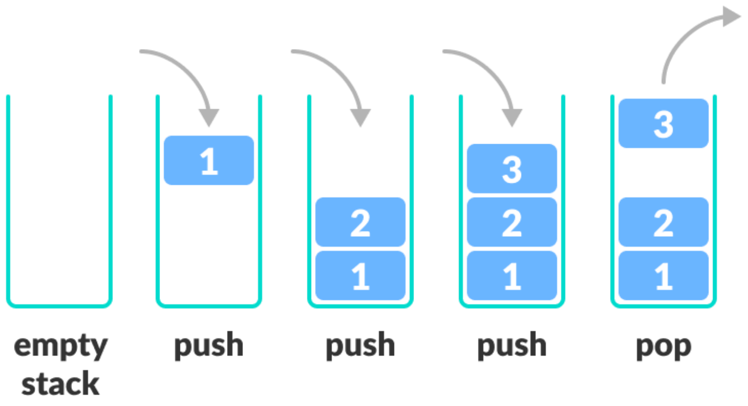
Queue.get(): basically same as dequeue()

Queue.empty(): basically same as is_empty()

Queue.full(): basically same as is_full()

4.3. Stack

Stacks have the LIFO pattern, just like a top-opened container, you can only access the top item. Once you push an item it is added at the top of the stack, and once you pop an item it is popped from the top, as shown in the following picture.



Standard functions:

pop() - remove an item from the top of the stack and return the item

push() - add an item to the top of the stack

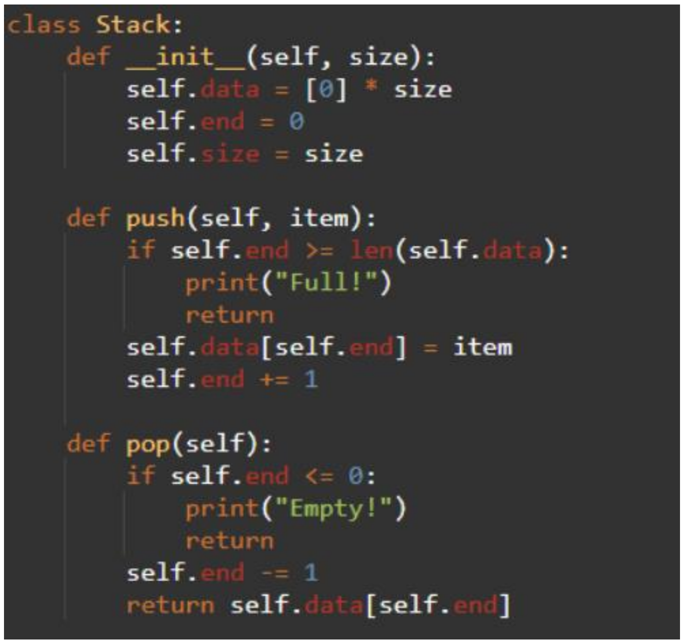
Extra functions:

is_empty() - return true when the stack is empty

is_full() - return true when the stack is full

peek() - return the top item(not removing)

Here is a simple implementation of a stack using a list in python.



```
class Stack:
    def __init__(self, size):
        self.data = [0] * size
        self.end = 0
        self.size = size

    def push(self, item):
        if self.end >= len(self.data):
            print("Full!")
            return
        self.data[self.end] = item
        self.end += 1

    def pop(self):
        if self.end <= 0:
            print("Empty!")
            return
        self.end -= 1
        return self.data[self.end]
```

You can also use the built-in Collections.deque module in python, here are some functions or syntax:

deque.append(): basically same as push()

deque.pop(): basically same as pop()

deque[-1]: basically same as peek()

deque: basically similar to not is_empty()

not deque: basically similar to is_empty()

4.4. Assignment Goals

Understand the principal of queue and stack

Explore the usage of stack and queue.

Practice using functions to achieve certain features and reduce code

4.5. Instructions

Objective 1: For the course waitlist program, import the python built-in module 'queue', create an integer variable additional_space initialized as 0(no available space), and an empty queue waitlist, with max size 5.

Import the built-in module:

```
from queue import *
```

Create a queue with maxsize as the waitlist:

```
waitlist = Queue(maxsize = 5)
```

Objective 2: For the course waitlist program, Create the following suggested functions:

print_waitlist():

print all the students on the waitlist

add_to_waitlist(student):

add a student to the waitlist.

drop_class(n):

n students drop the class, and additional space will be plus n, letting students on the waitlist register for the course automatically.

Call print_waitlist() at the end of the function.

request_register(student):

Send a request for course registration, in this function:

If there is available space for the course, available_space minus 1.

Else if there is no available space of course and the waitlist is not full, call the add_to_waitlist(student) function

Else if the waitlist is full, give a message that the waitlist is full.

Call print_waitlist() at the end of the function.

Objective 3: Test your functions for the course waitlist program

You can use a combination of functions to test the outcome and the supposed result.

For example:

```
drop_class(1)
request_register('student1')
request_register('student2')
request_register('student3')
request_register('student4')
request_register('student5')
drop_class(2)
```

And here is the supposed output for the test:

```
1 students has droped the class.            // drop_class(1)
------------------------------------------
student in waitlist:
no student in waitlist
------------------------------------------
student1 send a request for register.       //request_register('student1')
student1 is registered.
------------------------------------------
student in waitlist:
no student in waitlist
------------------------------------------
student2 send a request for register.       //request_register('student2')
There is no space for the course
student2 have been in waitlist
------------------------------------------
student in waitlist:
student2
------------------------------------------
student3 send a request for register.       //request_register('student3')
There is no space for the course
student3 have been in waitlist
------------------------------------------
student in waitlist:
student2
student3
------------------------------------------
student4 send a request for register.       //request_register('student4')
There is no space for the course
student4 have been in waitlist
------------------------------------------
student in waitlist:
student2
student3
student4
------------------------------------------
student5 send a request for register.       //request_register('student5')
There is no space for the course
student5 have been in waitlist
------------------------------------------
student in waitlist:
student2
student3
student4
student5
------------------------------------------
2 students has droped the class.            // drop_class(2)
student2 from waitlist is registered, available space for register: 1.
student3 from waitlist is registered, available space for register: 0.
------------------------------------------
student in waitlist:
student4
student5
```

Objective 4: For the pancake cooking program, import the python built-in module 'deque', and create a deque as the pancake plate.

Import the built-in module:

```
from collections import deque
```

Create a stack in the form of deque:

```
plate = deque()
```

Objective 5: For the pancake cooking program, Create the following suggested functions:

make_pancake(pancake):

make a pancake and put it on the top of the plate(or top pancake).

top_pancake():

see the pancake on the top.

serve_pancakes(n):

pick top n pancakes to serve.

check_plate():

print all the pancakes on the plate from bottom to top

Objective 6: Test your functions for the pancake cooking program.

You can use a combination of functions to test the outcome and the supposed result.

For example:

```
make_pancake('strawberry')
make_pancake('plain')
make_pancake('apple')
top_pancake()
check_plate()
serve_pancakes(2)
```

And here is the supposed output for the test:

```
strawberry pancake is made      // make_pancake('strawberry')
-------------------------------------------
plain pancake is made           //make_pancake('strawberry')
-------------------------------------------
apple pancake is made           //make_pancake('strawberry')
-------------------------------------------
top pancake is apple            //top_pancake()
-------------------------------------------
here are pancakes from bottom to top:              // check_plate()
strawberry pancake
plain pancake
apple pancake
-------------------------------------------
apple pancake is served         // serve_pancakes(2)
plain pancake is served
```

4.6. Keys

Please refer to the keys in the attached files:

cooking_pancakes.py

course_waitlist.py