CSC148: Week 4
http://www.cdf.utoronto.ca/~csc148h/summer/

Sophia Huynh
Summer 2018
Announcements

- Exercise 3 is due tomorrow @ 11PM
- Lab 4 + Exercise 4 are out
- Midterm 1 next week
  - 6:10 - 7PM
  - A - Mao (inclusive): EX300
  - Everyone else: EX310
- Assignment 1 due June 17th @ 11PM
Outline

● Linked Lists
● Assignment 1 overview
● Midterm (+ Review)
Linked Lists
"Linked" Concept

If we have the letter 'A' and we know that 'B' comes after 'A' we could draw it like:

A → B
Linked Lists

Linked Lists are similar in concept, but the "arrow" and letter are in 1 object.
Linked Lists

Linked Lists are similar in concept, but the "arrow" and letter are in 1 object.
Linked Lists

Linked List Node

Value

A

Next

B
class LinkedListNode:
    def __init__(self, value, next):
        self.value = value
        self.next_ = next
Linked List Node Class

class LinkedListNode:
    def __init__(self, value, next = None):
        self.value = value
        self.next_ = next

Default parameter: if next isn't provided, next = None
Linked List Node Class

class LinkedListNode:
    def __init__(self, value, next = None):
        self.value = value
        self.next_ = next

next is a reserved attribute name, so we name our attribute next_
Type of next? 

Linked List Node

Value

Next

An arrow pointing to another Linked List Node...
Type of next? A variable that references another Linked List Node!
Type of next_?

Could also be None if there is no 'next' node.
Creating Linked List Nodes

\[ a = \text{LinkedListNode}("A", \text{LinkedListNode}("B")) \]
Creating Linked List Nodes

\[ a = \text{LinkedListNode}("A", \text{LinkedListNode}("B")) \]

How do we get to B?
Problem: Add a new node

Suppose we have a new LinkedListNode that we want to add to the end.

\[
c = \text{LinkedListNode}("C")
\]
Adding a new node

What we have:

What we want:
Adding a new node

What we have:

What we want:
Adding a new node

How do we get to B?
B is a.next!
Adding a new node

\[ b = a.\text{next}_\_ \]
\[ b.\text{next}_\_ = c \]

Same as
\[ a.\text{next}_\_.\text{next}_\_ = c \]
Terminology

Front

A → B → C
Counting the size

How do we count how many nodes there are?
Counting the size

Front

Last Node points to None
Counting the size

● Loop while the node we're looking at is not None
● "Go through all of the nodes, adding 1 each time, until we get to None"
● Count is 0 if the front is None
Counting the size

\[
\text{count} = 0
\]
\[
\text{current\_node} = \text{front}
\]
Counting the size

```python
count = 0
current_node = front
while current_node is not None:
    count += 1
```
Counting the size

count = 0
current_node = front
while current_node is not None:
    count += 1
    current_node = current_node.next_
return count
Counting the size (diagrams)

Front

Count: 0

current_node
Counting the size (diagrams)

Front

Count: 0

current_node

Is this None?
Counting the size (diagrams)

Front

Count: 1

current_node

Is this None?
No.
Counting the size (diagrams)

Is this None?
current_node

Front

Count: 1
Counting the size (diagrams)

Front

Count: 2

current_node
Counting the size (diagrams)

Front

Count: 2

current_node
Counting the size (diagrams)

Front

Count: 3

current_node
Counting the size (diagrams)

Front

Count: 3

current_node = None
Counting the size (diagrams)

Front

Count: 3

Done! We've hit the end of the LinkedList! Count is 3!
Example: Add to the end

Want to add a value to the end of our Linked List.

Front

Want to add "Z" to the end of this
Example: Add to the end

1. Make new node
2. Add to the end
Example: Add to the end

1. new_node = LinkedListNode(value)
2. Add to the end (How?)
Traverse to the end

- Same steps as when counting
- Keep track of the **previous node** instead of keeping a counter
  - At the end, our current node is None after all!
  - We want to know what **points** to None!
Add to the end: Code

```
prev_node = None
current_node = front
```
Add to the end: Code

```python
prev_node = None
current_node = front
while current_node is not None:
    prev_node = current_node
    current_node = current_node.next_
```

prev_node always 1 behind cur_node
Add to the end: Code

prev_node = None
current_node = front
while current_node is not None:
    prev_node = current_node
    current_node = current_node.next_

At the end:
cur_node = None
prev_node = the last node
Add to the end

Front

prev_node

current_node
Add to the end

Front

prev_node
current_node
Add to the end

Front

prev_node

current_node
Add to the end

Front

prev_node
current_node
Add to the end

Front

prev_node

current_node

49
Add to the end

Front

prev_node
current_node
Add to the end

Front

prev_node

current_node
Add to the end

Set prev_node.next_ to our new node!

prev_node
Linked Lists

- Looping through the list to count is a hassle
- Adding the to end also needs us to go through everything
- Keeping track of the front of our linked list is messy
Linked List Class

- Make a Linked List Class
  - Has the **front** of the linked list
  - Also keep the **back** of the linked list
  - Keeps track of the **size**
    - When adding, just increase by 1
    - When removing, decrease by 1
    - Don't have to count every time!
Linked List Class

class LinkedList:
    def __init__(self):
        self.front = None
        self.back = None
        self.size = 0
LinkedList .append

- Have self.back now: Don't need to search for it!
LinkedList.append

def append(self, value):
    new_node = LinkedListNode(value)
    self.back.next_ = new_node
def append(self, value):
    new_node = LinkedListNode(value)
    self.back.next_ = new_node

What if self.back == None?
Empty Linked List

Front

Size: 0

Back

Front and Back both point to None.
Append to empty (expected result)

Front

Size: 1

Back

?
LinkedList.append

def append(self, value):
    new_node = LinkedListNode(value)
    if self.size == 0:
        self.front = new_node
        self.back = new_node
Add to the end

Front

Size: 3

Back
Add to the end

Front

Size: 3

Back
Add to the end

Front

Size: 3

Back
def append(self, value):
    new_node = LinkedListNode(value)
    if self.size == 0:
        self.front = new_node
    else:
        self.back.next_ = new_node
    self.back = new_node
def append(self, value):
    new_node = LinkedListNode(value)
    if self.size == 0:
        self.front = new_node
    else:
        self.back.next_ = new_node
    self.back = new_node
    self.size += 1
Add to the end

Front

Size: 3

Back
Add to the end

Front

Size: 3

Back
Add to the end

Front

Size: 3

Back
Add to the end

Front

Size: 4

Back
Add to the end (empty)
Add to the end (empty)
Tackling Linked List Problems

- **Draw a "before" and "after"!**
  - Lets you know which 'next' pointers you need to update
  - When you need to update front/back
  - Gives you a clear idea of what you need to do!
Add to the start

- "Prepend" method
- Add to the start instead of the end
We want to add "Z" to the start of this Linked List. What should it look like after?
a) Z → A → B → C
b) Z → A → B → C
c) Z → A → B → C
Left-side

Answer:

b)
What changes do we have to make?
a) Set front to the node with "Z"
b) Add 1 to the size
c) Set the next_of the node with "Z" to the current front
d) All of the above
Answer:

d) All of the above
Prepend code

```python
new_node = LinkedListNode(value)
a) Set front to the node with "Z"
   ```
   self.front = new_node
   ```
b) Add 1 to the size
   ```
   self.size += 1
   ```
c) Set the next_ of the node with "Z" to the current front
   ```
   new_node.next_ = self.front
   ```
```
Prepend code:

```python
new_node = LinkedListNode(value)
(a) Set front to the node with "Z"
    `self.front = new_node`
b) Add 1 to the size
    `self.size += 1`
c) Set the next_ of the node with "Z" to the current front
    `new_node.next_ = self.front`
```

(c) has to happen before (a)
Prepend code

```python
new_node = LinkedListNode(value)
c) Set the next_ of the node with "Z" to the current front
   ```
   new_node.next_ = self.front
```
a) Set front to the node with "Z"
   ```
   self.front = new_node
```
b) Add 1 to the size
   ```
   self.size += 1
```
def prepend(self, value):
    new_node = LinkedListNode(value)
    new_node.next_ = self.front
    self.front = new_node
    if self.size == 0:
        self.back = new_node
    self.size += 1
Removing a value

- Look for a value in the LinkedList
- Remove it if it's there
Removing a value

Remove "C".

Front

Size: 4

Back
Removing a value

Remove "C".

Front

Size: 3

Back
Removing a value

Remove "C".

C can still point to D. We'll never find C in our LinkedList though.
Removing a value

Need to find:

1. The node that points to the value we want to remove.
2. The node that comes after the value we want to remove.
Removing a value

1. The node that points to the value we want to remove.

We've done something similar earlier:

- Traverse through the linked list until we find our value or None (if value's not in the list)
- Keep track of prev_node
Old append loop

```python
while current_node is not None:
    prev_node = current_node
    current_node = current_node.next_
```
Our remove loop

while current_node is not None and current_node.value != value:
    prev_node = current_node
    current_node = current_node.next_
Our remove loop

while current_node is not None and current_node.value != value:
    prev_node = current_node
    current_node = current_node.next_

**We must use**

cur_node.value!

cur_node itself is an object, not its value!
remove("C")

Front

A

B

C

D

Back

prev_node

current_node

Size: 4
remove("C")

prev_node

current_node
remove("C")

Front

A

B

C

D

Back

Size: 4

prev_node

current_node
remove("C")

Front
A

Size: 4
B

Back
C

D

prev_node

current_node
remove("C")
remove("C")

Exit loop since current_node.value == "C"

prev_node

current_node
remove("C")

Front -> B -> C -> Back

Size: 4

prev_node

current_node

current_node.next_
remove("C")

Front

A

B

prev_node

current_node

current_node.next_

Size: 4

Back

C

D
remove("C")
Removing a value

current_node = self.front
while current_node is not None
    and current_node.value != value:
        prev_node = current_node
        current_node = current_node.next_
        prev_node.next_ = current_node.next_
Special cases

1. What if prev_node is None (and current_node is not None)?
2. What if current_node is None?
3. What if current_node.next_ is None?
Case 1. No prev_node

What if prev_node is None (and current_node is not None)?

- No node comes before the value we were looking for
Case 1. No prev_node

Front

Size: 3

Back

C → D → E
Case 1. No prev_node

Front

Size: 2

Back

C → D → E
Case 1. No prev_node

if prev_node == None and current_node is not None:
    self.front = current_node.next_
Case 2. current_node is None

What if current_node is None?

- We did not find the value in our LinkedList.
  - There is nothing to remove!
  - Don't do anything.
Case 3. Current_node.next_ is None

What if current_node.next_ is None?

- The value we're removing is at the back.
  - Update the back of our linked list.
  - Previous node is new back; points to None.
Case 3. `Current_node.next_ is None`

Front

Size: 3

Back
Case 3. Current_node.next_is_None

Front

Size: 2

Back

A  B  C

Size: 2
Case 3. Current_node.next_ is None

if current_node.next_ == None:
    self.back = prev_node

prev_node.next_ = None
    and
prev_node.next_ = current_node.next_
    are both the same!
Other: prev and next are None

Case is already covered in the other cases.
Linked Lists

- LinkedListNodes have a value and next_ pointer
- LinkedList keeps track of front, back, and size
Use of Linked Lists

Consider Queues from last week, implemented with lists:

- One of add() or remove() needed to shift over every element in the list
  - Shifting around is inefficient!
- Linked Lists don't need to shift!
Linked List Queues

- Queue's `.content` = a LinkedList
- `add()`: Adds to the end
- `remove()`: Removes from the front
**Linked List Queue**

Within the Queue with `.content` being a linked list:

- Add "A", "B", and then "C"

What does the LinkedList look like?

What if we called Queue's remove()?
Linked List Queue

Write the code for the Queue:

- `__init__()`: Make `self._content` a Linked List
- `add()`
- `remove()`
- `is_empty()`

Don't call LinkedList's append, prepend, or remove.
Linked List Queue

Front

Size: 3

Back

A -> B -> C
Linked List Queue

Front

Size: 2

Back

A → B → C

Return "A" at the end of remove()
remove()

1. Store the value of our current front
   
   self._content.front.value

2. Make the next node the new front
   The next node is simply
   
   self._content.front.next_

3. Decrease size by 1
   
   self._content.size -= 1
remove()

What if the item is the last one in the Queue?

- Both front and back should be None!
def remove(self):  # Assume it's not empty
    removed_value = self._content.front.value
    self._content.front =
        self.content.front.next_
    self._content.size -= 1
    if self._content.size == 0:
        self._content.back = None
    return removed_value
add()

- Exactly the same as our append!
  - Won't repeat the code, for time's sake.
- If it's an empty linked list: Set self._content.front and self._content.back to the new node.
- If not, set self._content.back.next_ to our new node.
  - Update self._content.back
is_empty()

- An Empty Linked List looks like this:

  Front  | Size: 0  | Back

So...

```python
return self._content.size == 0
```
Linked List Summary

Understand how to:
● Traverse a linked list
● Update a linked list
  ○ Identify which next_ pointers need to change
  ○ When back and front need to change

Drawing pictures helps!
Break for 10 minutes!
And then I'll talk about the assignment a little bit.
Assignment 1: Battle Game

- Read the handout
- Lots of starter and client code
  - a1_game.py is the main one for you to look at
[ Going through the starter code ]
BattleQueue Example

Front

Mage

Rogue
BattleQueue Example

Mage attacks.
Client code removes them from the Battle Queue.
BattleQueue Example

Mage attacks.
Your code adds them in to the end of the queue.
Rogue uses 'special attack'
Client code removes them from the front.
BattleQueue Example

Rogue uses 'special attack'
Your code adds them in to the end of the queue 2x
BattleQueue Example

Mage special attack behaves differently. Read the handout for details.
[ Demoing a finished assignment ]
When `special_attack()` is called, the next sprite is `mage_special_0`.

When `attack()` is called, the next sprite is `mage_attack_0`.

Arrows: The sprite name returned after the next call to `get_next_sprite()`.
Grading

● Provided unittests (40%)
  ○ I can add more on request if there's any cases that need clarification.
  ○ Make sure you downloaded and ran the newest version!
  ○ Tests Mage and Rogue

● No tests for BattleQueue provided.
Grading

● PythonTA (10%)
  ○ Excludes a1_game.py

● Documentation (10%)
  ○ Class docstrings, type annotations, method docstrings, docstring examples, etc.
  ○ No docstring examples needed for Playstyle.
  ○ Most of the documentation in a1_battle_queue.py is given.
Grading

- Hidden tests (40%)
  - Will test your code very, very thoroughly.
  - Descriptions of possible tests on the handout.
Midterm

- Covers everything up to and including Linked Lists except for Exceptions.
- 3 questions in total.
  - Class Design
  - Stacks/Queues
  - Linked Lists
Midterm

- The code for LinkedList and LinkedListNode are provided!

Below are the definitions from lecture for LinkedList and LinkedListNode for reference.

```python
class LinkedListNode:
    """
    A Node to be used in a LinkedList.
    """
    next_: Union["LinkedListNode", None]
    value: object

    def __init__(self, value: object,
                 next: Union["LinkedListNode", None] = None) -> None:
```
Midterm

- 3 questions
- 25 marks in total
  - 1st question = 10 marks
  - 2nd question = 8 marks
  - 3rd question = 7 marks
Preparing for the Midterm

● Do the labs!
  ○ Especially for Stacks/Queue and Linked Lists!

● midterm_practice.pdf
  ○ Should (ideally) over-prepare you for the midterm, and help for the final too.
  ○ Will be uploaded tomorrow morning

● Do past midterms (see test page)
Comparison: Past Midterms

- All questions will have a programming component
- Stack/queue question in recent midterms were "pick the order"-styled
  - Yours will be a programming one, so know how to use add(), remove() and is_empty()!
Stack/Queue Question

- You will **not** know the implementation of the Stack/Queue provided!
  - Do not try to use or assume anything about the _content attribute!
Class Design Question

● You will have to write docstrings and type annotations
  ○ Shortcut. Write "SEE SUPER" (and any small changes) -- I'll accept this in lieu of copying docstrings.
  ○ Write "EXTEND" or "OVERRIDE" in the docstring of any method you extend/override.
Logistics

- Rooms split by last names:
  - A - Mao: EX300
  - Everyone else: EX310
- 6:10PM - 7PM
- No aids allowed
- Bring your T-Card or some ID with you
- Lecture at 7:30PM (after midterm)
"Is the midterm hard?"

- I think it's "fair"/"easy" but I'm the one who wrote it!
- None of the questions should catch you off guard
  - You shouldn't need an epiphany to solve any of the questions; just know the material!
Review

- Super fast review of everything we've learned so far
  - Generally: my expectations of what you should know/should be able to do.
Class Design

- Given a problem/context/client code:
  - Identify which attributes and methods are needed
  - Design classes to work within the context/client code
Class Design

- Private attributes and encapsulation
- **Encapsulation**: How we hide attributes by making them private, accessing them via getters/setters
Encapsulation

- Consider our Queue:
  - _content was first introduced as a list
  - Client code only ever used add() or remove(), never accessing _content directly
- Can change the implementation of _content without breaking client code!
Encapsulatiuon

- If _content was public ("self.content"):  
  - People would be right in assuming the could access it directly  
  - Their code might assume it'll always be a list  
  - Could add to it in other ways, adjust things randomly, etc.
Encapsulation

- getters and setters let them access/modify attributes in a way that they can understand
  - Lets us adjust whatever they pass in/how we store things in order to give them what they want
Encapsulation

- **Queue.add()** is like a setter.
  - Client only ever needs to provide a value
  - If we stored it in a LinkedList, we create the LinkedListNode, etc.

- **Queue.remove()** is like a getter.
  - If we stored it in a LinkedList, we unpack the value from the LinkedListNode.
Getters/Setters and Exams

- If you're required to make getter/setters I'll say it explicitly.
  - Save yourself the time. If you can make a public variable, then do so.
Implementing Classes

class ClassName(SuperClass):
Inheritance

● When we **inherit** from a class, we get a copy of all of their methods
  ○ Don't have to rewrite it all

● Animals can Move
  ○ Cats are Animals: Inherits the Move behaviour; don't need to redefine it
Inheritance

- "Is A" relationship
  - Student is a Person
  - Cat is an Animal
  - Bird is not a Cat -- Bird should not inherit from Cat
  - Course is not a Person -- Course should not inherit from Person
Extending Methods

● Add on to what the super method did.
  ○ Calls the superclass' implementation within the subclass'
class Person:
    def __init__(self, name):
        self.name = name
Extending Methods

class Student(Person):
    def __init__(self, name, sn):
        super().__init__(name)
        self.student_number = sn
class Student(Person):
    def __init__(self, name, sn):
        super().__init__(name)
        self.student_number = sn

Person.__init__(self, name) is okay too!

self is needed in that call, but not in super()!

super().__init__(name)
self.student_number = sn
Extending Methods

class Person:
    def get_info(self):
        return self.name
Extending Methods

class Student(Person):
    def get_info(self):
        info = super().get_info()
        return info + " " + \
        str(self.student_number)
Overriding Methods

● Replace the super class' behaviour
  ○ Does not use it at all
Overriding Methods

class Student(Person):
    def get_info(self):
        return "\nstr(self.student_number)"
Extending vs Overriding

- **Extending**: Makes a call to
  
  `super().whatever(...)`
  
  ○ or `SuperClass.whatever(self, ...)`

- **Overriding**: No call to
  
  `super().whatever(...)`
  
  ○ or `SuperClass.whatever(self, ...)`
Composition

- Inheritance was "taking the behaviour of another class as your own"
- Composition is "using another class within your own"
- "Has A" relationship
  - Cat has a Home
  - Person has a Friend
Composition

- LinkedLists are composed of LinkedListNodes (as its front and back)
  - LinkedLists are very heavily based on composition/using other object as attributes!
Class Design

● Expect questions like the past (2018/2017) midterms
● Know when you should inherit, extend, and override
● Know how to use attributes within a class
Stacks and Queues

- Both types of Containers
- Can add() and remove() from them
- Can check is_empty()
- Multiple implementations for Stacks and Queues
Stacks

- First in, last out.
- Newest thing we add to the stack is the first thing we get when we call remove()
Queues

- First in first out
- The first thing we add to the Queue is the first thing we remove from it
Things to know

- How to add to a Stack or Queue
- How to remove from a Stack or Queue
- How to call is_empty()
- The concept is more important than the implementation
Stacks and Queues

- Lab was generally harder than midterm questions
- Example of something that would be fair game: "Implement an __eq__ method for Stacks"
  - Use only add(), remove(), is_empty().
  - Do not access ._content.
def __eq__(self, other):
    if type(other) != type(self):
        return False
    return True
Stack `__eq__`

- Compare the items in a Stack
- Have to empty the stacks out using `remove()`
- Same items removed in the same order = same content = equal Stacks
def __eq__(self, other):
    if type(other) != type(self):
        return False
    list_self = []
    while not self.is_empty():
        list_self.append(self.remove())

    list_other = []
    while not other.is_empty():
        list_other.append(other.remove())

    if len(list_self) != len(list_other):
        return False
    for i in range(len(list_self)):
        if list_self[i] != list_other[i]:
            return False
    return True
def __eq__(self, other):
    if type(other) != type(self):
        return False
    list_self = []
    list_other = []
    while not self.is_empty():
        list_self.append(self.remove())
    while not other.is_empty():
        list_other.append(other.remove())
Stack __eq__

def __eq__(self, other):
    if type(other) != type(self):
        return False
    list_self = []
    list_other = []
    while not self.is_empty():
        list_self.append(self.remove())
    while not other.is_empty():
        list_other.append(other.remove())
    is_equal = list_self == list_other
def __eq__(self, other):
    if type(other) != type(self):
        return False
    list_self = []
    list_other = []
    while not self.is_empty():
        list_self.append(self.remove())
    while not other.is_empty():
        list_other.append(other.remove())
    is_equal = list_self == list_other
    Can't return yet!  
    Both stacks are **empty** now!
Stack `__eq__`

- Have to re-add everything into the stacks before returning!
- Stacks are last in first out: if we want the first item we removed to be the first removed later, have to add it in last.
  - Add everything in reverse order.
def __eq__(self, other):
    ...
    while not other.is_empty():
        list_other.append(other.remove())
    is_equal = list_self == list_other
    list_self.reverse()
    # list_self = list_self[::-1]
    # also works fine.
    for item in list_self:
        self.add(item)
def __eq__(self, other):
    ...
    while not other.is_empty():
        list_other.append(other.remove())
    is_equal = list_self == list_other
    list_self.reverse()
    list_other.reverse()
    list_other.reverse()
    for item in list_self:
        self.add(item)
    for item in list_other:
        other.add(item)
Stack __eq__

def __eq__(self, other):
    ...
    for item in list_self:
        self.add(item)
    for item in list_other:
        other.add(item)
    return is_equal
Grading `__eq__`

- Out of 4 marks maybe:
  - 1 mark for emptying the stacks properly
  - 1 mark for finding out whether they're equal
  - 2 marks for correctly adding the items back to each stack
- -0.5 marks for each error
  - E.g. doing `list_self = list_self.reverse()`
Grading `__eq__`

- Out of 4 marks maybe:
  - 1 mark for emptying the stacks properly
  - 1 mark for finding out whether they're equal
  - 2 marks for correctly adding the items back to each stack
- -0.5 marks for each error
  - E.g. doing `list_self = list_self.reverse()`

**list.reverse() and list.sort() both return None!**
Be careful about using their return values!
Linked Lists

Know how to:

○ Assign to front/back/a node's next
○ Traverse the linked list
○ Add to it (in the front, back, between nodes, etc.)
○ Remove from it (front, back, between nodes)
Common Mistakes: `.value`

- Know the difference between a LinkedListNode and its value

Common mistake:

```python
node_a.next_ = node_b.value
```

Don't do this! It should just be `node_b`!
Linked Lists

- Draw out examples of before/after to help you identify what needs to be changed!
- All methods from today's lecture would be fair for a test (append, prepend, remove)
- Go through the lab and exercise!
Past Midterm

tinyurl.com/winter148midterm

- I'll go through this midterm, how I would've graded, etc.
- I'll put up the question
- Give you 5 ~ 10 minutes to try to find a solution
1. **[10 marks]** (~ 25 minutes) Below we have an implementation of class Employee. On the following pages, implement two subclasses:

   **SalariedEmployee** has an annual salary, which does not need to be in its string representation.

   **HourlyEmployee** has an hourly rate, and works a fixed number of hours per month (both potentially different per employee), which do not need to be in its string representation.

Your implementations should provide a string representation of Employee objects that shows the employee's name, phone, email, and monthly pay. You do **not** need to provide an `__eq__` method.

You must write docstrings for each class and method with type signatures/annotations for parameters and public attributes given in the format of the example code below.

No examples (such as doctests) are required. Indicate which methods are overriding others with a brief comment in the docstring of the method.

```python
class Employee:
    """Represent an Employee's information

    :param name: The employee's name
    :param phone: The employee's phone number
    :param email: The employee's email
    """
    name = None
    phone = None
    email = None

    def __init__(self, name: str, phone: int, email: str) -> None:
        """Initialize a new employee
        """
        self.name, self.phone, self.email = name, phone, email

    def __str__(self) -> str:
        """Return a string representation of the employee information.
        """
        return "Name: {}
          Phone: {}
          Email: {}
          Monthly Pay: {}"
          .format(self.name, self.phone, self.email,
                   self.get_monthly_payment())

    def get_monthly_payment(self) -> int:
        """Return the monthly payment (in cents) of the employee.
        """
        raise NotImplementedError
```
[ Live programming solution ]
Q1 Grading (Sophia's style)

● 1 mark for inheriting from Employee
● 1 mark for extending __init__ correctly
● 1 mark for adding SalariedEmployee's annual_salary
● 1 mark for adding HourlyEmployee's monthly_hours and hourly_rate
Q1 Grading (Sophia's style)

● 1 mark for correctly implementing SalariedEmployee's get_monthly_payment
● 1 mark for correctly implementing HourlyEmployee's get_monthly_payment
Q1 Grading (Sophia's style)

• 1 mark for correctly saying you extend __init__
• 1 mark for saying you override get_monthly_payment
• 2 marks for all type annotations/dostrings

Approx. -0.5 marks for each error
Class Design Question Goals

- Do you know how to implement a class?
- Inherit from a class?
- Understand how attributes work?
- Understand what overriding/extending is?
def swap(self, other: 'LinkedList') -> None:
   """ Swaps the values of two Linked Lists, leaving node ids intact.
   Raise LinkedListException if lists are different sizes.
   
   >>> lnk1 = LinkedList()
   >>> lnk1.prepend(0)
   >>> lnk1.prepend(1)
   >>> lnk1.prepend(2)
   >>> lnk2 = LinkedList()
   >>> lnk2.prepend(3)
   >>> lnk2.prepend(4)
   >>> lnk2.prepend(5)
   >>> lnk1.swap(lnk2)
   >>> print(lnk1.front)
   5 ->4 ->3 ->|
   >>> print(lnk2.front)
   2 ->1 ->0 ->|
   """
[ Live programming solution ]
Q2 Grading (Sophia's style)

- 1 mark for correctly comparing sizes
- 1 mark for correctly raising exception
- 1 mark for correcting starting at the front of each linked list
- 1 mark for having a correct condition
- 1 mark for correctly swapping
- 1 mark for going to the next node
3. [5 marks] (≈ 10 minutes) queues. Three empty Queues are created and then loaded with some strings:

```python
q1 = Queue()
q1.add("P")
q1.add("S")
q2 = Queue()
q2.add("H")
q2.add("A")
q3 = Queue()
q3.add("E")
```

Choose a sequence of commands from the table below to load `q3` so that it contains "S", "H", "A", "P", "E", in order, with "E" added last. When you're done the code at the bottom of the page should run as stated.

You may not use any other Python expressions except those in the table. You may use some of the commands in the table more than once, some of them not at all.

Hint: Try to draw what the queues contain to start with, and come up with the sequence of actions needed (in picture form, crossing out elements you remove) before writing any python code.

```
<table>
<thead>
<tr>
<th>q1.remove()</th>
<th>q1.add(q2.remove())</th>
<th>q1.add(q3.remove())</th>
</tr>
</thead>
<tbody>
<tr>
<td>q2.remove()</td>
<td>q2.add(q1.remove())</td>
<td>q2.add(q3.remove())</td>
</tr>
<tr>
<td>q3.remove()</td>
<td>q3.add(q1.remove())</td>
<td>q3.add(q2.remove())</td>
</tr>
</tbody>
</table>
```
[ Live programming solution ]
Q3 Grading (Sophia's style)

- 1 mark for moving P after A
- 1 mark for moving E after P
- 1 mark for moving S into the start of q3
- 1 mark for adding H, A, and P into q3
- 1 mark for having SHAPE in q3 in the correct order.
More Midterm Review

- Do midterm_practice.pdf
  - Solution posted on next Wednesday (morning)
  - Prefer you discuss and work it out amongst each other
  - I'll answer questions about it though
Homework

● Exercise 4 (due next Thursday @ 11PM)
● Midterm practice/studying :)
● TA Office Hours:
  ○ Tomorrow (Friday): 3-7PM in BA3289
  ○ More to be announced next week