CS 1110 Prelim 2, April 2024

This 90-minute closed-book, closed-notes exam has 6 questions worth a total of roughly 74 points (some point-total adjustment may occur during grading). Pace yourself accordingly.

You may separate the pages while working on the exam; we have a stapler available.

Note: throughout the exam you may assume that preconditions are met. No need to assert them.

Also note that `x += 1` means `x = x + 1`

It is a violation of the Academic Integrity Code to look at any exam other than your own, to look at any reference material besides the reference provided in the exam itself, or to otherwise give or receive unauthorized help.
We also ask that you not discuss this exam with students who are scheduled to take a later makeup.

Academic Integrity is expected of all students of Cornell University at all times, whether in the presence or absence of members of the faculty. Understanding this, I declare I shall not give, use or receive unauthorized aid in this examination.

Signature: _____ Date _____

Name (First Last):

Cornell NetID, all caps:

This is a comprehensive reference sheet that might include functions or methods not needed for your exam.

String methods

`s[i:j]`	Returns: if `i` and `j` are non-negative indices and `i ≤ j-1`, a new string containing the characters in `s` from index `i` to index `j-1`, or the substring of `s` starting at `i` if `j ≥ len(s)`
`s.count(s1)`	Returns: the number of times `s1` occurs in string `s`
`s.find(s1)`	Returns: index of first occurrence of string `s1` in string `s` (-1 if not found)
`s.find(s1,n)`	Returns: index of first occurrence of string `s1` in string `s` STARTING at position n. (-1 if `s1` not found in `s` from this position)
`s.index(s1)`	Returns: index of first occurrence of string `s1` in string `s`; raises an error if `s1` is not found in `s`.
`s.index(s1,n)`	Returns: index of first occurrence of string `s1` in string `s` STARTING at position n; raises an error if `s1` is not found in `s` from this position
`s.isalpha()`	Returns: `True` if `s` is *not empty* and its elements are all letters; it returns `False` otherwise.
`s.isdigit()`	Returns: `True` if `s` is *not empty* and its elements are all numbers; it returns `False` otherwise.
`s.islower()`	Returns: `True` if `s` is has at least one letter and all letters are lower case; returns `False` otherwise (*e.g.*, 'a123' is `True` but '123' is `False`).
`s.isupper()`	Returns: `True` if `s` is has at least one letter and all letters are upper case; returns `False` otherwise (*e.g.*, 'A123' is `True` but '123' is `False`).
`s.lower()`	Returns: a copy of `s`, all letters converted to lower case.
`s.join(slist)`	Returns: a string that is the concatenation of the strings in list `slist` separated by string `s`
`s.replace(a,b)`	Returns: a *copy* of `s` where all instances of `a` are replaced with `b`
`s.rfind(s1)`	Returns: the highest index in string `s` where string `s1` is found (-1 if not found)
`s.split(sep)`	Returns: a list of the "words" in string `s`, using `sep` as the word delimiter (whitespace if `sep` not given)
`s.strip()`	Returns: copy of string `s` where all whitespace has been removed from the beginning and the end of `s`. Whitespace not at the ends is preserved.
`s.upper()`	Returns: a copy of `s`, all letters converted to upper case.

List methods

`lt[i:j]`	Returns: if `i` and `j` are non-negative indices and `i ≤ j-1`, a new list containing the elements in `lt` from index `i` to index `j-1`, or the sublist of `lt` starting at `i` if `j ≥ len(s)`
`lt.append(item)`	Adds `item` to the end of list `lt`
`lt.count(item)`	Returns: count of how many times `item` occurs in list `lt`
`lt.index(item)`	Returns: index of first occurrence of `item` in list `lt`; raises an error if `item` is not found. (There's no "find()" for lists.)
`lt.index(y, n)`	Returns: index of first occurrence of `item` in list `lt` STARTING at position n; raises an error if `item` does not occur in `lt`.
`lt.insert(i,item)`	Insert `item` into list `lt` at position `i`
`lt.pop(i)`	Returns: element of list `lt` at index `i` **and also removes that element from the list `lt`**. Raises an error if `i` is an invalid index.
`lt.remove(item)`	Removes the first occurrence of `item` from list `lt`; raises an error if `item` not found.
`lt.reverse()`	Reverses the list `lt` in place (so, `lt` is modified)
`lt.sort()`	Rearranges the elements of `x` to be in ascending order.

Dictionary Operations

`d[k] = v`	Assigns value `v` to the key `k` in `d`.
`d[k]`	If value `v` was assigned to the key `k` in `d`, `d[k]` evaluates to `v`.
`del d[k]`	Deletes the key `k` (and its value) from the dictionary `d`.
`d.keys()`	**Returns**: an iterator of all the keys in dictionary `d`.
`d.values()`	**Returns**: an iterator of all the values in dictionary `d`.

Other useful functions

`s1 in s`	Returns: `True` if the substring `s1` is in string `s`; `False` otherwise.
`elem in lt`	Returns: `True` if the element `elem` is in list `lt`; `False` otherwise.
`y in d`	Returns: `True` if `y` is a key in dictionary `d`; `False` otherwise.
`y in d.values()`	Returns: `True` if `y` is a value in dictionary `d`; `False` otherwise.
`input(s)`	prompts user for a response using string `s`; returns the user's response as a string.
`isinstance(o, c)`	Returns: `True` if `o` is an instance of class `c`; `False` otherwise.
`len(s)`	Returns: number of characters in string `s`; it can be 0.
`len(lt)`	Returns: number of items in list `lt`; it can be 0.
`len(d)`	Returns: number of keys in dictionary `d`; it can be 0.
`list(range(n))`	Returns: the list [0 .. n-1]

1. [8 points] **Iteration and lists.** It's election time! Every vote counts, but let's make *some* votes count more than others. Implement the following function, making effective use of for-loops.

```python
def amplify_my_vote(ballots, chosen):
    """    Modifies the list `ballots` as follows:
    Every vote for the `chosen` candidate is added a second time
    to the end of the ballot list.

    Note: the votes should not be considered case sensitive. If the
    chosen candidate is "Matt", the function SHOULD double votes
    for "MATT" "matt" "MAtt"...  with their original cases.

    This function does not return anything.

    Examples:

    amplify_my_vote(["Ada", "Bob", "Caz", "Deb"], "Ada")
            ballots becomes ["Ada", "Bob", "Caz", "Deb", "Ada"]

    amplify_my_vote(["ada", "bob", "ADA", "deb"], "Ada")
            ballots becomes ["ada", "bob", "ADA", "deb", "ada", "ADA"]

    amplify_my_vote(["Leo", "Mik", "Leo"], "")
            ballots becomes/remains ["Leo", "Mik", "Leo"]

    amplify_my_vote([], "Ada")
            ballots becomes/remains []

    Preconditions:
       ballots: a (possibly empty) list of str
       chosen: a (possibly empty) str                      """
```

2. [14 points] **Dictionaries.** Implement this function according to its specification:

```
def get_mapping(word1, word2):
    """
    Returns the dictionary that contains the mapping that would turn
    `word1` into `word2` or None if no such mapping exists (meaning no
    1-for-1 character replacement can produce `word2` from `word1`)

    In this dictionary, the key is the character from `word1` and
    the value is the character it should be replaced with in order
    to produce `word2`

    A character may be replaced by itself ('love'->'love').
    More than 1 character may be replaced by the same character ('rap'->'ooo')
    One character may NOT be replaced by multiple characters ('moo'->'mop')

    Examples:

    word1: 'love'   , word2: 'love'
          the function returns {'l':'l','o':'o','v':'v','e':'e'}

    word1: 'book'   , word2: 'seem'
          the function returns {'b':'s', 'o':'e', 'k':'m'}

    word1: 'sassy'  , word2: 'daddy'
          the function returns {'s':'d', 'a':'a', 'y':'y'}

    word1: 'rap'    , word2: 'ooo'
          the function returns {'r':'o','a':'o','p':'o'}

    word1: ''       , word2: ''
          the function returns {}

    word1: 'lovely' , word2: 'lo'
          the function returns None

    word1: 'moo'    , word2: 'mop'
          the function returns None

    Precondition:  `word1` and `word2` are (possibly empty) strings
                   of exclusively lower case letters a-z

    """
```

Please implement your function on the next page!

```python
def get_mapping(word1, word2):
```

3. [14 points] **Recursion.** Imagine that we are representing Python modules Python classes. Let `Module` be a class whose objects have the following two attributes:

> name   [str] – unique non-empty name of module
> imps   [possibly empty list of Module] – modules that need to be
>   imported in order for this module to work correctly

Implement the following **function** (*not* an object method), making effective use of recursion. For-loops are allowed as long as your solution is fundamentally recursive.

```python
def depends_on(mod, target):
    """Returns: True if Module `mod` depends on a Module named `target`
                False otherwise

    Directly, a Module depends on itself and its imports. Indirectly, a
    Module also depends on all of the Modules that its imports depend on.
    So if A imports B, and B imports C, then A depends on C.

    Parameter `mod` : the Module we're interested in
    Precondition: `mod` is a Module object

    Parameter `target` : name of the Module we want to know if `mod` depends on
    Precondition: `target` is a str

    ------    Here is a simplified example from A6 ------
    m1 = Module("math",[])          # imports nothing
    m2 = Module("consts",[])        # imports nothing
    m3 = Module("cnlasserts", [m1]) # imports math
    m4 = Module("player", [m2, m3]) # imports consts, cnlasserts

    depends_on(m1, "math")  Returns True  (math depends on itself)
    depends_on(m2, "math")  Returns False (consts imports nothing)
    depends_on(m3, "math")  Returns True  (cnlasserts imports math)
    depends_on(m4, "math")  Returns True  (player depends on math (via cnlasserts))

    """
```
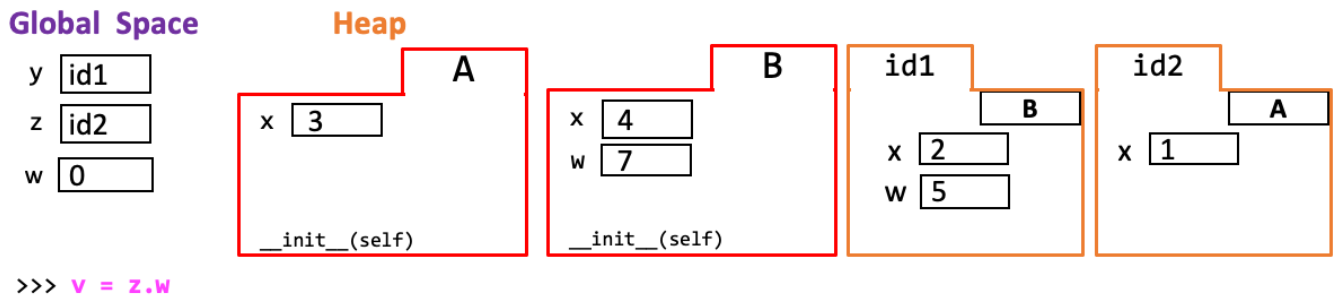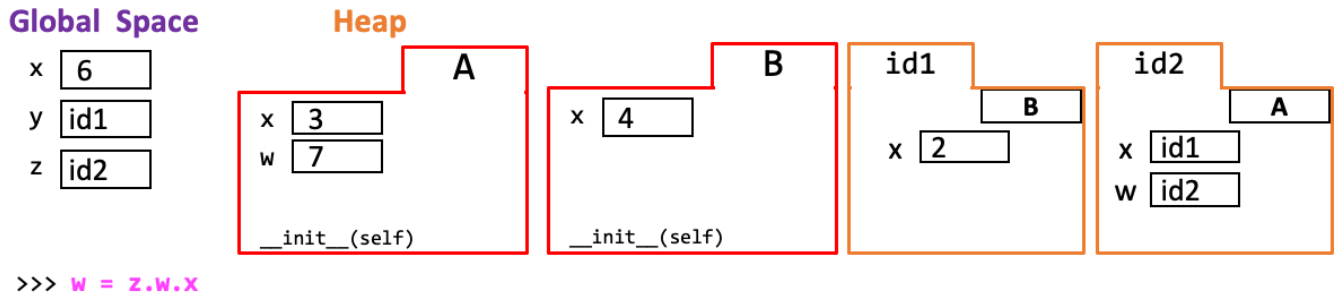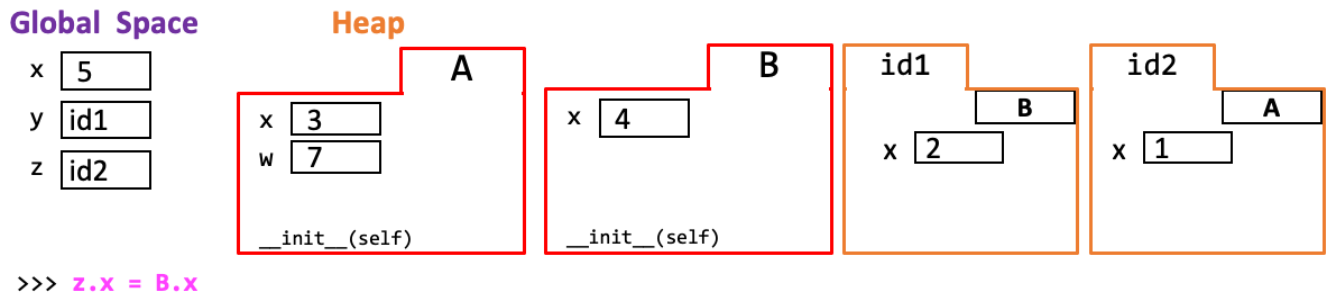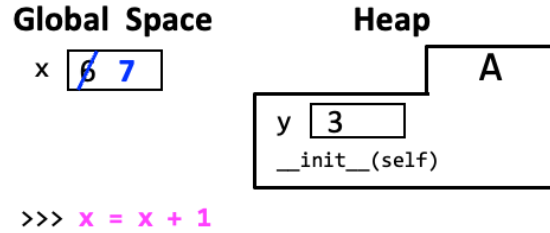
Please implement your function on the next page!

```python
def depends_on(mod, target):

    # STUDENTS: put your base case here:




    # STUDENTS: put the rest of your solution here:
```

4. [8 points] **Visualizing Python.** For this question, you will be shown the state of memory before a single assignment statement is executed. Modify the drawing to show how memory changes after that single assignment statement has been executed. If at any point an error is thrown, please write **ERROR** next to the assignment statement; only draw the changes to memory that would occur before the error occurs. Each part is independent.
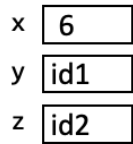
Notice: there is no Call Stack.
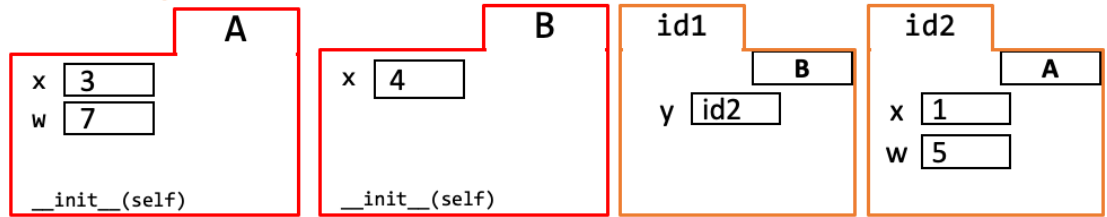*(The assignment statements are not inside functions or methods.)*
To the right is an example. The statement to execute is shown in **pink** and the answer is shown in **blue**.

Global Space

x   ~~6~~ 7

Heap

A
y   3
__init__(self)

>>> x = x + 1

Global Space

x   5
y   id1
z   id2

Heap

A
x   3
w   7
__init__(self)

B
x   4
__init__(self)

id1
B
x   2

id2
A
x   1

>>> z.x = B.x

Global Space

x   6
y   id1
z   id2

Heap

A
x   3
w   7
__init__(self)

B
x   4
__init__(self)

id1
B
x   2

id2
A
x   id1
w   id2

>>> w = z.w.x

Global Space

y   id1
z   id2
w   0

Heap

A
x   3
__init__(self)

B
x   4
w   7
__init__(self)

id1
B
x   2
w   5

id2
A
x   1

>>> v = z.w

Page 8

Global Space

x `6`
y `id1`
z `id2`

Heap

A
x `3`
w `7`
__init__(self)

B
x `4`
__init__(self)

id1
B
y `id2`

id2
A
x `1`
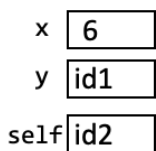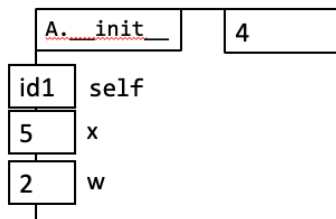w `5`

>>> y.w = y.x

5. [8 points] **Visualizing Methods.** For this question, you will be shown the state of memory before a single Python statement is executed. Modify the drawing to show how memory changes after that single statement has been executed. If at any point an error is thrown, please write **ERROR** next to the assignment statement; only draw the changes to memory that would occur before the error occurs. Do not worry about changing the Program Counter in the top right corner of the call frame. (Since we are not showing you the code, you can't know what the next line of executable code will be). Once again, each part is independent. Notice that there is a call stack: each line being executed exists inside a method.
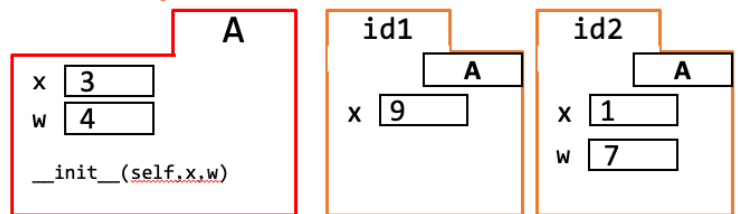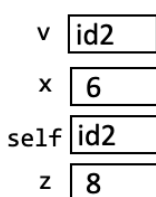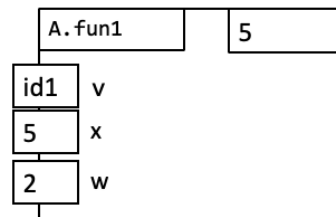

Global Space

x `6`
y `id1`
self `id2`

Call Stack

A.__init__   `4`
id1 | self
5 | x
2 | w

Heap

A
x `3`
w `4`
__init__(self,x,w)

id1
A
x `9`

id2
A
x `1`
w `7`

execute line 4 of A.__init__: **x = self.w**

Global Space

v `id2`
x `6`
self `id2`
z `8`

Call Stack

A.fun1   `5`
id1 | v
5 | x
2 | w

Heap

A
x `3`
w `7`
__init__(self)
fun1(v,x,w)

id1
A
x `1`

id2
A
x `9`
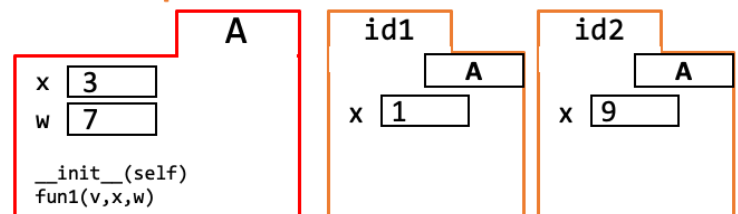
execute line 5 of A.fun1:   **z = self.x**

Global Space
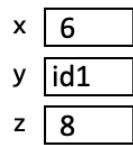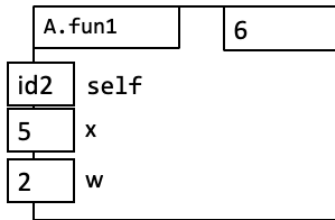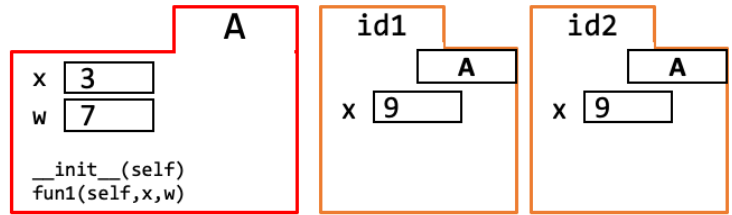
x `6`

y `id1`

z `8`

Call Stack

| A.fun1 | `6` |

| id2 | self |

| 5 | x |

| 2 | w |

execute line 6 of A.fun1:   return y == self

Heap

A

x `3`

w `7`

__init__(self)
fun1(self,x,w)

id1

A

x `9`

id2

A

x `9`

Global Space

x `6`

y `id1`

z `8`

self `id1`

Call Stack

| A.fun1 | `6` |

| id2 | self |

| 5 | x |

| 2 | w |

execute line 6 of A.fun1:   return y == self

Heap

A

x `3`

w `7`

__init__(self)
fun1(self,x,w)

id1

A

x `9`

id2

A

x `9`
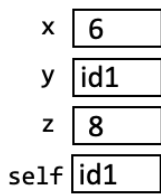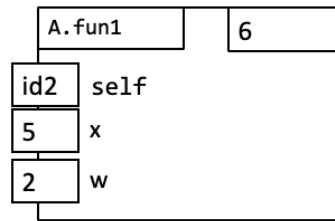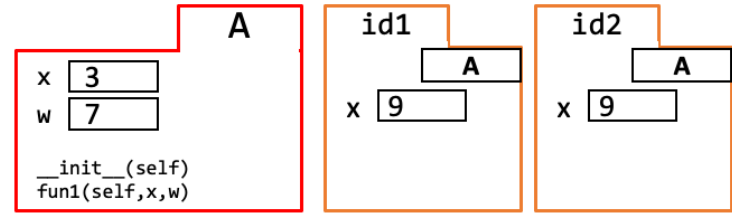
6. **Classes.** Consider the following class, defined as follows:

```python
class SandwichOrder:
    """A class to represent a sandwich order

    3 Class Attributes:
        total_orders:  [int] the total number of sandwich orders
        total_revenue: [float] total revenue from all sandwich orders
        PRICES:        [dict] prices of 'bread', 'protein', 'cheese', 'topping'

    6 Instance Attributes:
        name:         [str] name of the customer, for example 'Luke Calka'
        bread:        [str] name of the bread, for example 'Wheat'
        protein:      [str] name of the protein, for example 'Ham',
                         empty string ('') indicates no protein on the sandwich
        with_cheese:  [bool] True if the Sandwich should have cheese on it
        toppings:     [list of str without duplicates]
        price:        [float] should ALWAYS represent the price of this sandwich
    """

    # Class Attributes
    # These should ALWAYS accurately reflect the state of all orders made
    total_orders = 0                  # the number of sandwich orders
    total_revenue = 0.0               # the total prices across all orders


    PRICES = {'bread': 2.00, 'protein': 4.00, 'cheese': 1.50, 'topping': 0.50}
              # PRICES are by category:
              #               all proteins cost $4
              #               each topping costs $0.50
              #               ... and so forth
```

(a) [8 points] Implement this method of `SandwichOrder` according to its specification:

```python
def calculate_price(self):
    """
    Calculates the price of the sandwich order based on the PRICES of
    bread, protein, cheese, and toppings.

    The price should be assigned to the object's `price` attribute.
    Returns: Nothing!

    Example:
        If sandwich order has wheat bread, turkey protein, cheese,
            and 3 toppings: lettuce, tomato and mayo
    price would be: 2.00 (bread) + 4.00 (protein) + 1.50 (cheese) +
    + 0.50 (topping) + 0.50 (topping) + 0.50 (topping) = $9.00

    Precondition: the identifier self is an initialized SandwichOrder with
        attributes name, bread, protein, with_cheese, and toppings
    """
```

(b) [12 points] Implement the `__init__` method of `SandwichOrder` according to its specification. Avoid redundancies between your code here and the previous page.

```python
def __init__(self, name, bread, protein, cheese, tops=None):
    """
    Initializes a new instance of a SandwichOrder with given parameters.
    Calculates initial price of the sandwich.
    Also updates relevant Class Attributes as necessary.

    Example calls:
    s1 = SandwichOrder('John Wick', 'Wheat', '', True, ['Pickles', 'Mayo'])
    s2 = SandwichOrder('Max Rockatansky', 'Sourdough', 'Chicken', False)

    parameters name, bread, protein, and cheese can all be assigned to
    their respective instance attributes (see class specification)

    toppings: should be assigned the empty list [] if no list is provided
                (i.e., `tops` has the default value None)
              should be assigned the list id `tops` if a list IS supplied as
                an argument (do not create a new list)

    Preconditions: name: non-empty str
                   bread: non-empty str
                   protein: (possibly empty) str
                   cheese: bool, False means no cheese please
                   tops: None or [list of str] of chosen toppings
                            items in tops are guaranteed to be unique
    """
```

7. [10 points] **Debugging.** Consider the following two classes and 3 lines of code that use them:

```python
class Engine:
    """    A class to represent an engine.

    2 Instance Attributes:
    max_time,   INVARIANT: non-empty str of digits, example: '1000'
    curr_usage, INVARIANT: int >= 0
    """
    # Attributes can be changed, but the INVARIANTS must always be satisfied.
    # Example: curr_usage should never go negative or become a float.
    def __init__(self, maxi):
        """
        Initializes: the Engine with the given maximum time.
                     the current usage to zero.

        Precondition: maxi: a str, a positive number ending in 'h', like '25h'
        """
        self.max_time = maxi.replace('h', '')
        self.curr_usage = 0

class Airplane:
    """    A class to represent an airplane with some number of engines
    Instance Attribute `engines`: a (possibly empty) list of Engine
    """
    def __init__(self, engines):
        """  Initializes: the Airpline with the given engine list """
        self.engines = engines

    def use_engine(Engine, engine, hour):
        """ Increases `engine`'s current usage by `hour`
        Preconditions: engine is a int, 0 <= engine < len(self.engines)
                       hour is a int >= 0                              """
        Engine.engines[engine].curr_usage += Engine.engines[engine].hour


    def needs_service(self):
        """    True if any engine's usage reaches its max.    """
        i = 0
        while i < len(self.engines):
            eng = self.engines[i]
            if eng.curr_usage >= eng.max_time:
                return True
            i = i + 1
            return False

a = Airplane([Engine('2500h'), Engine('2500h')])
a.use_engine(0, 100)
print(a.needs_service())
```

When the given code is run in Python, the following error is reported:

```
Traceback (most recent call last):
  File "airplane.py", line 46, in <module>
    a.use_engine(0, 100)
  File "airplane.py", line 32, in use_engine
    Engine.engines[engine].curr_usage += Engine.engines[engine].hour
AttributeError: 'Engine' object has no attribute 'hour'
```

(a) Fix the code so that the line of code throwing the above error can successfully execute. **Fix only the code that is responsible for throwing the error.** Mark your fix(es) with the label **FIX1**.

(b) You fix the above error and rerun the code. Now a new error is reported:

```
Traceback (most recent call last):
  File "airplane.py", line 47, in <module>
    print(a.needs_service())
  File "airplane.py", line 40, in needs_service
    if eng.curr_usage >= eng.max_time:
TypeError: '>=' not supported between instances of 'int' and 'str'
```

Fix the code to remove only this new error. **Fix only the code that is responsible for throwing the error. Pay attention to the invariants that must remain satisfied.** Mark your fix(es) with the label **FIX2**.

(c) Now that you've dealt with these errors, let's address the functionality of one last method. The `needs_service` method should return `True` if at least one engine's `curr_usage` is greater than or equal to its `max_time`, otherwise return `False`. When we run the above code (after fixing parts (a) and (b)), it prints `False`. Is there a bug in the `needs_service` method?

Circle One: Yes   No

If you answered Yes, fix the bug and mark your fix(es) with the label **FIX3**. If you answered No, provide a test (like lines 45-47) that when run, will correctly print `True`.