In problems 1a-1c and 4a-4c, we can store a simplified **Scooter Rental** database as a nested **dict**. The **outer dict** has keys (**str**) that are scooter names: e.g., **'s1'**, each associated with an **inner dict** whose keys are **date 2-tuples** (month: **1-12** followed by day in the month: **1-31**), each associated with a **list** of **3-tuples**: the renter's **name** (**str**), the **minutes** rented (**int**), the **feet** travelled (**int**). For example, a small database has the form:

```
db = {'s1': {(11, 6): [('Amy',10,2000), ('Bob',8,1500), ('Bob',7,1500)],  # Bob rents twice
             (11, 7): [('Cam',12, 800)]
            },
      's2': {(10,31): [('Cam',12, 800), ('Ida',30,8000)],
             (11, 6): [('Gil',20, 400), ('Dan',17,3500), ('Eva',15,1800)]
            }
     }
```

1a. (6 pts) Write a function named **rentals**: it takes two arguments: **db** (a Scooter Rental database) and **date** (**2-tuple** in the form **(month,day)**; assume -don't check whether- it is a legal one). The **rentals** function returns the total number of rentals on that **date**. **Requirement**: You must write the body of this function as **one return statement**, but you can use **multiple lines** to write that statement clearly.

```
def rentals (db : {str:{(int,int): [(str,int,int)]}},  date : (int,int)) -> int:
```

1b. (6 pts) Write a function named **by_month**: it takes one argument: **db** (a Scooter Rental database). It returns a dictionary (**dict** or **defaultdict**) whose keys are months (**int**: 1-12) and whose associated values are the sum (**int**) of the **feet** travelled for all rentals in that month. There are **no statement requirements** for this problem. Hint: don't iterate over the 12 months; naturally process months by iterating over the **db**.

```
def by_month (db : {str:{(int,int): [(str,int,int)]}}) -> {int:int}:
```

1c. (8 pts) Write a function named **by_name**: it takes one argument: **db** (a Scooter Rental database). It returns a **sorted list** of renter names (**str**). These names are sorted in **decreasing order** by the **number of rentals** (**int**) for each renter; and if two renters have the same number of rentals, they appear in decreasing order of the **feet** (**int**) **they travelled**. There are **no statement requirements** for this problem. Hint: Solve this problem similarly to 1b, by building a useful dictionary, associating each renter name with data that are useful for sorting.

```
def by_name (db : {str:{(int,int): [(str,int,int)]}}) -> [str]:
```

2a. (6 pts) In the **Match?** column in the table below, circle whether or not the Regular Expression (RE) pattern string `"^x([ab]?c)*(x+)$"` matches each of the following test strings

String #	Test String	Match?
1	`"xxx"`	**(see answer sheet)**
2	`"xccccx"`	**(see answer sheet)**
3	`"xacbcxx"`	**(see answer sheet)**
4	`"xaacxx"`	**(see answer sheet)**
5	`"x"`	**(see answer sheet)**
6	`"xaax"`	**(see answer sheet)**
7	`"xcacbcxx"`	**(see answer sheet)**

2b. (9 pts) Recall that the `sub` (substitute) function in the `re` module is defined like

```
def sub(re_pattern : str,  rep_func : callable,  text : str)  ->  str:
```

It searches `text` for substrings matching `re_pattern`; for every substring that it finds, it replaces it with the result of calling `rep_func`, supplying the resulting match object as its argument.

For example `re.sub(r"a\d", (lambda m : "A#"), "a5bcayza7")` returns `"A#bcayzA#"`: every occurrence of `a` followed by any digit is replaced by `"A#"`. We don't use `^` or `$` in this pattern, because the pattern can match anywhere in `text`. Here `rep_func` is a simple `lambda` that **does not use** the match object it is passed, but `m` is **bound to the match object** so the **match object** could have been used inside the `lambda`.

We define a **big-number-list** (**BNL**) as a sequence of any number (including 0: none) of **two**(or more)**-digit integers** that are **unsigned** and appear between `<` and `>`; integers are separated by commas (`,`) and there are **no spaces** inside the **BNL**. For example, `<>` is a **BNL; so is** `<22>`; so is `<237,42,128>`. But `<87,5>` is not a **BNL** because `5` has only one digit. Recall that the characters `<` and `>` have no special meaning in RE.

Define a `pattern` and `rep_func` below, so the call `re.sub(pattern, rep_func, text)` replaces every **BNL** in `text` by its number of digits inside the `<>`. For example `re.sub(pattern, rep_func, 'By processing <237,42,128> and <>')` returns the substituted string `'By processing <8> and <0>'`

Hint: recall that if an option is discarded in a RE, then its match group is `None`, not an empty string.

```
pattern =
```

```
def rep_func(mo)  ->  str :    #mo is the match object for each time pattern matches in text
```

3. (15 pts) When we take a measurement that might include errors, we might record its value as `7.5 ± .25` (meaning a measured value of **`7.5` plus or minus the error bound `.25`**); we can also represent this same measurement as an interval `7.25` to `7.75` meaning a measurement somewhere between these two values: a number that is ≥ `7.25` and ≤ `7.75`. In this problem we will represent measurements in this second form: whose first number represents the lowest value the measurement can be, and whose second number represents the highest value the measurement can be: the first number must always be ≤ the second: note that an interval of `7.5` to `7.5` is legal: it means a measurement that is an exact value: its error bound is ±`0`.

The following class defines `__init__` for a `Measurement` from two numbers (either `int` or `float`). For example, to represent the measurement above, we can define `m = Measurement(7.25,7.75)`. In the class below, **(a)** define the method called by the `repr` function, **(b)** define the method called by the `<` operator to work for two `Measurement` objects: `a < b` is `True` if the highest possible value of `a` is less than the lowest possible value of `b`, and **(c)** overload the **addition** operator so that we can add two `Measurement`s and can also add a `Measurement` and an **exact value** (`int` or `float`): the `Measurement` can appear on either side of the **exact value**. Note that **addition is commutative**. Attempting to add any other type to a `Measurement` should fail.

Adding two `Measurement`s results in a `Measurement`: adding `Measurement(a,b)` to `Measurement(c,d)` results in `Measurement (a+c,b+d)`: adding its lowest and highest numbers. Adding a `Measurement` and an **exact value** results in a `Measurement:` adding `Measurement(a,b)` to **exact value** `v` results in `Measurement(a+v,b+v)`: adding the **exact value** to both the lowest and highest numbers; or, think of an **exact value** `v` as `Measurement(v,v)`: a `Measurement` that is exact: its lowest and highest numbers are the same.

```
class Measurement:
    def __init__(self,low,high):
        assert low<=high, f'Measurement.__init__: low({low})>high({high})'
        self.low  = low
        self.high = high


    # a) overload the method called by repr



    # b) overload the method called by < assuming both objects are Measurements



    # c) overload + allowing Measurement + Measurement, Measurement + int or float, and
    #   int or float + Measurement; addition fails for any other combinations
```

4. (15 pts) We can store a simplified **Scooter Rental** database as a nested `dict`. The **outer `dict`** has keys (`str`) that are scooter names: e.g., `'s1'`, each associated with an **inner `dict`** whose keys are `date 2-tuples` (month: `1-12` followed by day in the month: `1-31`), each associated with a `list` of `3-tuples`: the renter's `name` (`str`), the `minutes` rented (`int`), the `feet` travelled (`int`). For example, a small database has the form:

```
{'s1': {(11, 6): [('Amy',10,2000), ('Bob',8,1500), ('Bob',7,1500)],   # Bob rents twice
        (11, 7): [('Cam',12, 800)]
       },
 's2': {(10,31): [('Cam',12, 800), ('Ida',30,8000)],
        (11, 6): [('Gil',20, 400), ( 'Dan',17,3500),( 'Eva',15,1800)]
       }
}
```

Define the `Scooter_DB` class; it is initialized with a `list` of `5-tuples`, of the form `('s1',(11,6),'Amy',10,2000)`: representing all the information needed for one rental,. In its `__init__` method, it creates a `db` **attribute** to store such values in a nested dictionary (`dict` or `defaultdict`) of the form shown above, creating **no other attributes** for `Scooter_DB` objects. Assume `sdb = Scooter_DB(...)`

a. Write the `__contains__` method to determine whether or not a specified person has ever rented a specified scooter: e.g., checking whether `('Bob','s1') in sdb`. The first argument should be a `2-tuple` with the renter's name followed by the scooter's name. If either name is not in the Scooter Database; return `False`.

b. Overload the `__getitem__` method so that `s[date]` returns a `list` of `3-tuples` containing all rentals on that `date` in the database: `sdb[(11,6)]` returns a `list` containing (in any order) `[('Amy',10,2000), ('Bob',8,1500), ('Bob',7,1500), ('Gil',20,400), ('Dan',17,3500), ('Eva',15,1800)]`

c. Overload the `__delitem__` method so that `del sdb[date]` deletes all rentals on that `date` from the `Scooter_DB` object. Recall that trying to `del` a key not in a dictionary raises a `KeyError` exception.

When `sdb[11,6]` is translated the `date` argument is bound to the `2-tuple` `(11,6)`.

```
class Scooter_DB:
    def __init__(self,rentals):
        ... code to initialize self.db into nested dict form shown above
```

5. (5 pts) Answer each question directly, in one sentence.

5a1. (2 pts) How does Python recognize that code is a generator function?

5a2. (1 pts) When a generator function is called, what does it return?

5a3. (2 pts) When is the first statement in a generator function's body executed?

5b. (5 pts) Write the **`no_same_adjacent`** iterator decorator as a generator function: it has one parameter, **`iterable`**. It produces each value in **`iterable`**, but never the same value **twice in a row** (always producing the first value because there was no value before it). For example, calling **`list(no_same_adjacent('aaaabaccd'))`** returns **`['a', 'b', 'a', 'c', 'd']`**: note each value is different from the one before it; the **`a`** appears twice but not **in a row**. **Requirement**: write this generator function using a **`for`** loop: use no **`while`** loops; you can define local variables, but cannot create any new data structures.

```
def no_same_adjacent (iterable):
```

5c. (5 pts) Write the **`pairs`** iterator decorator as a generator function: it has one parameter, **`iterable`**. It produces a **`2-tuple`** that is the first two values in the **`iterable`**, then the second, two, etc. For example, calling **`list(pairs('abcd'))`** returns **`[('a','b'), ('c','d')]`**. If there are an odd number of values in **`iterable`**, it ultimately returns a **`2-tuple`** with the last value followed by **`None`**: **`list(pairs('abcde'))`** returns **`[('a','b'), ('c','d'), ('a',None)]`**. Also, **`None`** may appear in **`iterable`**: calling **`list(pairs([None,None,None]))`** returns **`[(None,None), (None,None)]`**. Recall that well-behaved generator functions **`return`** instead of raising a **`StopIteration`** exception. **Requirement**: write this generator function using a **`while`** loop: use no **`for`** loops; you can defome local variables, but cannot create any new data structures.

```
def pairs(iterable):
```

6a. (4 pts) Indicate the **places** where Python looks, in the **order** in which it looks at them, to **locate the value bound to a name `n`** used inside some **function body** (not an attribute of an object; for attributes see 6b). Assume no special declarations (e.g., no `nonlocal`) about the name in the function. You do not need to fill in all blanks for full credit.

(1st)

(2nd)

(3rd)

(4th)

(5th)

6b. (4 pts) Indicate the **places** where Python looks, in the **order** in which it looks at them, to **locate the value bound to an attribute** of an object `o`. You do not need to fill in all blanks for full credit.

(1st)

(2nd)

(3rd)

(4th)

(5th)

6c (7 pts) Answer each question in one sentence.

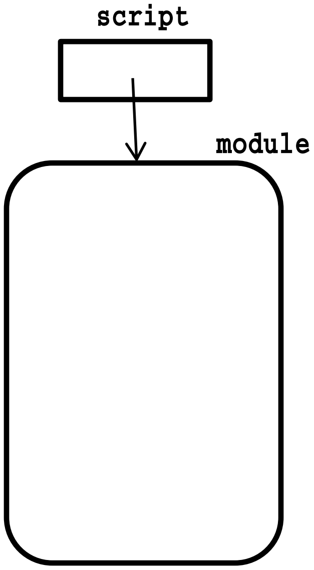
6c1. (3 pts) How can we control the meaning of an operator applied to an object constructed from a class we write?

6c2. (2 pts) When a method is called on an object, how does Python determine in which class the method is defined?

6c3. (2 pts) In what fundamental way does Python treat `x + y` different than `y + x`? Hint: think about something interesting you can say about objects `x` and `y`.

7a. (5 pts). Draw an appropriate box/oval/arrow picture that illustrates the following computation. All definitions occur in the module **script**. Label all boxes/variables with the name of the variable; label all ovals/objects with the name of its type. If a reference is changed, cross out the old reference and write in the new one.

```
x = [0,0]
y = [x,x[0]]
x[1] = y
x[1][0][0] = x[1]
x[0][1] = '2'
```



7b. (5 pts) Write a function named `of_all`, which takes any number of arguments that each refer to a function object: assume each function object takes **one argument** and returns an `int`; The `of_all` function **returns a function** that takes one argument; when called it returns the **sum** of all the values produced by the function objects passed as arguments to `of_all`. For example, if we write `oa = of_all (lambda x : x+1, lambda x : 2*x)` then calling `oa(3)` returns `10` because `3+1` is `4` and `2*3` is `6` and the **sum** of `4` and `6` is `10`.

Mistake on the Answer Sheet. Write the function header and its body on the answer sheet.

7c. (5 pts) The following code (lines are numbered) is **WRONG**. It is supposed to print `d`'s keys and their values in decreasing order of the values (see box on the right for correct output). Carefully examine the code.

```
1. d = {'a': 1, 'b': 3, 'c' : 2}
2. sorted_d = sorted(d.items(), key = lambda x : -x[1] )
3. for k,v in sorted_d.items():
4.     print(k,v)
```

Output should be:

```
b 3
c 2
a 1
```

(1) Explain where Python detects that something is wrong when executing this code. Explain what is wrong.

(2) Rewrite one line above (label it 1-4) so that the code produces the correct output: rewrite only one line.