Print only this page, answer problem #1 on it, and then submit it as a .pdf file by Thursday at 11:30pm on Gradescope. Leave a good amount of time for submitting on Gradescope.

When working on this quiz, recall the rules stated on the Academic Integrity statement that you signed. You can download the **q1helper** project folder (available for Friday, on the **Weekly Schedule** link) in which to write/test/debug your code. Submit your completed **q1solution.py** module online by Thursday, 11:30pm. I will post my solutions to EEE reachable via the **Solutions** link on Saturday morning (after Problem #1 is due).

---

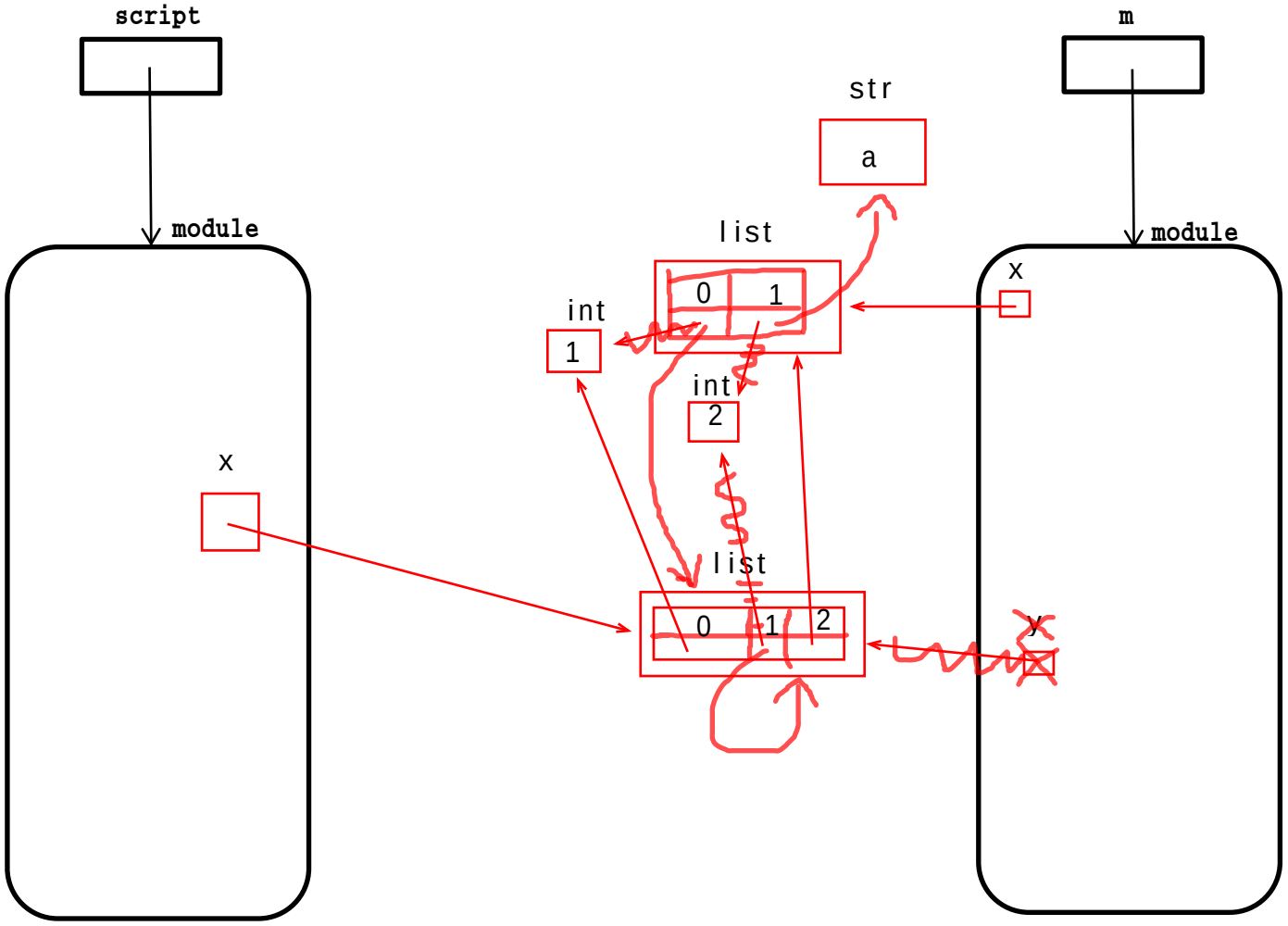
1. (5 pts) Draw a picture illustrating how Python executes the following code. Standard pictures show each name above a square box that contains the tail of an arrow, whose head points to an object. An object is shown as an oval (or rounded-corner square) labelled by the type of the object, whose inside shows its value (which may contain more names, arrows/references: **lists** can include or omit index numbers). When you change an existing reference, lightly cross out the old arrow and then draw the new one appropriately. Draw the picture with **no crossing arrows** (practice on a whiteboard and move things around if necessary).

script module (execution starts here)

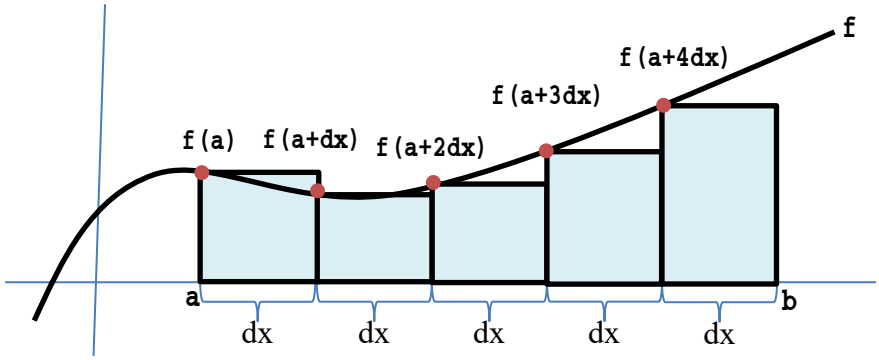
```
import m
x = m.y
x.append(m.x)
m.y[1] = x
m.x[0] = m.y[1]
x[1][2][1] = 'a'
del m.y
```

m module (imported by script)

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



2. (4+1 pts) Define the **integrate** function, which takes two arguments: (1) a function –call it **f**– (it is univariate: called with one numeric argument and returning a numeric result) and (2) an **int** value –call it **n**; if **n** is not an integer strictly greater than **0**, raise an **AssertionError** exception (with an appropriate error message). The **integrate** function returns another function that takes two **float** arguments –call them **a** and **b**; if **a** is not less than or equal to **b**, raise an **AssertionError** exception (with an appropriate error message). When the returned function is called, it returns an approximation to the definite integral $\int_a^b f(x)dx$. We approximate this integral by adding up the area of a sequence of **n** rectangles, whose widths are all **(b-a)/n** (call this number **dx**, illustrated below) and whose heights are **n** values of **f** starting at **f(a)** and ending at **f(a+(n-1)dx)**. If we define **int_f = integrate(f,5)**, Here is how to compute **int_f(a,b)**.



For the fifth/last point, determine how **integrate**'s returned function object can store an attribute named **zcc** (zero-crossing count) that computes the number of times **f**'s value crosses 0: from negative (< 0) to positive (≥ 0) or from positive to negative: For the picture, after calling **int_f(a,b)** the **int_f.zcc** attribute is **0**: here f(…) is always positive. Hint: function objects can store attributes: determine how they can be initialized and updated; there is a relatively simple Boolean/relational expression computing whether two values have opposite signs.

3-4. (16 pts) A stock trading company stores a **database** that is a dictionary (**dict**) whose keys are client names (**str**); associated with each client name is a **list** of **3-tuples** representing the transaction history of that client: in the **3-tuple**, the first index is a stock name (**str**), the second index is the number of shares traded for that transaction (**int**: positive means buying shares; negative means selling shares), and the third index is the price per share in **dollars** (**int** for simplicity) for that transaction. The **list** shows the **order** in which the transactions occurred. A simple/small database can look like

```
{
  'Carl': [('Intel',30,40), ('Dell',  20,50), ('Intel',-10,60), ('Apple', 20,55)],
  'Barb': [('Intel',20,40), ('Intel',-10,45), ('IBM',   40,30), ('Intel',-10,35)],
  'Alan': [('Intel',20,10), ('Dell',  10,50), ('Apple', 80,80), ('Dell', -10,55)],
  'Dawn': [('Apple',40,80), ('Apple', 40,85), ('Apple',-40,90)]
}
```

Define the following functions. For full credit, functions solving 3a, 3b, and 3c must have bodies **containing exactly one statement: a return statement**. Long return statements can be written over multiple lines by using the **** character: so, the requirement is one statement, not one line. Hint: write these functions using multiple statements first to get partial credit (all but 1 point), then if you can, transform that code into a single statement using combinations of **comprehensions** (sometime with multiple loops) and calls to **sorted**, to get full credit. No function should alter its argument.

- Use sequence unpacking (use _ for unneeded names); avoid indexing tuples anywhere but in a **key lambda**.
- In the notes, see how multiple statements can be rewritten as an equivalent statement comprehension.
- Sometimes use two loops in one comprehension. Complicated problems might require multiple comprehensions each with its own loop.

- For sorting problems, sometimes build the thing that must be sorted using a comprehension and call sorted on it, figuring out a key function; sometimes, sort something and then use the result in a comprehension to build what needs to be returned.

3a. (2+1 pts) The `stocks` function takes a **database** (`dict`) as an argument and returns a `set`: all stock names traded. E.g., if `db` is the **database** above, calling `stocks(db)` returns `{'Dell', 'Apple', 'Intel', 'IBM'}`.

3b. (2+1 pts) The `clients_by_volume` function takes a **database** (`dict`) as an argument and returns a `list`: client names (`str`) in decreasing order of their **volume** of trades (the sum of the number of shares they traded); if two clients have the same volume, they must appear in increasing alphabetical order. Important: selling `15` shares (appearing as `-15` shares) counts as `15` shares traded: add up the absolute values of the number of shares traded. E.g., if `db` is the **database** above, calling `clients_by_volume(db)` returns `['Alan', 'Dawn', 'Barb', 'Carl']`. `Alan` and `Dawn` each have a volume of `120` shares (they appear first in increasing alphabetical order); `Barb` and `Carl` each have a volume of `80` shares (they appear next in increasing alphabetical order).

3c. (3+1 pts) The `stocks_by_volume` function takes a **database** (`dict`) as an argument and returns a `list`: `2-tuples` of stock names (`str`) followed its **volume** (shares of that stock name traded among all clients) in decreasing order of volume; if two stocks have the same volume, they must appear in increasing alphabetical order. For example, if `db` is the **database** above, calling `stocks_by_volume(db)` returns `[('Apple',220), ('Intel',100), ('Dell',40), ('IBM',40)]`. `Dell` and `IBM` each had a volume of `40` shares (they appear last in increasing alphabetical order). Hint: you can use/call the `stocks` function you wrote in part 3a here.

4a. (3 pts) Define the `by_stock` function, which takes a **database** (`dict`) as an argument; it returns a dictionary (`dict` or `defaultdict`) associating a **stock** name with an inner dictionary (`dict` or `defaultdict`) of all the **clients** (`str`) as keys associated with their **trades** for that **stock** (a `list` of `2-tuples` of `int`). E.g., if `db` is the **database** above, calling `by_stock(db)` returns a dictionary whose contents are

```
{
 'Intel': {'Carl': [(30,40), (-10,60)], 'Barb': [(20,40), (-10,45), (-10,35)], 'Alan': [(20,10)]},
 'Dell':  {'Carl': [(20,50)], 'Alan': [(10,50), (-10, 55)]},
 'Apple': {'Carl': [(20,55)], 'Alan': [(80,80)], 'Dawn': [(40,80), (40,85), (-40,90)]},
 'IBM':   {'Barb': [(40,30)]}
}
```

4b. (3 pts) Define the `summary` function, which takes as arguments a **database** (`dict`) and a `dict` of current stock prices. It returns a dictionary (`dict` or `defaultdict`) whose keys are client names (`str`) whose associated values are `2-tuples`: the first index is a dictionary (`dict` or `defaultdict`) whose keys are stock names (`str`) and values are the number of shares (`int`) the client owns. The second index is the amount of money (`int`) the portfolio is worth (the sum of the number of shares of each stock multiplied by its current price). Assume each client starts with no shares of any stocks, and if a client owns 0 shares of a stock, the stock should not appear in the client's dictionary. Recall positive numbers are a purchase of the stock and negative numbers a sale of the stock. E.g., if `db` is the **database** above, calling `summary(db, {'IBM':65, 'Intel':60, 'Dell':55, 'Apple':70})` would return

```
{
  'Carl': ({'Intel': 20, 'Dell': 20, 'Apple': 20}, 3700),
  'Barb': ({'IBM': 40}, 2600),
  'Alan': ({'Intel': 20, 'Apple': 80}, 6800),
  'Dawn': ({'Apple': 40}, 2800)
}
```