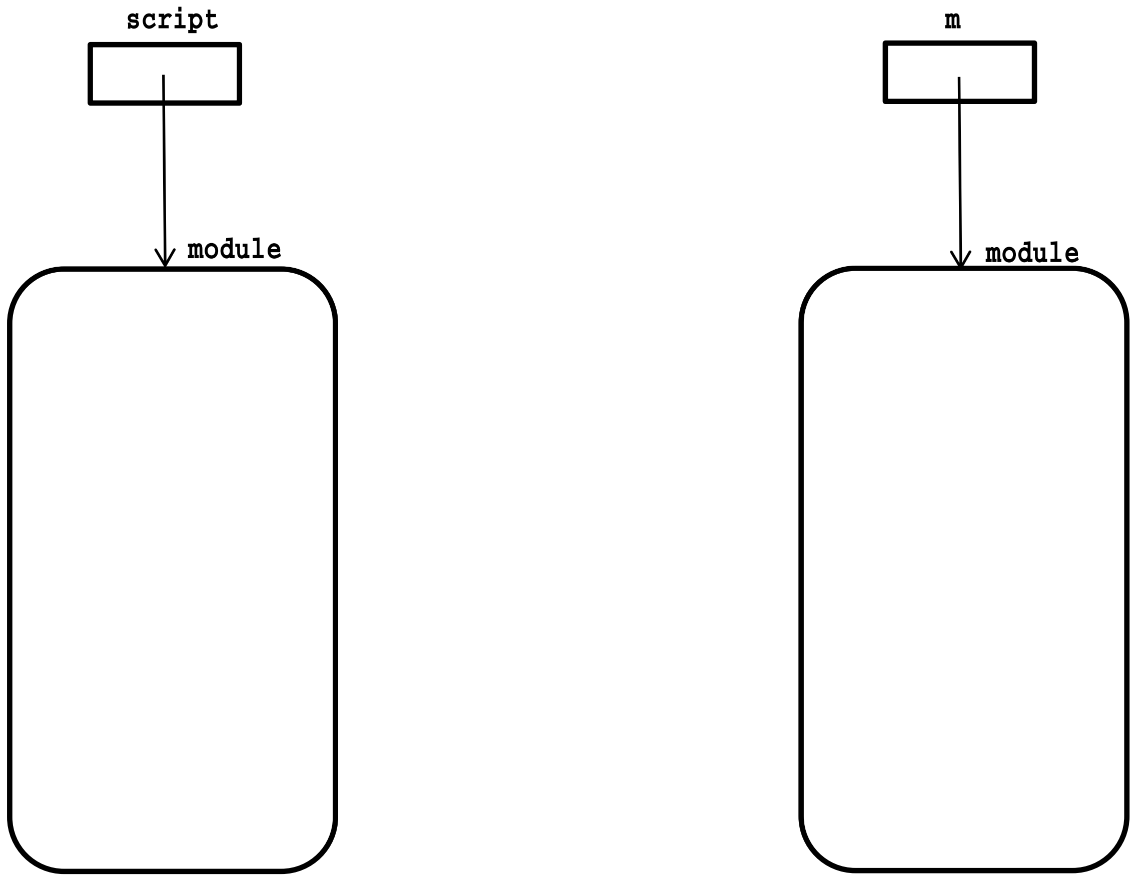
Print only this page, answer problem #1 on it, and then submit it as a .pdf file by Thursday at 11:30pm on Gradescope. You can also download this picture from Gradescope.

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. (4 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: **list**s 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).

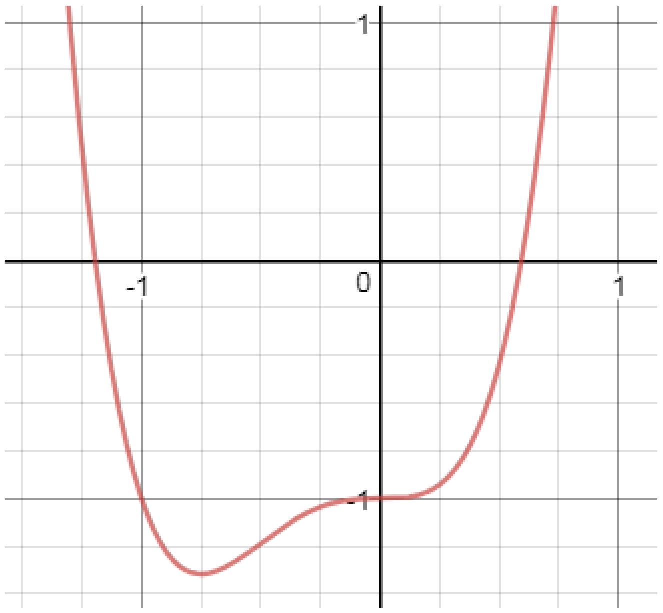
script module (execution starts here)	m module (imported by script)

`import m`
`x=['x',m.x]`
`m.x = x`
`del m.x[0]`
`m.y = m.x[0]`
`x[0] = x`
	`x = 2`



2. (4+1 pts) Define the **solve_root** function, which takes two arguments: (1) a function –call it **f**– (it is univariate, called with one numeric argument (**int** or **float**) and returns a numeric result) and (2) a numeric value –call it **error**; if **error** is not a positive value, raise an **AssertionError** exception (with an appropriate error message). The **solve_root** function returns a reference to a function defined inside it: this internal function takes two numeric arguments, call them **negf** and **posf**: **f(negf)** must be negative and **f(posf)** must be positive; if not raise an **AssertionError** exception (with an appropriate error message). When we call the returned function it returns a numeric value **x** such that **f(x)** is very close to 0.

The graph below shows a plot of $f(x) = 3x^4 + 3x^3 - 1$.



If we define

```
def f(x):
    return 3*x**4 + 3*x**3 - 1
```

Note that **f(0)** is negative and **f(1)** is positive. If we execute **rooter = solve_root(f, .0001)**, then calling **rooter(0,1)** returns a value close to **0.5936**; and **f(.5936)** is a value close to zero (**0.00003**).

Likewise note that **f(-1)** is negative and **f(-2)** is positive. Calling **rooter(-1,-2)** returns a value close to **-1.1952**; and **f(-1.1952)** is a value close to zero (**-0.00002**).

Algorithm: After verifying **negf** and **posf**, we know that some root (some **x** where **f(x) = 0**) lies between these values (because the sign of **f(x)** goes from negative at **negf** to positive at **posf**). Here is how we will narrow the location of that point. Compute the midpoint between **negf** and **posf**: rebind either **negf** or **posf** to the midpoint, depending on whether **f** evaluated at the midpoint is negative or positive (**f(negf)** must always be negative and **f(posf)** must always be positive: consider **0** positive for these purposes). Continue computing the midpoint and updating either **negf** or **posf** (shrinking the distance between them) until the distance between **negf** and **posf** is <= **error**: then return the midpoint between **negf** and **posf** as the best approximation to the root. When calling **rooter(0,1)** for the polynomial above, **negf** and **posf** are bound to the following sequence of values.

```
iteration 0: negf=0 / posf=1
iteration 1: negf=0.5 / posf=1
iteration 2: negf=0.5 / posf=0.75
iteration 3: negf=0.5 / posf=0.625
iteration 4: negf=0.5625 / posf=0.625
iteration 5: negf=0.5625 / posf=0.59375
iteration 6: negf=0.578125 / posf=0.59375
iteration 7: negf=0.5859375 / posf=0.59375
iteration 8: negf=0.58984375 / posf=0.59375
iteration 9: negf=0.591796875 / posf=0.59375
iteration 10: negf=0.5927734375 / posf=0.59375
iteration 11: negf=0.59326171875 / posf=0.59375
iteration 12: negf=0.593505859375 / posf=0.59375
iteration 13: negf=0.593505859375 / posf=0.5936279296875
iteration 14: negf=0.59356689453125 / posf=0.5936279296875
```

At this point **|negf-posf| ≤ .0001** so the function returns the midpoint of these two values.

For the fifth/last point, determine how **rooter**'s function object (see box to right of graph) can store attributes named **f** and **iterations**: **f** refers to the **callable** object bound to **solve_root**'s **f** parameter; **iterations** remembers the number of times that were necessary to call the function **f** when solving a problem: for example, after calling **rooter(0,1)** above, **rooter.iterations** evaluates to **14**. Hint: function objects can store attributes; initialize them carefully and use no globals.

Problems 3-4: Suppose that political parties wanted to store a database containing basic information about voters by their zipcode. We will represent this information in a **dictionary** whose keys specify a **zipcode** (**int**). Associated with each **zipcode** key is another **dictionary** whose keys specify a **party** (a political party, a **str**); associated with each **party** key is the number of registered **voters** of that **party** in the **zipcode** (**int** that is non-negative).

For example. a simple database might be as follows (note: the keys could appear in any order!)

```
db = {1: {'d': 15, 'i': 15,          'r': 15},
      2: {'d': 12,                   'r':  8},
      3: {'d': 10, 'i': 30, 'l': 20, 'r': 22},
      4: {'d': 30,          'l': 20, 'r': 30},
      5: {         'i': 15, 'l': 15, 'r': 15}}
```

I have printed this dictionary in an easy to read form. Python prints dictionaries on one line, and can print their key/values in any order, because dictionaries are not ordered. Assume that any distinct **str** can represent a political **party**: in the example above, **'d'** could be **democrat**, **'i'** could be **independent**, **'l'** could be **libertarian**, **'r'** could be **republican** (and there might be others).

3. (12 pts) Define the following three functions that each take a database in the form above as an argument. Write each function using combinations of **comprehensions** and calls to **sorted**. Each function is worth 4 points: 3 points for a correct solution, 4 points for a correct solution whose function body contains exactly one statement, a **return** statement (and does not call any helper functions that you write); 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.

(a) The **by_diversity** function returns a **list** of 2-tuples (whose first values are zipcodes and whose second values are different number of parties in that zipcode) whose values are decreasing by the number of parties; if two zipcodes have the same number of parties, these tuples should appear in increasing numerical order by zipcode. For example, if **db** is the database above, calling **by_diversity(db)** would return the **list** **[(3,4), (1,3), (4,3), (5,3), (2,2)]**.

(b) The **by_size** function returns a **list** of zipcodes whose values are decreasing by the number of registered voters in that zipcode; if the number of registered voters in two different zipcodes are the same, these zipcodes should appear in increasing numerical order by zipcode. For example, if **db** is the database above, calling **by_size(db)** would return the **list** **[3, 4, 1, 5, 2]**: zipcode **3** has **82** registered votes, zipcode **4** has **80**, zipcode **1** has **45**, zipcode **5** has **45**, and zipcode **2** has **20**.

(c) The **by_party** function returns a **list** of parties whose values are decreasing by the number of registered voters in that party, summed over all zipcodes; if the number of registered voters in two different parties are the same, these parties should appear in increasing alphabetical order. For example, if `db` is the database above, calling `by_party(db)` would return the **list** `['r', 'd', 'i', 'l']`: there are 90 registered **republicans**, 67 registered **democrats**, 60 registered **independents**, and 55 registered **liberals**. Hint: build a set of all the parties by using a comprehension iterating over the database.

4. (4 pts) Define the `registration_by_state` function with two arguments: (a) a database in the form of the one above and (b) a dictionary whose keys are `str` (states) each associated with a set of `int` (zipcodes in that state). This function returns an outer dictionary associating a `str` (state) with an inner dictionary that associates a **str** (party) with an `int` (the number of registered voters for that party for that state, over all zipcodes).

For example, if `db` is the database above calling `registration_by_state (db,{'CA': {1,3}, 'WA': {2,4,5}})` would return a dictionary whose contents are.

```
{'CA': {'d': 25, 'i': 45, 'r': 37, 'l': 20}),
 'WA': {'d': 42, 'r': 53, 'l': 35, 'i': 15})}
```

If a state in the dictionary has no zipcodes in the database, it should not appear in the returned dictionary.

Hint: You can use multiple `dict`, multiple `defaultdict`, or any combination so long as the contents of each dictionary is correct. I used `defaultdict`s for the outer and inner dictionaries, but then returned `dict`s for outer and inner dictionaries as the final result (building them with comprehensions); my function body is 6 lines (not a requirement). In the parameterless function argument in the outer `defaultdict`, I used a `lambda`.