When working on this quiz, recall the rules stated on the Academic Integrity Contract that you signed. You can download the **q2helper** project folder (available for Friday, on the **Weekly Schedule** link) in which to write your Regular Expressions and write/test/debug your code. Submit your completed files for **repattern1a.txt**, **repattern1b.txt**, **repattern2.txt**, **repattern4a.txt**, **repattern4b.txt**, and your **q2solution.py** module online by **Friday**, 11:30pm. I will post my solutions via the **Ed Discussion** link on Saturday morning.

For parts 1a, 1b, and 2a, use a text editor (I suggest using Eclipse's) to write and submit a **one line** file. The line should start with the **^** character and end (on the same line) with the **$** character. The contents of that **one line** should be exactly what you typed-in/tested in the online Regular Expression checker.

1a. (4 pts) Write a **regular expression** pattern that matches familial relationships involving a **m**other, **f**ather, **s**on, **d**aughter, and the words **G**reat, **g**rand, and **S**tep. To simplify the problem, we will use just the first letters (case is important) of these words with no spaces between them. The symbol **GGgs** means "great great grandson."

Legal: Should Match : **m**, **gf**, **Ggm**, **GGgf**, **Ss**, **SGgs**,

Illegal: Should Not Match: **mf**, **Gm**, **SSm**, **GSm**,

See **bscq2W20.txt** for other legal/illegal combinations. Put your answer in **repattern1.txt**.

1b. (2 pts) Write a **regular expression** pattern that matches the same strings described in part **1a**. But in addition for this pattern, ensure capture group **1** (also named **step**) is the **S** or **None**; group **2** (also named **great**) is some number of **G**s or **None**; group **3** (also named **grand**) is the **g** or **None**; group **4** (also named **basic**) is the **m**, **f**, **d**, or **s**. For example, if we execute **m = re.match(the-pattern, 'GGgf')** then **m.groups()** returns **(None, 'GG', 'g', 'f');** and **m.groupdict()** returns **{'step': None, 'great': 'GG', 'grand': 'g', 'basic': 'f'}**. There should be no other numbered capture groups. Hint **(?:...)** creates a parenthesized **regular expression** that is **not numbered** as a group. You can write one regular expression for both 1a and 1b, or you can write a simpler one for 1a (just use parentheses for grouping and ignore capture groups) and then update it for 1b by including only the necessary groups. Put your answer in **repattern1b.txt**.

2. (2 pts) Recall that parameter specifications in function definitions come in two flavors: **name-only** parameters and **default-argument** parameters, with the ability to preface at most one name-only parameter by a ***** (for this problem, we will not allow ****** written before a name-only parameter, although such parameters are legal in Python). All parameters are separated by commas. To simplify this problem, let's use **n** to abbreviate a name-only parameter and **d** to abbreviate a default-argument parameter. Write a **regular-expression** pattern that matches legal orders of these parameter lists in a function definition.

The general rule you should model is: a parameter list can (a) be empty, (b) contain one **n**, ***n**, or **d**, (c) can contain a sequence of **n**s and **d**s with at most one ***n** in the sequence (front, middle, or rear). These rules are a simplification of real Python, but are complex enough for this problem.

Legal: Should Match : **n**; **d**; ***n**; **n,d,n,d**; **n,d,*n,d,d**; ***n,d,d,n,d**; **n,n,*n**

Illegal: Should Not Match: **n,,d**; **nd**; ***n,*n**; **n,*nd**; **n,d*n,d**; ***n,d,d,*n**

Put your answer in **repattern2.txt**.

3. (7 pts) EBNF allows us to name rules and then build complex descriptions whose right-hand sides use these names. But Regular Expression (RE) patterns are not named, so they cannot contain the names of other patterns. It would be useful to have named REs and use their names in other REs. In this problem, we will represent named RE patterns by using a **dict** (whose **keys** are the names and whose **associated**

values are RE patterns that can contain names), and then repeatedly replace the names by their RE patterns, to produce complicated RE patterns that contains no names.

Define a function named **expand_re** that takes one **dict** as an argument, representing various names and their associated RE patterns; **expand_re** returns **None**, but mutates the **dict** by repeatedly replacing each name by its pattern, in all the other patterns. The names in patterns will always appear between **#**s. For example, if **p** is the **dict {'digit': r'[0-9]', 'integer': r'[+-]?#digit##digit#*'}** then after calling **expand_re(p)**, **p** is now the **dict {'integer': '[+-]?(?:[0-9])(?:[0-9])*', 'digit': '[0-9]'}**. Notice that **digit** remains the same, but each **#digit#** in **integer** has been replaced by its associated pattern and put **inside a pair of parentheses prefaced by ?:**. Hint: For every rule in the dictionary, substitute (see the **sub** function in **re**) all occurrences of its **key** (as a pattern, in the form **#**key**#**) by its associated value (always putting the value inside parentheses), in every rule in the dictionary. The order in which names are replaced by patterns is not important. Hint: I used **re.compile** for the **#**key**#** pattern (here no **^** or **$** anchors!), and my function was 4 lines long (this number is not a requirement).

> The **q2solution.py** module contains the example above and two more complicated ones (and in comments, the **dict**s that result when all the RE patterns are substituted for their names). These examples are tested in the **bscq2W21.txt** file as well.

4. (10 pts) In the review notes, examine the descriptions of **name-only** and **default-argument** parameters as well as **positional** and **named** arguments. You will write a function that takes a correctly written parameter list (from a function definition) and a correctly written argument list (from a function call) and implement the algorithm that matches the parameter/argument matching rules of Python. This function returns all the bindings for the parameters or raises an **AssertionError** if any of the binding rules are violated (e,g.: parameter fails to bind to argument or binds to more than one). To aid in this algorithm, you will first write two regular expressions for helping to decode a parameter/argument. This is a hard problem; I do not expect everyone to finish all the parts; the last parts are worth less than the early parts.

4a. (3 pts) Write a **regular expression** pattern that matches a single **argument**: an optional name and equal sign followed by an integer (for simplicity, we will assume here that all arguments are integers). Python names start with an alphabetic or underscore character, and continue with any alphabetic, numeric, or underscore characters. Integers have an optional sign followed by some non-zero number of digits. Allow no spaces in the text. Matches should have two named groups: **name** and **value**. For example, if we execute
m = re.match(the-pattern,'x=-2') then **m.groupdict()** returns **{'name': 'x', 'value': '-2'}**.

4b. (1 pts) Write a **regular expression** pattern that matches a single **parameter**: an optional ***** followed by a name (assume no spaces are between them) optionally followed by an equal sign and an integer (for simplicity, we will assume here that all default arguments are integers). Python names start with an alphabetic or underscore character, and continue with any alphabetic, numeric, or underscore characters. Allow no spaces in the text. Matches should have three named groups: **star** (might be **'*'** or **None**), **name**, and **value**. For example, if we execute
m = re.match(the-pattern,'x=-2') then **m.groupdict()** returns **{'star': None, 'name': 'x', 'value': '-2'}**. This pattern is actually very similar to the one you wrote in part 4a, which is why it is worth only 1 point.

4c. (6 pts) Now write the **match_param_args** function to take two strings as arguments, the first representing a legal list of parameters (from a function definition) and the second representing a legal list of arguments (from a function call). This function returns a **dict** with all the bindings for the parameters or raises an **AssertionError** if any of the binding rules are violated. **I strongly suggest that you write (conditional) print statements in this function to supply a trace of your function so that you can see how it is processing the arguments and better debug it. See my example at the end of this problem.** You can leave the tracing code in, delete it, or just comment it out when you submit your module for grading. Note that as the processing gets more complicated, the points go down, because I expect fewer students to correctly write that code. You don't have to follow my hints, but I expect that you will find them useful.

Hint 1: Split the parameter string and then create a **`list`** of match objects, with each parameter matched against the above-specified regular expression for a parameter. Do the same thing with the argument string. Also, build a set of all the parameter names. Build and ultimately return a dictionary of all the bindings (see below how to create each binding). Remember to convert all strings representing integers into actual integers.

Hint 2: As you solve each of these parts, archive your code so if you mess it up in the next part, you can restore your working code for submitting the previous part.

4c1. (2 pts) Correctly bind the next name-only or default-value parameters to positional arguments. Hint 3: It would be reasonable to think of using the **`zip`** function to process the parameter and argument lists from Hint 1. But to write code that can be extended to cover 4c2-4c4, instead write a **`for`** loop that processes each parameter; for processing arguments, keep an index (I called it **`ai`** for arg index) of what the current argument to process in the **`list`** of match objects (see Hint 1), incrementing it when each argument in the **`list`** is processed. **Important**: Ensure that you never try to process an argument beyond the last one in the match list.

Before returning the dictionary, ensure that (a) every positional argument is bound to some parameter (when the **`for`** loop that process each parameter terminates, there are no arguments left to process) and (b) every parameter name (see Hint 1, which specifies creating a set of them) is bound to some argument.

4c2. (2 pts) Detect a name-only parameter (prefaced by a **`*`**; there can be only one of these in a legal parameter list) and correctly handle it. Hint 4: When the looped-over parameter includes a *, process it by looping though all remaining argument indexes (there might be none!) that have positional (not named) arguments, binding it to a tuple of these values.

4c3. (1 pt) Detect and correctly handle named arguments (which in a legal list of arguments all appear after the positional arguments). Hint 5: Regardless of the parameter, when the next argument has a name, loop though all remaining argument indexes, binding each name to its value; but for each, test that the name is a parameter name and that it is not already bound to a value: if either test fails, immediately raise an **`AssertionError`**.

4c4. (1 pt) First, perform three special checks BEFORE starting the parameter/argument binding process. Check the list of parameter/argument match objects to ensure that

- There are not two parameters that are starred.
- A starred parameter does not have a default value.
- No positional argument follows a named-argument.

Raise an **`AssertionError`** if any of these conditions are violated.

Finally, detect and correctly handle default-value parameters. Hint 6: Do this only if there are no more arguments to process. For each remaining parameter (iterated over by the **`for`** loop), if it has a default value, bind its name to its default value, but only if the name **has not already been bound** (skipping the default value if the name has already been bound).

My Tracing Results (for part 4c2; edited for clarity: not required to trace but useful for debugging)

```
Binding parameter(s) a,b,*args to argument(s) 1,2,3,4
Binding param matches = [{'star': None, 'name': 'a', 'value': None},
                         {'star': None, 'name': 'b', 'value': None},
                         {'star': '*', 'name': 'args', 'value': None}]
  to arg matches      = [{'name': None, 'value': '1'},
                         {'name': None, 'value': '2'},
                         {'name': None, 'value': '3'},
                         {'name': None, 'value': '4'}]

In loop, handling parameter {'star': None, 'name': 'a', 'value': None} with ai = 0 and len(args) = 4
```

```
Binding next named parameter a to next positional argument
  bound to 1

In loop, handling parameter {'star': None, 'name': 'b', 'value': None} with ai = 1 and len(args) = 4
Binding next named parameter b to next positional argument
  bound to 2

In loop, handling parameter {'star': '*', 'name': 'args', 'value': None} with ai = 2 and len(args) = 4
Binding *named parameter args to remaining positional arguments
  bound to (3, 4)

Result to return = {'a': 1, 'b': 2, 'args': (3, 4)}
```

My Tracing Results (for part 4c4; edited for clarity: not required to trace but useful for debugging)

```
Binding parameter(s) a,b,*args,c=5,d=6 to argument(s) 1,2,3,4,d=5
Binding param matches = [{'star': None, 'name': 'a', 'value': None},
                         {'star': None, 'name': 'b', 'value': None},
                         {'star': '*', 'name': 'args', 'value': None},
                         {'star': None, 'name': 'c', 'value': '5'},
                         {'star': None, 'name': 'd', 'value': '6'}]
  to arg matches      = [{'name': None, 'value': '1'},
                         {'name': None, 'value': '2'},
                         {'name': None, 'value': '3'},
                         {'name': None, 'value': '4'},
                         {'name': 'd', 'value': '5'}]

In loop, handling parameter {'star': None, 'name': 'a', 'value': None} with ai = 0 and len(args) = 5
Binding next named parameter a to next positional argument
  bound to 1

In loop, handling parameter {'star': None, 'name': 'b', 'value': None} with ai = 1 and len(args) = 5
Binding next named parameter b to next positional argument
  bound to 2

In loop, handling parameter {'star': '*', 'name': 'args', 'value': None} with ai = 2 and len(args) = 5
Binding *name-only parameter args to remaining positional arguments
  bound to (3, 4)

In loop, handling parameter {'star': None, 'name': 'c', 'value': '5'} with ai = 4 and len(args) = 5
Current and remaining arguments all named; process each when bound to name that is an unbound parameter
  Binding  d
    bound to 5
Binding this default-value parameter c to 5
```

Hint 7: I wrote a few **`continue`** statements: when executed in a loop, Python will immediately begin the next iteration at the beginning of the loop, skipping all the statements following the **`continue`**.