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.

The **`q2helper`** project folder also contains a **`bm1.txt`**, **`bm2a.txt`** and **`bm2b.txt`** files (examine them) to use for batch-matching your pattern, via the **`bm`** option in the **`retester.py`** script (included in the download). These patterns are also tested automatically in **`q2solution.py`** script and similar examples in the **`q2helper`**'s **`bscq2F20.txt`** file.

1a. (3 pts) Write a **regular expression** pattern that matches times on a 12-hour clock written in the format **hour:minute:second**. Here **hour** can be any one- or two-digit number in the range 1-12 (with no leading 0 allowed); **:minute** is optional: if present it can be any two-digit number in the range 00-59; **:second** is optional: if present, it can be any two-digit number in the range 00-59; at the end is a mandatory **am/pm** indicator. Here are a few legal/illegal examples.

Legal: Should Match : `6pm`, `6:23pm`, `6:23:15am`, `12am`, `11:03am`, `8:40:04pm`

Illegal: Should Not Match: `6`, `06pm`, `14pm`, `6::pm`, `6:60pm`, `6:111pm`, `6:4pm`, `6:04:7pm`, `6:23:15:23am`

Put your answer in **`repattern1a.txt`**.

1b. (3 pts) Write a **regular expression** pattern that matches the same strings described in part **1a**. But in addition for this pattern , ensure group **1** is the **hour**; group **2** is the **minute** (or **`None`** if **:minute** is not present); group **3** is the **second** (or **`None`** if **:second** is not present); and group **4** is **am** or **pm**. For example, if we execute **`m = re.match(the-pattern, '6:23pm')`** then **`m.groups()`** returns **`('6', '23', None, 'pm')`**. There should be no other numbered 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 (ignore groups) and then update it for 1b by including the necessary groups. Put your answer in **`repattern1b.txt`**.

2a. (4 pts) When we print computer documents, there is a common form used to specify the page numbers to print. Generally, **commas separate page specifications**, where each **page specification** is a single page number, or a contiguous range of pages. In a page specification, a **dash** or a **colon** can separate two numbers: for a **dash**, these numbers specify the first and last pages in a range to print (inclusive); for a **colon**, they specify the first page and how many subsequent pages to print (so **`10:3`** means 3 pages starting at **`10`**: **`10`**, **`11`**, and **`12`**). Finally, if either of these forms is used, we can optionally write a **slash** followed by a number (call it **`n`**), which means for the page specification, print every **`n`**th page in the range (so **`10-20/3`** means **`10`** through **`20`** , but only every 3rd page: **`10`**, **`13`**, **`16`**, and **`19`**). Write a **regular expression** that ensures group **1** is the first **page**; group **2** is a **dash** or **colon** (or **`None`** if not present); group **3** is the **number** after the **dash** or **colon** (or **`None`** if not present); group **4** is the **number** after the **slash** (or **`None`** if not present).

Write a **regular expression** pattern that describes a single **page specification**: the integers you specify here must start with a non-0 digit. Here are examples that should match/should not match a single **page specification**:

Match `3` and `5-8` and `12:3` and `5-8` and `6:4` and `10-20/3` and `10:10/3`
Not Match `03` and `5-08` and `3 4` and `3 to 8` and `4/3` and `4-:3` and `4-6:3`

Put your answer in **`repattern2a.txt`**.

2b. (8 pts) Define a function named `pages` that takes one `str` as an argument, representing a list of **page specifications separated by commas**, and a `bool` value controlling whether the pages are `unique`: printed only once; it returns a `list`, sorted in ascending order, of all the pages (`int`s) in the page specifications. This function **must use the regular expression pattern you wrote for part 2a** and extract (using the `group` method) information to create the numbers in the page specification. For example, if we called the function as `pages('5-8,10:2,3,7:10/3',unique=True)` it would return the list `[3,5,6,7,8,10,11,13,16]`; if we called `pages('5-8,10:2,3,7:10/3',unique=False)` it would return the list `[3,5,6,7,7,8,10,10,11,13,16]`.

Here are some more examples of arguments to `pages` and their meanings:

`'3'`	page `[3]`	
`'3,5-8,12:3'`	pages `[3,5,6,7,8,12,13,14]`	(3, 5 to 8, 12 and 2 more pages)
`'6-10,4-8'`	pages `[4,5,6,6,7,7,8,8,9,10]`	(pages are ordered; assume `unique` is `False`)
`'6-10/2,4-10/2'`	pages `[4,6,8,10]`	(pages are ordered; assume `unique` is `True`)

Raise an `AssertionError` exception (using Python's `assert` statement) if any page specifications fails to match the regular expression, or if any **dash** separator separates a higher first number from a lower second one: e.g., `10-8` raises/prints the exception `AssertionError: pages: in page specification 10-8, 10 > 8.` The page specification `8-8` is OK: it means only page `8`.

Hint: My function body is 15 lines of code (this number is not a requirement). After using `split` to separate the `str` argument to get a `list` of **page specifications**, use the `re.match` function (using your regular expression solution from Problem #2a) on each, then call the `group` function to extract the required information, and finally process it. The `range` class is very helpful in determining the pages in each **page specification**.

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 `bscq2F20.txt` file as well.