Exercises-Wildcards

Description

This exercise is about shell wildcards. Wildcards (which may also be called shell metacharacters, or, more accurately, *globbing characters*) are special characters used in filename patterns. When the shell encounters a wildcard pattern on a command-line, it replaces it, if possible, with a list of paths that match the wildcard pattern. There are three things you should learn from this exercise:

- the globbing functions in the C library expand wildcards. The use of these functions is limited to a few programs - shells and the **`find`** command in particular. Almost no other commands will expand wildcards, For more information, check out **`glob(7)`** (use the command **`man 7 glob`**)
- the wildcard operators [globbing characters], what they mean and how to use them.
- ways to suppress the interpretation of wildcard operators by quoting them.

The Wildcard Operators

The following sequences of characters are interpreted as wildcards by the shell. Enclosing characters in any type of quotes (single- or double-) hides them from the shell.

Operator	*Meaning*	*Example*	*matches objects whose names are*
`*`	anything, including nothing	**`A*`**	capital **`A`** followed by any number (including zero) of any combination of characters
?	a single character	**`???`**	exactly three characters long
a string of characters enclosed in **`[ ]`**	a single character that is one of the characters enclosed in the brackets.	**`[abc]`**	a single character that is either **`a`** or **`b`** or **`c`**
two characters separated by a dash enclosed in **`[ ]`**	one character whose value is greater than or equal to the first character and less than or equal to the second character	**`[a-z]`**	a single character whose value is greater than or equal to '**`a`**' and less than or equal to '**`z`**' . (i.e., a lowercase character) Note: uppercase characters have consecutive values, as do lowercase characters, as do digits. However, **`[a-Z]`** does not do what you want. Use **`[a-zA-Z]`** instead.
a character class enclosed in [ ]	a single character that is a member of that character class	**[[:alpha:]]**	the character class is alpha. It is indicated by [:alpha:] The inner brackets indicate the character class. The outer brackets indicate the character set (that it is a single character) Both sets of brackets are needed. Classes include: alpha, upper, lower, digit, alnum, blank, space(whitespace), punct, print, ascii, cntrl, ascii
if the first character inside of the [ ] is ! or ^	! negates the character set. a single character that is not a member of the set.	**[^[:lower:]]**	a single character that is not a member of the character class **`[:lower:]`** Thus, any character that is not lowercase.

Note: ***`*`*** *and* ***`?`*** *will not match a leading* ***`.`*** *in a name (indicating a hidden file)*

Note2*:* ***`!`*** *is the traditional negation operator in character sets.* ***`^`*** *is also recognized by bash for this function to make it consistent with regular expressions, which we will cover later. We will use them interchangeably, but* ***`^`*** *is more important to know.*

bash also offers brace expansion, which uses braces to enclose a comma-separated list of alternatives. Brace expansion is not a wildcard, as this sequence is expanded whether or not each possibility exists, rather it is a pattern generator.

`{xyz,abc}` *expands to the list* **`xyz abc`**

`abc{xyz,abc}` *expands to the list* **`abcxyz abcabc`**

`*.{doc,html}` *expands to* **`*.doc *.html`** *which are then expanded as wildcards*

Exercises

First, interpret the following wildcards:

1. **`[[:alpha:]]*`**
2. **`[^012]?`**
3. **`z*a?`**
4. **`[0-9!a-z]*`**
5. **`*a*z`**
6. **`[[:alpha:]]`**
7. **`*[^[:digit:][:punct:]]*`**

Now, connect to the directory **`wildcards`** beneath the class public directory on hills to do the remainder of the exercises.

8. Take the wildcard pattern from #2 and issue the following commands:
   - precede the wildcard with **`ls`**. It looks like **`ls`** understands wildcards.
   - precede the wildcard with **`echo`**. This shows that the shell expands wildcards.
   - precede the wildcard with **`echo ls`**. This shows that the shell expands the wildcard before it passes it to **`ls`**.
   - precede the wildcard with **`ls`** and put single quotes around the wildcard. This suppresses the shell's wildcard expansion and shows you that **`ls`** doesn't know anything about wildcards.
9. Take the wildcard pattern from #3 and issue the following commands:
   - precede the wildcard with **`ls`**. Do the paths output seem to make sense?
   - precede the wildcard with **`echo`**. Can you reconcile the output with that using **`ls`**?
   - precede the wildcard with **`echo ls`**. This shows the command as it is executed.
   - Can you put the above together and come up with an explanation? Test your theory by using the command **`ls -dF`** followed by the wildcard. What do these options do?

From this point on we will be practicing using wildcards with the **`ls`** command. As you just learned, it will be less confusing if you use the options **`-dF`** when using **`ls`** in the remainder of this exercise.

Write commands to list objects in the current directory whose names

10. are three characters long
11. start with a letter, either upper- or lower-case
12. contain a digit anywhere in the name
13. end with a character that is not a digit
14. contain a blank
15. start with a **`.`**
16. start with a **`.`** and are a total of three characters long
17. contain at least one character that is neither a letter nor a digit
18. contains a left or right square bracket **`[`** or **`]`**

19. contain at least one instance of each of **a** and **q** in the name, in either order (you need two wildcards to do this!)

Next, write commands to list objects whose paths are

20. in a subdirectory of the current directory. The name of the object can be anything.
21. in a subdirectory of the current directory whose name starts with a digit. The name of the object must be five characters long and start with a letter.

Last, to show you that this really has nothing at all to do with **ls**:

22. output the contents of all objects in the current directory whose name starts with **a**

Answers

The set of wildcards match anything in the current directory whose name

1. begins with a lower-case letter
2. is a total of two characters long. The first character may not be **0 1** or **2**
3. begins with a **z** and ends with an **a** followed by any single character
4. starts with a digit, lowercase letter or the **!** character (the **!** must come at the beginning of the character set to negate it)
5. contains an **a** and ends with a **z**
6. is a single alphabetic character
7. contains a character that is neither a digit nor a punctuation character
8. Here's the commands that you should have used and some interpretation:
   - **ls [^012]?** - **ls** seems to understand wildcards (but it doesn't)
   - **echo [^012]?** - **echo** outputs the list of names that match the pattern!
   - **echo ls [^012]?** - **echo** outputs what an ls command looks like after the shell has expanded the wildcard. Are there any wildcards left for **ls** to worry about?
9. Here are the commands that you should have used and some interpretation:
   - **ls z*a?** - **ls** outputs names, but they don't appear to match the pattern!
   - **echo z*a?** - **echo** outputs the list of names that match the pattern!
   - **echo ls z*a?** - **echo** outputs what an ls command looks like after the shell has expanded the wildcard. If this is the command as it is executed, why does **ls** put out strange names?
   - adding the options **-dF** shows that the object that matches the wildcard is a directory. Remember that **ls** outputs the contents of the directory by default. The **-d** suppresses this behavior and the **-F** puts the **/** at the end to indicate the directory.

```
10. ls -dF ???
11. ls -dF [a-zA-Z]*    # or [[:alpha:]]*
12. ls -dF *[0-9]*      # or *[[:digit:]]*
13. ls -dF *[^0-9]      # or *[^[:digit:]]
14. ls -dF *' '*        # Note you must use quotes to 'hide' the blank OR
    ls -dF *\ *         # use a backslash before the blank. You can also use the
    ls -dF *[[:blank:]]*  # :blank: class. Careful, *[' ']* or *[\ ]* do not work!
15. ls -dF .*
16. ls -dF .??          # Why did this file not appear in the output of #8?
17. ls -dF *[^a-zA-Z0-9]*  # can you do this using character classes?
18. ls -dF *[][]*       # a character set containing the characters [ and ]
```

19. `ls -dF *a*q* *q*a*`
20. `ls -dF */*`
21. `ls -dF [0-9]*/[a-zA-Z]????   # or [[:digit:]]*/[[:alpha:]]????`
22. `cat a*`