ISTA 350 Memory Diagram Worksheet Name:

Regular expressions are how computer people do pattern matching. Python has the re module for regular expressions. The following code will get a list of all of the filenames in the working directory:

```
import os
fnames = os.listdir()
```

Write a function called `get_scripts` that takes one string argument representing a file extension with a default value of `"py"` and returns a list of all of the files of that type in the working directory.

Write a function called `get_domains` that takes a filename and returns all of the domain names in the file. The file object `read` method will return the entire contents of the file as a single string. For our purposes, the definition of a domain name is a sequence of one or more labels separated by dots followed by a dot and 2 or 3 letters (the top-level domain). Each label has the format: at least one letter followed by zero or more letters/digits/hyphens. If there is more than one character, the last character must a letter/digit.

Regular Expression Cheat Sheet

Guidelines:

- Use raw strings for your regexes, e.g. `r"\d+"`, to avoid backslash problems
- Use this syntax to store a regular expression for repeated use: `p = re.compile(<re>)`
- `p.match(str)` matches the beginning of the string
- `p.search(str)` finds the leftmost match
- `p.findall(str)` returns a list of all matching substrings

Special Characters

- `\` escape special characters
- `.` matches any character
- `^` matches beginning of string
- `$` matches end of string
- `[5b-d]` matches any chars '5', 'b', 'c' or 'd'
- `[^a-c6]` matches any char except 'a', 'b', 'c' or '6'
- `R|S` matches either regex R or regex S
- `()` creates a capture group and indicates precedence

Quantifiers

- `*` 0 or more (append ? for non-greedy)
- `+` 1 or more (append ? for non-greedy)
- `?` 0 or 1 (append ? for non-greedy)
- `{m}` exactly mm occurrences
- `{m, n}` from m to n. m defaults to 0, n to infinity
- `{m, n}?` from m to n, as few as possible

Special sequences

- `\A` start of string
- `\b` matches empty string at word boundary (between `\w` and `\W`)
- `\B` matches empty string not at word boundary
- `\d` digit
- `\D` non-digit
- `\s` whitespace: `[ \t\n\r\f\v]`
- `\S` non-whitespace
- `\w` alphanumeric: `[0-9a-zA-Z_]`
- `\W` non-alphanumeric
- `\Z` end of string
- `\g<id>` matches a previously defined group

Extensions

- `(?iLmsux)` Matches empty string, sets re.X flags
- `(?:...)` Non-capturing version of regular parentheses
- `(?P<name>...)` Creates a named capturing group.
- `(?P=name)` Matches whatever matched previously named group
- `(?#...)` A comment; ignored.
- `(?=...)` Lookahead assertion: Matches without consuming
- `(?!...)` Negative lookahead assertion
- `(?<=...)` Lookbehind assertion: Matches if preceded
- `(?<!...)` Negative lookbehind assertion
- `(?(id)yes|no)` Match 'yes' if group 'id' matched, else 'no'

`dot_product`: recall that the dot product of two vectors is the sum of the element-by-element product of the vectors. This function takes two numpy arrays (vectors) of equal length and returns their dot product. For example (pretend the lists are actually arrays):

```
dot_product([0,  4, 7], [45, 1, 3]) == 0 * 45 + 4 * 1 + 7 * 3 == 25
```

Draw a memory diagram of the function's namespace before being called as above, immediately after being called, just before exiting, and just after exiting. Rewrite the function to include a default argument of a one-element array containing 0 for the second parameter. Redo the drawings.