F28PL - Programming Languages

Question H

(Python lists, polymorphism, recursion, and reduce)

Tommy Lamb

H00217505

Where answers comprise of more than one function, each is listed and commented on separately, in an order approximating lowest to highest hierarchically. That is independent functions first and then in increasing number of dependencies.

Note:

Certain functions utilise other functions defined in earlier questions. Any such prerequisites are noted in blue for each applicable question.

1. Using iteration (a for-next or a while loop or similar), write a program `mult1` that if given a list of integers or reals will return their product (the result of multiplying all the numbers together).
   We do not care about behaviour if `mult1` is not given a list of integers or reals; likewise for `mult2` and `mult3` below.
   `mult1` should return 1 for the empty list, because 1 is the **multiplicative unit**. Thus.

   - `mult1([])` calculates `1`, and
   - `mult1([1,2,3,10])` calculates `60`.

```
def mult1(numberlist):
    accumulator = 1
    for x in numberlist:
        accumulator *= x
       return accumulator
```

   - This program initialises an accumulator variable (though non-recursive) to 1, before iterating through every element in the argument list, storing and multiplying the accumulator by each element.

2. Using recursion, write a program `mult2` that if given a list of integers or reals will return their product (warning: this question is very slightly harder than it seems; make sure to test your answer).

```
def mult2(numberlist):
    if len(numberlist) == 0:
        return 1
    else:
          return numberlist.pop() * (mult2(numberlist))
```

- The program first uses an *if* statement to evaluate the base case (list is empty), and returns 1. In the sole step case the last item from the list is removed, and multiplied by the results of successive program calls. The reason the last element is removed, rather than the first, is simply down to the behaviour of the *pop* method without a specified index.

3. Using range or otherwise, write a one-line piece of code that breaks `mult2` (i.e. make it crash when it "should" return a value). Why did this happen?

```
mult2(list(range(10 ** 100)))
```

- This is the overkill answer, and while it doesn't break `mult2` per se, it is included purely for interest. What this code actually does is break Python's implementation of lists, by exceeding the maximum permissible size. Despite integers being infinite precision in Python, the resulting error is an OverflowError - not a memory error as one may expect. The error message, though cryptic, hints that this overflow comes from the underlying C code within CPython, rather than the Python langauge itself.

```
mult2(list(range(1001)))
```

- Conversely this code is absolutely Python's fault. For various debateable reasons Python doesn't truly implement recursion, and due to the resource intensive nature of the imitation recursion it imposes an arbitrary limit of 1000 on recursion depth. Attempting to execute `mult2` on this range results in more than 1000 recursive calls, and so the recursion depth exception is raised by Python.

4. Using Python lambda and reduce (see specifically the example on line 2 of this link), write a program mult3 that if given a list of integers or reals, returns their product.

```
from functools import reduce
def mult3(numberlist):
        return reduce(lambda x, y: x * y, numberlist, 1)
```

- This program imports the `reduce` function from the `functools` standard library, before applying it to the list. The function cumulatively applies a function `f` to each element of a list `l`, such that the result is `(f ...(f (f` l_0 l_1`)` l_2`)` l_n`)`. In this case an initial value 1 is also used. In effect this is placed at the head of the list before applying the function, which provides for the case that the list is empty. The function used is a simple higher-order function multiplying two elements.

5. <u>What do `mult1`, `mult2`, and `mult3` return if applied to `[1.0,2,3.0,10]`? What would corresponding programs do if they were written in ML and applied to such an input? What does this tell us about ML vs Python typing?</u>

   - All functions return `60.0`. The equivalent functions and input in ML would simply result in a type error and no execution would be performed. The input itself would not be allowed in ML, never mind having a function equivalent to those here process it.

     It demonstrates that Python is very much a weakly typed language, whilst ML is known to be very strictly typed. Python has a habit of attempting to return an answer - any answer - where possible, which leads to unexpected behaviour when types are mixed. This requires good documentation accompany Python programs in order that others may understand the program's idiosyncrasies. ML on the other hand is very much strongly typed, and will only return the 'right' result. If the type declarations don't match up, an exception is thrown and nothing is done. This serves to make functions more predictable, often simply because the type system limits the amount of possible edge cases that the function has to deal with.

6. <u>Order `mult1`, `mult2`, and `mult3` from slowest to fastest and explain your order. There is no single “right” answer here, and this is a theoretical question, not an empirical one; you are welcome to measure performance but what interests me is your understanding.</u>

   - `mult2, mult1, mult3`

     With the stack-based overheads associated with non tail-recursive calls, `mult2` should prove to be the worst performer in terms in speed. In addition to this is Python's complete inability to properly handle recursion, and the limited nature of the interpreter/compiler optimisation possible. The iterative `mult1` should demonstrate marginally greater speed as it lacks much of recursion's overheads and is more able to be optimised by the translator, though such optimisation is still lacklustre. By comparison `mult3` should prove highly optimisable as it only uses a single function, and one of a standard library. Knowing exactly how the function works theoretically allows the translator to extract as much performance as possible. It should demonstrate appreciably superior execution speed, given sufficiently sized input.

7. An issue with the programs above is that they only work for numbers (integers or reals), however, there is no reason in principle they should not also work with strings or lists (by concatenation). Note that in Python you can check whether a variable `x` has type `str` (for example) with `if type(x) is str: print("x is a string!")`, and similarly for lists `if type(x) is list: print("x is a list!")`.
Write a program `multpoly` that outputs the product of a list of numbers, and the concatenation (i.e. chaining) of a list of strings or lists. So

- `multpoly([1,2,3,10])` calculates `60`, and
- `multpoly(["1","2","3","10"])` calculates `"12310"`, and
- `multpoly([[1],[2,3],[10]])` calculates `[1,2,3,10]`

```
def multpoly(polylist):
    accumulator = polylist[0]
    for entry in polylist[1:]:
        if type(entry) is int:
            accumulator *= entry
        else:
            accumulator += entry
    return accumulator
```

- Taking the first list element to initialise the accumulator, this function iterates through the list, checking the type of each element and performing the appropriate action. If the entry is an integer the accumulated value is multiplied by the integer value, otherwise addition of the element and accumulator is carried out. The *else* clause covers both the string and list cases specified in the question, as the addition operator is translated as concatenation for both types.

  As a result of checking each element individually, some interesting results can be achieved by mixing data types. Provided the first element is a list or string, then any integer values in the list multiply the number of instances of the accumulated string at that point. However if the first element is an integer, then an exception will be thrown. This is because accumulator would be initialised to type `int`, and `int += str` is not possible.

8. Using `if type(x) is list` or otherwise, write a recursive program `flatten` that inputs an arbitrarily nested list structure and outputs a "flattened" list consisting of the list of all the data in the list, but with any nested list structure removed. So for instance,

- `flatten([])` returns `[]`, and
- `flatten([["hi"],5])` returns `["hi",5]`, and
- `flatten([[[],[["hello"],[" "]],[1],[[["world"]],[]]],"!"])` returns `["hello"," ",1,"world!"]`.

```
def flatten(nestedlist):
    if type(nestedlist) is list:
        return [elements for lists in nestedlist for elements in flatten(lists)]
    else:
        return [nestedlist]  # nestedlist isn't actually a list in this case.
```

- It is necessary here to cite this Stack Overflow answer in conjunction with the answer given. Like Pandora's box, an innocent Google search for "flattening arbitrarily nested lists" and an impulsive click led to this webpage, and once the answer had been seen it could not be unseen. An exact Python implementation was not sought - rather a general idea, perhaps an algorithm in some other language or pseudocode. In order to fully comprehend this answer, and to demonstrate this understanding here, the list comprehension was expanded into the equivalent iterative code seen below (still featuring recursion):

```
def flatten(nestedlist):
    flattenedlist = []
    if type(nestedlist) is list:
        for nestedentry in nestedlist:
            for entry in flatten(nestedentry):
                flattenedlist.append(entry)
        return flattenedlist
    else:
          return [nestedlist]
```

- In this form the exact execution of the list comprehension can be seen clearly and better explained. Initialising an empty accumulator, if the function argument is a list then flatten it, otherwise return the argument (the base case). In flattening a given list every element in that list is iterated over, and then the result of flattening each of those individually (using a recursive call) is concatenated to the accumulator. This means that no matter how deeply nested an element is recursive calls will be made in sufficient number to reach it, whereby it will be returned to the preceding `flatten` call. That functional call represents the function attempting to flatten the list containing that element. Eventually at every depth level in the nested list the function will have returned to it a list comprising only of elements which represents a flattened sub-list. This flattened sub-list is then concatenated with all the other flattened sub-lists at this depth and point in the list, and passed up to the next level where the process is repeated. In this behaviour it vaguely resembles a depth-first search from Graph Theory, as opposed to a breadth-first approach.