F28PL - Programming Languages

Question 1 (Python lists, sets, and dictionaries)

Tommy Lamb

H00217505

1. State what the following Python programs calculate, and explain why:
   - `{ 0:"zero", 1:"one", 2:"two" }[1]`
   - `range(1000)[0]`
   - `for x in range(1,1000): print(list(enumerate(range(x)))[0][0])`
   - `"hello world"[:-2]`

```
>>> { 0:"zero", 1:"one", 2:"two" }[1]
'one'
```

   - The square bracketed number is used in Python to access particular elements of a tuple, list, or dictionary. In this case a single element is accessed represented by the *key* 1 of type `int`. Unlike with a list or tuple, the value in the square brackets represents the key value of an element, rather than numerical index within the structure. As proof of this, changing '1:"one"' to '3:"one"' will result in an error for the same input. Also note that slicing cannot be performed on a dictionary, at least not using the square bracket method.

```
>>> range(1000)[0]
0
```

   - The range function in Python is a lazy data structure which represents a list of integers of a particular sequence, and can be accessed in the same manner. With a single argument `n`, `range(n)` represents a list of integers from 0 (inclusive) to `n` (exclusive) in increments of 1. It is important to note that while the elements of the range can be accessed like that of a list, range is not itself a list. Rather `range` is a computation that when executed results in a list, and even then only constructs as much of the list as absolutely required. This can be proven by executing `list(range(10000000))` and `range(10000000)[100]`. The former code forces the entire list to be calculated and takes a noticeable amount of time to do so, whereas the latter code fragment returns virtually instantly, showing that it is not computing the entire list before returning the indexed value.

```
>>> for x in range(1,1000): print(list(enumerate(range(x)))[0][0])
```
999 zeros, each printed on a separate line.

- The reason for this output is twofold. Firstly, `enumerate` constructs a (lazy representation of a) list of tuples from an iterable object where the first value in the tuple is the integer index of the second tuple value in the iterable object - zero based. That is to say for a list `x = ['a','b','c']`, `enumerate(x)` would retum `[(0,'a'),(1,'b'),(2,'c')]`. Secondly is the double use of square bracket indexing. Where multiple square brackets are used, each set acts upon the result of the immediately preceding bracket set such that for `list[1][2]`, `[2]` acts upon the return value of `list[1]`. This can be mentally bracketed as `(list[1])[2]`.

  In this context the first element of the list resulting from `list(enumerate...)` is always `(0,0)`, regardless of the value of `x`. It is this first element which is returned by the first instance of `[0]`. The subsequent `[0]` then acts on this returned tuple, whose zeroeth element is always `0`. Thus regardless of the value of x and the rest of the list, `0` is returned for every value of x in the range from `1` to `1000`. Due to the inclusive/exclusive behaviour of `range`, this results in 999 zeros being printed to the screen.

```
>>> "hello world"[:-2]
'hello wor'
```

- Just as with the previous data structures strings also implement the square bracket functions. Unlike the previous examples this one demonstrates the syntax - namely a colon - for a splice , a method returning a sub-section of the structure upon which it was called. The numerical values before and after the colon indicate the numerical indexes between which is the substructure to be returned, the first being inclusive and the second exclusive. Where no value is specified the method assumes either the start or end of the superstructure to be the start or end point of the substructure as appropriate. A negative index value, not unique to splicing, indicates an index relative to the end of the list, rather than the start as normal; this 'reverse' indexing is not 0 based, and the last element is indexed as `-1`, not `-0`.

  In this usage the result is the substring of "`hello world`" which starts with the same index (`0`) as the string and ends with the antepenultimate (last but 2) index. The reason the `l` character is omitted is the second index value, `-2`, is exclusive rather than inclusive.

2. Explain the difference in behaviour between the following two code snippets:

```
x=[""] # Snippet 1
for i in x: x.append("")
x=[""] # Snippet 2
for i in x[:]: x.append("")
```

   - Both snippets of code iterate over the list x and both modify the contents of x, however the first will never end (until it crashes) whilst the second acts as was likely intended. This is because the first snippet iterates over x and adds an element to x at each stage of the iteration, such that x continually increases in size whilst it is being iterated over. The second snippet on the other hand technically iterates over a copy of x created by `[:]`. This copy is not modified, and more importantly, the modifications to x are also not reflected in the copy, such that the iteration does not iterate over the new entries to x as with the first snippet.

3. Explain the behaviour of the following code snippet:

```
x=[[]] ; x[0].extend(x)
while True:
   print(x)
   x = x[0]
```

   - This code creates a nested list of infinite depth, and subsequently prints to the screen every level of the infinitely nested list. The reason for this is much like with Question 2 previously, in that the list being iterated over is also being modified at the same time. The `extend()` function takes a list and appends all the elements within that list to the list from which it was called. In this case it adds the nested list of x to the first element in x, or attempts to. Every time it adds an element, the element it is trying to add changes as does the element it is adding to. The curious thing about this is that it does not trigger a never-ending loop, nor does it crash or throw an error. In fact Python executes it just like any other instruction and returns a list displayed only as `[[[...]]]`. It suggests that Python can implement nested lists to an infinite depth (within memory constraints), and as shown by the infinite while loop, perform at least some operations on them.

4. The following Python program returns an error.

```
{[0]:0}[[0]]
```

- Explain what the error is and why it arises.
- Suggest how to fix it.

  - The error results from the `list` type not being hashable, which is a requirement for dictionary keys. Lists are mutable, and at some point in their lives their hash code may change as a result. A key's hash code is used internally by Python in implementing a dictionary, and it requires an unchanging hash code for the life of the key - Python's definition of "hashable". All built in immutable types are hashable, and by default all objects of a user-defined class are hashable as well.

  - The obvious fix is to remove the superfluous square brackets from the code, such that it reads: `{0:0}[0]`. Since integers are an immutable type, and thus are hashable, they can be used as the key for the dictionary. Without more background information, an even more pithy correction would be `[0][0]`, acceptable because the key of `0` in the original dictionary simply corresponds to the value's numerical index and there is no suggestion that a list will not suffice.

5. Using list comprehensions and lambda, write a program that if given a number `n`, calculates the first `n` even numbers (starting from `0`).

```
def evennumbers(n):
        return [number for number in range(0,2*n,2)]
```

  - This program simply returns the list comprehension result of counting from `0` to `n` in steps of two, which results in `n` number of even numbers. `number` represents an accumulator list variable to which each value in the range object is concatenated.

6. Using list comprehensions and lambda, write a program that if given a list of numbers `l`, will return the sublist of those numbers in `l` that are divisible by 3.

```
def multiples(numberlist):
    return [sublist for sublist in numberlist if sublist % 3 == 0]
```

  - This program in essence uses a list comprehension to iterate through the provided list checking if each element is divisible by 3, and returning the list of all elements for which that holds true. One may rightly argue that this program makes no use of lambda as requested by the question, but the counter argument is made that to do so would decrease code readability with little to no advantage.

7. The zip function inputs two lists (which you may assume have the same length) and outputs the list of pairs of the lists 'zipped' together. For instance, `zip([1,2,3],["one","two","three"])` returns `[(1,"one"),(2,"two"),(3,"three")]`. Implement zip yourself, from first principles, using list comprehension (so **not** using a for-next loop and the append method; too easy).

```
def selfzip(listx, listy):
        return [(listx[a],listy[a]) for a in range(max(len(listx),len(listy)))]
```

   - Put simply this list comprehension iterates over the range from `0` to the length of the longest array, and accumulates a tuple using the range values to access the list elements in order by index. Though it takes the length of the longest list for iteration, it will throw an error should lists of differing lengths be passed as argument. This is irrelevant given the assumption permitted by the question that the lists are of the same length.

8. The following Python programs calculates the first ten squares and the first ten pentagonal numbers:
   `squares = [x*x for x in range(10)]`
   `pentagonals = [x*(3*x-1)//2 for x in range(10)]`
   (For this question, `0` is a square and a pentagonal number.)
   Based on those examples, and using list comprehensions if possible, write a Python program that calculates a list of numbers that are both square and pentagonal (up to some computational bound, i.e. the program is allowed to stop after a while). Test your program by calculating the first four such numbers.

```
def squarepentagonals(length):
        return [x for x in [p*(3*p-1)//2 for p in range(length)] if x in [s*s for s in
    range(length)]]
```

   - This program uses the two provided list comprehensions nested within a third to compute the square pentagonal numbers. Expanded upon, the overall comprehension can be read as: for each value in the list of pentagonal numbers, if that value appears in the list of square numbers, then append that value to the accumulator `x`. It is unclear how often each of the nested comprehensions are executed; though they need be and are likely executed only once, there is an outside chance that Python is not so efficient with the code in this form. The `length` argument is just to specify the "computational bound" for the program, though this application is a prime candidate for implementation by generator:

```
from math import floor, sqrt
def squarepentagonalgenerator():
    index = 0
    while True:
        x = index*(3*index-1)//2
        if floor(sqrt(x))==sqrt(x):
            yield x
        index += 1
```

   - The generator version here uses the logic from the given comprehension to calculate a pentagonal number form an index value. This number is then evaluated to be square or not, and yielded if so. For calculating a square pentagonal number beyond index of 4 there is a significant time cost, though memory usage is negligible.

9. Nonnegative integers can be implemented in Python by `nest(0)=[]` and `nest(n+1)=[nest(n)]`. Write programs `nest` and `unnest` which translate between nonnegative integers and the nesting implementation. We do not care what happens at incorrect input. Implement addition and multiplication as direct functions on this implementation (i.e. not by untranslating, adding the numbers, and retranslating).
(The student who finds this kind of thing amusing, might be interested in the surreal numbers.)

```
def nest(n):
    if n == 0:
        return []
    else:
        return [nest(n-1)]
```

- Implements the nesting representation of integers exactly as described in the question. After recursively calling n times, as control is passed up the stack the returned list is placed inside another empty list and passed further up. Thus with n recursive calls, there will be n−1 nesting operations which combine with the base case empty list to represent n in nested list notation.

```
def unnest(n):
    try:
        n[0]
        return 1+unnest(n[0])
    except IndexError:
        return 0
```

- Using a rather nifty technique this function recursively flattens the list, and then counts how many recursive calls were made as control moves back up the stack retuning this count. The try statement is used to handle the base case where the list is empty, as `n[0]` will throw an `IndexError` exception in this case. Should it not throw an error, a recursive call is made on the first element of the list, effectively stripping the outermost brackets away.

```
def nestadd(x,y):
    try:
        x[0]
        return [nestadd(x[0], y)]
    except IndexError:
        return y
```

- Acting in a similar way to `unnest`, this program simply nests the list `y` for each recursive call, rather than increasing the returned value. The function flattens list `x`, and nests list `y` as many times as `x` could be flattened, returning the result.

```
def nestmult(x,y):
    z = x
    accum = []
    while True:
        try:
            z[0]
            accum = nestadd(accum,y)
            z = z[0]
        except IndexError:
            return accum
```

- Unlike previous programs, this one does not make use of recursion, rather a theoretically infinite loop. Taking a copy of x to prevent modifying the original list contents and initialising an empty list accumulator, the program proceeds to continually flatten the copy of x and add list y to the accumulator as many times as copy z is flattened. Once z has been completely flattened it remains only to return the accumulated list.