Coursework PY1 (Questions on Python)
Deadline: 17 November 2017

Important note: In all cases code should be clearly-written and should include a brief explanation in English explaining the design of your code.

Your answer must take the form of a plaintext .py file including the program and a nontrivial collection of tests, which can be cut-and-pasted by your marker into the command line, to test that it works (I'll repeat this in a moment).

Consistent with the principle that *code is written for humans to read* in the first instance, and for computers to execute only in the second instance, marks will be awarded for *style, clarity and testing.* So all answers must include the following:

- clear code
- clear comments, and
- relevant tests with explanations of what the tests are expected to return.

Your answer must take the form of a commented .py file, call it "PY1-gabbay.py", such that your marker can read the code directly, but also cut-and-paste the text directly into the Python3 command line and it will run.

Since this is a single .py file, you may assume variables, procedures, and functions defined in earlier questions in your answers to later questions, though you should add comments in code explaining this if any clarification might help read your code.

You may not use a library function if it renders the question trivial.

1. **Complex number arithmetic**

The **complex numbers** are explained here (and elsewhere):
 http://www.mathsisfun.com/algebra/complex-number-multiply.html
Represent a complex integer as a pair of integers, so `(4,5)` represents 4+5i (or 4+5j, depending on the complex numbers notation you use).

1a. Using `def`, define functions `cadd` and `cmult` representing complex integer addition and multiplication.

For instance,
```
    cadd((1,0),(0,1))
```
should compute
```
    (1,1).
```

1b. Python has its own native implementation of complex numbers. Write translation functions `tocomplex` and `fromcomplex` that map the pair (x1,y1) to the complex number x1+(y1)j and vice versa. You may use the python methods `real` and `imag` without comment, but not `complex` (use j directly instead).

2. **Sequence arithmetic**

An **integer sequence** is a list of integers.

2a. Using `def`, write iterative functions `seqaddi` and `seqmulti` that implement pointwise addition and multiplication of integer sequences.

For instance
```
   seqaddi([1,2,3],[~1,2,2])
```
should compute
```
   [0,4,5]
```
You need not write error-handling code to handle the cases that sequences have different lengths.

2b. Do as for 2a, but make your functions recursive (as for ML).
Call them `seqaddr` and `seqmultr`.

2c. Do it again. This time use list comprehensions instead of iteration or recursion.

3. **Matrices**

Matrix addition and multiplication are described here:

- addition: http://www.mathsisfun.com/algebra/matrix-introduction.html
- Multiplication (dot product): http://www.mathsisfun.com/algebra/matrix-multiplying.html

Represent integer matrices as a list of list of integers. So a matrix is a column of rows of integers.

Write functions

- `ismatrix`
  This should input a list of list of integers (henceforth an **intmatrix**) and test whether a list of lists of integers represents a matrix (so the length of each row should be equal).
- `matrixshape`
  This should input an intmatrix and return the size of the matrix, which is the number of rows and the number of columns (traditionally the number of rows is given first, but if you have done it the other way around that's fine; just make sure to clearly explain your code).
- `matrixadd`
  Matrix addition, which is simply pointwise addition.

- `matrixmult`
  Similarly for matrix multiplication.

You do not need to write error-handling code, e.g. for the cases that inputs do not represent matrices or represent matrixes of the wrong shapes; so for instance your `matrixshape` may assume that the argument has successfully passed the test `ismatrix`.

Your answer can use iteration, recursion, list comprehension, or anonymous functions. You should not appeal to any libraries, e.g. for matrix processing. Don't use `zip`.

4. **Essay-style question**

Write an essay on Python data representation. Be clear, to-the-point, and concise. Convince your marker that you understand:

- Mutable vs immutable types. Give at least two examples of each, with explanation.
- Integer vs float types.
- Assignment = vs equality == vs identity `is`.
- The computational effects of a call to `list` on an element of range type, as in `list(range(5**5**5))`.
- Slices, with examples. Including an explanation of the difference in execution between
  `list(range(10**10)[10:10])` and
  `list(range(10**10))[10:10]`

Include short code-fragments where appropriate (as I do when lecturing) to illustrate your observations.

5. **Encoding**

Write a general encoding function `encdat` that will intput an integer, float, complex number, string, and return it as a string.

So

- `encdat(-5)` should return `'-5'`
- `encdat(5.0)` should return `'5.0'`
- `encdat(5+5j)` should return `'5+5j'` (not `'(5+5j)'`; see hint below).
- `encdat('5')` should return `'5'`

Hint: you may find it useful to consider the following code fragment

```
type(5) == type('5')
```

6. Cool bonus question

An encoding f of numbers in lists is as follows:

- f(0) = []   (0 maps to the empty list)
- f(n+1) = [f(n),[f(n)]]   (n+1 maps to the list that contains f(n) and singleton f(n))

Implement encode and decode functions in Python, that map correctly between nonnegative integers and this representation. Call them `fenc and fdec`.

7. Another cool bonus question

Implement a generator `cycleoflife` such that if we assign

```
    x = cycleoflife()
```

then repeated calls to

```
    next(x)
```

return the values

```
    eat
    sleep
    code
    eat
    sleep
    code
    ...
```

in an endless cycle. If you can't manage an endless cycle, do a program that runs for 1000 cycles then stops.

Note that this question is *not* asking you to program an endless loop that prints these values. The effect of this is to implement what is called a *stream* (infinite list)
  x = [eat, sleep, code, eat, sleep, code, ...].
This does not mean the whole infinite datastructure is in memory at one time (which is impossible for a machine with finite memory), but for any finite but unbounded number of calls to `next` your generator behaves as if it were the infinite datastructure illustrated above.

8. Really cool question

Call a **datum** something that is either an integer, or a list of data (datums).

Write a flatten function `gendat` that converts a datum to a list of integers.

So

- `gendat(5)` should return `[5]`
- `gendat([])`should return `[]`
- `gendat([5,[5,[]],[],[5]])` should return `[5,5,5]`

Do not use `str`. You may find it useful to consider `isinstance` or the following code fragment

```
   type(5) == type([])
```

9. Unmarked question

Implement the Sieve of Eratosthenes
  https://en.wikipedia.org/wiki/Sieve_of_Eratosthenes
as a Python generator `eratosthenes` such that if we assign

```
   x = eratosthenes()
```

then repeated calls to

```
   next(x)
```

return the primes, starting from 2.