**CPSC 189**
**Sample Final Exam Questions - SOLUTIONS**

**Q1.** What is the value of each of the following Python expressions?

a) `2 + 7 / 4`
`Ans: 3.75`

b) `10.0 * 3 / 4`
`Ans: 7.5`

c) `[i + 2 for i in loi]`
assuming that `loi = [1, 2, 3, 4]`
`Ans: [3, 4, 5, 6]`

d) `loi[1::2]`
assuming that `loi = [1, 2, 3, 4, 5]`
`Ans: [2, 4]`

e) `arr[lob]`
assuming that `lob = numpy.array([True, True, False, True])` and that `arr = numpy.array([1, 4, 3, 2]`
`Ans: array([1, 4, 2])`
(`[1, 4, 2]` is acceptable)

f) `[[x * y for y in range(1, 4)] for x in range(1, 3)]`
`Ans: [[1, 2, 3], [2, 4, 6]]`

g) `[x * y for y in range(1, 4) for x in range(1, 3)]`
`Ans: [1, 2, 2, 4, 3, 6]`

h) Assuming that `arr = numpy.array([1, -2, 3, -4])`
`arr + 4`
`Ans: array([5, 2, 7, 0])`
(`[5, 2, 7, 0]` is acceptable)

## Q2. Writing function signatures

**a)** Suppose you want to design a function that consumes a list of `float`s and a threshold value (also of type `float`) and that produces a list of only those `float`s that are greater than the threshold value. What is the function signature?
Ans: `(listof float), float -> (listof float)`

**b)** Suppose you want to design a function that consumes a list of strings and that removes from that list all the empty strings. Note: the function mutates the given list. What is the function signature?
Ans: `(listof str) -> NoneType`

**c)** Suppose you want to design a function that consumes:
- a function that consumes a `float` and produces a `bool`
- a list of `float`

and that produces a list of `bool`. What is the function signature?
Ans: `(float -> bool), (listof float) -> (listof bool)`

**Q3. Testing**

Suppose you want to design a function that consumes a list of volumes (each volume is represented by a float) and a density (also a float) and that produces a list of masses according to the formula: mass = density * volume. Design appropriate tests. It is **not** necessary to complete the body of the function.

```
def volumes_2_masses(lov, den):
    """
    (listof float), float -> (listof float)

    Produces a list of masses for each volume in lov
    assuming a density den

    >>> volumes_2_masses([], 1.0)
    []

    >>> act = volumes_2_masses([1.0, 2.5, 3.0], 2.0)
    >>> exp = [2.0, 5.0, 6.0]
    >>> all([abs(a - e) < e * EPS
    ...      for (e, a) in zip(exp, act)])
    True
    """

    return []      # stub
```

**Q4.** Use a list comprehension to design a function that consumes:

- a list of strings
- a width (of type `int`)
- a fill character (a string of length 1)

and that produces a list of strings by taking each string in the original list, centering it in a string of the given width and padding the remaining slots with the given fill character. We assume that the number of slots to be padded is an even number.

**Hint:** The following method of the string class will be useful:

> `center(width, fillchar)`
> Returns string centered in a new string of length `width` padded either side with `fillchar`

```
def center_str(los, width, pad):
    """
    (listof str), int, str -> (listof str)

    Produces a list of strings by taking each str in los,
    centering it in a string of size width and padding
    remaining slots with pad character.
    Requires: number of slots to be padded is an even
    Number.

    >>> center_str([], 10, '*')
    []

    >>> center_str(['189'], 7, '#')
    ['##189##']

    >>> center_str(['110', '189', '210'], 5, '^')
    ['^110^', '^189^', '^210^']
    """
    return [s.center(width, pad) for s in los]
```

**Q5.** For the function considered in Q4, suppose we want to make the `pad` parameter optional. If this parameter is not specified, we want a space to be used by default as the fill character. In point form, describe the changes that you would make to the design of the function. Do not rewrite the entire function.

- Change the function header to specify default value for the parameter:
  `def center_str(los, width, pad=' ')`
- Add a test to check use of the default parameter value:
  ```
  >>> center_str(['189'], 7)
  ['  189  ']
  ```

**Q6.** Consider the following data definition:

```
# Dog is dict(name=str, age=int, shots=bool)
# A dog with a name, age and shots status (shots is true if and only if
# the dog has been vaccinated)

D1 = dict(name='chunk', age=4, shots=True)
D2 = dict(name='molly', age=3, shots=True)
D3 = dict(name='monty', age=2, shots=True)
D4 = dict(name='meggy', age=1, shots=True)
D5 = dict(name='toby', age=3, shots=False)
D6 = dict(name='lulu', age=2, shots=False)

fn_for_dog(d):
    return ...d['name'] ...d['age'] ...d['shots']
```

Use a list comprehension to design a function that consumes a list of dogs and produces a list of the names of those dogs that are less than 3 years old and have been vaccinated. You may assume that the examples written as part of the data definition above are available for use in your tests.

```
def lonames_puppy_vacc(lod):
    """
    (listof Dog) -> (listof str)

    Produces a list of the names of those dogs in lod that are less than
    3 years old and have been vaccinated.

    >>> puppy_dogs_vacc([])
    []

    >>> puppy_dogs_vacc([D1, D2, D3, D4, D5, D6])
    ['monty', 'meggy']
    """
    return [d['name'] for d in lod if is_puppy_vacc(d)]


def is_puppy_vacc(d):
    """
    Dog -> bool

    Determines if dog is vaccinated and under 3

    >>> is_puppy_vacc(D6)
    False
    >>> is_puppy_vacc(D2)
    False
    >>> is_puppy_vacc(D3)
    True
    >>> is_puppy_vacc(D4)
    True
    """
    return d['age'] < 3 and d['shots']
```

**Q7.** Complete the design of the following function. Assume that the file to be read contains only integers separated by a known delimiter. Further assume that the file could be empty and that the file could contain rows that contain only whitespace.

Recall that the string class has the following method:

`split(sep)`
Returns a list of words in the string using `sep` as the delimiter.

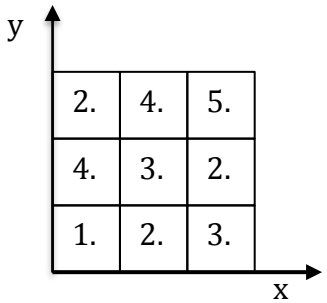
```
from StringIO import StringIO

def count_positive(f, delim):
    """
    file, str -> int

    Produces number of integers in file f that are
    positive.
    Requires: file contains only integers, separated by
    delim or lines containing only whitespace

    >>> f = StringIO('')
    >>> count_positive(f, ' ')
    0

    >>> f = StringIO('  \\n     \\n    \\n')
    >>> count_positive(f, ' ')
    0

    >>> f = StringIO('1 -4 5')
    >>> count_positive(f, ' ')
    2

    >>> f = StringIO('1, -3, 5\\n -2, 0, 6\\n 3, 4\\n')
    >>> count_positive(f, ',')
    5
    """
    count = 0

    for line in f:
        if line.strip() != '':
            count += count_positive_line(line, delim)

    return count
```

```
def count_positive_line(line, delim):
    """
    str, str -> int

    Produces number of integers in line that are positive
    Requires: line contains only integers, separated by
    delim

    >>> count_positive_line('0 1 2 3', ' ')
    3

    >>> count_positive_line('0, -1, 2, -3, 4, -5', ',')
    2
    """
    count = 0

    for next_str in line.split(delim):
        if int(next_str) > 0:
            count = count + 1

    return count
```

**Q8.** Suppose we use a 3D numpy array to represent temperatures measured in the atmosphere. Axis 0 of the numpy array is aligned with the *z*-axis, axis 1 with the y-axis and axis 0 with the *x*-axis. As the index on axis 0 increases z increases, as the index on axis 1 increases y increases and as the index on axis 2 increases x increases.

**a)** The following 2D array represents temperatures through one particular horizontal slice of the 3D array of temperatures. Write each temperature in the appropriate slot on the diagram below.

```
array([[1.0, 2.0, 3.0],
       [4.0, 3.0, 2.0],
       [2.0, 4.0, 5.0]])
```



**b)** *Using appropriate functions from the numpy library*, complete the design of a function that consumes such a 3D numpy array (of floats) and that produces a 2D array that contains the average of the temperatures along each column of temperatures that runs parallel to the *z*-axis. You may assume that each column of temperatures contains at least one temperature (so the average exists).

Note that you must complete the design of the test as well as the body of the function!

```
def temp_avg(temps):
    """
    np.ndarray(np.ndarray(np.ndarray(float))) -> np.ndarray(np.ndarray(float))

    Produces the average of temperatures along each column of
    temperatures aligned with the z-axis.

    >>> temps = np.array([[[1.0, 2.0, 3.0],
    ...                    [2.0, 3.0, 4.0],
    ...                     [3.0, 4.0, 5.0]],
    ...                    [[3.0, 4.0, 5.0],
    ...                     [4.0, 7.0, 8.0],
    ...                     [5.0, 8.0, 7.0]]])
    >>> act = temp_avg(temps)
    >>> exp = np.array([[2.0, 3.0, 4.0],
    ...                 [3.0, 5.0, 6.0],
    ...                 [4.0, 6.0, 6.0]])
    >>> np.all(abs(act - exp) < EPS * exp)
    True
    """
    return np.mean(temps, axis=0)
```

**Q9.** Consider the following function definition:

```
def map_arg(fn, lof, *args):
    """
    ((float, ...) -> float), (listof float), ... -> (listof float)

    Applies the function fn onto each of the numbers in lof with
    additional arguments in *args passed to fn

    <Tests omitted>
    """
    return [fn(f, *args) for f in lof]
```

What value is produced by the following function call?

```
map_arg(lambda x, y, z: x + y + z, [1.0, 2.0], 3.0, -1.0)
```

Ans: `[3.0, 4.0]`

**Q10.** Suppose you have a text file that contains an unspecified number of rows representing observational data for sea turtles. Each row starts with an integer that represents the ID of a tag attached to a sea turtle. This is followed by the latitude (`float`) and longitude (`float`) of the turtle at the time that it was observed. A comma separates the tag ID, latitude and longitude. There is no other data on the line. There are no header lines, no blank lines or lines that contain only whitespace.

Consider the following data definition:

```
Location is (float, float)
interp. (lat, long) is the latitude and longitude of a location

L1 = (23.7, 131.3)
L2 = (-12.3, -51.4)

fn_for_locn(loc):
    return ...loc[0]  ...loc[1]
```

Now complete the design of the following function. The function consumes a file (assumed to be open for reading in universal mode) and a tag ID and produces a list of locations where the turtle with the given ID was observed. Study the signature, purpose and tests carefully!

```
def find_locns(f, tag_id):
    """
    file, int -> (listof Location)

    Produces list of locations where turtle with ID tag_id was observed.

    >>> data = '1234, 23.8, 159.99\\n 3142, -42.3, 175.14\\n' \
               '1234, 23.84, 159.98\\n 1551, 32.6, 63.23\\n'
    >>> f = StringIO(data)
    >>> find_locns(f, 2315)
    []

    >>> f = StringIO(data)
    >>> find_locns(f, 1234)
    [(23.8, 159.99), (23.84, 159.98)]
    """
    locns = []
    for line in f:
        data = line.split(',')
        next_id = int(data[0])
        lat = float(data[1])
        lng = float(data[2])
        if next_id == tag_id:
            locns = locns + [(lat, lng)]

    return locns
```

**Q10b.** Suppose you have a directory structure, that contains files and subdirectories where each subdirectory contains files and other subdirectories (nested arbitrarily deep). Assume that this directory structure contains a number of files like the one described in part (a) of this question and that these files have a `.turtle` extension. You must also assume that the directory structure contains other files with different extensions that contain data that is not of interest in this question.

Design a function that consumes the path to a directory (as a string), a file `outf` that has already been opened for writing and the tag ID of a turtle. The function must then walk the given directory and, for each file with a `.turtle` extension, write to `outf` the locations where that turtle was observed, one location per line. So, for example, the output file might contain something like the following after a call to your function:

```
(23.8, 159.99)
(23.84, 159.98)
(23.8, 159.96)
(23.76, 159.92)
```

It is not necessary to include tests for any function that you design in this question.

```
def find_locns_dir(rootdir, outf, tag_id):
    """
    str, file, int -> NoneType

    Effect: writes to outf the locations where the turtle with tag_id
    was observed across all files with a .turtle extension in directory
    rooted at rootdir.
    """
    for path, lodn, lofn in os.walk(rootdir):
        find_locns_lofn(path, lofn, outf, tag_id)


def find_locns_lofn(path, lofn, outf, tag_id):
    """
    str, (listof str), file, int -> NoneType

    Effect: writes to outf the locations where the turtle with tag_id
    was observed for all files in lofn with a .turtle extension.
    """
    for fn in lofn:
        name, ext = os.path.splitext(fn)
        if ext == '.turtle':
            with open(os.path.join(path,fn), 'U') as inf:
                lolocn = find_locns(inf, tag_id)
                write_loloc(outf, lolocn)
```

```
def write_loloc(outf, lolocn):
    """
    file, (listof Location) -> NoneType

    Effect: writes list of locations to outf
    """
    for locn in lolocn:
        write_loc(outf, locn)


def write_loc(outf, locn):
    """
    file, Location -> NoneType

    Effect: writes location to outf
    """
    outf.write('(' + str(locn[0]) + ','
                   + str(locn[1]) + ')\n')
```