Practice Final Exam for CSCI-UA.0002
May, 2018

Name: ________________________________
Net ID: ___________________

There are 2 sections, the first section is worth 50 points and the second section is worth 50 points, for a total of 100. Answer all 4 questions in Section 1. Answer 2 out of the 3 questions in Section 2. It is essential that you

**PUT YOUR NAME AND NET ID ON ALL TEST MATERIALS.** It can be difficult to identify the author of an unsigned test and it would be better to avoid this problem.

This test includes 2 separate booklets.

Booklet 1 (this booklet) includes:

1. Section 1 questions with boxes to fill in your answers. Extra boxes/extra space is provided to allow room for cross-outs and corrections. You can use any part of the test (other than the boxes) for scrap paper. However, please note that the backs of pages will not be scanned into Gradescope and therefore not considered for credit—so do not include answers on the backs of pages. Please do not remove any pages from the test and please do not add any pages. Extra scrap paper is available if you need it.

2. Additional pages for answering Section 2 questions. Each part 2 question is allocated 3 blank pages, labeled with the question that they are intended for. Please make an effort to use these pages to answer the corresponding questions. You can use the back of pages as scrap paper, but the backs of pages will not be scanned in to Gradescope. If for some reason, you need to write an answer in a non-obvious place, please indicate this with a clear note, somewhere in the question, e.g., see pages marked for question 7 could be a note indicating that the rest of an answer (for question 5 or 6) is found in the space allocated for question 7.

*It is important that you do not remove any pages from this booklet. Removing pages may prevent Gradescope from working properly.*

Booklet 2 (the other booklet) contains:

1. Section 2 questions

2. A GLOSSARY OF TERMS – Please feel free to look up some of the basics in this glossary. I will also answer any reasonable look-up style questions as I am more interested in your ability to reason than your ability to memorize.

**Section 1:** (50 points) Each example consists of Python code, followed by questions and places to fill in answers. Please read all questions carefully and answer them as precisely as possible. Assume that there are no bugs in the code that will make the system crash, although the code may not solve a problem perfectly. If you find anything that you think is a bug, there is either a typo (and I should fix it for everyone) or you are mistaken.

**Sample Question A:**

```python
output = '1'+ '1'
```

Question: What is the value of the variable `output`?
Answer: '11'

Answer: '11'

Note: Attention to detail is important. The quotes indicate that it is a string. Partial credit is possible. For example, leaving out the quotes would have resulted in a small deduction, but answering 2, would have resulted in an incorrect answer.
def string_reverse(instring):
    output = ''
    for num in range(len(instring)):
        output = output + instring[-(num+1)]
    return(output)

def double_reverse(inlist):
    for index in range(len(inlist)):
        inlist[index] = string_reverse(inlist[index])
    inlist.reverse()

in_list = ['cat', 'dog', 'mouse']

def main():
    double_reverse(in_list)

main()

Question: After the above code executes, what is the value of the variable `inlist`?

Answer: }

Answer: ["mouse", "dog", "cat"]
def tack_on_str_positions(instring):
    output = ''
    for index in range(len(instring)):
        output = output + str(index) + instring[index]
    return(output)

def tack_on_list_position(list_of_strings):
    output = []
    for index in range(len(list_of_strings)):
        output.append(str(index) + '_' + list_of_strings[index])
    return(output)

def analyze_list_of_strings(list_of_strings):
    output = []
    for item in list_of_strings:
        output.append(tack_on_str_positions(item))
    return(tack_on_list_position(output))

output = analyze_list_of_strings(['cat', 'dog', 'mouse'])

Question: After the above code executes, what is the value of the variable `output`?

Answer: 3
def checking_ducks_in_a_row(input_list):
    output = []
    plural = False
    for item in input_list:
        item = item.lower()
        bird_type = 'other'
        if item[-1] == 's':
            plural = True
            check_item = item[:-1]
        elif item == 'geese':
            plural = True
            check_item = 'goose'
        else:
            plural = False
            check_item = item
        if check_item in ['duck','drake','duckling','mallard','widgeon']:
            bird_type = 'duck'
        elif check_item in ['goose','gander','gosling','greylag','brant','honker']:
            bird_type = 'goose'
        else:
            print(item+' is not a type of duck or goose! Revise and start again.
        bird_type = 'inappropriate_bird_type'
        record = [item,bird_type]
        if plural:
            record.append('multiple')
        else:
            record.append('single')
        output.append(record)
    return(output)

output1 = checking_ducks_in_a_row(test_1)
output2 = checking_ducks_in_a_row(test_2)

Question 3a: What is the values of output1 after the above code executes?

Answer: 

Question 3b: What is the values of output2 after the above code executes?

Answer: 

Question 3c: What if anything prints out during the execution of the code above?

Answer: 

Answer:
def get_lab_id(record, old_id=False):
    if old_id:
        return(old_id+'X')
    else:
        id_num = ''
        for field in record:
            if isinstance(field, str):
                id_num = id_num+field[0]
            elif isinstance(field, int):
                if field>99:
                    field = field//10
            elif isinstance(field, float):
                field = str(field)
            id_num_list = field.split('.')
            id_num = id_num+str(id_list[0])
    return(id_num)

def classify_record(record):
    record.reverse()
    id_num = get_lab_id(record)
    while id_num in lab_record_dictionary:
        id_num = get_lab_id(record,old_id=id_num)
    record.append(id_num)
    record.reverse()
    lab_record_dictionary[id_num]=record

def categorize_lab_specimens(list_of_records, outfile):
    import os
    global lab_record_dictionary
    lab_record_dictionary = {}
    for record in list_of_records:
        classify_record(record)
    list_of_records.sort()
    with open(outfile,'w') as outstream:
        for record in list_of_records:
            outstring = record[0]
            for field in record[1:]:
                outstring = outstring+'	'+str(field)
            outstring = outstring+os.linesep
            outstream.write(outstring)

def main():
    records = [['dutch elm',50,5000,'Australia'],['Bengal Tiger',10,800,'India'],
               ['Brenda Thompson',10,80,'Indiana']]
    categorize_lab_specimens(records,'specimens.tsv')

Question: On the next page, Write out lines as they would appear in the specimens.tsv file. Indicate tabs as spaces a few characters long.
Section 2 (50 points): Answer 2 of the 3 questions in this section (see booklet 2 for the questions). For each question, you do, write a Python program as specified. Please include the code in the pages allocated to that question, e.g., answer question 7 on the pages marked Question 7. Cross out any code that you do not want counted. If you do 3 (rather than 2) questions, cross out the answer to the question that you do not want to be counted. If for some reason, you need to write an answer in a non-obvious place, please indicate this with a clear note, somewhere in the question, e.g., see pages marked for question 7 could be a note indicating that the rest of an answer (for question 5 or 6) is found in the space allocated for question 7.

All questions are meant to handle general cases. If there are variables used as sample input parameters, please do not write programs that only handle those sample values. They should handle all inputs of that type, e.g., if the input parameter is a list of lists of the form [['fish',1,2,3]['dog',4,5,6]], that is an example of a list of records, where the first field of each record is a string, and the remaining fields are integers. Depending on the problem, records may or may not be limited to 4 items.

If variables or functions are provided for your use in the answer, you can use the name of these variables and call these functions without re-copying them into your answer.
This booklet contains:

- The questions for Section 2: Please answer these questions in the dotted pages provided at the end of Booklet 1.
- A Glossary of Terms

**Question 5:** Write a program using the *turtle* package. Given a number less than 1000, the turtle program should draw tally marks representing that number. Figure 1 illustrates tally marks corresponding to the number 199. Tally marks divide up a number into units of 5. Each unit of 5 consists of a sequence of 4 vertical lines, with one diagonal line cutting across from the top of the first line to the bottom of the last line. The final sequence may represent fewer than 5 as that number of vertical lines, without the diagonal, e.g., 4 is represented as 4 lines, 3 as 3 lines, etc.

Your program should begin with the turtle at position -300,0. Each vertical lines should be 20 units high and there should be a space of 10 between vertical lines within a group. There should be a distance of 20 between groups.

If the program ends a block of 5 and the X position is greater than 280, the program should advance to the next line of tally marks. This means that the program should go to the same X start position (-300), but 30 lower: (-300,-30) for the second line, (-300, -60) for the third line, etc.
Question 6: Write a program that determines the value of a roll of five dice in a game, which I will call Roll ’em.

The player rolls five dice, each using the following function:

```python
def die_roll():
    import random
    return(random.randint(1,6))
```

The program must identify the type and value of the hand. Types of hands are defined along with their point values, as follows (in order from highest to lowest).

1. five of a kind: five instances of the same number (60 points)
2. four of a kind: four instances of the same number (50 points)
3. full house: three instances of one number and two instances of another number (40 points)
4. straight: the roll includes five consecutive numbers in any order, e.g., 2,4,3,1,5 is a valid straight. (35 points)
5. three of a kind: three instances of the same number (30 points)
6. two pairs: two instances of one number and two instances of another number (20 points)
7. pair: two instances of the same number (10 points)
8. high: if none of the above, the value is determined by the highest roll, e.g., if the highest roll is 5, then the value of the roll is 5.

Extra credit: Complicate the above game by allowing the player to have one additional turn where they can reroll up to three dice. Add a computerized player that will also have this option, but whose choice is determined automatically. The computer will never reroll if it gets five of a kind or a straight and it will not reroll dice that are part of a grouping (pair, three of a kind, four of a kind, five of a kind).
**Question 7:** Write a program that reads in a text file and counts instances of the different characters in the file, storing the frequency information in a dictionary.

Then the program should sort the keys of the dictionary and write one entry for each key in an output file using any format you want. (use the `DICT.keys()` method to identify the keys of the dictionary; use the `list` function to convert it to a list; and use `list.sort()` to sort the list.)

For the space character, print 'SPACE' rather than the space character (as this will make it easier to read the file). Similarly, if a character is unprintable, write 'Unicode', followed by its unicode value instead of the character. The method `string.isprintable()` has a value of `True` if a character is printable and `False` otherwise, e.g., `os.linesep.isprintable()` evaluates to `False`. Sample lines from an output file that uses .tsv format:

```
Unicode 10 16
SPACE 142
" 2
( 2
) 2
, 28
- 4
. 7
A 2
C 1
```
Glossary of Terms

1. Some Basics

- **return**(X) causes the current function to exit and cause the expression represented by the function call to evaluate as X. For example given the following steps, the value of `output` would be 5:

```python
def add(num1, num2):
    return (num1+num2)
output = add(2, 3)
```

- **print**(X) prints X to the screen. This is only for the benefit of the user. It is not useful for having programs interact. **print**(a,b,c,d,...) will print all the items input and put a space in between the items by default, e.g., print(’Hello’,’world’,’it’, ’is’, ’me’) will print:

```
Hello world it is me
```

The key word parameter **sep** is used to separate the items by something other than a space. By default print ends the print out with a newline, but this default can also be changed with the **end** key word parameter. For example, print(’The ’,’blue’,’dog’,sep=’*’,end=’:
```
would print:
```
The*blue*dog:
```
and no newline would immediately follow (so a following print statement would print on the same line).

- The parameters of a function are the local variables inside of the parentheses in the function definition. They are useful when you have functions call functions.

- **input**(prompt) is used to ask a human being a question so that a program can interact with a human being. This is useful when you want a human being to enter information interactively. *input* statements should be used only when human interaction is appropriate. *input* statements return a string consisting of the characters that the user typed in. It may be necessary to convert this string to some other data type, e.g., an integer (with *int*) or a float (with *float*).

- The operator + will add two numbers or concatenate two strings
- The operator * will multiple two numbers or print a string some number of times.
- The operator ** will represent exponents, e.g., 5**2 == 25.

2. Division and Modulus

- 5 // 2 == 2
- 5/2 == 2.5
- 5%2 == 1

3. More Math

- round(4.5) == 4, round(4.6) == 5, round(-4.5) == -4, round(-4.6) == -5
- round(2/3, 2) == .67 ## The second argument of round indicates number of decimal places. The default is to round to the nearest whole number.
- math.floor(4.9) == 4, math.floor(-4.9) == 5
- math.ceil(4.9) == 5, math.ceil(-4.9) == -4
- math.trunc(4.9) == 4, math.trunc(-4.9) == -4
- math.pi == 3.141592653589793 – a variable for the value of pi

4. sequences

- object made up of other objects in an order
- the function len(sequence) returns the number of items in the sequence
- the operator **in** tests for membership in sequence, e.g., (’a’ in ’abc’) would have the value True.
- sequences are used in for loops (see below)
• indices and slices

– Indices in a sequence are numbers from zero to the length of the string. Zero refers to the
position before the first item in the string and the length of the string refers to the position
following the last item. Thus each item in the sequence are between two consecutive
indices. For example, the subscripted numbers indicate indices for the string 'The book':
0\text{T}_1 h_2 e_3 b_4 o_5 o_6 r_7 o_8 k_9. Similarly, the indices in [0, 'The', 'book', 'is', 'here'] indicate
positions in the list ['The', 'book', 'is', 'here'].

– negative indices can be used to count positions from the end. Thus -1 is equivalent to the
position at one less than the length of the string; -2 is equivalent to the position at two less
than the length of the string; etc. The the negative positions around The book would be
labeled as follows: '−8 T_{−7} h_{−6} e_{−5} b_{−4} o_{−3} r_{−2} o_{−1} k'.

– sequence[num] indicates an element in a sequence beginning at num (a number from zero
to one less than the length of the string), e.g., 'The book'[4] evaluates to 'b'; ['The', 'book',
'is', 'here'][0] evaluates to 'The'. The negative indices can be similarly applied, e.g., 'The
book'[-1] evaluates to 'k', the last character in the string.

– sequence[start_index:end_index] returns a subsequence between start_index and end_index.
For example, 'The book is red'[4:8] would return 'book' and ['The', 'book', 'is', 'about', 'roses'][2:4]
would return ['is', 'about']. Leaving out the number before the colon suggests that a subse-
quence begins at the beginning of the sequence and leaving out the number after the colon
suggests that the subsequence ends at the end of the list. Thus 'The book'[:3] evaluates as
'The' and ['The', 'book', 'is', 'here'][2:] evaluates as ['is', 'here'].

• ranges define a sequence of numbers based on the length of a sequence starting from 0. If
given 2 arguments, the second argument is the length of a sequence starting from 0 and the first
argument is a starting point within that sequence.

– range(5) is approximately equivalent to [0,1,2,3,4]
– range(1,5) is approximately equivalent to [1,2,3,4]

• Strings

– an empty string has zero characters ''
– strings are sequences of characters, e.g., 'Hello World!' consists of the items ['H', 'e',
'I', 'l', 'o', ' ', 'W', 'o', 'r', 'l', 'd', '!']
– string1.strip(string2) – removes instances of elements in string2 from beginning and end
of string. For example, '***Hello World***'.strip('*') will return 'Hello World'. '+'-The
giant wombat.*+'.strip('-+*') will return 'The giant wombat'.
– string.split(character) – creates a list by dividing a string at each instances of character. For
example, 'Hello World'.split(' ') will return the list ['Hello', 'World'].
– string.lower() converts string to lower case
– string.index(item) returns the position index item occurs in the list – it is an error if the item
is not in the string. This works with both characters and substrings.

• Lists

– A list is represented by square brackets surrounding a list of objects, divided by commas,
e.g., ['A','List','of','Strings']
– Lists are mutable. A function that takes a variable pointing to a list as an argument, can
change the list. For example, if the variable my_sentence is set to the list ['Today', 'is',
'the', 'final', 'exam'], the function call: my_sentence.reverse() will cause the value of
my_sentence to change to ['exam', 'final', 'the', 'is', 'Today'].
– You can add an object to the end of a list using the append method. For example, suppose
my_list = ['a','b','c']. Then my_list.append('d') will add 'd' to the end of my_list, setting it
to ['a','b','c','d'].
– You can change an item in a list as follows: list[index] = newitem, e.g., if my_list is set to
[1,2,3], then the statement my_list[1] = 57, will change the value of my_list to [1,57,3].
– You can pop an item off the end of a list with the `pop` method. `pop` will return the removed item. For example, if `my_list` is set to `['a', 'b', 'c', 'd']`, then `my_list.pop()` will return 'd' and shorten the list to `['a', 'b', 'c']`.

– You can pop off an item at a particular position if you use `pop` with an index. For example, suppose `my_list = ['a', 'b', 'c']`, then `next_letter = my_list.pop(1)` would result in `next_letter` being equal to 'b' and `my_list` being set to `['a', 'c']`.

– You can combine two lists together using `extend`. If `my_list` is set to `['a', 'b', 'c']`, then the command `my_list.extend(['d', 'e', 'f'])` will reset `my_list` to `['a', 'b', 'c', 'd', 'e', 'f']`. The operator `+=` functions the same as `append`, i.e., `my_list.extend += ['d', 'e', 'f']` acts the same as `my_list.extend(['d', 'e', 'f'])`.

– The operator `+` returns a new list that combines 2 old lists, e.g., given `list1` set to `['a', 'b', 'c']` and `list2`, set to `['d', 'e', 'f']`, the command `new_list = list1 + list2`, will result in a new list merging together the 2 old ones, i.e., `['a', 'b', 'c', 'd', 'e', 'f']`. Unlike most functions and operators used to combine lists, it does not change the input lists.

– The method `sort` will put a list in order based on how `<` applies to pairs of elements in the list. This will only work if the members of the list are comparable with each other, e.g., you cannot compare an integer to a string.

– The method `reverse` – turns a list backwards, e.g., If `my_list` is set to `['a', 'b', 'c']`, the command `my_list.reverse()` will cause `my_list` to be set to `['c', 'b', 'a']`.

5. `dictionaries` – are sets of key/value pairs used for representing attributes of the keys. For example, consider the dictionary `phonebook = {'Mary': '212-333-4444', 'John': '914-444-3333'}`. The keys `Mary` and `John` are linked to the values `212-333-4444` and `914-444-3333`. This dictionary provides a way to look up the phone numbers of the named people.

- `dictionary[key]` – returns value associated with key in dictionary
- `dictionary[key] = value` – sets dictionary entry for key to value
- The `in` operator can be used with dictionaries to check if a key is found, e.g., `(name in phone_book)` evaluates to True given a dictionary `phonebook` and some key `name` such that `phonebook[key]` would have a value.
- `len(dictionary)` – returns the number of key/value pairs found in the dictionary.
- `dictionary.keys()` – returns a sequence of keys from the dictionary (this can be coerced into a list using the `list` function.
- `dictionary.keys()`, `dictionary.values()`, and `dictionary.items()` – These return sequences of keys, values and items respectively. Items are key value pairs. These can be converted to lists using the `list` function or used as is (e.g., in for loops).

6. `for` loops

- First Line: `for VARIABLE in SEQUENCE`:
- `VARIABLE` is set to each item in the sequence one at a time
- The Indented body repeats once for each item in sequence (for each setting of `VARIABLE`).
- A `return` inside a loop, will exit both the loop and the function.
- A `break` inside a loop, will exit the loop only.
- It is common to initialize a variable outside a loop (called an accumulator) that then gets incremented inside the loop.
- `for` loops can also be used with nonsequences composed of multiple elements. For example, looping a variable through a dictionary, sets the loop variable to keys in the dictionary (one at a time, in an arbitrary order).
7. **while loops**

- First line *While (BOOLEAN-EXPRESSION):*
- The loop keeps executing the indented body until BOOLEAN-EXPRESSION evaluates as *False.*
- If BOOLEAN-EXPRESSION is always *True,* the loop is endless.
- Typically BOOLEAN-EXPRESSION contains one or more variable(s), such that some values of these variables (or this variable) cause BOOLEAN-EXPRESSION to evaluate as *True* and other values cause it to evaluate as *False.*
- The body of the loop can change these variables. The loop starts when BOOLEAN-EXPRESSION is *True.* Then, under most circumstances, BOOLEAN-EXPRESSION eventually evaluates as *False* and the loop halts. For example, if the BOOLEAN-EXPRESSION is `stop == False,` setting `stop to True` inside the loop, will cause the loop to finish.
- It is common to use accumulator variables in a similar manner as with *for loops.*
- *break* and *return* behave the same way for *while loops* as they do with *for loops.*

8. **if statements**

- the first line of an *if* statement consists of *if BOOLEAN-EXPRESSION:*
- the body of text indented under the first line is executed if the BOOLEAN-EXPRESSION evaluates to *True*
- the *if* statement can be followed by optional *elif* statements of the form *elif BOOLEAN-EXPRESSION:* Each *elif* statement is only evaluated if the BOOLEAN expressions in the *if* and *elif* statements leading up to this one are *False.*
- The block of *if* and optional *elif* statements can end with an optional *else* statement. The first line is simply *else:* The body of text under *else* executes if the Boolean expressions for all previous *if* and *elif* statements in the sequence evaluate to *False.*

9. **logical operators**

- *X and Y* returns *True* only if both X and Y are *True*
- *X or Y* returns *True* only if X is *True,* Y is *True* or both are *True*
- *X in Sequence* returns *True* if X is in a member of a sequence, e.g., `'a' in 'abcdefg'` would return *True*
- *X == Y* returns *True* if X and Y are the same
- *X != Y* returns *True* if X and Y are the different
- *X < Y* returns *True* if X is less than Y
- *X > Y* returns *True* if X is greater than Y
- *X <= Y* returns *True* if X is less than or equal to Y
- *X >= Y* returns *True* if X is greater than or equal to
- *Not X* returns *True* if X is *False*
- The operators `<, >, <=, >=` can apply to non-numeric objects. Characters are compared based on their unicode values (so `'a' < 'b'). *True* is assumed to be greater than *False.* Sequences are compared lexicographically – the first item in pairs of sequences are compared first, but if they are equal the second items are compared, and so on.
10. Turtles

- Screen and Turtle objects are created using the commands `turtle.Screen()` and `turtle.Turtle()`.
- The turtle is initially in the center of the screen facing rightward.
- `my_turtle.left(degrees)` – rotates the `my_turtle` `degrees` left (from its perspective).
- `my_turtle.right(degrees)` – rotates `my_turtle` `degrees` right.
- `my_turtle.fd(distance)` – moves the `my_turtle` `distance` units forward.
- `my_turtle.bk(distance)` – moves the turtle `distance` units backwards.
- `my_turtle.pu()` – picks the pen up
- `my_turtle.pd()` – puts the pen down (ready to write)
- `my_turtle.circle(radius)` – creates a circle with radius `radius`. The circle will be above the direction the turtle was facing when it started drawing. The turtle will move left and up in a circle and end up in the same place as before.
- `my_turtle.goto(X,Y)` – (or `my_turtle.setPosition(X,Y)`) moves the turtle to the position `(X,Y)` (and draws a line from the current position to `(X,Y)` if the pen is down).
- `my_turtle.pos()` – returns the X and Y values of the current position, e.g., `X,Y = my_turtle.pos()`

11. `time.sleep(sec)` – pauses for `sec` seconds (requires the module `sleep` to be imported)

12. random – the random module

- `random.random()` returns a number between 0 and 1
- `random.randint(num1,num2)` returns a number between `num1` and `num2` (inclusive).
- `random.choice(sequence)` returns member of `sequence`.

13. Input/Output

- os – module including global variables like `os.linesep` (end of line strings: `\n` or `\r\n`) and `os.sep` (path separators – forward slash `/` or backward slash `\`). The os module also includes functions that interact with the operating system. `os.getcwd()` returns the current working directory. `os.listdir(PATH)` returns a list of files in `PATH`; `os.path.isdir(PATH)` returns True if `PATH` is a directory and False otherwise; `os.path.isfile(PATH)` returns True if `PATH` is the name of an existing file and False otherwise.
- Streams – Python objects used for reading files and writing to files.
  - `instream = open('my_file.txt', 'r')` sets the variable `instream` to the contents of the file `my_file.txt`. For most applications, it makes sense to remove these.
  - `outstream = open('my_file.txt', 'w')` sets the variable `outstream` to an object that will ultimately be saved as the file `my_file.txt`. The method `outstream.write(string)` will write a string to that file. It is a good idea to include `\n` anywhere you would like a line break in the file as end of lines are not automatic. `\n` should be used, rather than `os.linesep`, even in Windows.
  - `stream.close()` will close an opened stream. This ends the connection between Python and a file. In the case of output streams (like `outstream`), the content of the stream is written to the file.
  - `with open(file,'r') as instream: or with open(file,'w') as outstream:` starts a block in which a stream is opened. The body of code indented under these statements can read from or write to the stream. After the block ends, the stream is closed.

14. Error Handling

- Try/Except – Two key words that begin blocks, similar to IF/Else statements. If the code indented under `Try:` does not cause any error, then the following `Except` statements are ignored. If an error is raised, the `Except` statements can “catch” an error. Rather than the error occurring (and the code halting), the code indicated under `Except` executes.
15. Extra topics – These topics are not required for the midterm, but can be optionally used in part 2.

- Recursion: A recursive function typically includes a base case, in which solving the problem consists of a single step and one or more recursive case. For the recursive case, the function solves a small part of the problem and the calls “itself” again, but on a smaller version of the problem. In the following `count_down` function: the function will print 1 if one is the argument of `count_down`. For the recursive case, `recursive_count_down` will print the current number argument and then recursively call itself with one less than the current number as an argument:

```python
def recursive_count_down(number):
    if number == 1:
        print(number)
    else:
        print(number)
        recursive_count_down(number-1)
```

- Object Oriented Programming: A style of programming in which methods (functions) and attributes (variables) are linked to objects which they help characterize and define. Methods with the same names can be linked to different object classes. However, they should have somewhat analagous functions. For example a `roll` method should choose a random number for any `die` object, but depending on the particular class of `die` object, the method may be different. For example, dice can have different numbers of sides (so the choice of numbers can vary). When rolling a “fair” die, one can expect that each possible number will have the same odds of landing on each side. However, it is also possible to implement an “unfair” die and put different weights on the different sides.