**Name:**____________________________________ **NetID:**_______________ 🌵🌵

# CSCI-UA.0002-008 – Midterm Exam #2

## November 16th, 2016

### Instructor: Joseph Versoza

| Ask the person to your left for their first name (leave blank if next to empty seat or wall): | Ask the person to your right for their first name (leave blank if next to empty seat or wall): |
|---|---|
| ____________________________ | ____________________________ |

**Keep this test booklet closed until the class is prompted to begin the exam**

- Computers, calculators, phones, textbooks, and notebooks are **not allowed** during the exam
- Please turn off your phone to avoid disrupting others during the exam
- The back of this cover sheet can be used as scratch paper

1. **Read the code sample** in the first column. **Answer the question** in the second column. (15 points)

**Code**

```
def double_it(s):

    print('A')
    print('B')

    if(len(s) > 0):
        return s * 3

    print('C')


print(double_it('cactus'))
```

**Question**

(4 points)

a) What is the output (shown on screen) of this code?

```
A
B
cactuscactuscactus
```

b) Explain why.

**return stops execution of function**

---

```
noise = 'DRZZZT'

def make_electronic_music(sound):
    noise = 'BLEEP'
    s = '{} BLOP {}'
    print(s.format(noise, sound))

result =
make_electronic_music('BLOOOP')
print(result)
print(noise)
```

What is the output (shown on screen) of this code? (4 points)

**BLEEP BLOP BLOOOP**
**None**
**DRZZZT**

---

```
def find(needle, haystack):
    for item in haystack:
        if item == needle:
            return True
        else:
            return False

print(find(4, [4, 3, 2, 1]))
```

(2 points)

a) **How many times will the body of the the for loop in the function run?**

**once**

b) There 's a logical error in this program; it's supposed to return `True` if the needle exists (can be found) in the list, haystack... or `False` otherwise.

What pair of arguments **should return** `True` based on the description above, but return False instead because of a logical error.

**4, [3, 4, 2, 1]**

---

```
words = ["baz", "qux"]
results = words.append("corge")
print(words)   # a -->
print(results) # b -->

more_words = ["foo", "bar"]
more_results = more_words.pop()
print(more_words)   # c -->
print(more_results) # d -->

stuff = ["blub", "blah", "bbbb"]
print(stuff.index("blah"))  # e -->
```

What is the output (shown on screen) of this code? (5 points)

```
a) ['baz', 'qux', 'corge']
b) None
c) ['foo']
d) bar
e) 1
```

2. Using the following variable declaration, write out the output (error is possible) of the print statements below. (5 points)

```
animals = [['dog', 'cat', 'ant'], ['bat'], ['goat'], ['cow', 'bee']]
```

```
a) print(animals[-3])            (a) ['bat']

b) print(animals[4])             (b) error

c) print(animals[1] * 2)         (d) ['bat', 'bat']

d) print(animals[0][1:100])      (e) ['cat', 'ant']

e) print("animals"[3])           (f) m
```

3. True or False (5 points)

a) (**True** / False) `[2, 5, 6, 1] > [2, 5, 3, 1]`

b) (True / **False**)
```
squashed = [1, 2] + [3, 4]
print(squashed == [[1, 2], [3, 4]])
```

c) (**True** / False) `'foo' in ['baz', 'bar', 'foo']`

d) (**True** / False) Both strings and lists support the repetition operator (multiplication).

e) (True / **False**) Local variables can be accessed outside of the function that they are defined in.

4. The intention of the following code is to create a function that would **take a non-empty list of ints (positive, negative, and mixed positive and negative ints), and return the largest int in that list**. For example, given [1, 2, 3], the function would return 3. Unfortunately, the implementation is slightly off! Test the program below. (4 points)

```
def max_int(numbers):
    largest_number = 0
    for n in numbers:
        if n > largest_number:
            largest_number = n
    return largest_number
```

Write assertions that will cover three unique test cases. There is a logical error in the program; find the error based on the implementation and description. Make your **last assertion** the one that uncovers the logical error.

a) `assert 9 == max_int([1, 5, 9]), 'only positive numbers'`

b) `assert 1 == max_int([1, -5, -9]), 'mixed positive and negative'`

c) `assert -1 == max_int([-1, -5, -9]), 'negative numbers only'`

(This assertion should uncover a logical error)

5. What is the output of the following program? Use the grid to the right of the program as a guide; **each individual character of output can be placed in a single box (an empty box implies a space)**. You do not have to use all of the boxes. (4 points)

```
def make_pattern(rows):
    for row_num in range(rows):
        row = ''
        for col_num in range(rows):
            if col_num == 0 or col_num == row_num:
                row += str(col_num)
            else:
                row += ' ' #space character
        print(row)

make_pattern(5)
```

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 0 | | | | | | | |
| 0 | 1 | | | | | | |
| 0 | | 2 | | | | | |
| 0 | | | 3 | | | | |
| 4 | | | | 4 | | | |

6. You're tired of inadvertently writing Python variable names that aren't valid, so you decide to **write a program that checks the validity of variable names**. To write this program, you'll create a function called **`is_valid_name`**. Your program will then continually ask the user for a variable name... and you'll use your function to determine whether or not it's valid. If the user enters an **invalid variable name 3 times in a row** or if they **enter a valid name**, **stop asking** for a variable name! (12 points)

   a) Create a function called **`is_valid_name`**
      - parameters: a string representing a variable name
      - processing: use the rules below to determine whether or not the variable is valid
      - return: either true or false depending on whether or not the variable name is valid
   b) A valid variable name:
      - starts with only an underscore or a letter
      - is only composed of underscores, letters or numbers
   c) Continually ask the user for a variable names
   d) Use your function to check if it's valid
   e) If the user enters a valid name... or if they enter 3 invalid names, stop asking

   Example usage:

```
print(is_valid_name('1asdf')) # False
print(is_valid_name('#foo'))  # False
print(is_valid_name('asdf1')) # True
print(is_valid_name('_foo'))  # True
print(is_valid_name('f_oo'))  # True
```

   Example Interaction:

```
Variable name plz
> $hello
Variable name plz
> hello
```

```
def is_valid_name(s):
    for c in s:
        if not c.isalnum() and c != '_':
            return False
    if s[0].isnumeric():
        return False
    return True


count = 0
while count < 3:
    if is_valid_name(input('Variable name plz\n> ')):
        break
    else:
        count += 1
```

7. Write a program that asks the user for 2 numbers… and then prints out the factors of all of the numbers between and including the 2 numbers entered. (10 points)
   a) ask for a start number and an end number
   b) only proceed to print out factors when: 1) the end number is at least equal to the start number, 2) both are greater than zero
   c) determine the factors for every number starting from the start number and ending at the end number (inclusive)
   d) print out the factors for a number in the following format:
      `The factors of <number> are: <factor 1> <factor 2> …  <factor n>`
      - the number and its factors should all be on the same line
      - list the factors in **descending order**
      - **separate** each factor with a
      - 
   e) (hint) a factor is a number that divides evenly into another number – for example, the factors of 4 are 4, 2, and 1
   f) (hint) one way to calculate all possible factors is to try every *potential* number that could possibly "fit" into the number

```
Start number
>60
End number
>64
The factors of 60 are: 60 30 20 15 12 10 6 5 4 3 2 1
The factors of 61 are: 61 1
The factors of 62 are: 62 31 2 1
The factors of 63 are: 63 21 9 7 3 1
The factors of 64 are: 64 32 16 8 4 2 1
```

```
start = int(input('Start number\n>'))
end = int(input('End number\n>'))
if start > 0 and end >= start:
    for i in range(start, end + 1):
        print('The factors of', i, 'are: ', end='')
        for j in range(i, 0, -1):
            if i % j == 0:
                print('{} '.format(j), end='')
        print()
```

8. You're a publisher of Star Trek fan fiction. Printed fan fiction doesn't sell in high numbers, so you decide to find creative ways to save money. One solution that you've come up with is to reduce the number of pages of each story by removing every vowel (regardless of case) in the works that you publish (it's totally still understandable without vowels, right?). So... you write a program to do it for you. (8 points total)

   Create a function called **`remove_vowels`**.

   a) It should take **two arguments**, the **string** that will have vowels removed from it, and an additional **boolean** value that determines whether or not the letter y should be considered a vowel
   b) It will **give back a new string** with all vowels removed
   c) If the second argument is True, it will count y as a vowel, and remove it from the incoming string.

   Example output:

```
>>> print(remove_vowels('Typical Picard', True))
Tpcl Pcrd
>>> print(remove_vowels('Typical Picard', False))
Typcl Pcrd
```

```
def remove_vowels(s, include_y):

    new_s = ''
    vowels = 'aeiouAEIOU'
    if include_y:
        vowels = vowels + 'yY'
    for c in s:
        if c not in vowels:
            new_s += c
    return new_s
```

9. Create a function called **`remove_last`**. It should take two arguments: (8 points)

   a) a **string** that specifies what character will separate words (for example, a dash: '-')
   b) another **string** composed of words **separated by the character specified** (for example: '`word1-word2-word3`')
   c) it will **give back everything but the last word**, **in all uppercase** (in the example above, `'WORD1-WORD2'`)
   d) if the separator doesn't occur in the string (including an empty string), then the original string is returned in uppercase
   e) Example usage:

```
print(remove_last('-', 'hi-how-are-you'))    # prints out — HI-HOW-ARE
print(remove_last('X', 'whyXsleep'))         # prints out — WHY
print(remove_last('-', 'hello'))             # prints out - HELLO
print(remove_last('-', ''))                  # does not print out anything
```

```
def remove_last(sep, s)
    count = 0
    i = len(s)
    for letter in s:
        if letter == sep:
            i = count
        count += 1
    return s[:i].upper()
```

10. Write a function called **`too_much_filler`**. (6 points)

   a) It should have **3 parameters**: a list of strings called **`words`**, a string called **`filler`**, and a number called **`limit`**.
   b) It should **return a boolean value**. If the number of times the string, `filler`, occurs in the list, `words`, is greater than the `limit`, give back `True`. Otherwise, give back `False`.
   c) Example usage:

```
>>> print(too_much_filler(['you', 'know', 'like', 'words', 'and','stuff'], 'like', 2)
False
>>> print(too_much_filler(['um', 'try', 'um', 'not', 'saying', 'um'], 'um', 2))
True
```

```
def too_much_filler(words, filler, limit):
  return words.count(filler) > limit
```

**Name:**____________________________________ **NetID:**______________

# Scratch Paper and Reference Material

## ASCII Chart

| Char | Dec | Char | Dec | Char | Dec | Char | Dec |
|---|---|---|---|---|---|---|---|
| (nul) | 0 | (sp) | 32 | @ | 64 | ` | 96 |
| (soh) | 1 | ! | 33 | A | 65 | a | 97 |
| (stx) | 2 | " | 34 | B | 66 | b | 98 |
| (etx) | 3 | # | 35 | C | 67 | c | 99 |
| (eot) | 4 | $ | 36 | D | 68 | d | 100 |
| (enq) | 5 | % | 37 | E | 69 | e | 101 |
| (ack) | 6 | & | 38 | F | 70 | f | 102 |
| (bel) | 7 | ' | 39 | G | 71 | g | 103 |
| (bs) | 8 | ( | 40 | H | 72 | h | 104 |
| (ht) | 9 | ) | 41 | I | 73 | i | 105 |
| (nl) | 10 | * | 42 | J | 74 | j | 106 |
| (vt) | 11 | + | 43 | K | 75 | k | 107 |
| (np) | 12 | , | 44 | L | 76 | l | 108 |
| (cr) | 13 | - | 45 | M | 77 | m | 109 |
| (so) | 14 | . | 46 | N | 78 | n | 110 |
| (si) | 15 | / | 47 | O | 79 | o | 111 |
| (dle) | 16 | 0 | 48 | P | 80 | p | 112 |
| (dc1) | 17 | 1 | 49 | Q | 81 | q | 113 |
| (dc2) | 18 | 2 | 50 | R | 82 | r | 114 |
| (dc3) | 19 | 3 | 51 | S | 83 | s | 115 |
| (dc4) | 20 | 4 | 52 | T | 84 | t | 116 |
| (nak) | 21 | 5 | 53 | U | 85 | u | 117 |
| (syn) | 22 | 6 | 54 | V | 86 | v | 118 |
| (etb) | 23 | 7 | 55 | W | 87 | w | 119 |
| (can) | 24 | 8 | 56 | X | 88 | x | 120 |
| (em) | 25 | 9 | 57 | Y | 89 | y | 121 |
| (sub) | 26 | : | 58 | Z | 90 | z | 122 |
| (esc) | 27 | ; | 59 | [ | 91 | { | 123 |
| (fs) | 28 | < | 60 | \ | 92 | \| | 124 |
| (gs) | 29 | = | 61 | ] | 93 | } | 125 |
| (rs) | 30 | > | 62 | ^ | 94 | ~ | 126 |
| (us) | 31 | ? | 63 | _ | 95 | (del) | 127 |

## String Methods

```
capitalize
count
endswith
find
format
index
isalnum
isalpha
isdecimal
isdigit
islower
isnumeric
isprintable
isspace
istitle
isupper
join
lower
replace
split
startswith
strip
title
upper
```

## List Methods

```
append
count
extend
index
insert
pop
remove
reverse
sort
```