CS 201R The Joy of Programming

Winter 2022

Practice Midterm

Name

Write your full name here

Directions

This test is closed book and closed note. You may use blank sheets of paper to draw on or to plan out your answer.

By taking this test, you agree to be honest and do your own work.

Reference

Bit

```
bit = Bit.load(filename)
bit.front_clear()
bit.left_clear()
bit.right_clear()
bit.move()
bit.left()
bit.right()
bit.paint('red')
bit.get_color()
bit.erase()
```

Images

```
image = Image(filename)
image = Image.blank(width, height)
image.show()
image.width
image.height
# foreach loop
for pixel in image:

# range/y/x loop
for x in range(image.width):
    for y in range(image.height):
        pixel = image.get_pixel(x, y)
```

Strings

```
isalpha()
isdigit()
isspace()
isupper()
islower()
index()
find()
upper()
lower()
strip()
s[start:end] – slicing
# looping
for character in s:
for i in range(len(s)):
```

General

```
len()
int()
str()
range()
```

What does this code print?

For this section, tell us what the code would print. We encourage you to use paper for drawing or scratch calculations. You should write exactly what the code should print. For example, the following code:

```
for i in range(3):
    print(i)
```

Would print:

0
1
2

1. What does the following code print?

```
n = 20
while n > 5:
    print(n)
    n = n - 3
```

20
17
14
11
8

2. What does the following code print?

```
n = 10
while n < 30:
    if n % 10 == 0:
      n += 8
    else:
        n += 1
    print(n)
```

18
19
20
28
29
30

3. What does the following code print?

```
word = "And see that ye have faith, hope, and charity"
result = ""
start_accumulating = False
for character in word:
    if character == 'f':
        start_accumulating = True
    if start_accumulating:
        result += character
print(result)
```

faith, hope, and charity

4. What does the following code print?

```
def function1(s):
    result = ""
    for character in s:
        if character.isdigit():
            result += character
    return result


def function2(s):
    result = ""
    for character in s:
        result += character + character
    return result


my_string = "23 plus 45 equals 68"
result1 = function1(my_string)
result2 = function2(result1)
print(result2)
```

223344556688

5. What does the following code print?

```
s1 = "my facebook life"
s2 = "this restaurant is terrible"
result = s1[3:12] + s2[16:]
print(result)
```

facebook is terrible

6. What does the following code print?

```
def get_chunky(s):
    position = s.find("chunky")
    if position >= 0:
        return s[position+7:]
    else:
        return "not chunky"


print(get_chunky("chunky monkey"))
print(get_chunky("I don't like chunky peanut butter"))
print(get_chunky("That looks kinda funky"))
```

monkey
peanut butter
not chunky

7. What does the following code print?

```
for y in range(2, 4):
    for x in range(10, 12):
        print(x, y)
```

10 2
11 2
10 3
11 3

Doctests

For this section, write doctests for the code we have provided.

8. Write four doctests for the following function, called **complex_word()**. Your doctests must meet the following criteria:
    a. One doctest that returns true
    b. Three doctests that return false
    c. Each doctest that returns false must test a different reason why the word is not complex
    d. For each invalid doctest include a comment that tells us the reason why that is not complex

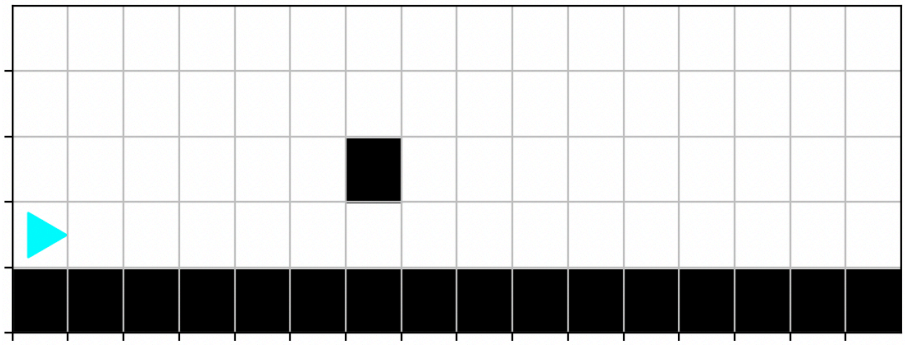
```
def complex_word(word):
    """ Checks if a word is complex. It must be have three different
vowels
    and five different consonants. It can only have alphabetic
characters.
    :param word: a word
    :return: True if the word is complex, False otherwise
    """
    # code not shown
```

```
>>> complex_word("hello")                # too short
False
>>> complex_word("lackadaisical")        # only 2 unique vowels
False
>>> complex_word("loquacious")           # only 4 unique consonants
False
>>> complex_word("accomplished")
True
```

Code

For this section, write code. We will not be running your code, so don't worry too much about syntax! We just want to see if you have the right idea.

Write a function called **green_squares(bit)** that will paint green squares to the right of the black box. This function takes one parameter, which is a bit world.



```
def green_squares(bit):
    while bit.left_clear():
        bit.move()
    bit.move()
    bit.left()
    bit.move()
    bit.right()
    while bit.front_clear():
        bit.paint("green")
        bit.move()
    bit.paint("green")
```

9. Given the function **repeat()**, write a new version of **repeat()** that can repeat text an arbitrary number of times.

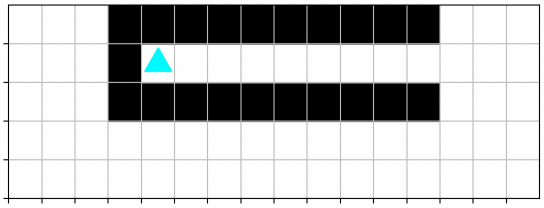
```
def repeat(text):
    result = ""
    for i in range(3):
        result += text
    return result
```

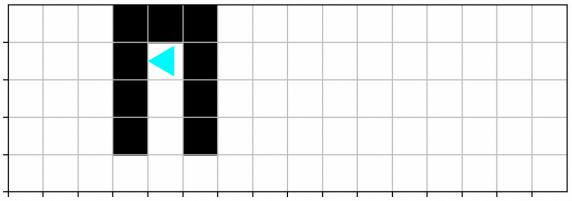
```
def repeat(text, n_times):
    result = ""
    for i in range(n_times):
        result += text
    return result
```

Which function is correct?

For this section, we want to know which function is the correct code for a given problem.

10. Which code block will move Bit out of a tunnel no matter which way it is facing? Circle the letter next to the correct version. Here are two possible tunnels:





a. Version (a)

```
def solve_tunnel_problem(bit):
    while not bit.front_clear():
        bit.move()
    while not bit.right_clear():
        bit.move()
```

b. Version (b)

```
def solve_tunnel_problem(bit):
    while not bit.front_clear():
        bit.left()
    while not bit.right_clear():
        bit.move()
```

c. Version (c)

```
def solve_tunnel_problem(bit):
    while not bit.front_clear():
        bit.right()
    while bit.right_clear():
        bit.move()
```

d. Version (d)

```
def solve_tunnel_problem(bit):
    while bit.front_clear():
        bit.left()
    while not bit.right_clear():
        bit.move()
```

11. Which function will copy an **image** into a **new_image**, offset downwards by **height** pixels? Circle the letter next to the correct version.

a. Version (a)

```
def copy_downward(new_image, image, height):
    for x in range(image.height):
        for y in range(image.width):
            pixel = image.get_pixel(x, y)
            new_pixel = new_image.get_pixel(x, y + height)
            new_pixel.red = pixel.red
            new_pixel.green = pixel.green
            new_pixel.blue = pixel.blue
```

b. Version (b)

```
def copy_downward(new_image, image, height):
    for x in range(image.width):
        for y in range(image.height):
            pixel = image.get_pixel(x, y)
            new_pixel = new_image.get_pixel(x + height, y)
            new_pixel.red = pixel.red
            new_pixel.green = pixel.green
            new_pixel.blue = pixel.blue
```

c. Version (c)

```
def copy_downward(new_image, image, height):
    for x in range(image.width):
        for y in range(image.height):
            pixel = image.get_pixel(x, y)
            new_pixel = new_image.get_pixel(x, y + height)
            pixel.red = new_pixel.red
            pixel.green = new_pixel.green
            pixel.blue = new_pixel.blue
```

d. Version (d)

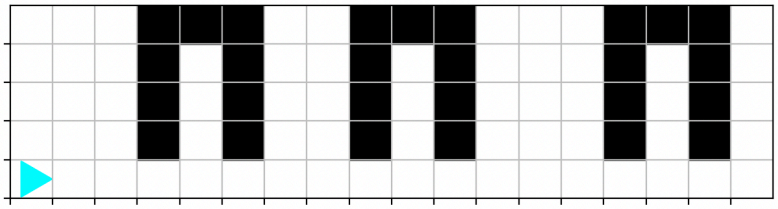
```
def copy_downward(new_image, image, height):
    for x in range(image.width):
        for y in range(image.height):
            pixel = image.get_pixel(x, y)
            new_pixel = new_image.get_pixel(x, y + height)
            new_pixel.red = pixel.red
            new_pixel.green = pixel.green
            new_pixel.blue = pixel.blue
```

Decomposition

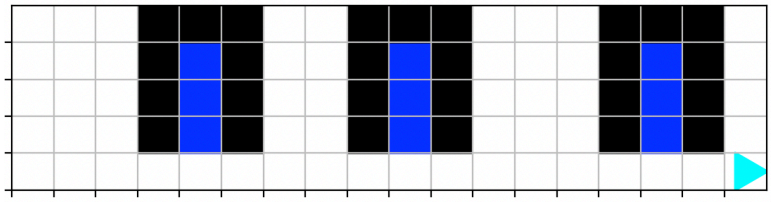
For this section, you will decompose a problem into a set of functions. We want you to write function definitions, but not any code for the function body.

12. Bit needs to fill the tunnels with blue.

Starting world



Desired outcome



Decompose this problem into a set of functions you would use to solve it. In the space below, (a) write the name and parameters of each function, and (b) describe the pre- and post-conditions for each function.

move_to_next_tunnel(bit)
- This function moves to the next tunnel
- Pre-condition: Bit is pointing to the right and the space above it is clear
- Post-condition: Bit is pointing up and the spaces above it are in the middle of the tunnel

fill_tunnel(bit)
- This function fills one tunnel
- Pre-condition: Bit is pointing up and the spaces above it are in the middle of the tunnel
- Post-condition: Bit is pointing right and is one space past the tunnel

fill_tunnels(bit)
- This function fills all the tunnels
- Pre-condition: Bit is in the starting position, pointing right, with space above it
- Post-condition: Bit is in the finishing position, pointing right, past the last tunnel

-

13. You want to create a collection of two images:

Decompose this problem into a set of functions you would use to solve it. In the space below, (a) write the name and parameters of each function, and (b) describe what each function does.

copy_image(new_image, image, start_x, start_y)
- Copies an image into a new_image. The copied image starts at the coordinates (start_x, start_y) in the new image.

fill_with_black(new_image)
- Fills an image with black

make_one_by_two_image(image1, image2, width)
- Makes a one-by-two image, with image 1 on top of image 2. Uses the width argument to indicate how much space should be around the borders and between the images