Question 1. True or False

Circle **T** if the statement is true, otherwise circle **F** if the statement is false.

1. Rust enum is an example of algebraic type. [✓ T] F
2. The purpose of generic programming is code reuse across different data types. [✓ T] F
3. Box forces the contained object to be heap allocated. [✓ T] F
4. Lifetime parameter must be added to all structures with non-static references. [✓ T] F
5. Lifetime elision optimizes the binary by eliminating the need to copy parameters. T [✓ F]

Question 2. Multiple Choices

Pick all answer(s) that are correct.

a) Which of the following statements are true regarding ownership and borrowing? [4 marks]

[✓ i.] You cannot access a variable after it has been moved.

[✓ ii.] You cannot change a variable while there are immutable references to it.

[✓ iii.] You can mutably borrow multiple non-overlapping slices of a sequence at one time.

[✓ iv.] You can have multiple immutable references to a variable at one time.

[✓ v.] You can change an immutable variable into a mutable one after moving it.

b) Which of the following traits have default implementation?

[✓ i.] PartialEq

ii. Display

iii. Not

[✓ iv.] Clone

[✓ v.] Default

Question 3. Short Answer

a) Describe two similarities and two differences between Python mixins and Rust traits.

Similarities:

- Both can supply default implementation.
- A structure/class can have multiple mixins or multiple traits.
- Both can be used for code reuse.

Differences:

- Traits cannot have data fields, mixins can.
- Traits are not affected by order of inheritance, mixins can be.
- Traits are supposed to be implemented (is-a), mixins are supposed to be included (has-a).
- Traits provides an interface, mixins only provides implementation.

b) If you try to compile the code below, it will generate errors/warnings. Identify the line(s) where error(s) would occur by specifying the line number(s), and if possible, write the code that would fix it. If a print statement would cause the error, write “DELETE” in the table.

```
1    fn create_model(company: String) -> String {
2        let model = String::from("Default, ");
3        if company == "Boeing" {
4            model = String::from("747, ");
5        }
6        else if company == "Airbus" {
7            model = String::from("A330, ");
8        }
9        model.push_str(&company);
10       model
11   }
12
13   fn create_airbus() -> &str {
14       let some_string = "Airbus";
15       &some_string
16   }
17
18   fn create_plane() {
19       let plane1 = create_airbus();
20       let plane2 = String::from("Boeing");
21       let plane3 = create_model(plane1);
22       let plane4 = create_model(plane2);
23       println!("Plane 1: {}", plane1);
24       println!("Plane 2: {}", plane2);
25       println!("Plane 3: {}", plane3);
26       println!("Plane 4: {}", plane4);
27   }
28
29   fn main() {
30       let mut height = 100;
31       let my_plane = "777, Boeing";
32       {
33           let b = my_plane;
34       }
35       println!("Plane is a: {}", b);
36       println!("Currently at {}", height);
37       {
38           let c = 2;
39           height += c;
40       }
41       println!("Ascending to {}", height);
42       create_plane();
43   }
```

Line #	Corrected line or "DELETE"
2	let mut model = String::from("Default, ");
13	fn create_airbus () -> &'static str {
21	let plane3 = create_model(plane1.to_string());
24	DELETE
35	DELETE
(33)	(optional) leads to a warning of unused variable 'b'

c) Assuming all the errors were fixed in the above program, write the output of the program.

```
Currently at 100
Ascending to 102
Plane 1: Airbus
Plane 3: A330, Airbus
Plane 4: 747, Boeing
```

Question 4. Programming Questions

a) Write structure named Matrix which supports a 2x2 matrix of type f64, and implements the determinant method, the transpose method, and the inverse method. The transpose method should modify the existing matrix, but the inverse method should return a new matrix if the matrix is invertible, otherwise it should return None (Hint: use Option<T>). Write a new static method which creates and initializes the matrix. Implement the Display trait for Matrix.

```
struct Matrix {
    values: [f64; 4],
}

impl Matrix {
    fn new(a: f64, b: f64, c: f64, d: f64) -> Matrix {
        Matrix { values: [a, b, c, d] }
    }

    fn determinant(&self) -> f64 {
        self.values[0] * self.values[3] –
        self.values[1] * self.values[2]
    }

    fn transpose(&mut self) {
        self.values.swap(1, 2);
    }

    fn inverse(&self) -> Option<Matrix> {
        let det = self.determinant();
        if det == 0. { None }
        else {
            Some(Matrix::new(self.values[3]/det, -self.values[1]/det,
                             -self.values[2]/det,  self.values[0]/det))
        }
    }
}

impl fmt::Display for Matrix {
    fn fmt(&self, f: &mut fmt::Formatter) -> fmt::Result {
        write!(f, "[ {:.2}, {:.2} \n {:.2}, {:.2} ]", self.values[0],
            self.values[1], self.values[2], self.values[3])
    }
}
```

b) Write a generic function, sum_of_squares, which calculates the sum of squares of a generic array slice. Hint: you may need some trait bounds.

```
fn sum_of_squares<T: Default + AddAssign + Mul<Output=T> + Copy>
    (list: &[T]) -> T
{
    let mut sum = Default::default();
    for &n in list {
        sum += n * n;
    }
    sum
}
```

c) Write a function that takes two string slices, text and word, and return a vector of all occurrences of the word in text in string slices. Note, you may need to “fix” the function signature. You may assume the text to consist of only ascii characters. Hint: look up the find method for a string slice.

```
fn findall(text: &str, word: &str) -> Vec<&str>
```

```
fn findall<'a, 'b>(text: &'a str, word: &'b str) -> Vec<&'a str> {
    let mut ret = vec![];
    let mut s = text;

    loop {
        let i = match s.find(word) {
            Some(i) => i,
            None => break,
        };

        ret.push(&s[i..i+word.len()]);
        s = &s[i+word.len()..];
    }

    ret
}
```

d) Implement a *sorted* singly linked list of i64 elements where the insert method will automatically place the new element such that the list remains in ascending order. Complete the remove method such that all elements with the specified value is removed from the list. Lastly, implement the Display trait to print all elements of the list.

```
#[derive(Clone)]
enum List { Node(i64, Box<List>), Tail, }

impl fmt::Display for List {
    fn fmt(&self, f: &mut fmt::Formatter<'_>) -> fmt::Result {
        match self {
            List::Node(v, next) => {
                write!(f, "{} -> ", v)?;
                write!(f, "{}", *next)
            },
            List::Tail => write!(f, "[]"),
        }
    }
}

fn insert_recursive(curr: & mut List, value: i64) {
    if let List::Node(other, next) = curr {
        if value > *other {
            insert_recursive(&mut *next, value);
        } else {
            let next = Box::new(curr.clone());
            *curr = List::Node(value, next);
        }
    }
    else {
        *curr = List::Node(value, Box::new(List::Tail));
    }
}

fn remove_recursive(curr: & mut List, value: i64) {
    if let List::Node(other, next) = curr {
        if value >= *other {
            remove_recursive(& mut *next, value);
    }
        if value == *other { *curr = *next.clone(); }
    }
}
```