1. PYTH9: Map, Filter, Reduce, Lambdas

```
def strip_characters(sentence, chars_to_remove):
    return "".join([char for char in sentence if char not in chars_to_remove])
```

2. PYTH10: Closures

Yes, Python supports closures. Closures occur when a function retains access to the variables from its lexical scope, even after the outer function has finished executing.

```
def outer_function(x):
    def inner_function(y):
        return x + y  # inner_function uses x from outer_function's scope
    return inner_function

# Create a closure by calling outer_function with a specific value for x
closure = outer_function(10)

# Now, calling closure will still have access to x = 10 from outer_function
print(closure(5))  # Output: 15
print(closure(7))  # Output: 17
```

3. PYTH11: List Comprehensions

a.

```
def convert_to_decimal(bits):
    exponents = range(len(bits)-1, -1, -1)
    nums = [bit * (2 ** exp) for bit, exp in zip(bits, exponents)]
    return reduce(lambda acc, num: acc + num, nums)
```

b.

```
def parse_csv(lines):
    return [(word, int(num)) for line in lines for word, num in [line.split(",")]]
```

c.

```
def unique_chars(sentence):
    return {char for char in sentence}
```

d.

```python
def squares_dict(lower, upper):
    return {num: num ** 2 for num in range(lower, upper + 1)}
```

4. HASK17: Algebraic Data Types, Recursion
    a.

```cpp
int longestRun(vector<bool> boolVec) {
    int longestAnswer = 0;
    int answer = 0;

    for (int i = 0; i < boolVec.size(); i++) {
        if (boolVec[i]) {
            answer++;
        } else {
            longestAnswer = max(longestAnswer, answer);
            answer = 0;
        }
    }

    return max(longestAnswer, answer);
}
```

    b.

```haskell
longest_run :: [Bool] -> Int
longest_run xs = maximum (0 : map length (filter (all (== True)) (group xs)))
```

    c.

```cpp
unsigned maxTreeValue(Tree* root) {
    if (root == nullptr) {
        return 0;
    }

    unsigned maxValue = root->value;
    stack<Tree*> nodes;
    nodes.push(root);

    while (!nodes.empty()) {
        Tree* current = nodes.top();
        nodes.pop();

        // Update maxValue if the current node's value is greater
        maxValue = max(maxValue, current->value);

        // Add all children to the stack
        for (Tree* child : current->children) {
            if (child != nullptr) {
                nodes.push(child);
            }
        }
    }

    return maxValue;
}
```

d.

```
data Tree = Empty | Node Integer [Tree]

max_tree_value :: Tree -> Integer
max_tree_value Empty = 0
max_tree_value (Node value subtrees) = maximum (value : map max_tree_value subtrees)
```

5. HASK19: Tail Recursion
    a.

```
sumSquares :: Integer -> Integer
sumSquares 0 = 0
sumSquares n = n^2 + sumSquares (n - 1)
```

b.

```
sumSquares :: Integer -> Integer
sumSquares n = helper n 0
    where
    helper 0 acc = acc
    helper n acc = helper (n - 1) (acc + n ^ 2)
```

6. DATA1: Static vs Dynamic Typing
    a. Most likely dynamically typed because you can switch between the integer and string types for user_id.
    b.
        i. Static since when you print only 3 for 3.5, meaning that it got coerced into being an integer.
        ii. Conversion since we are doing this on a primitive. The conversion would be narrowing since we are converting a larger type (double) into an int
        iii. Would likely be an error since we can't cast a double to an int
        iv. It's dynamic since you can change the type from int to string and the compiler doesn't care
7. DATA2: Strong vs Weak Typing
    a. It demonstrates the importance of a strongly typed language, since we have no clue how 123 acts if we try to perform an operation that may not be explicitly defined for it.
    b. I'm going to be honest I'm not too sure since I don't know what union is doing and why "simple union" is not being used

8. DATA3: Casting and Conversions
    a.
        Line 6: narrowing cast since we static_cast a long to int
        Line 7: widening conversion since we return an integer as a larger class (double)
        Line 11: widening cast since we are returning a person superclass when passed in a student object
        Line 15: widening cast since you are casting me into my person superclass
        Line 16: narrowing conversion since we are converting int into a short class
        Line 17: widening conversion since we pass in a short as a double
        Line 18: widening conversion since we widen s to double
        Line 19: narrowing cast since we cast a person to a student

    b.
        Line 10: widening conversion, turning an integer to a float
        Line 21: widening conversion, adds int and float
        Line 22: widening cast, converts small integer to a student object