Best Coding Practices

Lecture Flow

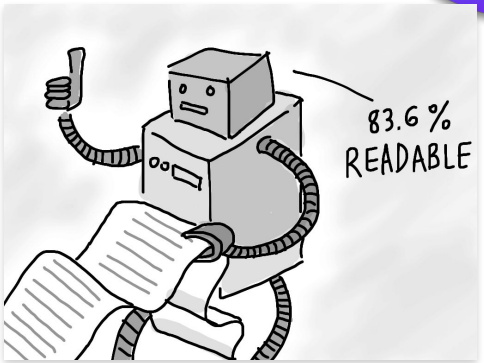
- Why Best Practices?
- Meaningful Naming
- Writing Modular Code
- Consistent Indentation
- Essential Comments
- The Good the Bad and the Ugly
- Recommendations

Why Best Practices?

Why Best Practices?

- Do things faster
- Reduce bugs
- Concise code
- Readability

```
while(alive) {
    eat();
    code();
    sleep();
    repeat();
}
```



A2SV

Meaningful Naming

Meaningful Naming

Bad Practice

```python
class Solution:
    def isPal(self, s: str) -> bool:
        l = 0
        r = len(s)-1

        while l < r :
            while l < r and not s[l].isalnum():
                l += 1
            while l < r and not s[r].isalnum():
                r -= 1
            if s[l].lower() != s[r].lower():
                return False
            l += 1
            r -= 1

        return True

```

Good Practice

```python
class Solution:
    def isPalindrome(self, s: str) -> bool:

        left,right = 0,len(s)-1
        while left < right:
            while left < right and not s[left].isalnum():
                left += 1
            while left < right and not s[right].isalnum():
                right -= 1
            if s[left].lower() != s[right].lower():
                print(s[left],s[right])
                return False
            left += 1
            right -= 1
        return True
```

Meaningful Naming

1. Interviewers seriously care
2. Reduces ambiguity and bugs
3. Helps debugging and readability

Writing Modular Code

Writing Modular Code

Bad

```
class Solution:
    def maxSum(self, grid: List[List[int]]) -> int:
        def calculateSum(top, right):
            curSum = sum(grid[top][right - 2:right + 1]) + grid[top+1][right -1] + sum(grid[top +
2 ][right - 2:right + 1])
            return curSum
        top, right = 0, 2
        maxSum = 0
        while top + 2 < len(grid):
            curSum = 0
            right = 2
            while right < len(grid[0]):
                curSum += calculateSum(top, right)
                right += 1
                maxSum = max(curSum, maxSum)
                curSum = 0
            top += 1
        return maxSum
```

Good

```
class Solution:
    def calculateSum(self, top, right):
        top_sum = sum(grid[top][right - 2:right + 1])
        bottom_sum = sum(grid[top + 2 ][right - 2:right + 1])
        curSum =  top_sum + grid[top+1][right -1] + bottom_sum
        return curSum

    def maxSum(self, grid: List[List[int]]) -> int:
        top = 0
        right = 2
        maxSum = 0
        while top + 2 < len(grid):
            curSum = 0
            right = 2
            while right < len(grid[0]):
                curSum += self.calculateSum(top, right)
                right += 1
                maxSum = max(curSum, maxSum)
                curSum = 0
            top += 1
        return maxSum
```

Writing Modular Code

- Helps transiting from solution to code
- Helps seeing the commonalities between similar problems
- Interviewers seriously care
- Reduces bugs
- Helps debugging
- Reusable

Consistent Indentation

Consistent Indentation

Bad Practice

```
class Solution:
    def leastInterval(self, tasks: List[str], items_count: int) -> int:
        counts = [0] * 26
        for i in tasks: counts[ord(i) - ord('A')] += 1
        max_, max_count = max(counts), 0
        for count in counts: max_count += (count == max_)
        return max((max_ - 1) * (items_count+1) + max_count, len(tasks))
```

Good Practice

```
class Solution:
    def leastInterval(self, tasks: List[str], items_count: int) -> int:
        counts = [0] * 26
        for i in tasks:
            counts[ord(i) - ord('A')] += 1
        max_, max_count = max(counts), 0
        for count in counts: max_count += (count == max_)
        return max((max_ - 1) * (items_count+1) + max_count, len(tasks))
```

Consistent Indentation

Bad Practice

```
class Solution:
    def leastInterval(self, tasks: List[str], items_count: int) -> int:
        counts = [0] * 26
        for i in tasks:
            counts[ord(i) - ord('A')] += 1
        max , max_count = max(counts), 0
        for count in counts: max_count += (count == max_)
        return max((max_ - 1) * (items_count+1) + max_count, len(tasks))
```

Good Practice

```
class Solution:
    def leastInterval(self, tasks: List[str], items_count: int) -> int:
        counts = [0] * 26
        for i in tasks:
            counts[ord(i) - ord('A')] += 1
        max_, max_count = max(counts), 0
        for count in counts:
            if count == max_:
                max_count += 1
        return max((max_ - 1) * (items_count+1) + max_count, len(tasks))
```

A2SV

Consistent Indentation

Bad Practice

```
class Solution:
    def leastInterval(self, tasks: List[str], items_count: int) -> int:
        counts = [0] * 26
        for i in tasks:
            counts[ord(i) - ord('A')] += 1
        max_, max_count = max(counts), 0
        for count in counts:
            if count == max_:
                max_count += 1
        return max((max_ - 1) * (items_count+1) + max_count, len(tasks))
```

Good Practice

```
class Solution:
    def leastInterval(self, tasks: List[str], items_count: int) -> int:
        counts = [0] * 26
        for i in tasks:
            counts[ord(i) - ord('A')] += 1

        max_, max_count = max(counts), 0
        for count in counts:
            if count == max_:
                max_count += 1

        return max((max_ - 1) * (items_count+1) + max_count, len(tasks))
```

Consistent Indentation

Bad Practice

```
def isValid(row, col, grid):
    return 0 <= row < len(grid) and 0 <= col < len(grid[0]) and grid[row][col] == "."
```

Good Practice

```
def isValid(row, col, grid):
    if not (0 <= row < len(grid) and 0 <= col < len(grid[0]):
        return False

    if grid[row][col] != ".":
        return False

    return True
```

Consistent Indentation

Bad Practice

```
class Solution:
    def maxSum(self, grid: List[List[int]]) -> int:
        row,col=len(grid),len(grid[0])

        for i in range(row):
            tot=0
            for j in range(col):
                grid[i][j]=tot+grid[i][j]
                tot=grid[i][j]

        ans=0
        for i in range(row-2):
            for j in range(col-2):
                if j>=1:ans=max(ans,grid[i][j+2]-grid[i][j-1]+grid[i+2][j+2]-grid[i+2][j-1]+grid[i+1][j+1]-grid[i+1][j])
                else:ans=max(ans,grid[i][j+2]+grid[i+2][j+2]+grid[i+1][j+1]-grid[i+1][j])

        return ans
```

Consistent Indentation

Good Practice

```python
class Solution:
    def maxSum(self, grid: List[List[int]]) -> int:
        row = len(grid)
        col = len(grid[0])

        for i in range(row):
            total = 0
            for j in range(col):
                grid[i][j] = total + grid[i][j]
                total = grid[i][j]

        ans = 0
        for i in range(row-2):
            for j in range(col-2):
                if j>=1:
                  ans= max(ans,
                           grid[i][j+2]-grid[i][j-1]+
                           grid[i+2][j+2]-grid[i+2][j-1]+
                           grid[i+1][j+1]-grid[i+1][j])

                else:
                  ans= max(ans,
                          grid[i][j+2]+grid[i+2][j+2]+
                          grid[i+1][j+1]-grid[i+1][j])

        return ans
```

Consistent Indentation

- Increases code quality and readability
- Interviewers **seriously** care
- Reduces bugs
- Helps debugging

Essential Comments

Essential Comments

```python
class Solution:
    def loudAndRich(self, richer: List[List[int]], quiet: List[int]) -> List[int]:
        people_size = len(quiet)
        graph = [[] for _ in range(people_size)]
        indegree = [0 for _ in range(people_size)]

        quieter_person = [person for person in range(people_size)]
        queue = deque()

        for rich, poor in richer:
            graph[rich].append(poor)
            indegree[poor] += 1

        #push nodes with 0 degrees into queue
        for person in range(people_size):
            if indegree[person] == 0:
                queue.append(person)

        while(queue):
            current_person = queue.popleft()

            for neighbour in graph[current_person]:

                #if parent node having more money is quieter , update
                if quiet[quieter_person[current_person]] <= quiet[quieter_person[neighbour]]:
                    quieter_person[neighbour] = quieter_person[current_person]

                indegree[neighbour] -= 1

                if indegree[neighbour] == 0:
                    queue.append(neighbour)

        return quieter_person
```

Essential Comments

- Helps understanding
- Shows your care to code quality
- Impresses the interviewer

The Good the Bad and the Ugly

```python
# Check if i, j coordinate is in boundaries of the matrix
def isInside(i, j, n, m):
    if i < 0 or i >= n or j < 0 or j >= m:
        return False
    return True

# Runs dfs and returns true if there is a rectangle
def dfs(i, j, n, m, grid, start_i, start_j, start_letter, nodes_count, directions):

    # Mark current cell as visited
    prev_letter = grid[i][j]
    grid[i][j] = '*'

    answer = False

    for direction in directions:
        ni = i + direction[0]
        nj = j + direction[1]

        # If we hit to the start point, return true
        if ni == start_i and nj == start_j and nodes_count + 1 >= 4:
            return True

        # If new explored cell is inside and satisfies condition, go to that cell
        if isInside(ni, nj, n, m) and start_letter == grid[ni][nj]:
            answer = dfs(ni, nj, n, m, grid, start_i, start_j, start_letter, nodes_count + 1, directions)
            if answer:
                break

    # Revert back the letter, backtracking
    grid[i][j] = prev_letter

    return answer

def main():
    n, m = list(map(int, input().split()))

    grid = []
    for _ in range(n):
        row = [ *input().strip() ]
        grid.append( row )

    directions = [[0, 1], [1, 0], [-1, 0], [0, -1]]

    for i in range(n):
        for j in range(m):
            is_possible = dfs(i, j, n, m, grid, i, j, grid[i][j], 0, directions)
            if is_possible:
                print("Yes")
                sys.exit(0)

    print("No")

if __name__ == "__main__":
    main()
```

```python
def dfs(i, j, n, m, g, s_i, s_j, s_l, cnt, dirs):
    p_l, g[i][j] = g[i][j], '*'
    ans = False
    for d in range(1, 5):
        ni, nj = i + dirs[d], j + dirs[d - 1]
        if ni == s_i and nj == s_j and cnt + 1 >= 4:
            return True
        if (ni >= 0 and ni < n and nj >= 0 and nj < m) and s_l == g[ni][nj]:
            ans = dfs(ni, nj, n, m, g, s_i, s_j, s_l, cnt + 1, dirs)
            if ans:
                break
    g[i][j] = p_l
    return ans

def main():
    n, m = list(map(int, input().split()))
    g = []
    for i in range(n):
        row = [*input().strip()]
        g.append(row)
    dirs = [0, 1, 0, -1, 0]
    for i in range(n):
        for j in range(m):
            if dfs(i, j, n, m, g, i, j, g[i][j], 0, dirs):
                print("Yes")
                sys.exit(0)
    print("No")

if __name__ == "__main__":
    main()
```

The Good

```python
# Check if i, j coordinate is in boundaries of the matrix
def isInside(i, j, n, m):
    if i < 0 or i >= n or j < 0 or j >= m:
        return False
    return True

# Runs dfs and returns true if there is a rectangle
def dfs(i, j, n, m, grid, start_i, start_j, start_letter, nodes_count, directions):

    # Mark current cell as visited
    prev_letter = grid[i][j]
    grid[i][j] = '*'

    answer = False

    for direction in directions:
        ni = i + direction[0]
        nj = j + direction[1]

        # If we hit to the start point, return true
        if ni == start_i and nj == start_j and nodes_count + 1 >= 4:
            return True

        # If new explored cell is inside and satisfies condition, go to that cell
        if isInside(ni, nj, n, m) and start_letter == grid[ni][nj]:
            answer = dfs(ni, nj, n, m, grid, start_i, start_j, start_letter, nodes_count + 1, directions)
            if answer:
                break

    # Revert back the letter, backtracking
    grid[i][j] = prev_letter

    return answer

def main():
    n, m = list(map(int, input().split()))

    grid = []
    for _ in range(n):
        row = [ *input().strip() ]
        grid.append( row )

    directions = [[0, 1], [1, 0], [-1, 0], [0, -1]]

    for i in range(n):
        for j in range(m):
            is_possible = dfs(i, j, n, m, grid, i, j, grid[i][j], 0, directions)
            if is_possible:
                print("Yes")
                sys.exit(0)

    print("No")

if __name__ == "__main__":
    main()
```

The Bad

```python
def dfs(i, j, n, m, g, s_i, s_j, s_l, cnt, dirs):
    p_l, g[i][j] = g[i][j], '*'
    ans = False
    for d in range(1, 5):
        ni, nj = i + dirs[d], j + dirs[d - 1]
        if ni == s_i and nj == s_j and cnt + 1 >= 4:
            return True
        if (ni >= 0 and ni < n and nj >= 0 and nj < m) and s_l == g[ni][nj]:
            ans = dfs(ni, nj, n, m, g, s_i, s_j, s_l, cnt + 1, dirs)
            if ans:
                break
    g[i][j] = p_l
    return ans

def main():
    n, m = list(map(int, input().split()))
    g = []
    for i in range(n):
        row = [*input().strip()]
        g.append(row)
    dirs = [0, 1, 0, -1, 0]
    for i in range(n):
        for j in range(m):
            if dfs(i, j, n, m, g, i, j, g[i][j], 0, dirs):
                print("Yes")
                sys.exit(0)
    print("No")

if __name__ == "__main__":
    main()
```

A2SV

ThE Ugly.

```ruby
#changes case, I like this function name better
def change_casing(str)str.swapcase; end
```

```
//quick bf interpreter
bF=(A,B,C,D,E,F,G,H)=>{for(E=[C=D=F=0],G='',H={'>':_=>++F<E.length||E.push(0),'<':_=>--F,'+':_=>++E[F]<256||(E[F]=
String.fromCharCode(E[F]),',':_=>E[F]=B.charCodeFt(D++),'[':T=>{if(!E[F])for(T=1;T;)'['==A[++C]?++T:']'==A[C]&&--
for(T=1;T;)']'==A[--C]?++T:'['==A[C]&&--T}};C<A.length;++C)H[A[C]]();
return G}
```

```javascript
var func = (function func(x) {
  collection = []
  for (let thing = 0; thing < 122; ++thing) {
    if (x[thing])
      collection.push('this is' + x[thing] + " ")
    else
      break;
  }
})
collection
```

```
#get divs
divs=lambda num: [x for x in range(2,int(num/2)+1) if num%n < 1] o
```

```
//don't forget 2 use recursion
```

```
#replace if/else w/nested ternaries!
```

```javascript
function memAlloc(banks) {
  var rec={},max=Math.max(...banks),maxi=banks.indexOf(m
  rec[banks]=1
  while(1){
    var m = -1,mi = -1,il=maxi+banks.length,add=max/len|
    banks[maxi]=0
    for(var i=maxi+1;i<=maxi+len;i++){
```

A2SV

A2SV Example - 1

50. Pow(x, n)

Medium 5925 6459 Add to List Share

Implement pow(x, n), which calculates `x` raised to the power `n` (i.e., x^n).

Example 1:

```
Input: x = 2.00000, n = 10
Output: 1024.00000
```

Example 2:

```
Input: x = 2.10000, n = 3
Output: 9.26100
```

Example 3:

```
Input: x = 2.00000, n = -2
Output: 0.25000
Explanation: 2^-2 = 1/2^2 = 1/4 = 0.25
```

A2SV Example - 1

Code example taken from G31 - Submission

Bad Practice

```python
class Solution:
    def myPow(self, x: float, n: int) -> float:
        if n == 0:
            return 1
        elif n % 2 == 0:
            result = self.myPow(x,n//2)
            return result ** 2
        elif n == 1:
            return x
        elif n == -1:
            return 1/x
        else:
            return self.myPow(x,n//2) * self.myPow(x,n-n//2)
```

Good Practice

```python
class Solution:
    def myPow(self, x: float, n: int) -> float:
        ## getting power of x to absolute value of n
        result = self.myPositivePow(x, abs(n))

        ## if n is negative reverse the result
        if(n < 0):
            result = 1 / result

        return result

    def myPositivePow(self, x: float, n: int) -> float:
        # any number rasied to 0 is 1.0
        if(n == 0):
            return 1.0

        # do the half computation
        halfPower = self.myPositivePow(x, n // 2)
        fullPower = halfPower * halfPower

        # if the power is odd multiply fullPower by x
        if(n % 2 != 0):
            fullPower *= x;

        return fullPower
```

A2SV

A2SV Example - 2

20. Valid Parentheses

Easy 16147 814 Add to List Share

Given a string `s` containing just the characters `'('`, `')'`, `'{'`, `'}'`, `'['` and `']'`, determine if the input string is valid.

An input string is valid if:

1. Open brackets must be closed by the same type of brackets.
2. Open brackets must be closed in the correct order.
3. Every close bracket has a corresponding open bracket of the same type.

Example 1:

```
Input: s = "()"
Output: true
```

Example 2:

```
Input: s = "()[]{}"
Output: true
```

Example 3:

```
Input: s = "(]"
Output: false
```

A2SV Example - 2

Code example taken from G33 - Submission

Bad Practice

```python
class Solution:
    def isValid(self, s: str) -> bool:
        dc = { '(': ')',
               '{': '}',
               '[' : ']'

        }
        stack = []
        for i in s:
            if i in dc.keys():
                stack.append(i)
            else:
                if len(stack) == 0:
                    return False
                op = stack.pop()

                if dc[op] != i:
                    return False
        if len(stack) != 0:
            return False
        return True
```

God Practice

```python
class Solution:
    def isValid(self, s: str) -> bool:
        # making valid pairs to identify them
        validPiars = { '(': ')', '{': '}', '[' : ']'}
        stack = []

        for char in s:
            if char in validPiars.keys():
                # if the character is opening, add to stack
                stack.append(char)

            elif len(stack) == 0 or validPiars[stack.pop()] != char:
                # if the character is closing,
                # we need to have opening pairs in the stack
                return False

        # check if we have opening parenthesis left in the stack
        return len(stack) == 0
```

Function Parameters

Good Practice

```
class Solution:
    def isValidHelper(self, current lower_bound, upper_bound):
        if current == None:
            return True

        if not (lower_bound < current.val < upper_bound):
            return False

        left_answer = self.isValidHelper(current.left, lower_bound, current.val)
        right_answer = self.isValidHelper(current.right, current.val, upper_bound)

        return left_answer and right_answer

    def isValidBST(self, root: Optional[TreeNode]) -> bool:
        return self.isValidHelper(root, float("-inf"), float("inf"))
```

Bad Practice

```
class Solution:
    def isValidHelper(self, root, lower_bound, upper_bound):
        if root == None:
            return True

        if not (lower_bound < root.val < upper_bound):
            return False

        return self.isValidHelper(root.left, lower_bound, root.val) and
self.isValidHelper(root.right, root.val, upper_bound)

    def isValidBST(self, root: Optional[TreeNode]) -> bool:
        return self.isValidHelper(root, float("-inf"), float("inf"))
```

A2SV

Other Recommendations

Bad Practice

```
i=i+1
submitted +=1
x = x * 2 - 1
hypot2 = x * x + y * y
c = (a + b) * (a - b)
```

Good Practice

```
i = i + 1
submitted += 1
x = x*2 - 1
hypot2 = x*x + y*y
c = (a+b) * (a-b)
```

Quote of the day

““Any fool can write code that a computer can understand. Good programmers write code that humans can understand.”

– Martin Fowler