Styling Your Code

Introduction

This module will cover some guidelines for styling your code logically. Failure to follow these guidelines will not necessarily result in a loss of points, but always be sure to review the assignment instructions for each and every major assignment to ensure you do not lose points unnecessarily.

Why a Style Guide?

Having a guideline to follow in terms of code style will help both yourself as the code author, and anyone who happens to read the code you write. Some teams in professional projects have their own style guidelines so that code appearance is uniform between authors, improving readability and reducing the amount of time spent trying to figure out what code is doing.

Applying a style guide to your code is most effective when done consistently. Applying it consistently inside a function or module is of greatest importance, followed by applying it consistently across an entire assignment. There are times when you should not apply the guidelines laid out here, specifically:

- If the guideline makes your code less readable.
- If you are trying to be consistent with older code that does not adhere to the guideline.
- Certain situations not listed here, where you make the decision to not apply the style guidelines based on your individual needs.
- Just because an example here does something, does not mean you're allowed to use it in the assignments. For example, list comprehension is not allowed for any of the assignments, but an example below uses it to showcase a styling topic.

General

Lines should not exceed 79 characters

Less is more, such that, it is usually possible to condense the code you are writing by removing unnecessary conditionals, refraining from commenting on every line of code, and not overthinking it. After writing a function, think of it as a rough draft and go back to improve on your code after you've discovered the logic behind the task at hand.

Built-In Functions

See a list of permitted built-in functions **here (https://canvas.oregonstate.edu/courses/1923248/pages/built-in-python-restrictions#built-in-header)**

Returning None

If your code does not return a value, then avoid explicitly returning 'None'. This is more about readability than it is about resource impact. A function that does not require a returned value will terminate on its own after the last line of said function executes.

```
def reduceValues(self, reduction : int) -> None:
    '''
    Reduces all the values in self._data by value reduction
    '''

    for x in range(len(self._data)):
        self._data[x] -= reduction

    #This return is NOT needed!
    return

def reduceValues(self, reduction : int) -> None:
    '''
    Reduces all the values in self._data by value reduction
    '''

    for x in range(len(self._data)):
        self._data[x] -= reduction

    #This return is NOT needed!
    return None
```

Leaving 'pass' in function

Always delete 'pass' from your functions if it is not required for some conditional logic (it almost never is - if you are using pass you might need to refactor that logic). As with the guideline above, this is more about code readability than resource impact. 'Pass' is included in most of the function skeletons for the course assignments; it is added to prevent your code from throwing errors because of declared functions with no logic.

```
def myFunction(self):
    '''
    TODO : Implement this function
    '''

    #REMOVE THIS PASS AFTER IMPLEMENTATION!
    pass
```

Returns and Exceptions over Else

Conditionals are powerful tools when used properly, but can reduce the readability of your code by creating unnecessary indentation and clutter. Some examples include adding an ‘else’ statement after a conditional that would return out of the function, such as raising an exception. Because the else would trigger regardless, and there are no other conditional checks needed, the ‘else’ does not need to be written as it is assumed by the program. Another example is when your function can do 1 of 2 things, you can remove the else statement by adding a return inside the if conditional to reduce unnecessary indentation.

```
# Bad Practice
def foo():
    if index > arr.length():
        raise DynamicArrayException
    else:
        foo(x, y)
        bar(x, y)

# Good Practice
def foo():
    if index > arr.length():
        raise DynamicArrayException

    foo(x, y)
    bar(x, y)
```

```
# Bad Pracitce
def foo():
    if self.head is None:
        self.head = node(value)
    else:
        newNode = node(value)
        foo(newNode)
        bar(newNode)

# Good Practice
def foo():
    if self.head is None:
        self.head = node(value)
        return

    newNode = node(value)
    foo(newNode)
    bar(newNode)
```

For & While

- For loops iterate a set number of times, whereas while loops iterate until a specific condition is met.

- For loops are used to iterate through, while loops until a condition is met.

- For loops know the number of times they will loop, while loops do not.

```
# Iterates a known number of times
for x in range(10):
    print(x)
```

```
# Iterates across a known objects length
# Note, this will NOT work on your Static and Dynamic Arrays
for value in tempArr:
    print(value)

# Iterates until a condition is met
while x > 0:
    print(x)
    x = x // 2

# Iterates until a condition is met
while arr.length() > 0:
    print(arr.pop())
```

Range

Range should only be used when we need something performed a certain number of times, not for when we want to iterate across a list or conditional requirements (such as a while loop).

```
# Bad Practice
def produce_evens(total):
    result = []
    num = 0
    while len(result) < total:
        result.append(num)
        num += 2

# Good Pracitce
def produce_evens(total):
    result = []
    for x in range(0, total*2, 2):
        result.append(x)
```

Conditional Compression

When applicable, you should compress conditional statements into a single line. There are many reasons for this, including readability and control flow.

```
# Bad Practice
if x > 0:
    if x // 2 == 0:
        print(x)

# Good Practice
if x > 0 and x // 2 == 0:
    print(x)
```

```
# Bad Pracitce
if x > 0:
    print(x)
elif x // 2 == 0:
    print(x)

# Good Pratice
```

```
if x > 0 or x // 2 == 0:
    print(x)
```

```
# Bad Practice
def bar():
    if x > 0:
        if x // 2 == 0:
            foo(x)
            return
    if x > 0:
        if x // 3 == 0:
            foo(x+1)
            return
    if x > 0:
        if x // 5 == 0:
            foo(x+2)

# Good Practice
def bar():
    if x > 0 and x // 2 == 0:
        foo(x)
    elif x > 0 and x // 3 == 0:
        foo(x+1)
    elif x > 0 and x // 5 == 0:
        foo(x+2)
```

Breaks and Continues

Breaks and Continues are tempting when writing a loop, especially ones that seem to never stop growing. It may be hard to write a loop condition that meets all the provided restrictions, and with that, a break may feel like the best way to get out of your loop. However, these one-token statements bloat your code with unneeded conditionals and lines of code, and can actively hinder the debugging process as your loop condition is now embedded somewhere deep within the loop, instead of attached directly to it. It also reduces readability. Imagine it like so. You as a reader see while x < y, and you know this loop runs until x is greater than or equal to y, which will then break the loop. However, 15 lines down, you find the conditional statement that if x*y // 2 == 0, we want to break from the loop. Now this can change your initial assumptions of the loop's actual functionality.

```
# Bad Practice
randNum = random.randrange(1,10)
# Adds random integers to a list to a max size of 10
# If randNum is a 4, we break from the loop
while len(myList) < maxLength:

    if randNum == 4:
        break

    if randNum // 2 == 0:
        myList.append(randNum * 2)
        randNum = random.randrange(1,10)
        continue

    if randNum // 3 == 0:
        myList.append(randNum * 3)
        randNum = random.randrange(1,10)
        continue
```

```
    else:
        myList.append(randNum * randNum)
        randNum = random.randrange(1,10)

# Good Practice
randNum = random.randrange(1,10)
# Adds random integers to a list to a max size of 10
# If randNum is a 4, we break from the loop
while len(myList) < maxLength and randNum != 4:

    if randNum // 2 == 0:
        myList.append(randNum * 2)

    elif randNum // 3 == 0:
        myList.append(randNum * 3)

    else:
        myList.append(randNum * randNum)

    randNum = random.randrange(1,10)
```

```
# Adds values from an array to a new array
# until it comes across a value x
def foo(x):
    tempList = []
    for node in myList:
        if node.value() == x:
            break

        tempList.append(node)

    return tempList
```

Type Hinting

Type hints are an optional, but highly recommended, tool in python used to identify parameter and return types.

In this example, our parameter num is an integer(num : int) and we return an integer(-> int:)

```
def square(num : int) -> int:
    return num * num
```

In this example, our parameter is a string (name : str) and we do not return anything from the function (-> None:)

```
def hello_name(name : str) -> None:
    print("Hello " + name)
```

In this example, our parameter is an array of integers (arr : list[int]) and we return a tuple of 2 lists of integers (-> tuple[list[int], list[int]])

```
def split_list(arr : list[int]) -> tuple[list[int], list[int]]:
    return (arr[:len(arr)//2], arr[len(arr)//2:])
```

In this example, our parameter is an array of integers and strings (arr : list[int, str]) and we return a tuple of two list, one with integers, the other with strings (-> tuple[list[int], list[str]]). The reason we can safely assume 'else' here is because our type hint clarifies we will receive a list of integers and strings only. Python will make a fuss if this condition is not met, due to the type hinting.

```
def split_strings_from_ints(arr : list[int, str]) -> tuple[list[int], list[str]]:
    intArr = []
    strArr = []
    for ele in arr:
        if type(ele) is int:
            intArr.append(ele)
        else:
            strArr.append(ele)

    return (intArr, strArr)
```

Refactoring

Refactoring is a vital component to creating easily readable code while simultaneously improving your codes logic flow. Functions generally shouldn't be longer than 10-20 lines. This is not a hard-set rule - rather, if you have a section of code that goes beyond this ask yourself: 'Can this be broken down into a function on its own' or 'Can I remove redundancy'? Another reason you may want to break code apart into helper functions is if you have a section of code you'd use more than once. This can be twice within the same function, or used across multiple different functions. However, don't break your code apart into functions with no reasoning. If you surpass 20 lines of code, don't feel like its necessary to create a helper function just for the sake of it. Refactoring is to help with logical flow, and by breaking code apart without logic, you aren't doing yourself or the readers a service.

Below is an example (and explanation) of how you can go about breaking code apart into smaller helper functions. You can also copy-paste this into your IDE and follow along via the debugger (add fooMain() at the bottom of your code and add a break point there).

- fooMain -
  - This section is the main portion of our program, which in essence is creating a matrix (a list of list) based on 3 different inputs from the user via get_input.
  - After we get the user input, our program calls create_matrix and stores the returned value (a matrix whose dimensions and values were given to use via get_input) into the variable matrix.
  - Finally, our program calls print_matrix to send the matrix to the terminal in an easily readable format.
- get_input -

    - Notice how in fooMain this is one line of code. However, if we look at it, it is actually a little beefy function.
    - First, we ask the user for 3 different inputs. The first two are our X-axis and Y-axis of our matrix. The third is a Boolean expression to see if we want our values inside the list to be squared.
    - After getting the initial 3 inputs, we have a while conditional check to make sure we get a True or False response from the user.
    - Then, we transform the squared value to a Boolean.
    - Finally, we return our user inputs in the form of a tuple, an easy-to-use data structure that is able to hold and pass information in a compact form through out our functions.
- create_matrix -
    - The tuple from user_input got broken into 3 sections, and passed to this function with our X-axis, Y-axis, and our squared boolean.
    - This method actually utilizes ANOTHER helper function, create_list. Because a matrix is a list of list, we initially have to create the list inside our main list.
    - We use a for loop with a range() to create matrixX number of list (our rows), and append to our arrs variable an array that is returned by create_list.
- create_list-
    - create_list is a short and simple function that returns one of two things.
        - In the case our squared variable is False, we return via list comprehension that takes skip (x value passed by create_matrix that is represented by its position in the for loop) multiplied by x (defined in the list comprehension). This means in a matrix of 3x3 we'd have the values [0, 0, 0] in the first list, [0, 1, 2] in the second list, and [0, 2, 4] in the third list.
        - In the case our squared variable is True, we return via list comprehension that multiplies skip*x by itself. This means in a matrix of 3x3 we'd have [0, 0, 0], [0, 1, 4], [0, 4, 16].
        - Note: This function uses list comprehension, but you will not be allowed to use list comprehension in this class.
- print_matrix -
    - Finally, we print our matrix by calling this function. Again, it is just a simple loop that prints each row on a new line.

```
def fooMain():
    '''
    Get input from a user to create a matrix of inputted size
    '''

    # (matrixX, matrixY, squared(T/F))
    userInput = get_input()

    matrix = create_matrix(userInput[0], userInput[1], userInput[2])
    print_matrix(matrix)

def get_input():
    '''
    Prompt user for matrix size input
```

```
    '''

    # Prompts
    matrixX = input("How many rows do you want your matrix to have    :")
    matrixY = input("How many columns do you want your matrix to have :")
    squared = input("Do you want your Matrix Squared (T or F)         :")

    # Squared needs to be True or False
    while squared not in ("T", "F"):
        squared = input("Do you want your Matrix Squared (T or F)    :")

    # Set sqaured to boolean value
    if squared == "T":
        squared = True
    else:
        squared = False

    return (int(matrixX), int(matrixY), squared)

def create_matrix(matrixX, matrixY, squared = False):
    '''
    Create a list of list representing a matrix based on user input
    '''
    arrs = []

    # Fills matrix
    for x in range(matrixX):
        arrs.append(create_list(matrixY, x, squared))

    return arrs

def create_list(total, skip, squared = False):
    '''
    Creates a single list of size total
    '''
    if squared is False:
        return [x*skip for x in range(total)]

    return [(x*skip)*(x*skip) for x in range(total)]

def print_matrix(matrix):
    '''
    Prints the matrix in a readable way
    '''
    for row in matrix:
        print(row)
```