Getting Marks for Code Style

Greetings!

You may have noticed that code style and structure account for roughly **10%** of your total assignment mark. While the idea of code style can seem very nebulous and subjective, a lot of it boils down to making your code readable and maintainable - which are always very desirable things in any sort of large scale project. This document will talk about some pointers on what code style can mean for you, and how you can refine your work so that it helps you on this assignment (and, hopefully, other future projects!)

Table of Contents

Rule 0: Avoid `Hardcoding`

> A program has been **hardcoded** in the case that it has been written to only work for a very narrow, very specific set of inputs.

If you've been keeping up with you assessments so far, this should be something you're already doing - but it bears repeating! Much of the value of a program comes from its capacity to respond to a wide range of possible inputs with a correct, corresponding output (or equivalent behaviour). Hardcoding restricts the range of inputs that your program can respond to correctly to a very small, very specific subset of inputs, which reduces its applicability and reliability.

Consider the following function, which has been hardcoded:

```
"""Specification: Returns true if number is odd."""
def is_odd(n):
    if n == 1:
        return True
    if n == 3:
        return True
    if n == 5:
        return True
    if n == 7:
        return True
    if n == 9:
        return True
    return False
```

The problem here is pretty clear to see: if you give the function any number that isn't `1`, `3`, `5`, `7`, or `9`, it will return `False` - even if the number is odd! We want our programs to be reliable: they should work on any valid input, not just a small subset of those inputs - so we'd best avoid hardcoding as much as we can.

With regards to your assignment, consider how giving your program different board configurations will change its output. If we supply your program with a completely new board configuration that hasn't been given in the scaffold (or in the milestone test cases), will it still work the way you expect it to?

Readability

Maintaining good code style means keeping your code readable and comprehensible; in most cases, you want people to be able to easily understand your code once they have a chance to read it. There are a lot of different aspects to readability, and a lot of it revolves around making your future life less difficult. You may find these ideas to be quite intuitive - in which case, good! Please continue to do these things.

Organising Your Code

Keep related sections of code close to one another - that way, looking up specific functions or classes remains a quick and easy affair. You will notice that the assignment scaffold already does a lot of this for you: each class or class type is stored in its own file, all functions used for parsing from an external config file are kept in the same place, you have an entire other section for converting your board state into a readable string, and so on. You don't *always* need to make a new file every time you add new functionality to your system, but keeping them grouped up like this makes things easier to find overall.

Code organisation also extends to how you structure each file. It's important to be consistent about this, and there are some generally accepted conventions on how you should do this:

```
# File comments go at the top if appropriate.
# - Include author, date, intent, any licenses where applicable.
"""
Author: Victor Kuo
Date: 18 May 2020
Purpose: A quick example on what this file contains.
License: MIT Open Source License. (Only include if appropriate.)
"""

# Imports come after the file comments, but before anything else.
# This is so you know immediately what kinds of dependencies are necessary for the file
you're working on.
from player import Player
import sys

# Classes
# Ideally, each class is in its own file. In the case that this isn't true, you will
want them after your import statements, but before all other code.
class Game:
    def __init__():
        self.player = "???"

# Top-level and static functions.
# Placed after import statements (and classes, if applicable), but before the main body
of code.
# It's not a bad idea to put these in their own file/module, too.
def find_starting_coordinate():
    return 0, 0

# Main code - avoid putting main code in files that also contain a class. It's better to
define a class in its own file and have it imported it in.
print("Hello, world!")
```

Spacing

Putting spaces between logical sections of your code helps organise it into readable, manageable chunks - which makes the code easier to interpret, modify, and maintain as necessary. Consider the following implementation of the **splice** exercise from the Week 7 tutorial:

```
def slice(input_string, start, end):
    if not (isinstance(input_string, str) and isinstance(start, int) and isinstance(end,
int)):
        raise TypeError("Parameters given must follow order: str, int, int.")
    if start < 0 or start >= len(input_string) or end > len(input_string):
        raise ValueError("Invalid range [{}, {}) specified.".format(start, end))
    if end < 0:
        end += len(input_string)
    invariant_specifier = 'F'
    invariant = start < end
    step = 1
    if start > end:
        invariant_specifier = 'B'
        invariant = start > end
        step = -1
    substring = ""
    while invariant:
        substring += input_string[start]
        start += step
        if invariant_specifier == 'F':
            invariant = start < end
        else:
            invariant = start > end
    return substring
```

The individual lines are pretty short, and don't invoke any custom method calls - so it's still pretty readable. However, the lack of breaks between logical sections make it exhausting to read, and locating specific ideas is difficult. Consider instead, the *exact same* code, but with appropriate spacing. Note how it neatly divides the code up into manageable sections:

```
def slice(input_string, start, end):
    # Section 1: Error checking.
    if not (isinstance(input_string, str) \
            and isinstance(start, int) and isinstance(end, int)):
        raise TypeError("Parameters given must follow order: str, int, int.")

    if start < 0 or start >= len(input_string) or end > len(input_string):
        raise ValueError("Invalid range [{}, {}) specified.".format(start, end))

    # Section 2: Accounting for negative end value.
    if end < 0:
        end += len(input_string)

    invariant_specifier = 'F'
    invariant = start < end
    step = 1

    # Section 2.5: Setting up invariant based on start > end (or vice versa).
    if start > end:
        invariant_specifier = 'B'
```

```
        invariant = start > end
        step = -1

    # Section 3: Traverse `input_string`, build return value.
    substring = ""
    while invariant:
        substring += input_string[start]
        start += step

        if invariant_specifier == 'F':
            invariant = start < end
        else:
            invariant = start > end

    return substring
```

Notice also that the spaces aren't arbitrary: they're between logical sections, like paragraphs in a passage. Dividing up logical ideas like this will make it easier when you need to debug your code (because you'll be spending a lot less time *looking* for the right section).

Worth Reading: Max has written up a guide on how to keep your lines of code short and readable.

Write Self-Documenting Code

Self-documenting code is code that is comprehensible, readable, and transparent. As previously mentioned, it should be reasonably easy for someone to read your code and identify its components, goals, and approaches to achieve said goals. Here's what you can do to facilitate this:

Use Comments Appropriately

Comments are useful tools to explain sections of code that may be complicated, obscure, or otherwise unclear in purpose. Keep your comments concise, and assume that anyone reading your code already has a solid understanding of programming concepts (so avoid saying the obvious/overexplaining).

- As in the examples above, you can also use comments to mark out logical sections of code.
- Avoid using *too many* comments: it's not desirable to obscure your code with non-code text.
- As a rule, leave a space between the `#` character and the contents of your comment.
    - E.g. `# Section N: Calculates some value.` as opposed to `#Section N: Calculates some value.`
- You may have noticed comments that start with: `# TODO:` - which are exactly what they look like: reminders that you still have something left to implement. Delete these comments when you've done what they want.

Docstrings

> **Docstrings** - literally *document strings* - are string literals that exist to document the specification or functionality of classes, functions, or other such constructs within a system.

Some of these were included as part of your scaffold. You may have noticed them:

```
def grid_to_string(grid, player):
    """Turns a grid and player into a string representing the game board.

    Arguments:
        grid -- list of lists of Cells
        player -- a Player with water buckets

    Returns:
        string: A string representation of the grid and player.
    """
```

Docstrings specify a function or class' intended functionality, their arguments/attributes, and any relevant return values (in the case of a function or a method). You don't need docstrings for getters and setters (but it won't hurt if you include them either).

Use the multiline string specifier `"""` to begin and end your docstrings. You can put them before or after the class or function definition, so long as you're consistent about it.

Write clear variable names

Avoid single-character or obscure variable names. Use clear and descriptive variable names so that it's not necessary to trace a variable back to where it's been initiated to figure out what it's supposed to represent.

- Since we're coding in **Python**, snake case is recommended for function and variable names. This convention enforces only the use of lowercase letters and using underscores `_` to replace spaces where necessary.
- Class names follow a `CapitalLetters` convention (capitalise your class names, no spaces; capitalise the start of new words).
- For constant, immutable values (i.e. values you never intend to change), it may be acceptable to use fully uppercase variable names.
    - e.g. `PI = 3.1415`.

Modularity

Any sufficiently large task can be decomposed into smaller sub-tasks that are easier to process and write a solution for. This is a good principle to apply to your coding approach: writing a different function, class, or module to be responsible for an aspect of your program's functionality allows you to divide up your work into manageable chunks, and to guarantee the functional correctness of one element of your product without needing *everything* to be complete.

You will notice that, for your assignment, this has partially been done for you: classes have their own files, `grid.py` and `game_parser.py` exist as independent modules, `solver.py` isn't necessary for any of the other components of the program, and so on. You will probably want to create additional functions, methods, or modules to be responsible for different aspects of your code.

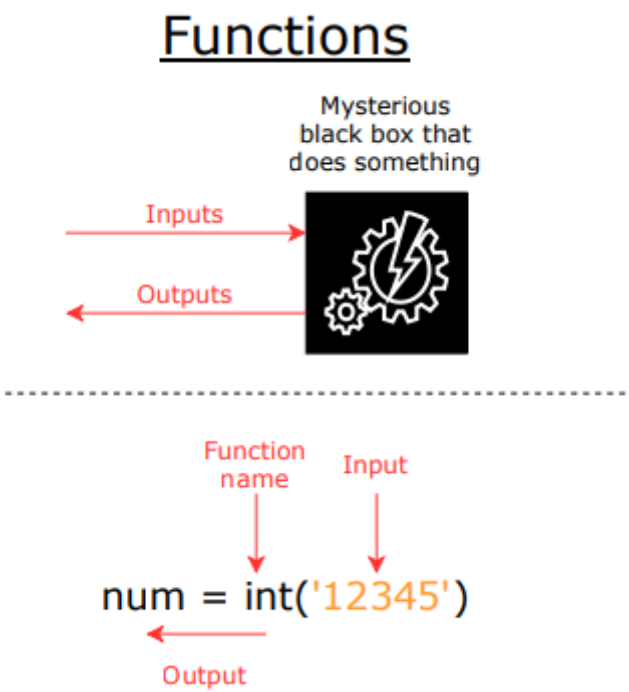
A lot of the benefits conferred by this approach have already been discussed previously:

- Dividing up your code into sections makes it easier to build and check for correctness.
- Your program becomes more readable because of it.
- Errors are easier to isolate, your code becomes easier to maintain.
- You can reuse your code if it ends up being applicable in several places.

As a general guideline, when a contiguous section of code begins to reach around **~60 lines**, it's probably a good time to consider whether you can get a function to take care of some of it.

Don't Break Scope

Recall this diagram from Week 1:



It describes a function as a **black box** that takes in some input, performs operations on or derived from those inputs, and produces some desired output. This can take the form of a return value, or - in the case of class setter methods - a modification to the state of the class (instance) it's attached to.

It's important to observe that this **black box** is completely independent of anything else in the code, outside of its inputs - so a function **really shouldn't be trying to interact with anything that isn't given to it as input.**

- Avoid using the `global` keyword. If you find that your function needs additional information, or that it needs to interact with an object that hasn't been given to it as an argument, consider simply modifying the function so that it receives this object or information as a parameter.

  So instead of this:

```
def do_something(param_1, param_2):
    global banned_parameters
    i = 0
    while i < len(banned_parameters):
        ...

banned_parameters = [
    "This is", "an important",
    "list", "with some crucial", "information"
]
do_something("information", "data")
```

Just do this:

```
def do_something(param_1, param_2, banned):
    i = 0
    while i < len(banned):
        ...

banned_parameters = [
    "This is", "an important",
    "list", "with some crucial", "information"
]
do_something("information", "data", banned_parameters)
```

- On the flipside, be careful not to overwrite the inputs you're receiving when writing your function. Consider the following code:

```
def add_if_no_duplicates(words, addition):
    """
    Adds `addition` to the list `words` if it isn't already part of the list.
    Parameters:
        - words -- a list.
        - addition -- string to add to `words`.
    """
    words = []
    if addition not in words:
        words.append(addition)

banned = ["pineapples on pizza", "hating on clowns", "calling things 'moist'"]
add_if_no_duplicates(banned, "murder hornets")
# What happens to `banned` here? Does the function work as we expect it to?
```

Miscellaneous

- Avoid repeating yourself. If you find yourself writing very similar code more than twice, it's time to just put it in a function (especially if it's long - see: modularity). This helps you stay consistent, too.
- Go for the simplest solution first - they often work best.
- Get rid of code you don't need.
- If changing one thing in one section of code means that you'll have to make a cascading series of changes in lots of other sections of code, reorganise your code so that this isn't necessary. This may lead to needlessly complicated problems otherwise.
- Don't be afraid to refactor your code when you need to. You'll already know how things are supposed to fit together, so it won't take as much time as designing your solution in the first place.