Python Coding Standards

I. General

Defining Constants: All constants should be defined in a separate file within the relevant code area. Refer to the 'Variables' section on how to properly name the constants.

Importing Classes: When importing a file that contains class definitions, ensure that only the classes that are desired are imported. For example if DbClassName is defined in db_className, the class should be imported as: [from db_className import DbClassName].

Importing Constants: When importing a file that contains constant definitions, the constants should be imported with a name consisting of an abbreviation of the coding area and def, in all caps and separated by an underscore [DB_DEF]. The example is in the database code area.

Importing External Modules: When importing external python modules, only import the functions being used. For example, if importing ceil() from math, import as: [from math import ceil]. Do not import the entire math library and call 'math.ceil()'.

Indentations: Tabs should be set to four spaces for any nesting of functions, conditionals, or loops.

Semicolons Ending Lines: Semicolons should be used to end lines, even when not syntactically necessary. This includes after imports, assignment, function calls, etc. Follow C++ / C syntax on approaches to semicolons.

Keep Comments Succinct: Unless describing a complex section of code, keep comments of functionality to condensed, generalized comments. Having more comments than code on a page is undesirable. At the same time, make sure that comments are distributed usefully throughout the code base to ensure that the code is understandable from external users.

II. Classes

Class Names: Class names should contain an abbreviation of the current system and a descriptive name with no spaces in Pascal Case [DbClassName]. This example is in the Database code area, and is the "ClassName" class.

Getter and Setter Methods: All classes should have getter and setter methods for all variables that will be publicly accessible. The methods should be named in camelCase with no spaces [getVariableName() or setVariableName()].

Class Methods: Class methods should be in camelCase with no spaces [myClassMethod()].

Class Members: Class members should be descriptive in camelcase, no spaces, followed by an underscore [myVariableName_]

Class Local Variables: Class local variables should be descriptive in camelcase, no spaces [myLocalVariable]

Shared Class Variables: Shared class variables should be descriptive separated by underscores, all lowercase, followed by an underscore [my_variable_ name]

III. Variables

Local Variables: Local variables should be descriptive in camelcase, no spaces [myVariableName]

Global Constants: Global constants should be descriptive in all capitals separated by underscores [MY_VARIABLE_NAME]

IV. Functions

Function Naming: Functions should be named descriptively in camelCase with no spaces [myFunctionName()].

Function Arguments Naming: Function arguments in a function definition should be camelcase with no spaces. They should also be generalized to the function.

V. Commenting

Requirement Commenting:
Where a requirement is implemented in the code, it should be noted above the block of code with a descriptive comment and a list of the requirements that were implemented:

```
# Description of Implementation
# REQs: <list of requirement numbers>
```

File Header Commenting:
All files should contain an overall block comment at the top of the file that includes the file name, the purpose of the file, the creation date, and the author of the file:

```
##########################################################
# File: <File Name>
#
# Purpose of File:
#
# Creation Date:
```

```
#
# Author:
#
#########################################################
```

<u>File Section Commenting</u>:
Each file should be broken into different sections with corresponding comments for each section. The sections are: File Header, Notes, Import Files, and Code. The outline is below:

```
<File Header Commenting Comments>

# Notes
<Notes>

# Import Files
<Import Files>

# System Functions (or other descriptive comment of section)
```

<u>Function or Class method Commenting</u>:
Each function or class method should include a description including its name, purpose, requirements met in the function, inputs, and outputs:

```
#########################################################
# Function:
# Purpose:
# Requirements:
# Inputs:
# Outputs:
#########################################################
```

Next.tsx Coding Standards

Morgann: Next.tsx (React, TypeScript). Since the interaction and generation are real-time, we want to run most stuff client-side which means JavaScript. Probably, we'll make an API call for data at the start and every few minutes, so we can definitely pre-process the data however we want through a backend server at each API call. I'm learning this rn and it seems like the easiest and modern, but lmk about express.js or other options.

or if we can generate videos or images backend then play or interpolate or interact with them in a simple way for minutes?

Git Branches:

1. Use local branch (e.g backend, morgann, feature-X),
1. Pull from more core branch (e.g. main)
2. Resolve merge conflicts locally (use your discretion and just ask whoever wrote it if you have a conflicting design decision)
3. Push your branch to remote
4. Pull Request to core branch (main) to update it

Naming Conventions:

1. root = github-project-kebab-case
2. root/components/ComponentInPascalCase.tsx

```
interface ComponentPascalProps {
    children?: ReactNode;
    a: number;
    b-optional?: string;
}

export default function ComponentPascal({ children, a, b = 'hi' } : ComponentPascalProps) {
    return (
        <div>
        ...
        </div>
    )
}
```

3. root/app/<more-structure>/page-in-kebab-case.ts
4. root/styles/<more-structure>/styles-matching-page?
5. root/types/type-pascal.ts
    a. [quense/yup: Dead simple Object schema validation]
    b. export const typeSchemaCamel = object({

c. export type TypePascal = InferType<typeof typeSchemaCamel>;

```
import { object, string, number, date, InferType } from 'yup';

let userSchema = object({
    name: string().required(),
    age: number().required().positive().integer(),
    email: string().email(),
    website: string().url().nullable(),
    createdOn: date().default(() => new Date()),
});

// parse and assert validity

const user = await userSchema.validate(await fetchUser());

type User = InferType<typeof userSchema>;

/* {
name: string;
age: number;
email?: string | undefined
website?: string | null | undefined
createdOn: Date
} */
```

6. root/constants.ts
    a. CONSTANT_UPPER_CASE = ...

Ordering Conventions

Global Variables
…
Functions
…
Main

File System Organisation

```
root ->
    code->
        FrontEnd ->
            UI ->
            Data I/O ->
            GenerativeFrontEnd ->
        BackEnd ->
            Databases ->
                src->
            GenerativeBackEnd ->
                src->
            …
    docs->
        Requirements ->
        Notes ->
        Statuses ->
    bins->
```