Clean Code Development (CCD)

1. Meaningful Names

- Using meaningful names for variables, functions, classes, and methods.
- It is important to avoid using ambiguous names.
- Choosing names that convey the main purpose and functionality of the code.

Here is the example of meaningful names in my code:

```
> def generate_password(length=PASSWORD_LENGTH_DEFAULT, use_upper=True, use_digits=True, use_special=True, pronounceable=False, entropy=None): ···

> def generate_pronounceable_password(length): ···

> def check_password_strength(password): ···
    passphrase = [random.choice(word_list) for _ in range(num_words)]
    return delimiter.join(passphrase)

> def save_passwords_to_file(passwords, filename="passwords.txt"): ···
```

2. Modularity

- Breaking code to modular and smaller functions and classes.
- It is important to have functions with no more than 20 lines of code.
- Each function or class should have a single responsibility.

Here is the example of modularity in my code:

```
def generate_password(length=PASSWORD_LENGTH_DEFAULT, use_upper=True, use_digits=True, use_special=True, pronounceable=False, entropy=None):
    if pronounceable:
        return generate_pronounceable_password(length)

    characters = string.ascii_lowercase
    if use_upper:
        characters += string.ascii_uppercase
    if use_digits:
        characters += string.digits
    if use_special:
        characters += string.punctuation

    if entropy:
        min_length = math.ceil(entropy / math.log2(len(characters)))
        if length < min_length:
            raise ValueError(f"Password length must be at least {min_length} for the specified entropy")

    password = [secrets.choice(characters) for _ in range(length)]
    return ''.join(password)

def generate_pronounceable_password(length):
    vowels = 'aeiou'
    consonants = 'bcdfghjklmnpqrstvwxyz'
    password = ''

    for i in range(length):
        password += random.choice(consonants) if i % 2 == 0 else random.choice(vowels)

    return password
```

3. Comments

- Using comments or focusing on explaining why something is done rather than what is done.
- Making sure comments are up-to-date and update your comments time-to-time for reflecting the current state of the code.

Here is the example of commenting in my code:

```
#Generating Methods
def generate_password(length=PASSWORD_LENGTH_DEFAULT, use_upper=True, use_digits=True, use_special=True, pronounceable=False, entropy=None): ...
#Generating Methods
def generate_pronounceable_password(length): ...
#Calculation Methods
def check_password_strength(password): ...
```

4. Consistent Style

- Adopting a consistent coding style throughout the project and adhering to chosen style.
- Trying to put tabs and spaces in the same manner through the code.

Here is the example of consistent style in my code:

```
def main():
    word_list = ["apple", "banana", "cherry", "dog", "elephant", "flower", "giraffe", "honey", "ice", "jungle"]

    while True:
        print("Password Generator")
        print("1. Generate Random Password")
        print("2. Generate Passphrase")
        choice = int(input("Choose an option (1/2): "))

        if choice == 1:
            num_passwords = int(input("Enter the number of passwords to generate: "))
            length = int(input(f"Enter the length of the password (default is {PASSWORD_LENGTH_DEFAULT}): ") or PASSWORD_LENGTH_DEFAULT)
            use_upper = input("Include uppercase letters? (y/n): ").strip().lower() in ('y', 'yes')
            use_digits = input("Include digits? (y/n): ").strip().lower() in ('y', 'yes')
            use_special = input("Include special characters? (y/n): ").strip().lower() in ('y', 'yes')
            entropy = float(input("Specify password entropy (optional): ") or 0)

            passwords = []
            for _ in range(num_passwords):
                try:
                    password = generate_password(length, use_upper, use_digits, use_special, entropy=entropy)
                    passwords.append(password)
                    print(f"Generated Password: {password}")
                    strength = check_password_strength(password)
                    print(f"Password Strength: {strength}/4")
                except ValueError as e:
```

5. Error Handling

- Implementing proper error handling mechanisms.
- Avoid using exceptions for flow control unless in exceptional situations.
- Printing meaningful error messages to the user.

6. Version Control

- Using version control systems to track changes and collaborate effectively. In My project I am using Git.
- Committing changes with meaningful commit messages.

7. Refactoring

- Review and refactoring code to improve its structure and readability.
- Refactoring when you find areas that can be improved without changing the behavior.

Here is the example of refactoring my code:

4 RPG

```diff
@@ -89,8 +89,8 @@ def main():
89          elif choice == 2:
90              num_passphrases = int(input("Enter the number of passphrases to generate: "))
91              num_words = int(input("Enter the number of words per passphrase: "))
92 -            delimiter = input("Enter the word delimiter (default is space): ")
93 -            delimiter = delimiter if delimiter else ' '
94
95              passphrases = [generate_passphrase(word_list, num_words, delimiter) for _ in range(num_passphrases)]
96              for passphrase in passphrases:
```

```diff
89          elif choice == 2:
90              num_passphrases = int(input("Enter the number of passphrases to generate: "))
91              num_words = int(input("Enter the number of words per passphrase: "))
92 +            delimiter = input("Enter the word delimiter (default is space): ") or ' '
93 +
94
95              passphrases = [generate_passphrase(word_list, num_words, delimiter) for _ in range(num_passphrases)]
96              for passphrase in passphrases:
```

8. Documentation

- Providing clear and concise documentation for code, especially complex algorithms.
- Documents should Include information on how to use and extend code.

9. Testing

- Writing unit tests to ensure the correctness of the code.
- Test-driven development (TDD) can be beneficial in creating clean and functional code.

Here is the written unit test for my code:

```python
class TestRPGFunctions(unittest.TestCase):

    def test_generate_password_default(self): ...

    def test_generate_password_custom_length(self): ...

    def test_generate_password_with_entropy(self): ...

    def test_generate_pronounceable_password(self): ...

    def test_check_password_strength(self): ...

    def test_generate_passphrase(self): ...


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

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

```
PS C:\Users\MAAHSHAD> & C:/Users/MAAHSHAD/.conda/envs/tmda/python.exe c:/Project/test_RPG.py
......
----------------------------------------------------------------------
Ran 6 tests in 0.002s

OK
PS C:\Users\MAAHSHAD> & C:/Users/MAAHSHAD/.conda/envs/tmda/python.exe c:/Project/test_RPG.py
......
----------------------------------------------------------------------
Ran 6 tests in 0.003s

OK
PS C:\Users\MAAHSHAD>
```

10. Code Duplication

- Eliminating redundant code by promoting code reuse.
- Follow SOLID principles (Single Responsibility, Open/Closed, Liskov Substitution, Interface Segregation, Dependency Inversion) to create maintainable and scalable code.