**Запреты**

1. Не использовать ***from module import \**** для импорта PyQt-модулей, так как это засоряет пространство имен и может привести к уменьшению читаемости кода и конфликтам имен, особенно в больших проектах.

**Пример неправильного кода:**

| from PyQt5.QtWidgets import \*  window = QWidget() |
| --- |

**Пример правильного кода:**

| from PyQt5.QtWidgets import QWidget  window = QWidget() |
| --- |

2. Не писать длинные строки кода, длина которых превышает длину в 79 символов, так как это нарушает PEP8 и затрудняет чтение кода, особенно в PyQt, где вызовы методов могут быть многословными.

**Пример неправильного кода:**

self.button = QPushButton("Нажми на кнопку", self, toolTip="Это текст всплывающей подсказки")

**Пример правильного кода**:

| self.button = QPushButton("Нажми на кнопку",   self,   toolTip="Это текст всплывающей подсказки") |
| --- |

3. Не игнорировать обработку исключений при работе с PyQt, так как незакрытые ошибки (например, при загрузке ресурсов) могут привести к аварийному завершению приложения.

**Пример неправильного кода:**

self.setStyleSheet(open("styles.qss").read())

**Пример правильного кода:**

| try:  with open("styles.qss", "r") as file:  self.setStyleSheet(file.read()) except FileNotFoundError:  print("Ошибка: Файл стилей не найден") |
| --- |

**Рекомендации**

1. Использовать понятные имена для виджетов и переменных, так это улучшает читаемость кода. Также PyQt-код часто включает множество объектов (кнопки, окна и т.д.), и их имена должны отражать назначение.

**Пример плохого кода:**

| w1 = QWidget() b1 = QPushButton("Принять", w1) |
| --- |

**Пример хорошего кода:**

| main\_window = QWidget() ok\_button = QPushButton("Принять", main\_window) |
| --- |

2. Использовать комментарии для объяснения сложных PyQt-структур, так использование методов библиотеки PyQt может быть запутанным для новичков.

**Пример без комментариев:**

| layout = QVBoxLayout() layout.addWidget(self.button) self.setLayout(layout) |
| --- |

**Рекомендуемый подход:**

| # Создаем вертикальный layout для размещения виджетов layout = QVBoxLayout() layout.addWidget(self.button) # Добавляем кнопку в layout self.setLayout(layout) # Применяем layout к окну |
| --- |

3. Разделять логику приложения и интерфейс, так как это упрощает поддержку и тестирование кода.

**Пример плохого кода:**

| class MyWindow(QWidget):  def \_\_init\_\_(self):  super().\_\_init\_\_()  self.button = QPushButton("Нажать", self)  self.button.clicked.connect(lambda: self.resize(400, 300)) |
| --- |

**Рекомендуемый подход:**

| class MyWindow(QWidget):  def \_\_init\_\_(self):  super().\_\_init\_\_()  self.button = QPushButton("Нажать", self)  self.button.clicked.connect(self.on\_button\_click)   def on\_button\_click(self):  self.resize(400, 300) |
| --- |

**Требования**

1. Каждая функция или метод должны иметь четкое назначение и не превышать 20-30 строк, так как длинные функции ухудшают читаемость и усложняют отладку.

**Пример неправильного кода:**

| def setup\_ui(self):  self.button = QPushButton("Нажать", self)  self.button.clicked.connect(lambda: print("Button clicked!"))  # Еще 50 строк кода... |
| --- |

**Пример правильного кода:**

| def setup\_ui(self):  self.button = QPushButton("Нажать", self)  self.button.clicked.connect(self.handle\_click)  def handle\_click(self):  print("Кнопка нажата!") |
| --- |

2. Использовать подсказки типов данных для всех функций и методов, так как это улучшает читаемость и упрощает рефакторинг.

**Пример неправильного кода:**

| def add\_numbers(a, b):  return a + b |
| --- |

**Пример правильного кода:**

| def add\_numbers(a: int, b: int) -> int:  return a + b |
| --- |

**Чек-лист**

**1. Соответствие PEP 8**

Проверьте, соответствует ли код стандартам PEP 8 (отступы, пробелы, названия переменных, функций и классов).

**2. Корректность импортов**

Убедитесь, что все необходимые библиотеки (PyQt, sys, os и др.) импортированы корректно и нет неиспользуемых импортов.

**3. Оптимизация кода**

Проверьте, что код не содержит избыточных вычислений или ненужных циклов.

**4. Комментарии и документация**

Проверьте, что код содержит комментарии для сложных участков и документацию для основных функций и классов.

**5. Соответствие требованиям задачи**

Убедитесь, что реализованный функционал полностью соответствует поставленной задаче.