Импортируем необходимые библиотеки и адаптеры

import psycopg2

import sys

from PyQt5.QtWidgets import (QApplication, QWidget,

QTabWidget, QAbstractScrollArea,

QVBoxLayout, QHBoxLayout,

QTableWidget, QGroupBox,

QTableWidgetItem, QPushButton,

QMessageBox)

Создаем класс MainWindow с конструктором

class MainWindow(QWidget):

def \_\_init\_\_(self):

super(MainWindow, self).\_\_init\_\_()

self.\_connect\_to\_db()

self.setWindowTitle("Shedule")

self.vbox = QVBoxLayout(self)

self.tabs = QTabWidget(self)

self.vbox.addWidget(self.tabs)

self.\_create\_shedule\_tab()

Класс QTabWidget создает структуру, которую можно заполнять

вкладками.

Вкладки это подстраницы в окне приложения. Аналогом вкладок в

оконных приложениях являются вкладки в веб-браузере.

Создаем метод для подключения к базе данных

def \_connect\_to\_db(self):

self.conn = psycopg2.connect(database="<название

вашей базы данных>",

user="postgres",

password="1234",

host="localhost",

port="5432")

self.cursor = self.conn.cursor()

Создаем метод для отображения вкладки с расписанием

def \_create\_shedule\_tab(self):

self.shedule\_tab = QWidget()

self.tabs.addTab(self.shedule\_tab, "Shedule")

self.monday\_gbox = QGroupBox("Monday")

self.svbox = QVBoxLayout()

self.shbox1 = QHBoxLayout()

self.shbox2 = QHBoxLayout()

self.svbox.addLayout(self.shbox1)

self.svbox.addLayout(self.shbox2)

self.shbox1.addWidget(self.monday\_gbox)

self.\_create\_monday\_table()

self.update\_shedule\_button = QPushButton("Update")

self.shbox2.addWidget(self.update\_shedule\_button)

self.update\_shedule\_button.clicked.connect(self.\_update\_shedule)

self.shedule\_tab.setLayout(self.svbox)

Класс QWidget() создает виджет, который будет являться

вкладкой в нашем приложении. Данный класс может также

использоваться для создания отдельных окон, но в нашем случае

будет вкладкой.

self.tabs.addTab(self.shedule\_tab, "Shedule") добавляет в

структуру с вкладками новую вкладку с названием "Shedule".

Класс QGroupBox() может группировать виджеты, он

предоставляет

рамку, заголовок вверху и может отображать несколько виджетов

внутри. В нашем случае он служит исключительно в

декоративных

целях

Создаем метод для отображения таблицы с расписанием на

понедельник

def \_create\_monday\_table(self):

self.monday\_table = QTableWidget()

self.monday\_table.setSizeAdjustPolicy(QAbstractScrollArea.AdjustToConten

ts)

self.monday\_table.setColumnCount(4)

self.monday\_table.setHorizontalHeaderLabels(["Subject", "Time",

"", ""])

self.\_update\_monday\_table()

self.mvbox = QVBoxLayout()

self.mvbox.addWidget(self.monday\_table)

self.monday\_gbox.setLayout(self.mvbox)

Класс QTableWidget() создает пустую пользовательскую таблицу

аналогичную таблицам Excel.

setSizeAdjustPolicy(QAbstractScrollArea.AdjustToContents)

устанавливает возможность изменения размера под размер данных

в ячейке.

Метод setColumnCount() задает таблице количество колонок.

Метод setHorizontalHeaderLabels(["Название", "Название"])

задает колонкам подписи.

Создаем метод для обновления таблицы с расписанием на

понедельник

def \_update\_monday\_table(self):

self.cursor.execute("SELECT \* FROM timetable WHERE

day='wednesday'")

records = list(self.cursor.fetchall())

self.monday\_table.setRowCount(len(records) + 1)

for i, r in enumerate(records):

r = list(r)

joinButton = QPushButton("Join")

self.monday\_table.setItem(i, 0,

QTableWidgetItem(str(r[0])))

self.monday\_table.setItem(i, 1,

QTableWidgetItem(str(r[2])))

self.monday\_table.setItem(i, 2,

QTableWidgetItem(str(r[4])))

self.monday\_table.setCellWidget(i, 3, joinButton)

joinButton.clicked.connect(lambda ch, num=i:

self.\_change\_day\_from\_table(num))

self.monday\_table.resizeRowsToContents()

Заполнение таблицы происходит в цикле for для того, чтобы

динамически обрабатывать изменения в количестве записей.

Метод setRowCount() задает таблице количество строк.

Кнопка joinButton не является отдельным свойством класса

MainWindow, так как нам не нужно ее "запоминать". Далее

интерпретатор запоминает ее с помощью функции-обработчика

clicked.connect().

Метод setItem(<Номер строки>, <Номер колонки>, <Строка с

данными>) записывает в ячейку с определенным адресом

строковые данные.

Метод setCellWidget(<Номер строки>, <Номер колонки>,

<Виджет>)

помещает в ячейку с определенным адресом виджет.

В нашем случае это кнопка "Join".

Метод resizeRowsToContents() автоматически адаптирует размеры

ячеек таблицы под размер данных внутри этой ячейки. Это

необходимо использовать для экономии визуального

пространства.

Создаем метод изменяющий запись в базе данных по нажатию на

кнопку "Join"

def \_change\_day\_from\_table(self, rowNum):

row = list()

for column in range(self.monday\_table.columnCount()):

try:

row.append(self.monday\_table.item(rowNum,

column).text())

except:

row.append(None)

try:

self.cursor.execute("UPDATE SQL запрос на

изменение одной строки в базе данных", (row[0],))

self.conn.commit()

except:

QMessageBox.about(self, "Error", "Something

wrong!")

Метод columnCount() возвращает количество колонок таблицы.

Конструкция item(<Номер строки>, <Номер столбца>).text()

возвращает текст, записанный в определенной ячейке.

Создаем метод обновляющий все таблицы на вкладке

def \_update\_shedule(self):

self.\_update\_monday\_table()

…

Ваши методы обновления таблиц

"Запускаем" наше приложение

app = QApplication(sys.argv)

win = MainWindow()

win.show()

sys.exit(app.exec\_())