5. Data on Event Driven Application

5.1

Structured and Unstructured Data Storage:

Structured data refers to data that is organized in a predefined manner, typically using a schema. Examples of structured data include spreadsheets, SQL databases, and other tabular data formats. Unstructured data, on the other hand, refers to data that is not organized in a predefined manner. Examples of unstructured data include emails, text documents, images, and video files.

In event-driven programming, you may encounter both structured and unstructured data. For example, you might use a structured SQL database to store user account information, while using unstructured text files to store log data.

File Handling:

File handling refers to the process of reading from and writing to files on disk. In PyQT5, you can use the built-in QFile and QTextStream classes to handle file I/O.

Here's an example of how to read from a text file:

```
from PyQt5.QtCore import QFile, QTextStream

file = QFile("example.txt")
if file.open(QFile.ReadOnly | QFile.Text):
    stream = QTextStream(file)
    content = stream.readAll()
    file.close()

print(content)
```

In this example, we're opening a text file called "example.txt" and reading its contents into a variable called "content". We're using the QFile class to open the file and the QTextStream class to read the file's contents.

File Types: Text, CSV, XML:

There are many different types of files that you might encounter in event-driven programming, including text files, CSV files, and XML files.

A text file is a file that contains plain text, without any formatting or structure. Text files are often used to store configuration data or log information.

A CSV file is a file that contains comma-separated values. CSV files are often used to store tabular data, such as data exported from a spreadsheet.

An XML file is a file that contains structured data in the form of XML tags. XML files are often used to exchange data between different systems or applications.

Here's an example of how to read from a CSV file:

```
import csv
with open('example.csv', 'r') as csvfile:
 reader = csv.reader(csvfile)
 for row in reader:
 print(row)
```

In this example, we're using Python's built-in csv module to read from a CSV file called "example.csv". We're using a "with" statement to automatically close the file when we're done with it, and we're iterating over the rows in the file using a for loop.

Overview RDBMS: MySQL:

RDBMS stands for "Relational Database Management System", which is a type of software used to manage relational databases. A relational database is a database that organizes data into tables, which consist of rows and columns.

MySQL is an open-source RDBMS that is often used in web applications to manage and store data. In PyQT5, you can use the built-in QtSql module to connect to a MySQL database and perform database operations such as querying, updating, and deleting data.

To connect to a MySQL database using QtSql, you first need to create a database object and set its properties, such as the host name, database name, username, and password. Once you've set up the database object, you can open the connection to the database and execute queries using a QSqlQuery object.

Here's an example of how to connect to a MySQL database using QtSql:

```
from PyQt5.QtSql import QSqlDatabase, QSqlQuery

db = QSqlDatabase.addDatabase('QMYSQL')
db.setHostName('localhost')
db.setDatabaseName('mydatabase')
db.setUserName('myusername')
db.setPassword('mypassword')

if db.open():
    query = QSqlQuery()
```

In this example, we first create a QSqlDatabase object using the addDatabase method and specifying the database driver as 'QMYSQL'. We then set the hostname, database name, username, and password properties of the database object.

Next, we open the connection to the database using the open method. We then create a QSqlQuery object to execute a SELECT query on the 'mytable' table in the database. Finally, we iterate over the query results using a while loop and print out the value of the first column in each row.

5.2 Connection of Application with Database: Driver & Port

When developing a software application, you often need to store and retrieve data from a database. To connect your application to a database, you need to use a database driver, which is a software component that allows your application to communicate with the database server.

In PyQT5, you can use the QtSql module to connect your application to a database using a database driver. The QtSql module provides a set of classes for working with SQL databases, including the QSqlDatabase class for managing database connections and the QSqlQuery class for executing SQL queries.

To connect to a database using a driver, you first need to choose the appropriate driver for the database you're using. For example, if you're using MySQL, you would choose the QMYSQL driver, while if you're using SQLite, you would choose the QSQLITE driver.

Here's an example of how to connect to a MySQL database using the QMYSQL driver in PyQT5:

```
from PyQt5.QtSql import QSqlDatabase, QSqlQuery, QSqlTableModel

db = QSqlDatabase.addDatabase("QMYSQL")
db.setHostName("localhost")
db.setDatabaseName("mydatabase")
db.setUserName("myusername")
db.setPassword("mypassword")
db.setPort(3306) # Optional: Set the port number if needed

if db.open():
    print("Database connection established")
    # Do something with the database, such as execute a query or display data in a table
else:
    print("Error connecting to the database")
```

In this example, we first import the required classes from the QtSql module. We then create a QSqlDatabase object and set its properties, including the hostname, database name, username, and password. We also set the port number to 3306, which is the default port number for MySQL.

Next, we open the database connection using the open method. If the connection is successful, we print a message to indicate that the connection has been established. If the connection fails, we print an error message.

Once the database connection is established, we can execute SQL queries using a QSqlQuery object or display data in a table using a QSqlTableModel object. We can also use a data adapter, such as the QSqlRelationalTableModel or the QSqlQueryModel, to customize the way data is displayed in the table and handle relationships between tables.

Overall, connecting your application to a database using a driver and port in PyQT5 is a simple process that allows you to store and retrieve data from a database and build powerful applications with a graphical user interface.

5.3 Data Binding with User Interface

In event-driven programming, data binding is the process of synchronizing the data in your application with the user interface controls that display that data. In PyQT5, you can use

data binding to connect your user interface controls to a data source, such as a database or a list of objects, and automatically update the controls when the data changes.

To use data binding in PyQT5, you first need to identify the controls in your user interface that you want to bind to data. These controls are known as bindable controls and include widgets such as QLineEdit, QComboBox, and QTableView.

Once you've identified the bindable controls, you can use the setData and data methods to bind the controls to the data source. The setData method sets the value of a property in the control, while the data method retrieves the value of a property from the control.

Here's an example of how to bind a QLineEdit control to a property of an object in PyQT5:

```
from PyQt5.QtWidgets import QApplication, QMainWindow, QLineEdit, QLabel

class Person:
    def __init__(self, name="", age=0):
        self._name = name
        self._age = age

    def get_name(self):
        return self._name

    def set_name(self, value):
        self._name = value

    def get_age(self):
        return self._age

    def set_age(self, value):
        self._age = value

class MainWindow(QMainWindow):
    def __init__(self, person):
        super().__init__()

        self.person = person

        self.name_label = QLabel("Name:", self)
        self.name_label.move(10, 10)

        self.name_edit = QLineEdit(self)
        self.name_edit.move(80, 10)

        self.age_label = QLabel("Age:", self)
        self.age_label.move(10, 40)

        self.age_edit = QLineEdit(self)
        self.age_edit.move(80, 40)

        self.bind_controls()

        self.setGeometry(100, 100, 250, 100)
        self.setWindowTitle("Data Binding Example")

    def bind_controls(self):
        self.name_edit.setText(self.person.get_name())
        self.name_edit.textChanged.connect(self.on_name_changed)

        self.age_edit.setText(str(self.person.get_age()))
        self.age_edit.textChanged.connect(self.on_age_changed)

    def on_name_changed(self, value):
        self.person.set_name(value)

    def on_age_changed(self, value):
        self.person.set_age(int(value))

if __name__ == "__main__":
    app = QApplication([])
    person = Person("John Doe", 30)
    window = MainWindow(person)
    window.show()
    app.exec_()
```

In this example, we create a Person class that has two properties, name and age. We then create a MainWindow class that contains two QLineEdit controls for editing the name and age properties, as well as two QLabel controls for displaying the labels.

In the MainWindow class, we first create the controls and set their positions. We then call the bind_controls method to bind the controls to the data source. In this method, we set the initial values of the controls to the values of the properties in the Person object and connect the textChanged signal of each control to a callback method that updates the corresponding property in the Person object.

When the user changes the value of a control, the corresponding callback method is called, which updates the corresponding property in the Person object. The changes are automatically reflected in the other controls, thanks to the data binding.

Overall, data binding in PyQT5 is a powerful technique for creating dynamic and interactive user interfaces that respond to changes in data.

5.4. Data Manipulation & Storage

In event-driven programming, data manipulation and storage are essential for creating applications that can store, manipulate, and display data. In PyQT5, you can use various data manipulation and storage techniques to handle data generated by your application.

To store generated data with validation in PyQT5, you can use various storage options, such as files, databases, or memory-based data structures. For example, you can use CSV or XML files to store data in a human-readable format, or you can use a relational database such as MySQL to store data in a structured format that supports efficient querying and indexing.

Here's an example of how to store and validate data in a PyQT5 application:

```
from PyQt5.QtWidgets import QApplication, QMainWindow, QLabel, QLineEdit, QPushButton, QVBoxLayout, QWidget

class MainWindow(QMainWindow):
    def __init__(self):
        super().__init__()

        self.data = []

        self.name_label = QLabel("Name:", self)
        self.name_edit = QLineEdit(self)

        self.email_label = QLabel("Email:", self)
        self.email_edit = QLineEdit(self)

        self.add_button = QPushButton("Add", self)
        self.add_button.clicked.connect(self.on_add_clicked)

        self.layout = QVBoxLayout()
        self.layout.addWidget(self.name_label)
        self.layout.addWidget(self.name_edit)
        self.layout.addWidget(self.email_label)
        self.layout.addWidget(self.email_edit)
        self.layout.addWidget(self.add_button)

        self.widget = QWidget()
        self.widget.setLayout(self.layout)
        self.setCentralWidget(self.widget)

        self.setGeometry(100, 100, 250, 150)
        self.setWindowTitle("Data Manipulation & Storage Example")

    def on_add_clicked(self):
        name = self.name_edit.text()
        email = self.email_edit.text()

        if not name or not email:
            return

        self.data.append({"name": name, "email": email})
        self.name_edit.setText("")
        self.email_edit.setText("")

        # Save the data to a file
        with open("data.txt", "a") as f:
            f.write(f"{name},{email}\n")

if __name__ == "__main__":
    app = QApplication([])
    window = MainWindow()
    window.show()
    app.exec_()
```

In this example, we create a simple PyQT5 application that allows the user to enter a name and email address and stores them in a list called data. We also add a button that the user can click to add the data to the list.

In the on_add_clicked method, we first retrieve the name and email values from the QLineEdit controls. We then check if both values are non-empty and add them to the data list if they are. We also clear the QLineEdit controls to allow the user to enter new data.

Finally, we save the data to a file called "data.txt" using the with statement to automatically close the file after writing to it.

To perform data manipulation operations such as insert, update, delete, and search, you can use various techniques such as SQL queries or list comprehensions. For example, you can use SQL queries to insert data into a MySQL database or use list comprehensions to filter and search data in a list.

Overall, data manipulation and storage are crucial aspects of event-driven programming that allow you to create robust and scalable applications that can handle data generated by your users.

Here's an example of how to perform data manipulation operations in MySQL using PyQt5:

```
import sys
from PyQt5.QtWidgets import QApplication, QWidget, QLabel, QLineEdit,
QPushButton
import mysql.connector

class Example(QWidget):
    def __init__(self):
        super().__init__()
        self.initUI()

    def initUI(self):
        self.lbl1 = QLabel('Name:', self)
        self.lbl1.move(15, 10)
        self.lbl2 = QLabel('Age:', self)
        self.lbl2.move(15, 40)
        self.lbl3 = QLabel('ID:', self)
        self.lbl3.move(15, 70)
```

```
        self.name = QLineEdit(self)
        self.name.move(80, 10)
        self.age = QLineEdit(self)
        self.age.move(80, 40)
        self.id = QLineEdit(self)
        self.id.move(80, 70)

        add_button = QPushButton('Add', self)
        add_button.clicked.connect(self.add_to_database)
        add_button.move(15, 100)

        update_button = QPushButton('Update', self)
        update_button.clicked.connect(self.update_database)
        update_button.move(105, 100)

        delete_button = QPushButton('Delete', self)
        delete_button.clicked.connect(self.delete_from_database)
        delete_button.move(195, 100)

        search_button = QPushButton('Search', self)
        search_button.clicked.connect(self.search_database)
        search_button.move(285, 100)

        self.setGeometry(300, 300, 350, 150)
        self.setWindowTitle('Data Manipulation Operations')
        self.show()

    def add_to_database(self):
        name = str(self.name.text())
        age = int(self.age.text())
        id = int(self.id.text())

        mydb = mysql.connector.connect(
            host="localhost",
            user="yourusername",
            password="yourpassword",
            database="mydatabase"
            )

        mycursor = mydb.cursor()

        sql = "INSERT INTO customers (name, age) VALUES (%s, %s)"
        val = (name, age)

        mycursor.execute(sql, val)
```

```
    mydb.commit()

def update_database(self):
  name = str(self.name.text())
  age = int(self.age.text())
  id = int(self.id.text())

  mydb = mysql.connector.connect(
      host="localhost",
      user="yourusername",
      password="yourpassword",
      database="mydatabase"
      )

  mycursor = mydb.cursor()

  sql = "UPDATE customers SET name=%s WHERE id=%s"
  val = (name,id)

  mycursor.execute(sql,val)

  mydb.commit()

def delete_from_database(self):
  id=int(self.id.text())

  mydb=mysql.connector.connect(
      host="localhost",
      user="yourusername",
      password="yourpassword",
      database="mydatabase"
      )

  mycursor=mydb.cursor()

  sql="DELETE FROM customers WHERE id=%s"
  val=(id,)

  mycursor.execute(sql,val)

  mydb.commit()

def search_database(self):
  id=int(self.id.text())

  mydb=mysql.connector.connect(
      host="localhost",
```

```
            user="yourusername",
            password="yourpassword",
            database="mydatabase"
            )

        mycursor=mydb.cursor()

        sql="SELECT * FROM customers WHERE id=%s"
        val=(id,)

        mycursor.execute(sql,val)

        result=mycursor.fetchall()

        for row in result:
            print(row)

if __name__ == '__main__':
    app = QApplication(sys.argv)
    ex = Example()
    sys.exit(app.exec_())
```

This is a simple PyQt5 application that allows you to perform data manipulation operations such as Inserting data into a MySQL database table using the INSERT INTO statement, Updating data in a MySQL database table using the UPDATE statement, Deleting data from a MySQL database table using the DELETE statement, and Searching for data in a MySQL database table using the SELECT statement