Kivy vs React vs Flutter:

Flutter is a framework developed by Google that uses the Dart language and provides hot reloading, tight VS Code integration and extensive debugging tools.

React Native is a framework developed by Facebook that uses JavaScript and React and allows developers to reuse code across web and mobile platforms2.

Kivy is a library written in Python that supports multiple input devices and can run on Windows, Linux, Android, iOS and Raspberry Pi.

Documentation: Flutter has comprehensive and well-organized documentation, while React Native has less detailed and sometimes outdated documentation. Kivy has decent documentation but it is not as beginner-friendly as Flutter.

Dynamic vs static programming: React Native and Kivy use dynamic languages (JavaScript and Python) that are easier to write and debug, while Flutter uses a static language (Dart) that is faster and more secure but requires more code.

Project size: Flutter tends to produce larger app sizes than React Native and Kivy, which may affect the download time and storage space of your app

Layout: Flutter uses its own rendering engine (Skia) to create widgets that look the same on any device, while React Native and Kivy use native components or platform-specific libraries to create the UI.

Python libraries that has drag and drop feature:

PyQt, PySimpleGUI, MatDeck

Scheduling techniques in OS:

CPU scheduling, disk scheduling, and multiple processor scheduling

SOLID principles and Examples:

1. Single Responsibility Principle
2. Open and Closed Principle
3. Liskov Subtsituation Principle
4. Interface Segregation Principle
5. Dependency Inversion Principle

Single Responsibility Principle(SRP):

A class should have only one responsibility and only one reason to change.

That means a class does not perform multiple jobs.

Example:

Violation of SRP

```
class Account:
  “””Demo bank account class “””
  def __init__(self, account_no: str):
      self.account_no = account_no
  def get_account_number(self):
      “””Get account number”””
      return self.account_no
  def save(self):
    “””Save account number into DB”””
    pass
```

How does it violate the SRP?

In the account class, I am performing two tasks. One is stored data and another one gets an account number. So it violates the SRP.

Solution:

```
class AccountDB:
  """Account DB management class """
  def get_account_number(self, _id):
     """Get account number"""
     pass
  def account_save(self, obj):
     """Save account number into DB"""
     pass

class Account:
  """Demo bank account class """

  def __init__(self, account_no: str):
     self.account_no = account_no
     self._db = AccountDB()
  def get_account_number(self):
     """Get account number"""
     return self.account_no
  def get(self, _id):
     """
     :param _id:
     :return:
     """
     return self._db.get_account_number(_id=_id)

  def save(self):
     """account save"""
     self._db.account_save(obj=self)
```

Open and Closed Principle(OCP):

Software entities (classes, function, module) open for extension, but closed for modification

Example:
Violation of OCP

```
class Discount:
    """Demo customer discount class"""
    def __init__(self, customer, price):
        self.customer = customer
        self.price = price
```

```
  def give_discount(self):
    """A discount method"""
    if self.customer == 'normal':
      return self.price * 0.2
    elif self.customer == 'vip':
      return self.price * 0.4
```

Solution:

```
class Discount:
  """Demo customer discount class"""
  def __init__(self, customer, price):
    self.customer = customer
    self.price = price
  def get_discount(self):
    """A discount method"""
    return self.price * 0.2
class VIPDiscount(Discount):
  """Demo VIP customer discount class"""
  def get_discount(self):
    """A discount method"""
    return super().get_discount() * 2
class SuperVIPDiscount(VIPDiscount):
  """Demo super vip customer discount class"""
  def get_discount(self):
    """A discount method"""
    return super().get_discount() * 2
```

Liskov Substitution Principle(LSP):

if S is a subtype of T, then objects of type T may be replaced by objects of type S, without breaking the program.

Interface Segregation Principle(ISP):

Actually, This principle suggests that "A client should not be forced to implement an interface that it does not use".
It splits interfaces that are very large into smaller and more specific ones so that clients will only have to know about the methods that are of interest to them. It prevents code from being forced to depend on methods it does not use.

Dependency Inversion Principle(DIP):

This principle suggests the below two points.
a. High-level modules should not depend on low-level modules. Both should depend on abstractions.
b. Abstractions should not depend on details. Details should depend on abstractions.

Architecture patterns:

1. Layered pattern

Usage:

- General desktop applications.
- E commerce web applications.

2. Client-server pattern

Usage:

- Online applications such as email, document sharing and banking.

3. Master-slave pattern

Usage:

- In database replication, the master database is regarded as the authoritative source, and the slave databases are synchronized to it.
- Peripherals connected to a bus in a computer system (master and slave drives).

4. Pipe-filter pattern

Usage

- Compilers. The consecutive filters perform lexical analysis, parsing, semantic analysis, and code generation.
- Workflows in bioinformatics.

5. Broker pattern

Usage

- Message broker software such as Apache ActiveMQ, Apache Kafka, RabbitMQ and JBoss Messaging.

6. Peer-to-peer pattern

Usage

- File-sharing networks such as Gnutella and G2)
- Multimedia protocols such as P2PTV and PDTP.
- Cryptocurrency-based products such as Bitcoin and Blockchain

7. Event-bus pattern

Usage

- Android development
- Notification services

8. Model-view-controller pattern

Usage

- Architecture for World Wide Web applications in major programming languages.
- Web frameworks such as Django and Rails.

9. Blackboard pattern

Usage

- Speech recognition
- Vehicle identification and tracking
- Protein structure identification
- Sonar signals interpretation.

10. Interpreter pattern

Usage

- Database query languages such as SQL.

- Languages used to describe communication protocols.