1S. Analise o trecho de código e proposições a seguir.

```
1  int? conta ({int? a, int? b}){
2    print (a! + b!);
3  }
4
5  void main(){
6    print(conta(a: 2, b: 2));
7    print(conta(a: null, b: null));
8  }
```

I A função “conta” utiliza três parâmetros nomeados.
II A linha 7 causa um erro em tempo de compilação.
III Ainda que a e b sejam null, o operador ! usado na linha 2 não causa um erro em tempo de compilação.

É correto apenas o que se afirma em

I
II
III
I e II
II e III

Feedback
A proposição I é falsa. A função “conta” usa apenas dois parâmetros nomeados: a e b. O tipo de retorno dela não tem nada a ver com isso.
A proposição II é falsa. Os parâmetros a e b foram marcados com ? e, portanto, são opcionais, admitindo a atribuição de null. Seu uso, na linha 2, causa um erro, porém ele acontece em tempo de execução pois, ali, já “saltamos do bote que não afunda”, dizendo para o compilador que garantimos que os valores de a e b não serão e que, portanto, podem ser utilizados.
A proposição III é verdadeira. O uso do operador ! caracteriza nosso salto do “bote que não afunda”, ou seja, prometemos para o compilador que as expressões a e b são diferentes de null. Por isso, o compilador sai de cena e compila o código. Na chamada da linha 7, as variáveis a e b recebem null, causando um erro em tempo de execução.

2S Analise o trecho de código e proposições a seguir.

```
import 'package:flutter/material.dart';
void main() {
  var app = MaterialApp(
    home: Scaffold(
      appBar: AppBar(
        title: const Text("Minhas Imagens"),
      ),
      floatingActionButton: FloatingActionButton(
          child: const Icon(Icons.add),
          onPressed: () {
            print("Hello!");
          },
      ),
    ),
  );
  runApp(app);
}
```

I. O código revela que o widget Scaffold possui um parâmetro nomeado chamado home.
II. O widget Scaffold desempenha o papel de “esqueleto” da aplicação e, do ponto de vista visual (alteração de cores, por exemplo) seu uso não causa nenhuma alteração na aplicação.
III. A função associada a onPressed pode ser substituída por uma arrow function sem que isso altere o funcionamento do programa.

É correto apenas o que se afirma em

I
II
III
I e II
II e III

Feedback
A proposição I é falsa. home é, na verdade, um parâmetro nomeado de MaterialApp. O código mostra uma instância de Scaffold associada a ele.
A proposição II é falsa. Sem um Scaffold, a aplicação mostra, por padrão, uma tela com um texto misturando as cores vermelho e amarelo que a documentação classifica como “feia”. Um Scaffold é utilizado para construir um objeto Material cuja propriedade textStyle define o estilo textual da aplicação. Veja a documentação de MaterialApp.

> MaterialApp configures its WidgetsApp.textStyle with an ugly red/yellow text style that's intended to warn the developer that their app hasn't defined a default text style. Typically the app's Scaffold builds a Material widget whose default Material.textStyle defines the text style for the entire scaffold.

A proposição III é verdadeira. As arrow functions estão limitadas a uma única linha e, neste caso, a função exibida tem apenas uma linha. A arrow function seria assim:
( ) => print(“Hello!”);

3S Analise o trecho de código e proposições a seguir.

```dart
import 'package:flutter/material.dart';
class App extends StatelessWidget {
  int variavel = 2;
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      home: Scaffold(
        appBar: AppBar(
          title: Text("$variavel"),
        ), // AppBar
        floatingActionButton: FloatingActionButton(
          child: const Icon(Icons.add),
          onPressed: () {
            variavel++;
          },
        ), // FloatingActionButton
      ), // Scaffold
    ); // MaterialApp
  }
}
```

I A aplicação mostra o valor 2 como título. Quando o botão é clicado, esse valor é incrementado e a aplicação é atualizada graficamente, passando a exibir 3.
II A aplicação mostra o valor 2 como título. Se a função associada à propriedade onPressed for reescrita da seguinte forma

```dart
onPressed: () {
  setState(() => variavel++);
},
```

esse valor é incrementado e a aplicação é atualizada graficamente, passando a exibir 3.
III Ainda que possua uma variável de instância, esse é um widget sem estado.

É correto apenas o que se afirma em

I
II
III
I e II
II e III

Feedback
A proposição I é falsa. A atualização gráfica não acontece: o widget é sem estado e a variável está sendo atualizada diretamente.
A proposição II é falsa. A atualização gráfica não acontece: o widget é sem estado e sequer possui uma função chamada setState.
A proposição III é verdadeira. A classe herda de StatelessWidget e, portanto, o widget é sem estado. A existência da variável de instância é irrelevante.

4S Analise o trecho de código e as proposições a seguir.

```
1  class ImageModel {
2    late String url;
3    late String alt;
4    ImageModel(this.url, this.alt);
5    ImageModel.fromJSON(Map <String, dynamic> decodedJSON) {
6      url = decodedJSON['photos'][0]['src']['medium'];
7      alt = decodedJSON['photos'][0]['alt'];
8    }
9  }
```

I Tanto chaves quanto valores do mapa “decodedJSON” podem ser de tipos quaisquer.
II A classe possui dois construtores.
III A linha 4 define um construtor e ele faz atribuições implícitas. Ou seja, embora não tenhamos escrito nenhuma atribuição (usando o operador “=”) explícita, ele faz pelo menos uma dessas implicitamente.

É correto apenas o que se afirma em

I
II
III
I e II
II e III

Feedback
A proposição I é falsa. As chaves do mapa somente podem ser do tipo String.
A proposição II é verdadeira. A linha quatro define um construtor com o nome da classe. A linha 5 define um construtor nomeado chamado “fromJSON”.
A proposição III é verdadeira. O construtor da linha 4 recebe valores de url e alt como parâmetro e os atribui às variáveis de instância. As atribuições são ditas “implícitas” já que não aparecem explicitamente no código.

5S Analise o trecho de código e proposições a seguir. Suponha que o trecho de código está envolvido na definição de um Widget com estado que funciona corretamente.

```
1  Widget build(BuildContext context) {
2    return MaterialApp(
3      home: Scaffold(
4        appBar: AppBar(
5          title: const Text("Minhas Imagens"),
6        ),
7        floatingActionButton: FloatingActionButton(
8          child: const Icon(Icons.add),
9          onPressed: () {
10           setState(() => numeroImagens++);
11         },
12       ),
13       body: Text('$numeroImagens'),
14     ),
15   );
16 }
```

I Um clique no botão causa uma atualização gráfica que envolve o widget descrito na linha 13.

II A função associada à propriedade onPressed pode ser reescrita como uma arrow function da seguinte forma, sem que isso altere o comportamento do programa.

```
onPressed: () => numeroImagens++
```

III Por estarmos lidando com a definição de um Widget com estado, o método build exibido deve fazer parte de classe que herda de StatefulWidget.

É correto apenas o que se afirma em

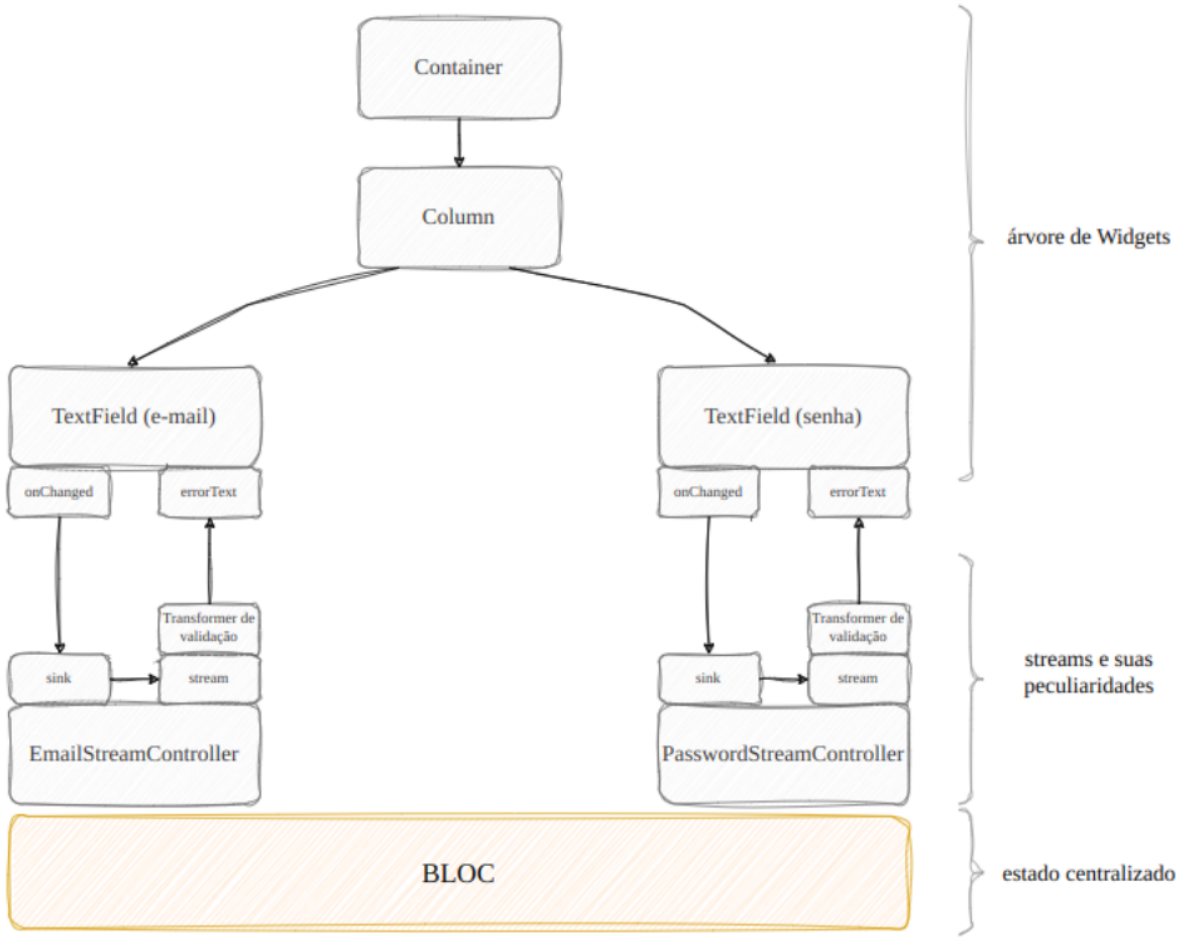
I
II
III
I e II
II e III

Feedback

A proposição I é verdadeira. O clique no botão atualiza a variável utilizando corretamente o mecanismo de estado do widget. Isso, por definição, atualiza a tela.

A proposição II é falsa. Embora a arrow function sugerida seja válida sintaticamente, ela deixa de utilizar o método setState, que é fundamental para que a atualização gráfica aconteça.
A proposição III é falsa. O método build deve fazer parte de uma classe que herda de State.

6S Analise a figura e as proposições a seguir.



I A figura mostra um Widget do tipo Column que possui duas variáveis de instância do tipo StreamController.

II Os fluxos de validação descritos pela figura operam de maneira assíncrona.

III A figura mostra que as validações são acionadas quando o usuário clica em um botão.

É correto apenas o que se afirma em

I
II
III
I e II
II e III

Feedback
A proposição I é falsa. Os controllers são propriedades do BLOC, elas não moram na árvore de widgets.

A proposição II é verdadeira. Utilizamos streams que são acionados quando eventos acontecem. Quando o tratamento de um evento termina, uma “emissão” acontece e, assim, uma atividade (atualização ou não do campo de exibição de mensagem de erro, neste exemplo) pode ser executada.
A proposição III é falsa. Os eventos tratados são do tipo “onChanged” e eles acontecem quando o usuário digita algo num campo textual.

7S Analise o trecho de código e as proposições a seguir.

```
1  from rest_framework import generics
2  from .models import Filme
3  from .serializers import FilmeSerializer
4  class FilmeListCreate(generics.ListCreateAPIView):
5    queryset = Filme.objects.all()
6    serializer_class = FilmeSerializer
7
8  class FilmeRetrieveDestroy(generics.RetrieveDestroyAPIView):
9    queryset = Filme.objects.all()
10   serializer_class = FilmeSerializer
```

I Uma das views lida com o método POST do protocolo HTTP.
II Ambas as views lidam com o método GET do protocolo HTTP.
III Ambas as views apresentam um erro em tempo de execução causado pela inexistência da classe interna chamada Meta.

É correto apenas o que se afirma em

I
II
III
I e II
II e III

Feedback
A proposição I é verdadeira. A view definida na linha 4 herda de ListCreateAPIView. O método HTTP utilizado para criar recursos é o POST.
A proposição II é verdadeira. A view da linha 4 lida com o método GET do protocolo HTTP para obter a lista de recursos. A view da linha 8 lida com o método GET do protocolo HTTP para obter um recurso pelo seu id, muito embora esse mapeamento não seja feito neste arquivo.
A proposição III é falsa. As views não requerem tal classe interna. Essa é uma necessidade das classes Serializers.

8S Analise o comando e a proposição a seguir e assinale verdadeiro ou falso.

```
python manage.py makemigrations filmes_app
```

O comando tem o potencial de alterar a estrutura da base de dados.

A proposição é falsa. O comando tem o potencial de gerar arquivos .py que descrevem diferenças eventualmente existentes entre aquilo que as classes de modelo descrevem e a estrutura da base existente no momento. Entretanto, não cabe a este comando a execução destes arquivos.

9S Analise o trecho de código e as proposições a seguir.

```
from django.db import models
class Genero(models.Model):
  descricao = models.CharField(max_length=100)
    def __str__(self):
      return self.descricao

class Filme(models.Model):
  titulo = models.CharField(max_length=100)
  descricao = models.TextField()
  diretor = models.CharField(max_length=100)
    def __str__(self):
      return self.titulo
```

I Apesar da estrutura definida pela classe Filme, o código não revela quais campos o usuário deve incluir no JSON a ser enviado caso deseje cadastrar um novo filme.
II Por serem classes de modelo, a existência do método __str__ em ambas é obrigatória.
III As classes de modelo estão implicitamente mapeadas aos padrões de acesso host:port/generos/ e host:port/filmes/.

É correto apenas o que se afirma em

I
II
III
I e II
II e III

Feedback

A proposição I é verdadeira. A decisão sobre quais campos devem ser incluídos no JSON cabe a uma classe Serializer.
A proposição II é falsa. O método __str__ é equivalente ao toString das linguagens Java e Dart, por exemplo. Ele somente calcula a representação textual de um objeto. O fato de estarmos lidando com classes de modelo do DRF não torna a sua existência obrigatória.
A proposição III é falsa. Não há mapeamento implícito. Ele deve ser feito explicitamente e envolve a definição do padrão de acesso, bem como da classe view correspondente.

10S Analise o trecho de código e as proposições a seguir.

```python
from django.contrib.auth.models import User
from rest_framework import serializers
import re
class UserSerializer(serializers.ModelSerializer):
  password = serializers.CharField(write_only=True)
  def validate_password(self, password):
    if not re.search('[A-Z]', password):
      raise serializers.ValidationError("A senha deve conter pelo menos uma letra maiúscula")
    if not re.search('[a-z]', password):
      raise serializers.ValidationError("A senha deve conter pelo menos uma letra minúscula")
    if not re.search('[0-9]', password):
      raise serializers.ValidationError("A senha deve conter pelo menos um número")
    if not re.search('[^a-zA-Z0-9]', password):
      raise serializers.ValidationError("A senha deve conter pelo menos um caracter especial")
    if len(password) < 8:
      raise serializers.ValidationError("A senha deve conter pelo menos oito caracteres")
    return password
  class Meta:
    model = User
    fields = ('id', 'username', 'email', 'password')
```

I O código revela validações padrão realizadas internamente pelo DRF.
II O código revela que o programador escreveu uma classe chamada User que possui, pelo menos, os campos id, username, email e password.
III Basta que uma das condições de um if do método validate_password seja avaliada como True para que um ValidationError seja lançado e a execução do método seja encerrada.

É correto apenas o que se afirma em

I
II
III
I e II
II e III

Feedback
A proposição I é falsa. As validações exibidas foram explicitamente escritas pelo desenvolvedor, elas não são padrão do DRF.
A proposição II é falsa. A classe User faz parte das classes de modelo de uma aplicação Django.
A proposição III é verdadeira. Se uma condição de um dos if for avaliada como true, uma instrução raise será executada. Em Python, as instruções raise causam o encerramento da execução do método atual e, neste caso, um ValidationError é lançado por cada uma delas.