***O QUE SÃO MODELS*?**

***Models*** se refere à parte do framework que lida com a definição e manipulação de dados do aplicativo. Aqui está uma descrição dos "models" no contexto do Django:

Eles são responsáveis por definir a estrutura e o comportamento dos dados em um aplicativo Django.

**Definição de Dados**: Os "models" do Django são classes Python que representam as tabelas do banco de dados. Cada classe de modelo geralmente mapeia diretamente para uma tabela no banco de dados, e os atributos da classe representam as colunas dessa tabela.

**Campos de Modelo**: Os atributos de uma classe de modelo Django são geralmente campos de modelo que definem o tipo de dado e outras características da coluna correspondente no banco de dados. Django fornece uma variedade de tipos de campo, como IntegerField, CharField, DateField, ForeignKey, entre outros.

**Relacionamentos**: Os modelos do Django podem definir relacionamentos entre si, como chaves estrangeiras (ForeignKey), chaves primárias (OneToOneField), e relações de muitos-para-muitos (ManyToManyField). Isso permite representar estruturas complexas de dados e associar objetos uns aos outros.

**Administração de Dados**: O Django fornece uma interface de administração automática para os modelos, onde os usuários podem visualizar, adicionar, modificar e excluir registros do banco de dados por meio de uma interface de usuário pré-construída. Isso é muito útil durante o desenvolvimento e para tarefas de gerenciamento básico de dados.

**Migrações**: O Django inclui um sistema de migrações que permite gerenciar as alterações na estrutura do banco de dados ao longo do tempo. Quando você altera a definição de um modelo, o Django pode gerar automaticamente as migrações necessárias para atualizar o banco de dados para refletir essas alterações.

Em resumo, os "models" no Django fornecem uma maneira conveniente e poderosa de definir a estrutura de dados de um aplicativo, facilitando a manipulação e o acesso aos dados do banco de dados por meio de uma interface Pythonic.

**Model CharField** é um tipo de campo de modelo usado para armazenar strings de comprimento limitado. Aqui está uma descrição detalhada do **CharField**:

Parâmetro obrigatório: Definir o comprimento máximo da string usando o argumento max\_length(valor máximo é 254)

nome = models.CharField(max\_length=100)

**Armazenamento de Strings Curtas**: O **CharField** é usado para armazenar strings de comprimento limitado, como nomes, títulos, descrições curtas, etc. Ele é adequado para dados que não excedem um determinado número de caracteres.

**Definição de Comprimento Máximo**: Ao definir um **CharField**, você pode especificar o comprimento máximo permitido para a string. Por exemplo, **CharField(max\_length=100)** limitará a entrada a 100 caracteres.

**Validação Automática**: O Django valida automaticamente os dados inseridos em um **CharField** para garantir que eles não excedam o comprimento máximo especificado. Se uma string ultrapassar o limite, uma exceção será levantada e o dado não será salvo no banco de dados.

**Armazenamento Eficiente**: No banco de dados, um **CharField** é normalmente armazenado como um tipo de dado de caractere fixo (CHAR), o que pode ser mais eficiente em termos de espaço de armazenamento em comparação com tipos de dados de string variáveis como VARCHAR.

**Uso em Formulários**: Os **CharFields** são frequentemente usados em formulários Django para receber entrada de texto curto dos usuários. Eles são renderizados como caixas de texto simples em formulários HTML.

**Tradução de Dados**: O **CharField** também suporta tradução de dados em modelos multilíngues, permitindo armazenar diferentes traduções de um campo de texto em vários idiomas.

No geral, o **CharField** é uma ferramenta versátil e amplamente utilizada em modelos Django para armazenar e manipular dados de texto curtos.

**Model TextField** é um tipo de campo de modelo utilizado para armazenar strings de comprimento variável, geralmente para textos longos. Aqui estão algumas características importantes do **TextField**:

Não possui parâmetro obrigatório

sobre\_voce = models.TextField()

**Armazenamento de Texto Longo**: O **TextField** é ideal para armazenar grandes volumes de texto, como descrições, comentários, conteúdo de postagens de blog, entre outros. Ele não tem um limite rígido de comprimento como o **CharField**.

**Sem Limite de Comprimento Explícito**: Ao contrário do **CharField**, o **TextField** não requer a especificação de um comprimento máximo. Isso permite armazenar grandes blocos de texto sem preocupações com limites arbitrários de tamanho.

**Adequado para Texto Formatado**: O **TextField** é frequentemente usado quando o conteúdo precisa ser formatado de forma mais complexa, como parágrafos, quebras de linha, listas, etc. Ele suporta texto formatado usando HTML, Markdown ou outros formatos de marcação.

**Armazenamento Eficiente**: Embora o **TextField** possa armazenar grandes volumes de texto, ele é eficiente em termos de armazenamento de banco de dados. Os dados são armazenados de forma otimizada, ocupando apenas o espaço necessário para o conteúdo do texto.

**Uso em Formulários**: Assim como o **CharField**, o **TextField** também é usado em formulários Django para capturar entrada de texto longo dos usuários. Ele é frequentemente renderizado como uma área de texto multi-linha em formulários HTML.

**Tradução de Dados**: O **TextField** também suporta tradução de dados em modelos multilíngues, permitindo armazenar diferentes traduções de um campo de texto longo em vários idiomas.

o **TextField** é uma ferramenta poderosa para armazenar e manipular grandes volumes de texto em modelos Django, oferecendo flexibilidade e eficiência para lidar com conteúdo textual variado.

**Model IntegerField** é um tipo de campo de modelo utilizado para armazenar números inteiros. Aqui estão algumas características importantes do **IntegerField**:

Não possui parâmetro obrigatório

idade = models.IntegerField()

**Armazenamento de Números Inteiros**: O **IntegerField** é usado para armazenar valores inteiros, como números de identificação, quantidades, pontuações, entre outros.

**Sem Decimais**: O **IntegerField** não suporta valores decimais. Se precisar de suporte para valores decimais, você deve usar **DecimalField**.

**Definição de Intervalo de Valores**: Você pode opcionalmente definir limites mínimo e máximo para os valores que o campo **IntegerField** pode armazenar. Por exemplo, você pode definir **IntegerField(min\_value=0, max\_value=100)** para limitar os valores armazenados entre 0 e 100.

**Validação Automática**: O Django valida automaticamente os dados inseridos em um **IntegerField**, garantindo que sejam números inteiros válidos dentro do intervalo especificado, se houver.

**Armazenamento Eficiente**: No banco de dados, um **IntegerField** é armazenado como um tipo de dado de inteiro, ocupando um espaço de armazenamento mínimo.

**Uso em Formulários**: Os **IntegerFields** são frequentemente usados em formulários Django para receber entrada numérica dos usuários. Eles são renderizados como caixas de entrada de texto para números inteiros em formulários HTML.

**Relacionamentos de Chave Estrangeira**: O **IntegerField** pode ser usado para representar chaves estrangeiras em modelos, embora seja mais comum usar **ForeignKey** para esse fim.

o **IntegerField** é uma ferramenta útil para armazenar e manipular números inteiros em modelos Django, oferecendo validação automática e eficiência de armazenamento.

**Model FloatField** é um tipo de campo de modelo utilizado para armazenar números de ponto flutuante, ou seja, números que podem conter parte decimal. Aqui estão algumas características importantes do **FloatField**:

Não possui parâmetro obrigatório

altura = models.FloatField()

**Armazenamento de Números de Ponto Flutuante**: O **FloatField** é usado para armazenar valores numéricos que podem incluir parte decimal, como valores monetários, medidas precisas, entre outros.

**Precisão Decimal Limitada**: Embora os **FloatFields** possam armazenar números com parte decimal, a precisão é limitada devido à natureza binária da representação de ponto flutuante em computadores. Isso significa que eles podem não ser apropriados para valores que requerem alta precisão, como cálculos financeiros.

**Definição de Intervalo de Valores**: Assim como o **IntegerField**, você pode opcionalmente definir limites mínimo e máximo para os valores que o campo **FloatField** pode armazenar.

**Validação Automática**: O Django valida automaticamente os dados inseridos em um **FloatField**, garantindo que sejam números de ponto flutuante válidos dentro do intervalo especificado, se houver.

**Armazenamento Eficiente**: No banco de dados, um **FloatField** é armazenado como um tipo de dado de ponto flutuante, ocupando um espaço de armazenamento mínimo.

**Uso em Formulários**: Os **FloatFields** são frequentemente usados em formulários Django para receber entrada numérica com parte decimal dos usuários. Eles são renderizados como caixas de entrada de texto para números de ponto flutuante em formulários HTML.

**Precisão Limitada**: Devido à precisão limitada dos números de ponto flutuante, é importante ter cuidado ao lidar com cálculos que envolvem valores monetários ou outras medidas que requerem alta precisão.

Em resumo, o **FloatField** é uma ferramenta útil para armazenar e manipular números de ponto flutuante em modelos Django, oferecendo validação automática e eficiência de armazenamento.

**Model BooleanField** é um tipo de campo de modelo utilizado para armazenar valores booleanos, ou seja, valores que representam verdadeiro ou falso. Aqui estão algumas características importantes do **BooleanField**:

Parâmetro OPCIONAL: Definir o valor padrão usando o argumento default

possui\_filhos = models.BooleanField(default=False)

**Armazenamento de Valores Booleanos**: O **BooleanField** é usado para armazenar valores booleanos, que podem ser verdadeiros (**True**) ou falsos (**False**). Este tipo de campo é frequentemente usado para representar opções binárias, como "ativo/inativo", "sim/não", "verdadeiro/falso", etc.

**Flexibilidade de Representação**: Internamente, o **BooleanField** é armazenado no banco de dados como um tipo de dado booleano (geralmente um bit), ocupando um espaço de armazenamento mínimo. Na interface de usuário, ele é geralmente representado como uma caixa de seleção (checkbox) ou como um par de botões de opção.

**Ausência de Valores Nulos**: O **BooleanField** não suporta valores nulos (**NULL**). Em vez disso, ele pode ter um valor padrão definido, geralmente **True** ou **False**, para indicar o estado inicial do campo se nenhum valor for explicitamente fornecido.

**Uso em Formulários**: Os **BooleanFields** são frequentemente usados em formulários Django para representar opções binárias que os usuários podem selecionar. Eles são renderizados como caixas de seleção (checkbox) em formulários HTML.

**Validação Automática**: O Django valida automaticamente os dados inseridos em um **BooleanField**, garantindo que sejam valores booleanos válidos (**True** ou **False**).

**Expressões de Consulta Simples**: O **BooleanField** é útil para consultas simples, como filtrar registros com base em um estado booleano específico, por exemplo, recuperar todos os objetos onde o campo **ativo** é **True**.

o **BooleanField** é uma ferramenta útil para representar valores booleanos em modelos Django, oferecendo uma forma simples e eficiente de armazenar e manipular opções binárias.

**Model DateField** é um tipo de campo de modelo usado para armazenar datas (sem hora) em um formato de data. Aqui estão algumas características importantes do **DateField**:

Não possui parâmetro obrigatório

data\_nascimento = models.DateField()

**Armazenamento de Datas**: O **DateField** é usado para armazenar datas específicas, como datas de nascimento, datas de eventos, prazos, etc. Ele não inclui informações de hora, apenas o dia, mês e ano.

**Formato de Data**: Internamente, o **DateField** armazena as datas no formato "AAAA-MM-DD" (ano-mês-dia), compatível com o formato padrão de data usado pelo banco de dados.

**Validação Automática**: O Django valida automaticamente os dados inseridos em um **DateField**, garantindo que sejam datas válidas no formato esperado.

**Uso em Formulários**: Os **DateFields** são frequentemente usados em formulários Django para receber entrada de datas dos usuários. Eles são renderizados como campos de entrada de data em formulários HTML, permitindo que os usuários selecionem uma data de um calendário ou a digitem manualmente.

**Suporte a Null**: Por padrão, o **DateField** aceita valores nulos (**NULL**), o que significa que um campo de data pode estar vazio. Você pode opcionalmente definir **null=False** para exigir que o campo seja preenchido.

**Definição de Intervalo de Datas**: Você pode opcionalmente definir limites mínimo e máximo para as datas que o campo **DateField** pode armazenar, utilizando os argumentos **auto\_now** e **auto\_now\_add**.

**Trabalhando com Datas**: O Django fornece uma variedade de métodos e funções úteis para trabalhar com datas em modelos, como filtrar registros com base em datas específicas, calcular a diferença entre datas, formatar datas para exibição, etc.

o **DateField** é uma ferramenta útil para armazenar e manipular datas em modelos Django, oferecendo validação automática e facilitando o trabalho com informações de calendário.

**Model** **DateTimeField** é um tipo de campo de modelo usado para armazenar datas e horas juntas. Aqui estão algumas características importantes do **DateTimeField**:

Não possui parâmetro obrigatório

data\_de\_criacao = models.DateTimeField()

**Armazenamento de Data e Hora**: O **DateTimeField** é usado para armazenar informações de data e hora em um único campo. Ele inclui informações sobre o ano, mês, dia, hora, minuto, segundo e microssegundo.

**Formato de Data e Hora**: Internamente, o **DateTimeField** armazena as datas e horas no formato "AAAA-MM-DD HH:MM:SS.ssssss" (ano-mês-dia hora:minuto:segundo.microssegundo), compatível com o formato padrão de data e hora usado pelo banco de dados.

**Validação Automática**: O Django valida automaticamente os dados inseridos em um **DateTimeField**, garantindo que sejam datas e horas válidas no formato esperado.

**Uso em Formulários**: Os **DateTimeFields** são frequentemente usados em formulários Django para receber entrada de datas e horas dos usuários. Eles são renderizados como campos de entrada de data e hora em formulários HTML, permitindo que os usuários selecionem uma data e hora de um calendário ou as digitem manualmente.

**Suporte a Null**: Por padrão, o **DateTimeField** aceita valores nulos (**NULL**), o que significa que um campo de data e hora pode estar vazio. Você pode opcionalmente definir **null=False** para exigir que o campo seja preenchido.

**Definição de Intervalo de Data e Hora**: Você pode opcionalmente definir limites mínimo e máximo para as datas e horas que o campo **DateTimeField** pode armazenar, utilizando os argumentos **auto\_now** e **auto\_now\_add**.

**Trabalhando com Datas e Horas**: O Django fornece uma variedade de métodos e funções úteis para trabalhar com datas e horas em modelos, como filtrar registros com base em datas e horas específicas, calcular a diferença entre datas e horas, formatar datas e horas para exibição, etc.

o **DateTimeField** é uma ferramenta útil para armazenar e manipular datas e horas em modelos Django, oferecendo validação automática e facilitando o trabalho com informações de data e hora.

**Model ForeignKey** é um tipo de campo de modelo usado para estabelecer uma relação de chave estrangeira entre dois modelos. Aqui estão algumas características importantes do **ForeignKey**:

Parâmetro obrigatório 1: “To”, Indica o modelo ao qual a chave estrangeira se refere.

Parâmetro obrigatório 2: “on\_delete”, Especifica o que acontece com os objetos relacionados quando o objeto referenciado é excluído.

autor = models. ForeignKey(Pessoa, on\_delete=models.CASCADE)

**Estabelecimento de Relacionamento**: O **ForeignKey** é usado para estabelecer uma relação de "muitos para um" entre dois modelos. Isso significa que cada instância do modelo que contém o campo **ForeignKey** está associada a uma única instância do modelo referenciado.

**Representação da Chave Estrangeira**: O **ForeignKey** armazena a chave primária da instância relacionada do modelo referenciado. Essa chave é usada para conectar os dois modelos no banco de dados.

**Uso de Nomes de Classe**: Ao definir um **ForeignKey**, você especifica o nome da classe do modelo que está sendo referenciado. Por exemplo, **ForeignKey(ClasseModeloReferenciado)**.

**Atributos Opcionais**: Além de especificar o modelo referenciado, você pode usar vários atributos opcionais para personalizar o comportamento do **ForeignKey**, como **on\_delete**, **related\_name**, **limit\_choices\_to**, entre outros.

**Atributo on\_delete**: Este atributo especifica o que acontece com os objetos relacionados quando o objeto referenciado é excluído. As opções incluem **CASCADE**, **PROTECT**, **SET\_NULL**, **SET\_DEFAULT** e **SET()**, entre outras.

**Atributo related\_name**: Este atributo permite especificar um nome personalizado para o relacionamento inverso do **ForeignKey**. Isso é útil quando há mais de um campo **ForeignKey** apontando para o mesmo modelo.

**Uso em Consultas**: Você pode usar campos **ForeignKey** em consultas para acessar objetos relacionados e filtrar registros com base em campos do modelo referenciado.

**Uso em Templates**: No Django, é fácil acessar e exibir dados de objetos relacionados em templates usando a sintaxe de ponto para acessar os atributos do modelo relacionado.

o **ForeignKey** é uma ferramenta poderosa para estabelecer relacionamentos entre modelos em Django, permitindo representar de forma eficiente associações complexas entre diferentes tipos de dados.

**Model ManyToManyField** é um tipo de campo de modelo usado para estabelecer uma relação de muitos para muitos entre dois modelos. Aqui estão algumas características importantes do **ManyToManyField**:

Parâmetro obrigatório: “To”, Indica o modelo ao qual o campo many-to-many está associado

livros\_produzidos = models.ManyToManyField(Livro)

**Estabelecimento de Relacionamento**: O **ManyToManyField** é usado para estabelecer uma relação de "muitos para muitos" entre dois modelos. Isso significa que cada instância de um modelo pode estar associada a várias instâncias do outro modelo, e vice-versa.

**Representação da Relação**: Internamente, o **ManyToManyField** cria uma tabela de associação separada no banco de dados para armazenar as relações entre os dois modelos. Essa tabela de associação contém pares de chaves estrangeiras que conectam as instâncias dos dois modelos.

**Uso de Nomes de Classe**: Ao definir um **ManyToManyField**, você especifica o nome da classe do modelo que está sendo relacionado. Por exemplo, **ManyToManyField(ClasseModeloRelacionado)**.

**Atributos Opcionais**: Além de especificar o modelo relacionado, você pode usar vários atributos opcionais para personalizar o comportamento do **ManyToManyField**, como **related\_name**, **symmetrical**, **through**, entre outros.

**Atributo related\_name**: Este atributo permite especificar um nome personalizado para o relacionamento inverso do **ManyToManyField**. Isso é útil quando há mais de um campo **ManyToManyField** apontando para o mesmo modelo.

**Atributo symmetrical**: Por padrão, o **ManyToManyField** é simétrico, o que significa que se um objeto A estiver relacionado a um objeto B, então o objeto B também estará relacionado ao objeto A. Você pode desativar essa simetria definindo **symmetrical=False**.

**Atributo through**: Este atributo permite especificar um modelo intermediário personalizado para a tabela de associação. Isso é útil quando você precisa adicionar campos extras à tabela de associação.

**Manipulação de Relacionamentos**: O Django fornece uma API conveniente para adicionar, remover e verificar objetos relacionados usando métodos como **add()**, **remove()**, **clear()**, **set()**, entre outros.

**Uso em Consultas**: Você pode usar campos **ManyToManyField** em consultas para acessar e filtrar registros com base em objetos relacionados.

o **ManyToManyField** é uma ferramenta poderosa para representar relacionamentos de muitos para muitos entre modelos em Django, permitindo modelar de forma eficiente associações complexas entre diferentes tipos de dados.

**Model PositiveIntegerField** é um tipo de campo de modelo usado para armazenar números inteiros positivos. Aqui estão algumas características importantes do **PositiveIntegerField**:

“Não possui parâmetro obrigatório”

idade = models.PositiveIntegerField()

**Armazenamento de Números Inteiros Positivos**: O **PositiveIntegerField** é usado para armazenar valores inteiros que devem ser positivos, ou seja, números maiores que zero.

**Validação Automática de Positividade**: O Django valida automaticamente os dados inseridos em um **PositiveIntegerField**, garantindo que sejam números inteiros positivos válidos.

**Definição de Intervalo de Valores**: O **PositiveIntegerField** não suporta valores negativos. Você pode opcionalmente definir limites mínimo e máximo para os valores que o campo pode armazenar, usando os argumentos **min\_value** e **max\_value**.

**Armazenamento Eficiente**: No banco de dados, um **PositiveIntegerField** é armazenado como um tipo de dado de inteiro, ocupando um espaço de armazenamento mínimo.

**Uso em Formulários**: Os **PositiveIntegerFields** são frequentemente usados em formulários Django para receber entrada numérica positiva dos usuários. Eles são renderizados como caixas de entrada de texto para números inteiros em formulários HTML.

**Uso em Consultas**: Você pode usar campos **PositiveIntegerField** em consultas para acessar e filtrar registros com base em valores inteiros positivos.

o **PositiveIntegerField** é uma ferramenta útil para armazenar e manipular números inteiros positivos em modelos Django, oferecendo validação automática e eficiência de armazenamento. Ele é particularmente útil quando você precisa garantir que apenas valores positivos sejam inseridos em um campo específico.

**Model EmailField** é um tipo de campo de modelo usado para armazenar endereços de e-mail. Aqui estão algumas características importantes do **EmailField**:

“Não possui parâmetro obrigatório”

email = models.EmailField()

**Armazenamento de Endereços de E-mail**: O **EmailField** é usado para armazenar endereços de e-mail válidos.

**Validação Automática de E-mail**: O Django valida automaticamente os dados inseridos em um **EmailField**, garantindo que sigam o formato padrão de um endereço de e-mail válido.

**Restrições de Formato**: O **EmailField** verifica se o formato do endereço de e-mail é válido, incluindo um nome de usuário seguido por "@" e um domínio válido. Além disso, ele pode verificar se o domínio do e-mail possui um registro MX válido.

**Armazenamento Eficiente**: No banco de dados, um **EmailField** é geralmente armazenado como uma string de texto, ocupando um espaço de armazenamento mínimo.

**Uso em Formulários**: Os **EmailFields** são frequentemente usados em formulários Django para receber entrada de endereços de e-mail dos usuários. Eles são renderizados como caixas de entrada de texto em formulários HTML.

**Uso em Consultas**: Você pode usar campos **EmailField** em consultas para acessar e filtrar registros com base em endereços de e-mail.

**Personalização da Mensagem de Erro**: Você pode personalizar a mensagem de erro exibida quando um endereço de e-mail inválido é fornecido, definindo o parâmetro **error\_messages** ao definir o campo **EmailField**.

o **EmailField** é uma ferramenta útil para armazenar e manipular endereços de e-mail em modelos Django, oferecendo validação automática e facilitando a captura de informações de contato dos usuários.

**Model ImageField** é um tipo de campo de modelo usado para armazenar imagens. Aqui estão algumas características importantes do **ImageField**:

Parâmetro OPCIONAL: “upload\_to”, Cria uma pasta para salvar somente os arquivos deste campo.

foto\_perfil = models.ImageField(upload\_to="perfil/")

**Armazenamento de Imagens**: O **ImageField** é usado para armazenar arquivos de imagem, como JPEG, PNG, GIF, entre outros formatos de imagem suportados.

**Upload de Imagens**: Os **ImageFields** permitem que os usuários façam upload de arquivos de imagem por meio de formulários web. Eles são frequentemente usados em conjunto com o widget de upload de arquivo do Django.

**Armazenamento Eficiente**: No banco de dados, um **ImageField** armazena apenas o caminho para o arquivo de imagem no sistema de arquivos, não a imagem em si. A imagem em si é armazenada no sistema de arquivos do servidor.

**Validação Automática**: O Django valida automaticamente os dados inseridos em um **ImageField**, garantindo que sejam arquivos de imagem válidos e que atendam aos requisitos de tamanho e formato especificados, se houver.

**Redimensionamento e Recorte de Imagens**: O Django fornece suporte para redimensionamento e recorte de imagens diretamente no nível do modelo, usando as opções **width\_field**, **height\_field** e **upload\_to**.

**Uso em Formulários**: Os **ImageFields** são frequentemente usados em formulários Django para permitir que os usuários façam upload de imagens. Eles são renderizados como campos de upload de arquivo em formulários HTML.

**Uso em Consultas**: Você pode usar campos **ImageField** em consultas para acessar e filtrar registros com base em imagens.

**Manipulação de Imagens**: O Django fornece várias ferramentas para manipulação de imagens, incluindo redimensionamento, recorte, rotação e conversão de formato de imagem.

o **ImageField** é uma ferramenta útil para armazenar e manipular arquivos de imagem em modelos Django, oferecendo validação automática e funcionalidades avançadas de manipulação de imagem. Ele é frequentemente usado em aplicativos da web que precisam lidar com upload e exibição de imagens, como galerias de fotos, avatares de usuário e mídias sociais.

**Model** **FileField** é um tipo de campo de modelo usado para armazenar arquivos. Aqui estão algumas características importantes do **FileField**:

Parâmetro OPCIONAL: “upload\_to”, Cria uma pasta para salvar somente os arquivos deste campo

documentos = models.FileField(upload\_to="documentos/")

**Armazenamento de Arquivos**: O **FileField** é usado para armazenar arquivos de qualquer tipo, como documentos, planilhas, PDFs, imagens, entre outros.

**Upload de Arquivos**: Os **FileFields** permitem que os usuários façam upload de arquivos por meio de formulários web. Eles são frequentemente usados em conjunto com o widget de upload de arquivo do Django.

**Armazenamento Eficiente**: No banco de dados, um **FileField** armazena apenas o caminho para o arquivo no sistema de arquivos, não o arquivo em si. O arquivo é armazenado no sistema de arquivos do servidor.

**Validação Automática**: O Django valida automaticamente os dados inseridos em um **FileField**, garantindo que sejam arquivos válidos e que atendam aos requisitos de tamanho e formato especificados, se houver.

**Uso em Formulários**: Os **FileFields** são frequentemente usados em formulários Django para permitir que os usuários façam upload de arquivos. Eles são renderizados como campos de upload de arquivo em formulários HTML.

**Uso em Consultas**: Você pode usar campos **FileField** em consultas para acessar e filtrar registros com base em arquivos.

**Manipulação de Arquivos**: O Django fornece várias ferramentas para manipulação de arquivos, incluindo salvamento, exclusão, renomeação e leitura de arquivos.

**Limite de Tamanho e Formato**: Você pode opcionalmente definir um limite de tamanho máximo e uma lista de formatos de arquivo permitidos usando os parâmetros **upload\_to** e **validators**.

o **FileField** é uma ferramenta útil para armazenar e manipular arquivos em modelos Django, oferecendo validação automática e funcionalidades avançadas de manipulação de arquivos. Ele é frequentemente usado em aplicativos da web que precisam lidar com upload e gerenciamento de diversos tipos de arquivos.

**Model** **OneToOneField** é um tipo de campo de modelo usado para estabelecer uma relação de um para um entre dois modelos. Aqui estão algumas características importantes do **OneToOneField**:

Parâmetro obrigatório 1: “To”, Indica o modelo ao qual a campo OneToOneField se refere.

Parâmetro obrigatório 2: “on\_delete”, especifica o que acontece com os objetos relacionados quando o objeto referenciado é excluído.

aluno = models.OneToOneField(Aluno, on\_delete=models.CASCADE)

**Estabelecimento de Relacionamento**: O **OneToOneField** é usado para estabelecer uma relação onde cada instância de um modelo está associada a uma única instância do outro modelo, e vice-versa.

**Representação da Chave Estrangeira**: O **OneToOneField** armazena a chave primária da instância relacionada do modelo referenciado. Essa chave é usada para conectar os dois modelos no banco de dados.

**Uso de Nomes de Classe**: Ao definir um **OneToOneField**, você especifica o nome da classe do modelo que está sendo referenciado. Por exemplo, **OneToOneField(ClasseModeloReferenciado)**.

**Atributos Opcionais**: Além de especificar o modelo referenciado, você pode usar vários atributos opcionais para personalizar o comportamento do **OneToOneField**, como **on\_delete** e **related\_name**.

**Atributo on\_delete**: Este atributo especifica o que acontece com o objeto relacionado quando o objeto referenciado é excluído. As opções incluem **CASCADE**, **PROTECT**, **SET\_NULL**, **SET\_DEFAULT** e **SET()**.

**Atributo related\_name**: Este atributo permite especificar um nome personalizado para o relacionamento inverso do **OneToOneField**. Isso é útil quando há mais de um campo **OneToOneField** apontando para o mesmo modelo.

**Uso em Consultas**: Você pode usar campos **OneToOneField** em consultas para acessar e filtrar registros com base em objetos relacionados.

o **OneToOneField** é uma ferramenta útil para estabelecer relacionamentos de um para um entre modelos em Django, permitindo representar de forma eficiente associações específicas e exclusivas entre diferentes tipos de dados.