🧱 1. Setting up a Python Virtual Environment

A **virtual environment (venv)** is an isolated workspace that keeps your project's dependencies separate from your global Python installation.

🧩 Step 1: Create a Project Folder

Choose or create a folder for your bot project:

```
mkdir telegram_bot
cd telegram_bot
```

🧩 Step 2: Create a Virtual Environment

Run this command:

```
python -m venv venv
```

This creates a folder named `venv` containing its own Python interpreter and packages.

🧩 Step 3: Activate the Virtual Environment

🪟 On Windows (PowerShell):

```
venv\Scripts\activate
```

🐧 On macOS/Linux:

```
source venv/bin/activate
```

Once activated, your terminal prompt will look like this:

```
(venv) C:\Users\YourName\telegram_bot>
```

This indicates you're working *inside* the virtual environment.

🧩 Step 4: Install Dependencies

Install the **Telegram bot library** and other essentials:

```
pip install python-telegram-bot
```

(Optional for advanced bots):

```
pip install requests python-dotenv
```

🧩 Step 5: Freeze Requirements (Best Practice)

Save the dependencies in a file for deployment or sharing:

```
pip freeze > requirements.txt
```

This generates a file like:

```
python-telegram-bot==21.4
requests==2.31.0
```

When deploying elsewhere, just run:

```
pip install -r requirements.txt
```

🤖 2. Working with Bot Message Handlers

Once your bot environment is ready, you'll define **handlers** that tell the bot *how to respond to messages*.

🧩 The Core Idea

Every Telegram bot listens for **updates** (new messages, commands, buttons, etc.). Each type of interaction is processed by a **Handler** — like a router.

Handler Type	Purpose
`CommandHandler`	Handles commands (e.g., `/start`, `/help`)
`MessageHandler`	Handles plain text messages
`CallbackQueryHandler`	Handles inline button clicks
`ConversationHandler`	Manages multi-step chats

🧠 Basic Bot Structure with Handlers

Create a file called `bot.py` inside your project folder and write:

```
from telegram import Update
from telegram.ext import ApplicationBuilder, CommandHandler, MessageHandler,

BOT_TOKEN = "YOUR_BOT_TOKEN_HERE"

# --- COMMAND HANDLERS ---
async def start(update: Update, context: ContextTypes.DEFAULT_TYPE):
    await update.message.reply_text("👋 Hello! I’m your Python bot. Use /he

async def help_command(update: Update, context: ContextTypes.DEFAULT_TYPE):
    await update.message.reply_text(
        "/start - Start the bot\n"
        "/help - Show available commands\n"
        "/about - Learn about this bot"
    )

async def about(update: Update, context: ContextTypes.DEFAULT_TYPE):
    await update.message.reply_text("🤖 I’m a demo Telegram bot built with

# --- MESSAGE HANDLER (Non-commands) ---
async def echo_message(update: Update, context: ContextTypes.DEFAULT_TYPE):
    text = update.message.text
    await update.message.reply_text(f"You said: {text}")

# --- MAIN APP ---
app = ApplicationBuilder().token(BOT_TOKEN).build()

# Add handlers
app.add_handler(CommandHandler("start", start))
app.add_handler(CommandHandler("help", help_command))
app.add_handler(CommandHandler("about", about))
app.add_handler(MessageHandler(filters.TEXT & ~filters.COMMAND, echo_message

print("✅ Bot is running...")
app.run_polling()
```

⚙️ How This Works

- `ApplicationBuilder()` → creates the bot application.
- `.token()` → connects the bot using your API token.
- `CommandHandler()` → listens for commands like `/start` or `/help` .
- `MessageHandler()` → handles regular messages (anything not starting with `/`).

- `app.run_polling()` → continuously polls Telegram servers for updates.

🧩 Advanced Message Handlers

✅ Handle Stickers or Photos

```
app.add_handler(MessageHandler(filters.PHOTO, lambda u, c: u.message.reply_t
app.add_handler(MessageHandler(filters.STICKER, lambda u, c: u.message.reply
```

✅ Handle Specific Keywords

```
async def keyword_reply(update: Update, context: ContextTypes.DEFAULT_TYPE):
    text = update.message.text.lower()
    if "hello" in text:
        await update.message.reply_text("Hey there!")
    elif "bye" in text:
        await update.message.reply_text("Goodbye 👋")
    else:
        await update.message.reply_text("Hmm, I don’t understand that yet!")

app.add_handler(MessageHandler(filters.TEXT & ~filters.COMMAND, keyword_repl
```

✅ Handle Inline Buttons (Callback Queries)

```
from telegram import InlineKeyboardButton, InlineKeyboardMarkup
from telegram.ext import CallbackQueryHandler

async def menu(update: Update, context: ContextTypes.DEFAULT_TYPE):
    keyboard = [
        [InlineKeyboardButton("About", callback_data="about"),
         InlineKeyboardButton("Help", callback_data="help")]
    ]
    await update.message.reply_text("Choose an option:", reply_markup=Inline

async def handle_callback(update: Update, context: ContextTypes.DEFAULT_TYPE
    query = update.callback_query
    await query.answer()
    if query.data == "about":
        await query.edit_message_text("🤖 I am a Python Telegram bot demo.")
    elif query.data == "help":
        await query.edit_message_text("🧠 Use /start or /about for info.")

app.add_handler(CommandHandler("menu", menu))
app.add_handler(CallbackQueryHandler(handle_callback))
```

🧠 Error Handling for Handlers

You can define a global error handler:

```
from telegram.error import TelegramError

async def error_handler(update: Update, context: ContextTypes.DEFAULT_TYPE):
    print(f"⚠️ Update {update} caused error {context.error}")

app.add_error_handler(error_handler)
```

🧩 Testing Your Handlers

Run the bot:

```
python bot.py
```

Go to Telegram → find your bot → type `/start` → try `/help`, `/about`, or send messages. You'll see how different handlers respond.

🧠 Common Handler Use Cases

Scenario	Handler
`/start`, `/help`, `/about`	`CommandHandler`

Scenario	Handler
Text messages like “hi”, “bye”	`MessageHandler`
Button clicks	`CallbackQueryHandler`
Multi-step chat (e.g., quiz)	`ConversationHandler`
File uploads	`MessageHandler(filters.Document)`

⚙️ Deactivate the Virtual Environment (when done)

When you’re finished working:

```
deactivate
```

This returns your terminal to the global environment.

🧩 Practice Tasks (15 Exercises)

1. Create a bot that replies “Welcome!” when a user types `/start` .
2. Add `/help` and `/about` commands.
3. Echo any non-command text message.
4. Add a handler for photos — reply “Nice photo!”.
5. Respond “Hi there!” when user says “hello”.
6. Create a `/menu` with buttons “About” and “Help”.
7. Implement callback queries for menu buttons.
8. Add error logging for failed updates.
9. Store every user message in a local file (`messages.txt`).
10. Add `/time` command that shows the current time.
11. Add `/weather` that returns “Sunny 🌞 ” (mock).
12. Add `/sum` that takes two numbers (e.g. `/sum 3 7`) and returns 10.
13. Create a `/clear` command to reset a file log.
14. Add `/feedback` command to collect user feedback (save in file).
15. Deploy the bot using Render or PythonAnywhere and test live.