# Chapter 3 - MCP basics in python

**[Intro Transition]**

"Alright, now that we've already covered what MCP is and why it matters, let's actually build something with it.

In this section, we're going to create a simple MCP server and client in Python. Our little demo server will have two tools: a **Calculator** with basic add and subtract functions. This will give us a hands-on look at how the pieces come together."

---

## Step 1 – Setting Up the MCP stdio Server

"First, let's set up a server using the Python MCP SDK.

We're going to start with the simplest transport: **stdio**.

This means our server will communicate with the client using standard input and output – nice and simple for demos and testing."

There are several ways to set up an MCP stdio server, but we'll use the `FastMCP` class because it's the simplest for beginners.
Let's take a look at the code:

**Show code snippet:**

```
from mcp.server.fastmcp import FastMCP

mcp = FastMCP(
   name="Calculator",
)
```

"All you need to do is import `FastMCP` and create an instance with the name you want."

---

### Step 2 – Defining Tools (e.g., Calculator)

Next, let's add some tools. Tools are just functions we register with the server that clients can call.

Here’s a basic calculator example:

**Calculator Example:**

```
@mcp.tool("add")
def add(a: int, b: int) → int:
    """
    Adds two integers together.

    Args:
        a (int): The first number.
        b (int): The second number.

    Returns:
        int: The sum of a and b.
    """
    return a + b

@mcp.tool()
def subtract(a: int, b: int) → int:
    return a - b

@mcp.tool()
def divide(a: int, b: int) → float:
    if b == 0:
        raise ValueError("Division by zero is not allowed")
    return a / b

@mcp.tool("multiply")
def multiply(a: int, b: int) → int:
    return a * b

if __name__ == "__main__":
    mcp.run()
```

Pretty simple — we registered two tools, `add` and `subtract`, and each one just takes in numbers and returns the result.

Here are some important tips when creating tools:

1. **Specify input and output types.**

   Python doesn't require type hints, but since these tools are used by LLMs, specifying types helps the model understand what to send.

   For example, if you don't indicate that `add` expects integers, the LLM might provide a string instead. That could lead to unexpected results, which we want to avoid.

2. **Add a clear comment or docstring explaining the tool.**

   Just like when you ask ChatGPT something, it helps to describe in detail what the tool does and why it exists. These comments are fed to the LLM, so it can easily understand when and how to use the function.

   Here are some tips for comment.

   - Write Comprehensive Tool Descriptions
   - Provide Detailed Parameter Documentation
   - Include Usage Examples in Docstrings

3. **Handle Errors Gracefully**

   - Just like humans, an LLM can correct its own mistakes if it understands the error.
   - Clear error messages help the LLM identify and fix issues (e.g., database connection errors, division by zero, or API key error).

4. **Choose descriptive names.**

   Naming is important — a clear, meaningful name helps both you and the LLM understand the purpose of the tool.

"Now, this server setting is done for client  and ready to accept requests from an MCP client."

---

## Step 3 – Setting Up the MCP stdio Client

"Now that our server is running, let's see how we can actually call these tools from a Python client.

First, let's set up the MCP server for stdio MCP server.

```
from mcp import StdioServerParameters

server_params = StdioServerParameters(
    command="uv",
    args=["run", "servers/stdio_servers/calculator.py"],
)
```

Here, we're creating `StdioServerParameters` to tell Python how to start our server.

- `command` is the command you would normally run in the terminal to start the server.
- `args` are any additional arguments needed to run your server script.

Once we have the server parameters set up, we can connect to the server asynchronously using `stdio_client`. This is useful if you want non-blocking calls and better integration with other async code.

```
async def run():
    async with stdio_client(server_params) as (read, write):
        async with ClientSession(read, write) as session:
            # Initialise session
            await session.initialize()

if __name__ == "__main__":
    asyncio.run(run())
```

*Here's what's happening step by step:*

- `stdio_client(server_params)` connects to our running MCP server using stdio.
- `ClientSession(read, write)` creates a session for sending requests and receiving responses.
- `await session.initialize()` sets up the session so it's ready to call tools.

From here, you can start calling any of your registered tools through this session.

---

## Step 4 –Calling MCP server tools from the MCP client

At this point, even if it doesn't look like much, our async client is already connected to the MCP server.

Now, let's check which tools are available on the server and inspect their details

```
async def run():
    async with stdio_client(server_params) as (read, write):
        async with ClientSession(read, write) as session:
            # Initialise session
            await session.initialize()

                    tools = await session.list_tools()
            for tool in tools.tools:
                print("=" * 40, "\n")
                print(
                    f"Tool Name: {tool.name}\n"
                    f"Description: {tool.description}\n"
                    f"Input Schema: {tool.inputSchema}\n"
                    f"Title: {tool.title}\n"
                )
                print("=" * 40, "\n")
            # Example: call the 'add' tool
            result = await session.call_tool("add", {"a": 5, "b": 3})
            print("Add result:", result)
```

*Here's what's happening:*

- `await session.list_tools()` fetches all the tools registered on the server.
- We loop through each tool and print out its name, description, input schema, and title. This is really helpful to understand what each tool does.
- Finally, we call the `add` tool asynchronously using `session.call_tool()` and print the result.

With just a few lines of async code, we can fully inspect and interact with all the tools on our MCP server.

---

## Step 5 – Setting Up the MCP streamable http Server

By now, you probably have a sense of how to set up an MCP stdio server and communicate with it locally.

But you might be wondering — how can we set up an MCP HTTP server so we can call the tools not just locally, but from the internet?

Let's take a look at setting up an MCP **streamable HTTP server**.

First, it's very similar: we instantiate the `FastMCP` class and define our tools

```
from mcp.server.fastmcp import FastMCP

mcp = FastMCP("StatefulServer")

@mcp.tool()
def greet(name: str = "World") -> str:
    """Greet someone by name."""
    return f"Hello, {name}!"

# Run server with streamable_http transport
if __name__ == "__main__":
    mcp.run(transport="streamable-http")
```

The main difference is how we call the `run` method.

By default, `FastMCP.run()` uses the `stdio` transport, which is why we didn't need to specify it before.

But if we want to use **streamable HTTP**, we simply pass `transport="streamable-http"` when running the server. This allows clients to connect over HTTP and interact with our tools from anywhere.

---

## Step 6 – Setting Up the MCP http Client

Next, we can call tools on the MCP HTTP server from the MCP client. Since the transport has changed from stdio to HTTP, the client setup is slightly different.

```
import asyncio
from mcp import ClientSession
from mcp.client.streamable_http import streamablehttp_client

async def main():
    # Connect to a streamable HTTP server
    async with streamablehttp_client("http://localhost:8000/mcp") as (
```

```
        read_stream,
        write_stream,
        _,
    ):
        # Create a session using the client streams
        async with ClientSession(read_stream, write_stream) as session:
            # Initialize the connection
            await session.initialize()

if __name__ == "__main__":
    asyncio.run(main())
```

*The main differences compared to the stdio client are:*

1. We no longer need to instantiate `StdioServerParameters`.
2. We specify the **URL of the server** in `streamablehttp_client` instead of using a local command.
    - Since our tools are running on the local server, we need to specify the URL as `localhost`.
    - By default, the server runs on port 8000 and the endpoint is `/mcp`

Everything else works the same way as with the stdio client.

This makes it easy to call your MCP tools over the internet without changing how the tools themselves are defined.

---

## Recap

"So, that's it – you've just built your first MCP server in Python. We created a calculator and exposed it as tools, and showed how to call them from clients.

In the next section, we'll take this further and look at how to build richer functionality, but this is the foundation you'll keep coming back to."