

# Interactive Python Notebooks

## 1. What are interactive Python notebooks?

An interactive Python notebook is where you can write and run Python code, add explanations, and create visualizations—all in one place. Think of it as a digital lab notebook for coding!

Why Use a Notebook?

- **Interactivity:** Run code step-by-step and see results instantly.
- **Documentation:** Mix code with text to tell a story.
- **Visuals:** Add charts and plots effortlessly.
- **Sharing:** Share your work with friends, colleagues, or the world.

## 2. Understanding the notebook interface

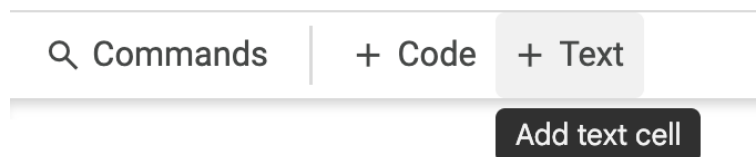
A Jupyter Notebook is made up of cells—little boxes where you write stuff. There are two main types: **markdown text cells** and **code cells**:

- **Markdown text cells:** These are for text. Use them to write explanations, headings, lists, or even add links and images. Think of them as your storytelling tool.
- **Code cells:** These are where the magic happens—write and run Python code here.

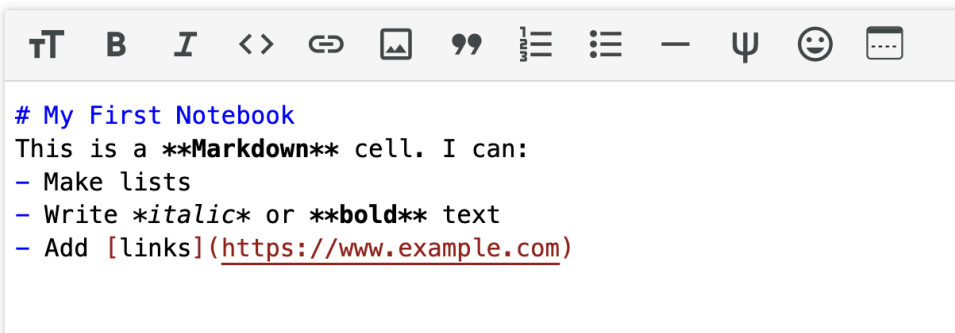
## 3. Your first Markdown cell

Let's try it out!

Click the "+" button in the toolbar to add a new text cell.



Type this:



```
# My First Notebook
This is a Markdown cell. I can:
- Make lists
- Write italic or bold text
- Add [links](https://www.example.com)
```

Depending on your setup, hit "Run" (the play button) or press **Shift + Enter**. Boom! The text formats beautifully in the cell.

## ✓ My First Notebook

This is a **Markdown** cell. I can:

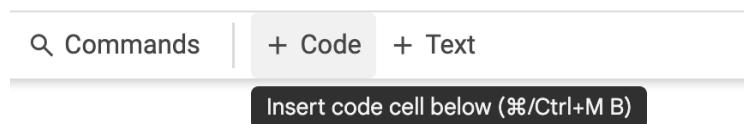
- Make lists
- Write *italic* or **bold** text
- Add [links](https://www.example.com)

Markdown keeps your notebook readable and organized.

## 4. Writing and running code

Now, let's run some Python!

Click the "+" button in the toolbar to add a new code cell.

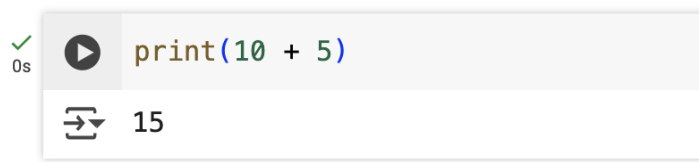


Type this:



```
print(10 + 5)
```

Depending on your setup, hit "Run" (the play button) or press **Shift + Enter**. That code ran!



Each code cell is like a mini-program. When you "run" it, Python executes the code inside and shows the output right below.

## 5. Running code cells

You can run cells **one at a time** or in any order. To run a cell:

- Select the cell you want to run.
- Press **Shift + Enter** or click the "Run" button on the cell.

To run **all** cells in **sequential** order, Use the "Run all" option in the menu.

Runtime	Tools	Help
Run all	⌘/Ctrl+F9	
Run before	⌘/Ctrl+F8	
Run the focused cell	⌘/Ctrl+Enter	
Run selection	⌘/Ctrl+Shift+Enter	
Run cell and below	⌘/Ctrl+F10	

## 6. Reordering cells

Sometimes, your notebook needs a little reorganization.

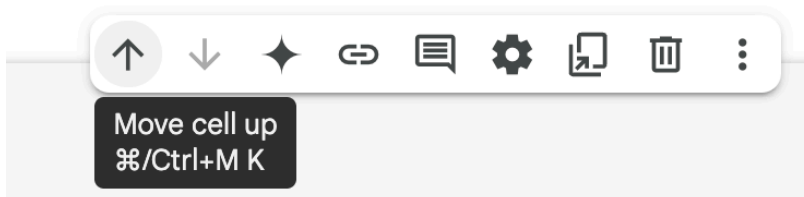
```
✓ [1] length = 10  
0s      width = 5  
        area = length * width  
        area
```

⇌ 50

```
✓ [2] my_string = "This is a string"  
0s      my_string
```

⇌ 'This is a string'

To reorder a cell, click on that cell, then click on the up or down arrow. Here we clicked on the **second** cell, then clicked on the "Move cell up" arrow.



Now the second cell has been moved to the first position.

```
✓ [2] my_string = "This is a string"  
0s      my_string
```

⇌ 'This is a string'

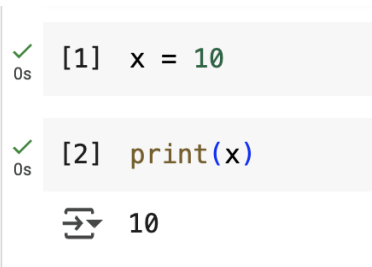
```
✓ [1] length = 10  
0s      width = 5  
        area = length * width  
        area
```

⇌ 50

## 7. Variables in notebooks

**Variables** are a big deal in notebooks because they "stick around" after you define them. Let's experiment.

In two separate code cells, type the following then run both cells:



```
[1] x = 10
```

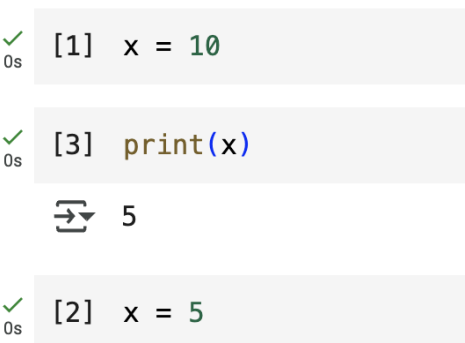
```
[2] print(x)
```

10

Nothing prints under the first cell because it only assigns the value of **10** to **x**.

The second cell outputs **10** because it retrieves and prints the value of **x**. This demonstrates that the **variable x** is stored in your notebook's session and **remains accessible across cells**.

In addition, the **order that the cells are run** matters and may not match their physical order on the page. Each cell has an **execution counter** in brackets next to it, indicating the sequence in which it was run, not its position in the notebook. Consider this example:



```
[1] x = 10
```

```
[3] print(x)
```

5

```
[2] x = 5
```

In this case, the output of **print(x)** is **5**, not 10, because the most recent assignment to **x** (from Cell 3) overwrites the earlier value. This highlights how the **order that the cells are run**, tracked by the counters, determines the final value of persistent variables like **x**.

## 8. Rerunning cells

The first cell below **loads a csv** with two columns: **'sport'** and **'number\_of\_players'**. The second cell **filters** the dataframe to show only rows where the **'number\_of\_players'** equals 15. The third cell **renames** the column **'number\_of\_players'** to **'num\_players'**.

What would happen if we **rerun** the second cell?

✓  
0s

[1] `import pandas as pd`  
`df = pd.read_csv('sports-num-players.csv')`  
`df.head(3)`



	sport	number_of_players
0	baseball	9
1	basketball	5
2	cricket	11



Next steps: [Generate code with df](#) [View recommended plots](#) [New interactive sheet](#)

✓  
0s

[2] `df.query('number_of_players == 15')`



	sport	number_of_players
6	rugby	15



✓  
0s

[3] `df = df.rename(columns={'number_of_players': 'num_players'})`  
`df.head(3)`



	sport	num_players
0	baseball	9
1	basketball	5
2	cricket	11



We now encounter an **error**!

✓  
0s

[1] `import pandas as pd`  
`df = pd.read_csv('sports-num-players.csv')`  
`df.head(3)`

↗

	sport	number_of_players
0	baseball	9
1	basketball	5
2	cricket	11

⌵

Next steps: [Generate code with df](#) [View recommended plots](#) [New interactive sheet](#)

⚠  
0s

[4] `df.query('number_of_players == 15')`

↗

[Show hidden output](#)

Next steps: [Explain error](#)

✓  
0s

[3] `df = df.rename(columns={'number_of_players': 'num_players'})`  
`df.head(3)`

↗

	sport	num_players
0	baseball	9
1	basketball	5
2	cricket	11

⌵

The error message will include **KeyError: 'number\_of\_players'**.

Why does this happen? After renaming the column in Cell 3, **'number\_of\_players'** no longer exists in **df**. It is now **'num\_players'**.

**Key Takeaway:** variables including dataframes persist across cells in a Python notebook session. Their state reflects the **most recent executed changes**.

## 9. Resetting runtime

Occasionally, you may need to **reset your runtime** in a Python notebook. Resetting **clears** all variables stored in the current session, giving you a fresh start. Let's walk through an example.

In two separate code cells, enter and run the following code:

```
✓ 0s [1] length = 10  
      width = 5  
  
✓ 0s [2] area = length * width  
      area  
  
⇒ 50
```

Now, go back to the first cell and modify it by replacing **'length'** with **'l'** and **'width'** with **'w'**.

Run both cells again.

```
✓ 0s [3] l = 10  
      w = 5  
  
✓ 0s [4] area = length * width  
      area  
  
⇒ 50
```

The code above has an error since **'length'** and **'width'** are not defined. Surprisingly, the output of the second cell remains 50. Why does this happen?

The notebook session still remembers the original variables **'length'** and **'width'** from the first run, and the second cell continues to use them to compute the area.



To catch errors like this, you sometimes need to reset the runtime.

Runtime	Tools	Help
Run all	⌘/Ctrl+F9	
Run before	⌘/Ctrl+F8	
Run the focused cell	⌘/Ctrl+Enter	
Run selection	⌘/Ctrl+Shift+Enter	
Run cell and below	⌘/Ctrl+F10	
<hr/>		
Interrupt execution	⌘/Ctrl+M I	
Restart session	⌘/Ctrl+M .	
Restart session and run all		

Now there are no variables stored by the notebook session. Also note that **execution counter** brackets are empty.

```
[ ] l = 10
    w = 5
```

```
[ ] area = length * width
    area
```

Now if we **rerun** the cells, we see the error in our code.

✓  
0s

[1] l = 10  
w = 5

⌚  
0s

[2] area = length \* width  
area

↩

-----

**NameError** Traceback (most recent call last)

<ipython-input-2-c1a5c3399503> in <cell line: 0>()  
----> 1 area = length \* width  
 2 area

**NameError:** name 'length' is not defined

Next steps:


[Explain error](#)

## 10. Opening a notebook



When you open a Python notebook, it may appear as though the code has already been executed. For instance, you might see outputs like `df.head(3)` displayed below a cell. However, the **execution counter** brackets—typically showing a number like `[1]`—are **empty**, indicating that the **session's variables have been cleared**.

If you attempt to run a later cell, such as the second one, without first running the initial cell, you'll encounter an error like `NameError: name 'df' is not defined`. This happens because the notebook hasn't yet defined the variables in the current session.

```
[ ] import pandas as pd
    df = pd.read_csv('sports-num-players.csv')
    df.head(3)
```




	sport	number_of_players
0	baseball	9
1	basketball	5
2	cricket	11



Next steps: [Generate code with df](#) [View recommended plots](#) [New interactive sheet](#)

0s

```
df.query('number_of_players == 15')
```



```
-----
NameError                                Traceback (most recent call last)
<ipython-input-1-39334de378b9> in <cell line: 0>()
----> 1 df.query('number_of_players == 15')
```

**NameError:** name 'df' is not defined

Next steps: [Explain error](#)

To resolve this, simply rerun the cells in sequential order.