Python Minimal Console App

Contents

Minimal Console App Start

Practical Lab — venv + dictionaries + modular functions

Goal

Build a **minimal** console app:

- **Menu A:** Simple Calculator (2 numbers, choose op)
- **Menu B:** File Stats (open raw text file, choose **sum** or **median**)
- Uses:
    - **venv** for isolation
    - **dictionary dispatch** (menu maps choices → functions)
    - **modular functions** (separate modules; main() controls flow)
    - **file I/O with with open(...)**

Lab Setup (venv) Project Virtual Environment

1. From your project folder create the Virtual Environment by running the commands for your OS. See Lesson 6 ex 20 p 143 for details.

Windows (PowerShell)

PowerShell

python -m venv .venv

.\.venv\Scripts\Activate.ps1

python -V

macOS / Linux

PowerShell

python3 -m venv .venv

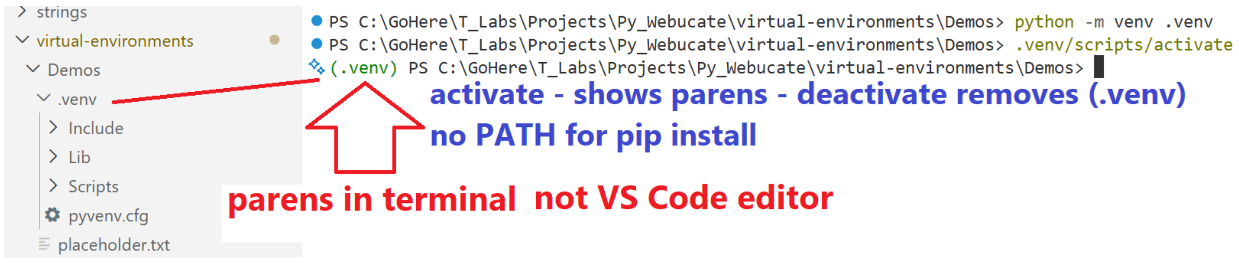
source .venv/bin/activate

python -V

- No external packages needed for this lab (standard library only), so **no pip installs required**.
- Deactivate when done – the parenthesis will be gone – no PATH for pip install unless reactivated.

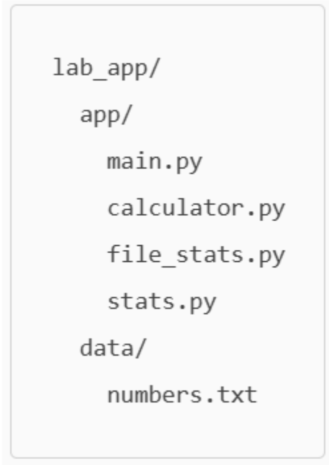
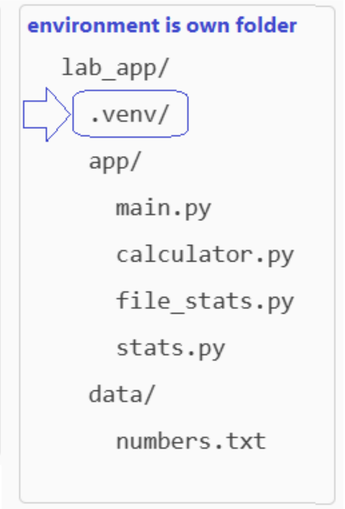
Shell

Deactivate



```
PS C:\GoHere\T_Labs\Projects\Py_Webucate\virtual-environments\Demos> python -m venv .venv
PS C:\GoHere\T_Labs\Projects\Py_Webucate\virtual-environments\Demos> .venv/scripts/activate
(.venv) PS C:\GoHere\T_Labs\Projects\Py_Webucate\virtual-environments\Demos>
```

Create Project Folders (outside of .venv)

lab_app/ (your project name)
 	
app/
 	
main.py
 	
calculator.py
 	
file_stats.py
 	
stats.py
 	
data/
 	
numbers.txt
	
	```
	lab_app/
	  app/
	    main.py
	    calculator.py
	    file_stats.py
	    stats.py
	  data/
	    numbers.txt
	```
	
	environment is own folder
	```
	lab_app/
	  .venv/
	  app/
	    main.py
	    calculator.py
	    file_stats.py
	    stats.py
	  data/
	    numbers.txt
	```

Create Files

Data File (sample)

Create data/numbers.txt:

```
10

20

30

40

50
```

(One number per line.)

Code — app/stats.py

Shared “math/stats” functions (sum + median). Median implemented manually (no dependencies).

Python

```
def sum_all(nums):
return sum(nums)



def median(nums):
if not nums:
raise ValueError("No numbers provided.")

s = sorted(nums)
n = len(s)
mid = n // 2

if n % 2 == 1:
return s[mid]

return (s[mid - 1] + s[mid]) / 2
```

Show more lines

Code — app/calculator.py

Simple 2-number calculator using dictionary dispatch.

Python

```
def add(a, b):
return a + b


def sub(a, b):
return a - b


def mul(a, b):
return a * b


def div(a, b):
return a / b


OPS = {
"1": ("add", add),
"2": ("sub", sub),
"3": ("mul", mul),
"4": ("div", div),
}
```

```
def run():

print("Calculator")

print("1) add 2) sub 3) mul 4) div")


op_choice = input("> ").strip()

op = OPS.get(op_choice)

if not op:

print("Invalid choice.")

return


try:

a = float(input("a: ").strip())

b = float(input("b: ").strip())

name, fn = op

result = fn(a, b)

print(f"{name} result: {result}")

except ZeroDivisionError:

print("Cannot divide by zero.")

except ValueError:

print("Invalid number.")
``
```

Show more lines

This follows the course style of **functions + main flow** and using dict.get() safely.

Code — app/file_stats.py

Reads a raw text file, parses numbers, runs sum/median via dictionary dispatch.

Python

```
from .stats import sum_all, median
```

```
ACTIONS = {
"1": ("sum", sum_all),
"2": ("median", median),
}



def _read_numbers(path):
nums = []
with open(path, "r", encoding="utf-8") as f:
for line in f:
line = line.strip()
if not line:
continue
nums.append(float(line))
return nums



def run():
print("File Stats")
print("Enter path to a text file with one number per line.")

path = input("> ").strip()

print("1) sum 2) median")
action_choice = input("> ").strip()

action = ACTIONS.get(action_choice)
if not action:
```

```
print("Invalid choice.")
return


try:
nums = _read_numbers(path)
name, fn = action
result = fn(nums)
print(f"{name} result: {result}")
except FileNotFoundError:
print("File not found.")
except ValueError as e:
print(f"Bad data: {e}")
```

Show more lines

This directly uses the PDF's file approach (with open(...)) and reinforces why relative paths can fail depending on run location (Lesson 1 note).

Code — app/main.py

Main menu: calculator OR file ops, using dictionary dispatch.

Python

```
from . import calculator
from . import file_stats



MODES = {
"1": ("calculator", calculator.run),
"2": ("file stats", file_stats.run),
}
```

```
def main():

print("1) calculator")

print("2) file stats")

choice = input("> ").strip()


mode = MODES.get(choice)

if not mode:

print("Invalid choice.")

return


_, fn = mode

fn()



if __name__ == "__main__":

main()
```

Show more lines

This matches the course pattern: **define functions, then call main()** for flow control.

Run It

From lab_app/:

Shell

```
python -m app.main
```

Try It:

- Calculator mode → 1
- File stats mode → 2 and then path:
  - data/numbers.txt (relative)
  - or full absolute path

If you see path issues (FileNotFoundError), it’s the exact behavior discussed in Lesson 1 about where VS Code runs scripts from and how relative paths can break.

Update Main with break on q loop

Add a while True: in main() and **break on q**

✅ **Update: app/main.py (loop + quit on q)**

Replace your main() with this version:

Python

```
from . import calculator

from . import file_stats



MODES = {

"1": ("calculator", calculator.run),

"2": ("file stats", file_stats.run),

}



def main():

while True:

print("1) calculator")

print("2) file stats")

print("q) quit")

choice = input("> ").strip().lower()


if choice == "q":

break


mode = MODES.get(choice)
```

```
if not mode:

print("Invalid choice.")

continue


_, fn = mode

fn()

print() # blank line between runs



if __name__ == "__main__":

main()
```

``

Show more lines

What this does (no fluff):

- Loops forever until the user enters **q**
- Uses the MODES dictionary to dispatch the correct function
- Uses continue to re-prompt on invalid input
- Keeps output readable with a single blank line after each run