

Intro to Coding with Python—main()

Dr. Ab Mosca (they/them)

Slides based off slides courtesy of Jordan Crouser (<https://jcrouser.github.io/>)

Plan for Today

- The `main()` function

A reminder from the syllabus

- **“References”**
 - You should use resources when you need help!
 - And you must cite them! (Give them credit for helping you)
 - In-line citations to any resources you used, including page numbers (if a printed resource) or a direct URL (if an online resource).
- Ex.

Example

```
*documentations.py - /Users/jcrouser/Google Drive/Teaching/Course Material/CSC111/CSC111/demos/documentations...

#-----
#           Names: Jordan Crouser & Morganne Crouser
#           Date: 26 September 2018
#       Filename: demo.py
# Description: This is a demonstration of how to
#               properly attribute help on a
#               CSC111 assignment
#-----

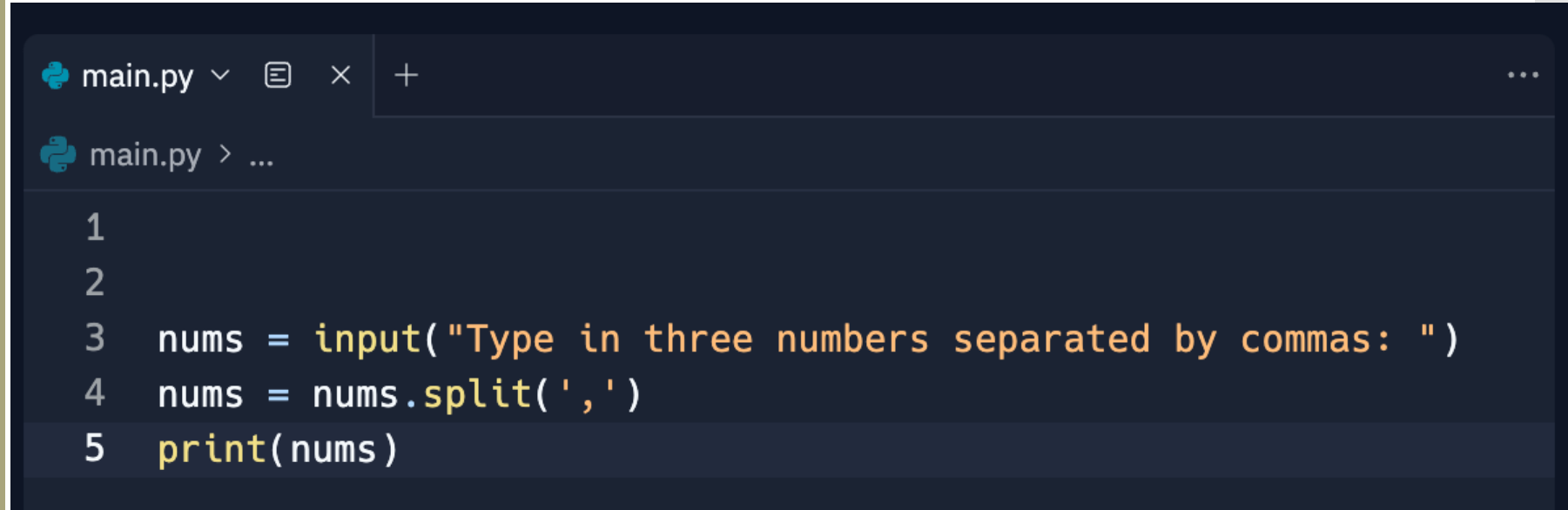
name = input("Enter your name: ")
formatted_string = "{0:>10}".format(name)
print(formatted_string)

# REFERENCES
# I googled how to use the str.format(...) method
# and found the Python documentation here:
# https://docs.python.org/3/library/stdtypes.html#str.format
```

Ln: 17 Col: 60

Recap

- So far, we've been writing code in files as if we were writing it on the console:

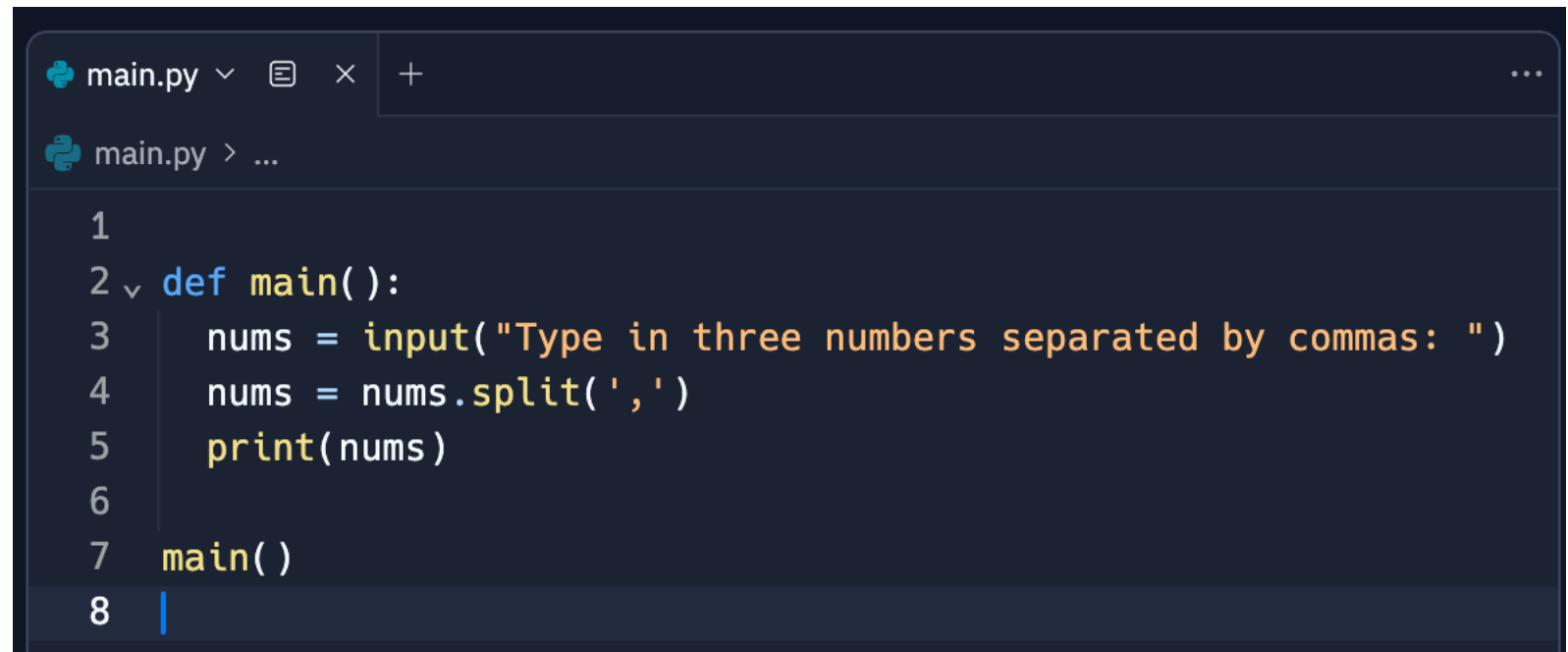
A screenshot of a code editor window. The title bar shows 'main.py' with a dropdown arrow, a list icon, a close icon, and a plus icon. The editor content shows a Python script with five lines of code, numbered 1 to 5 on the left. The code is: 1 (blank), 2 (blank), 3 'nums = input("Type in three numbers separated by commas: ")', 4 'nums = nums.split(',')', and 5 'print(nums)'. The file path 'main.py > ...' is visible in the breadcrumb area.

```
1
2
3  nums = input("Type in three numbers separated by commas: ")
4  nums = nums.split(',')
5  print(nums)
```

- When we do this, the Python interpreter executes everything from the **top down**

An alternative

- It is better practice to write the code you want to execute inside a `main()` function, e.g.

A screenshot of a Python IDE window titled 'main.py'. The code inside the editor is as follows:

```
1
2 def main():
3     nums = input("Type in three numbers separated by commas: ")
4     nums = nums.split(',')
5     print(nums)
6
7 main()
8
```

- This lets the interpreter “read ahead” and **then** execute

How this works

- **Remember:** the interpreter reads from the top down, which means that it reads the **definition** first

```
main.py  v  [icon]  x  +  ...
main.py  >  ...

1
2  v  def main():
3      nums = input("Type in three numbers separated by commas: ")
4      nums = nums.split(',')
5      print(nums)
6
7  main()
8  |
```

How this works

- Then it reads each line inside the definition, but these don't get **executed** yet

```
main.py  ▾  [icon]  ×  +  ...
main.py  >  ...

1
2  def main():
3      nums = input("Type in three numbers separated by commas: ")
4      nums = nums.split(',')
5      print(nums)
6
7  main()
8  |
```


How this works

- Then it reads each line inside the definition, but these don't get **executed** yet

```
main.py  ▾  [icon]  ×  +  ...
main.py  >  ...

1
2 ▾ def main():
3     nums = input("Type in three numbers separated by commas: ")
4     nums = nums.split(',')
5     print(nums)
6
7     main()
8     |
```

How this works

- Then it reads each line inside the definition, but these don't get **executed** yet

```
main.py  v  [icon]  x  +  ...
main.py  >  ...

1
2  v  def main():
3      nums = input("Type in three numbers separated by commas: ")
4      nums = nums.split(',')
5      print(nums)
6
7  main()
8  |
```

How this works

- At this stage, we've given python a "recipe" for what we want it to do when we call `main()`

```
main.py  v  [icon]  x  +  ...
main.py  >  ...

1
2  def main():
3      nums = input("Type in three numbers separated by commas: ")
4      nums = nums.split(',')
5      print(nums)
6
7  main()
8  |
```

- If we stop here, nothing will actually happen

How this works

- The real work happens only when we actually **call** the **main()** function

```
main.py  v  [icon]  x  +  ...
main.py  >  ...

1
2  def main():
3      nums = input("Type in three numbers separated by commas: ")
4      nums = nums.split(',')
5      print(nums)
6
7  main()
8  |
```

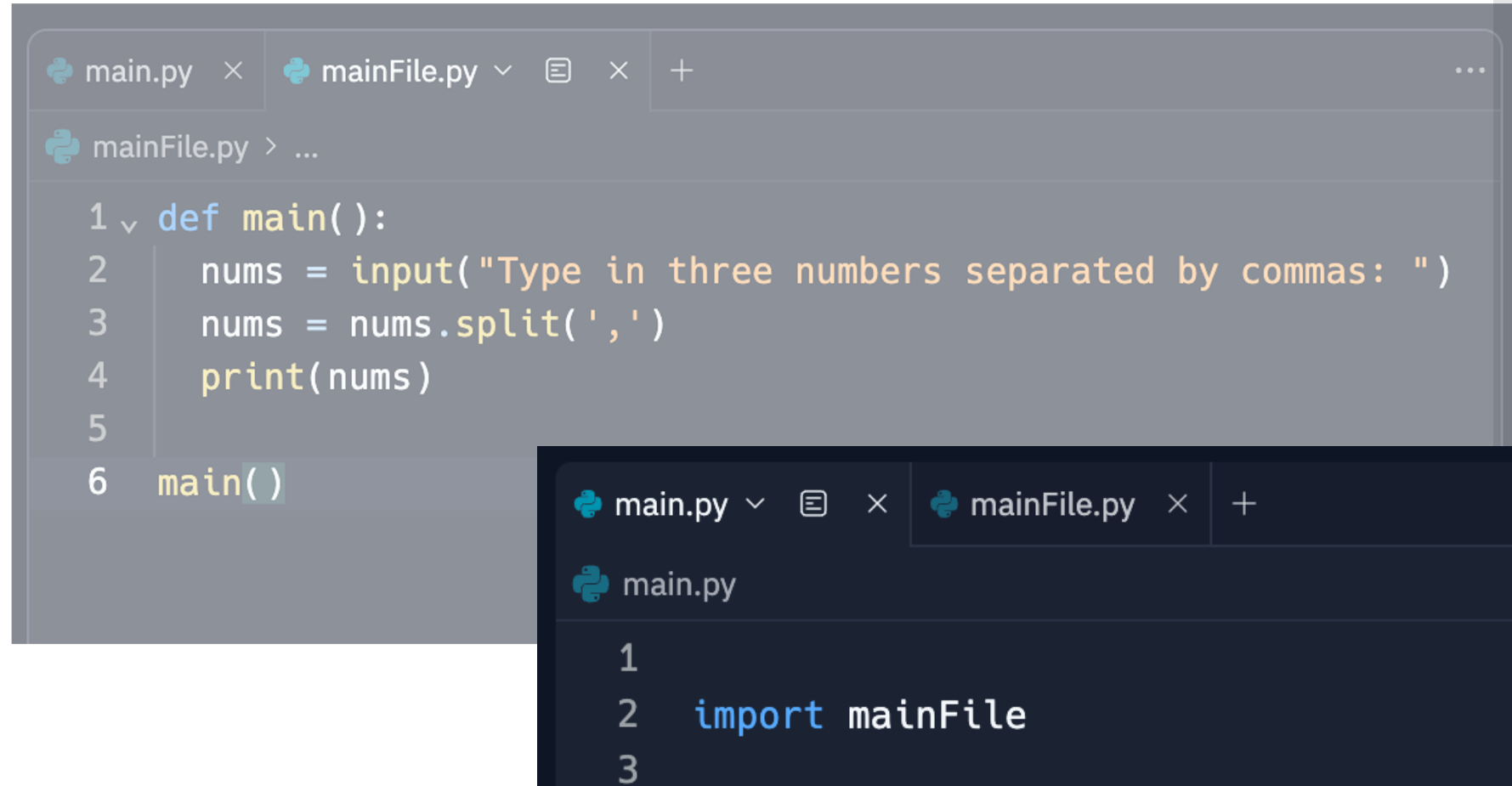
- When we do, python goes to the **main()** function definition and follows the instructions it finds there

Discussion

Why bother?

Just one more
thing...

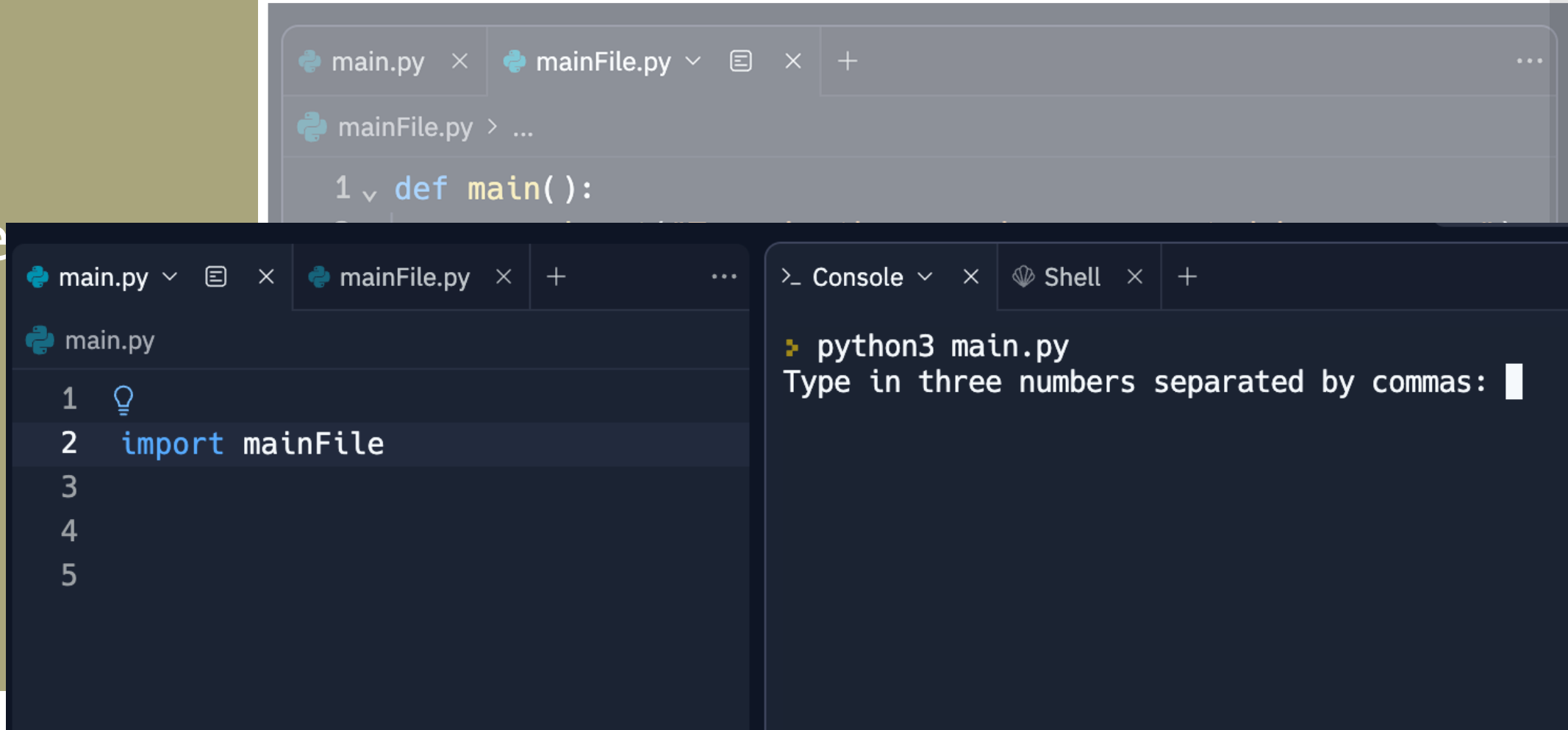
- What happens if someday we want to use the code in this file as **part of another program**?



```
main.py × mainFile.py ▾ ☰ × +  
mainFile.py > ...  
1 ▾ def main():  
2     nums = input("Type in three numbers separated by commas: ")  
3     nums = nums.split(',')  
4     print(nums)  
5  
6 main()  
  
main.py ▾ ☰ × mainFile.py × +  
main.py  
1  
2 import mainFile  
3
```

Just one
thing...

- What happens if someday we want to use the code in this file as **part of another program**?



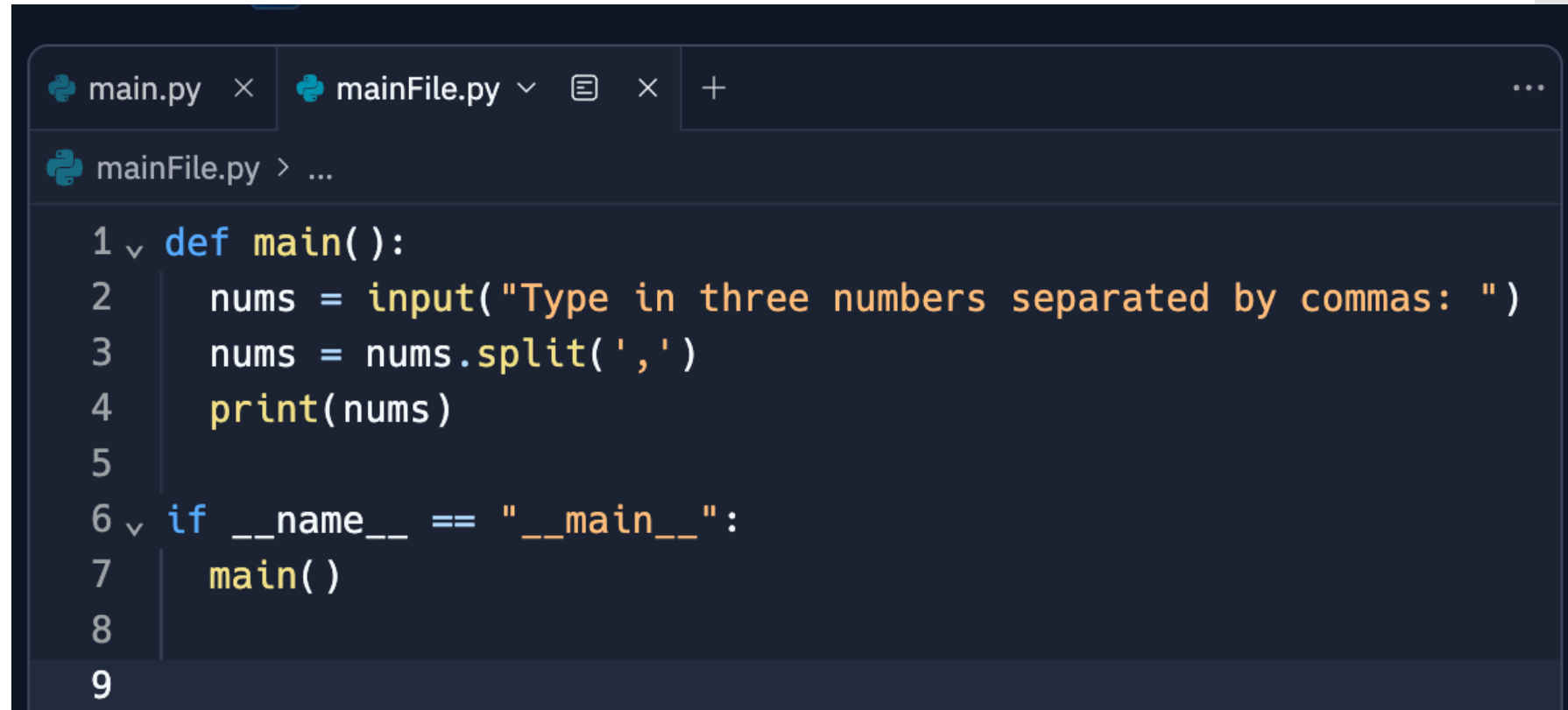
Discussion

- **What we need:** a way to tell python to behave one way when we **run it as a “stand-alone” program**, and a different way when we **import** it

Ideas?

Python convention

- We can use an **if** statement to tell python to call the **main()** function only if the program is being run directly



```
main.py × mainFile.py ▾ ☰ × +  
mainFile.py > ...  
1 ▾ def main():  
2     nums = input("Type in three numbers separated by commas: ")  
3     nums = nums.split(',')  
4     print(nums)  
5  
6 ▾ if __name__ == "__main__":  
7     main()  
8  
9
```

Python convention

- This is a little bit **confusing**: we named the function we created to hold our program was called **main()**

```
main.py × mainFile.py ▾ ... +
mainFile.py > ...
1 ▾ def main():
2     nums = input("Type in three numbers separated by commas: ")
3     nums = nums.split(',')
4     print(nums)
5
6 ▾ if __name__ == "__main__":
7     main()
8
9
```

Python convention

- In our **if** statement, we're asking whether some variable called **__name__** is equal to the string **"__main__"**

```
main.py ×  mainFile.py ▾  ...  
mainFile.py > ...  
1  def main():  
2      nums = input("Type in three numbers separated by commas: ")  
3      nums = nums.split(',')  
4      print(nums)  
5  
6  if __name__ == "__main__":  
7      main()  
8  
9
```

- (not to mention I don't recall initializing anything called **__name__**...)

To the
documentation!

The screenshot shows a web browser window displaying the Python documentation for the `__main__` module. The browser's address bar shows the URL `https://docs.python.org/3/library/__main__`. The page header includes navigation links for Python version (3.7.0), language (English), and documentation sections. The main content area features the title `__main__` — Top-level script environment. Below the title, a paragraph explains that `'__main__'` is the name of the scope in which top-level code executes. A code block illustrates a common idiom for conditionally executing code when a module is run as a script. The footer contains copyright information, a search bar, and links to report bugs or donate.

Python Software Foundation [US] | https://docs.python.org/3/library/__main__

For quick access, place your bookmarks here on the bookmarks bar. [Import bookmarks now...](#)

Python » English » 3.7.0 » Documentation » The Python Standard Library » Python Runtime Services » [previous](#) | [next](#) | [modules](#) | [index](#)

Quick search

`__main__` — Top-level script environment

`'__main__'` is the name of the scope in which top-level code executes. A module's `__name__` is set equal to `'__main__'` when read from standard input, a script, or from an interactive prompt.

A module can discover whether or not it is running in the main scope by checking its own `__name__`, which allows a common idiom for conditionally executing code in a module when it is run as a script or with `python -m` but not when it is imported:

```
if __name__ == "__main__":  
    # execute only if run as a script  
    main()
```

For a package, the same effect can be achieved by including a `__main__.py` module, the contents of which will be executed when the module is run with `-m`.

Python » English » 3.7.0 » Documentation » The Python Standard Library » Python Runtime Services » [previous](#) | [next](#) | [modules](#) | [index](#)

© Copyright 2001–2018, Python Software Foundation.

The Python Software Foundation is a non-profit corporation. [Please donate.](#)

Last updated on Sep 26, 2018. [Found a bug?](#)
Created using [Sphinx](#) 1.7.6.

DEMO
TIME

15-minute exercise

- Write a program that contains a **main()** function, which contains instructions for printing out the phrase:

`"Today is not Friday :-(`

- Use an **if** statement combined with checking the value of the `__name__` variable to call **main()** only when the program is run directly
- Add an **else** statement so that whenever the program ("module") is **imported**, it prints out the phrase:

`"Maybe today...?"`

Discussion

What did you come up with?

Takeaways

- Programs (“modules”) that are well-organized are **easier to read**, more **versatile**, and potentially **more efficient**
- The first step we’ll take toward organizing our code is to include a **main()** function, which includes the instructions we want our program to run
- To make it easier to **import** code we write now into later modules, we will follow the convention of including:

```
if __name__ == "__main__":  
    main()
```

at the end of each module

Helpful tip:
have a starter
template

```
starter.py - /Users/jcrouser/Google Drive/Teaching/Course Material/CSC111/CSC111/demos/start...
#-----
#           Names: Jordan Crouser & <PARTNER>
#           Date: <DATE>
#           Filename: starter.py
# Description: This is a demonstration of how to
#               organize your starter code (incl.
#               a main function and scope check)
#-----

def main():
    # This is where my code will go

if __name__ == "__main__":
    main()

# REFERENCES
```

Ln: 3 Col: 21