# BIOS-584 Python Programming (Non-Bios Student)

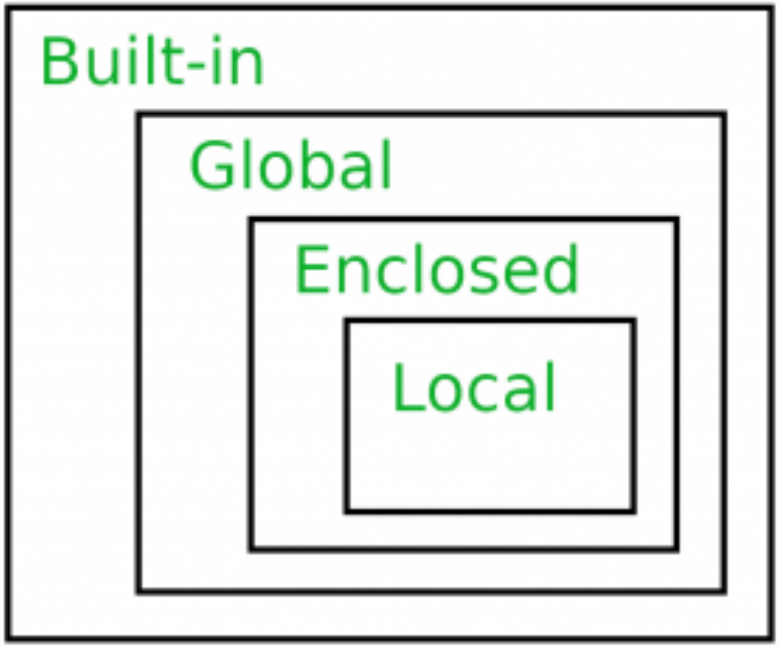
Week 08

Instructor: Tianwen Ma, Ph.D.

Department of Biostatistics and Bioinformatics,
Rollins School of Public Health,
Emory University

# Lecture Overview

- Stack array continued
- Variable scope in Python
- week-08-global-local.ipynb



# Join arrays

- You can “merge” or join arrays.
- This is super helpful in practice
- A couple of functions listed in the following slide

# Join a seq of arrays along an existing axis


- np.concatenate([a1, a2, ...], axis=0, ...)
- np.concat([a1, a2, ...], axis=0, ...)
- The input arrays must have the same shape, except in the dimension corresponding to axis.
  - Preferably saved them as a list.

# Join a seq of arrays along a new axis

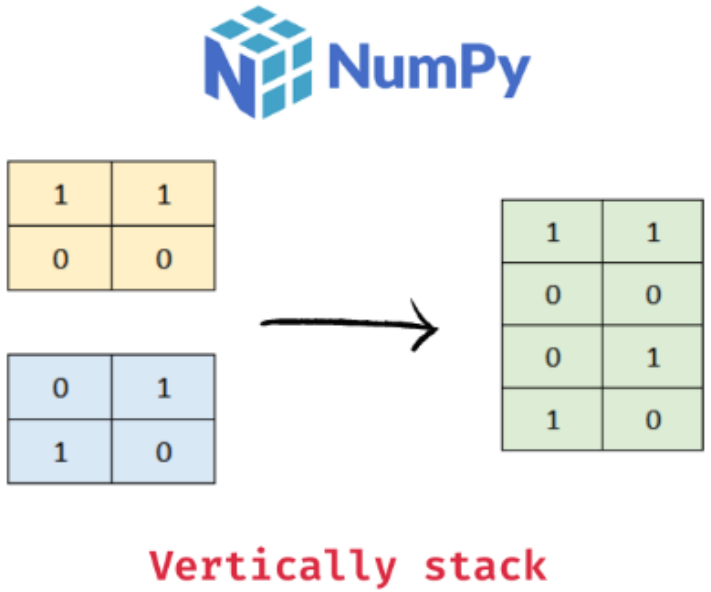

- np.stack([a1, a2, ...], axis=0, ...)
- Each array input must have the same shape
  - Preferably saved them as a list.
- The axis parameter specifies an index of the new axis in the dimensions of the resulting array.
- axis=0 is the first dimension of the new array
- axis=-1 is the last dimension of the new array

# Join arrays


- Stack arrays in sequence vertically
  - np.vstack([a1, a2, ...])
- Stack arrays in sequence horizontally
  - np.hstack([a1, a2, ...])
- The above two functions make most sense for arrays with up to 3 dimensions.
  - No need to specify the axis

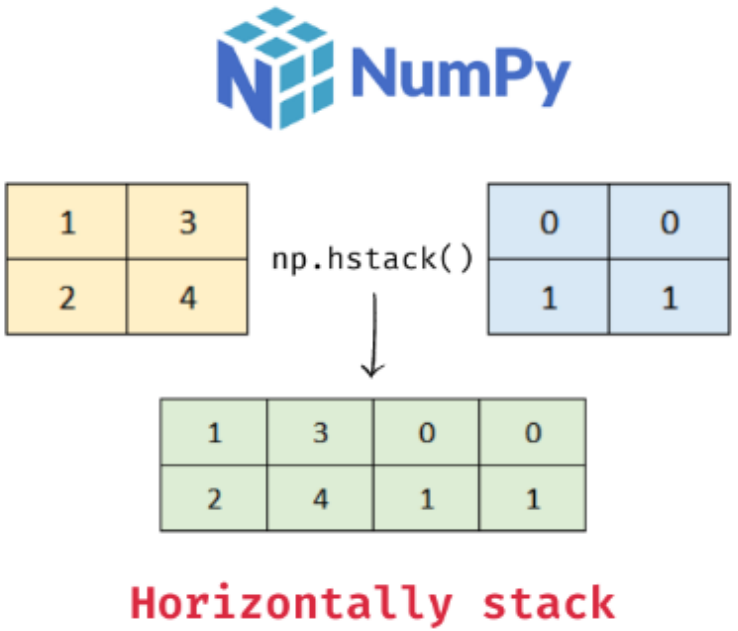
# Vertical stacking (Within 3-Dim)

- np.vstack([a1, a2, ...])
- Equivalently, suppose a1, a2 are 2-dim arrays:
- np.stack([a1, a2, ...], axis=0)



# Horizontal stacking (Within 3-Dim)

- np.hstack([a1, a2, ...])
- Equivalently, suppose a1, a2 are 2-dim arrays:
- np.stack([a1, a2, ...], axis=1)



# Intuitive Thought on Axis

- You can think of axis as performing operations along that dimension.
- The output shape for that particular dimension is collapsed due to the operation and thus disappeared.

# A Practical Example


- Let's call back from the blood pressure problem.
- Suppose we create a 3-dim array with a shape of (2, 10, 5).
- It refers to two sites, 10 patients each site, and 5 time points per patient

| Site ID | Patient ID | Time 1 | Time 2 | Time 3 | Time 4 | Time 5 |
|---|---|---|---|---|---|---|
| 1 | 1 | | | | | |
| … | … | | | | | |
| 1 | 10 | | | | | |
| 2 | 1 | | | | | |
| … | … | | | | | |
| 2 | 10 | | | | | |

# What are shapes when we compute means with the following axes?


- Axis=0
- Axis=1
- Axis=2
- Axis=-1
- What does Axis=-2 have?
- What about Axis=-3?
- Some may not have practical interpretations.

| Site ID | Patient ID | Time 1 | Time 2 | Time 3 | Time 4 | Time 5 |
|---|---|---|---|---|---|---|
| 1 | 1 | | | | | |
| … | … | | | | | |
| 1 | 10 | | | | | |
| 2 | 1 | | | | | |
| … | … | | | | | |
| 2 | 10 | | | | | |

# What are shapes when we compute means with the following axes?


- Axis=(1, 2)
- Axis=(0, 2)
- Axis=(0, 1)
- What does Axis=(-2, -1) have?

| Site ID | Patient ID | Time 1 | Time 2 | Time 3 | Time 4 | Time 5 |
|---|---|---|---|---|---|---|
| 1 | 1 | | | | | |
| … | … | | | | | |
| 1 | 10 | | | | | |
| 2 | 1 | | | | | |
| … | … | | | | | |
| 2 | 10 | | | | | |

# What are shapes when we compute means with the following axes?


- Axis=(0, 1, 2)
- Axis=None

| Site ID | Patient ID | Time 1 | Time 2 | Time 3 | Time 4 | Time 5 |
|---|---|---|---|---|---|---|
| 1 | 1 | | | | | |
| … | … | | | | | |
| 1 | 10 | | | | | |
| 2 | 1 | | | | | |
| … | … | | | | | |
| 2 | 10 | | | | | |

# Question?

- This principle applies to all NumPy functions with axis parameter.

# What is variable scope?

- Scope: area of program where a variable is accessible
- A variable's visibility in different parts of your code
- Apply the LEGB rule to determine variable scope

A practical example

```
x = 10  # Global scope

def print_x():
    x = 20  # Local scope
    print(x)  # Prints 20 (local)

print_x()

print(x)  # Prints 10 (global)
```

# What is variable scope?

- Local: Inside the current function
- Enclosing: Inside enclosing/nested functions
- Global: at the top level of the module
- Built-in: In the built-in namespace

A practical example

```
x = 10  # Global scope

def print_x():
    x = 20  # Local scope
    print(x)  # Prints 20 (local)

print_x()

print(x)  # Prints 10 (global)
```

# Global scope (Variables defined outside a function)

- Most variables we see are in the global scope
  - x=10
- They are stored in the global namespace and are accessible from anywhere in the code
- Global variables are created when you assign them with values and are destroyed when you close Python.

```
message_hello = "hello"
number3       = 3
```

```
print(message_hello + " world")
print(number3 * 2)
```

```
hello world
6
```

# Global scope (Variables defined outside a function)

- Global variables can be used in your code, but you should be careful when writing functions.
- Functions can change the value of global variables, leading to unexpected results
- It is recommended to include all variables that a function needs as parameters

```
message_hello = "hello"
number3       = 3
```

```
print(message_hello + " world")
print(number3 * 2)
```

```
hello world
6
```

# Global scope (recommended and non-recommended practices)

- Purpose: write a function to sum up three numbers $f(x, y, z) = x + y + z$.
- Pass **all numbers** as arguments to the function

```
# Correct Example:
def fn_add_recommended(x,y,z):
    return(x + y + z)

print(fn_add_recommended(x = 1, y = 2, z = 5))
print(fn_add_recommended(x = 1, y = 2, z = 10))
```

```
8
13
```

# Global scope (recommended and non-recommended practices)

- Purpose: write a function to sum up three numbers $f(x, y, z) = x + y + z$.
- Pass **all numbers** as arguments to the function

```
# Example that runs (but not recommended)
# Python will try to fill in any missing inputs
# with variables in the working environment
def fn_add_notrecommended(x,y):
    return(x + y + z)

z = 5
print(fn_add_notrecommended(x = 1, y = 2))
z = 10
print(fn_add_notrecommended(x = 1, y = 2))
```

```
8
13
```

# Global scope (recommended and non-recommended practices)

- Purpose: write a function to sum up three numbers $f(x, y, z) = x + y + z$.
- Pass **all numbers** as arguments to the function

```
# Example that runs (but not recommended)
# Python will try to fill in any missing inputs
# with variables in the working environment
def fn_add_notrecommended(x,y):
    return(x + y + z)

z = 5
print(fn_add_notrecommended(x = 1, y = 2))
z = 10
print(fn_add_notrecommended(x = 1, y = 2))
```

```
8
13
```

del z: remove variables from a global scope

```
del z
print(fn_add_notrecommended(x = 1, y = 2))
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
Cell In[2], line 2
      1 del z
----> 2 print(fn_add_notrecommended(x = 1, y = 2))

Cell In[1], line 5, in fn_add_notrecommended(x, y)
      4 def fn_add_notrecommended(x,y):
----> 5     return(x + y + z)

NameError: name 'z' is not defined
```

# Local scope (variables defined inside a function)

- Variables defined inside a function are local to that function
- They are NOT accessible outside the function
- Local variables are destroyed when the function returns
- If you try to access a local variable outside a function, you will receive a NameError
  - It includes parameters and variables created inside the function

# Local scope (Example 1)

```
def print_x():
    x = 20 # local scope
    print(x)

# call the function
print_x()
print(x)
```

```
20
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
Cell In[1], line 7
      5 # call the function
      6 print_x()
----> 7 print(x)

NameError: name 'x' is not defined
```

# Local scope (Example 2)

- Local variables supersede global variables
- When a local variable within a function has the same name as a global variable, the local variable takes precedence within that function's scope.

```
# This is an example where we define a quadratic function
# (x,y) are both local variables of the function
#
# When we call the function, only the arguments matter.
# any intermediate value inside the function are "invisible"
# or "inaccessible" when exiting the function.

def fn_square(x):
    y = x**2
    return(y)

x = 5
y = -5

print(fn_square(x = 1.2))

print(x)
print(y)
```

```
1.44
5
-5
```

# Local scope (Example 3)

- Local variables are not stored in the working environment
- When exiting the function, changes to x and y are not updated
  - It shows the values associated with global variables

```
# The following code assigns a global variable x
x = 5
y = 4

print("Example 3.1:")
# x and y values change inside the function only
print(fn_square(x = 10))
# when exiting the function,
# the values are still associated with
# global variables.
print(x)
print(y)

print("Example 3.2:") # another similar example
print(fn_square(x = 20))
print(x)
print(y)
```

```
Example 3.1:
100
5
4
Example 3.2:
400
5
4
```

# Permanent changes to global variables

- Use global keyword to permanently change a global variable inside a function
- It tells Python you want to use the global variable instead of a new (but temporary) local variable

```
def modify_x():
    global x
    x = x + 5

x = 1
print('x={} before modification'.format(x))
# Now, running the function wil permanently increase x by 5.
modify_x()
print('x={} after modify_x function'.format(x))
```

```
x=1 before modification
x=6 after modify_x function
```

# Permanent changes to global variables

- This is just a technical trick to realize the permanent change
- You should avoid using it in practice because it adds complexity to the structure and interpretation of code

```
def modify_x():
    global x
    x = x + 5

x = 1
print('x={} before modification'.format(x))
# Now, running the function wil permanently increase x by 5.
modify_x()
print('x={} after modify_x function'.format(x))
```

```
x=1 before modification
x=6 after modify_x function
```

# In-class practice


- What if you run the function modify_x() twice?
- Write out the output sequentially.

- What if we add global y and create a new fn_square_2()?
- Use the same syntax and argument in Example 2
- Print out values of x and y before and after running fn_square_2()

# In-class practice


What if you run the function modify_x() twice?

```
# Write your own code here
print('x={} after modify_x function once'.format(x))
modify_x()
print('x={} after modify_x function twice'.format(x))
modify_x()
print('x={} after modify_x function three times'.format(x))
```

```
x=6 after modify_x function once
x=11 after modify_x function twice
x=16 after modify_x function three times
```

What if we add global y and create a new fn_square_2()?

```
def fn_square_2(x):
    global y
    y = x**2
    return(y)

x = 5
y = -5

print('before running function')
print(x)
print(y)
print('after running function')
print(fn_square_2(x = 1.2))
print(x)
print(y)
```

```
before running function
5
-5
after running function
1.44
5
1.44
```

# Built-in scope (variables defined in Python)

- We have seen many built-in functions in Python, such as print(), len(), sum(), isinstance(), etc
- They are available in ANY PART of your code, so no need to define them
- Python has a list of variables that are always available to prevent you from using the same names
- Most of them are error names.

# Built-in scope

- You can check the error names by importing builtins

```
import builtins
print(dir(builtins))
```

```
['ArithmeticError', 'AssertionError', 'AttributeError', 'BaseException', 'BaseExceptionGroup', 'BlockingIOError', 'BrokenPipeError', 'Buff
erError', 'BytesWarning', 'ChildProcessError', 'ConnectionAbortedError', 'ConnectionError', 'ConnectionRefusedError', 'ConnectionResetErro
r', 'DeprecationWarning', 'EOFError', 'Ellipsis', 'EncodingWarning', 'EnvironmentError', 'Exception', 'ExceptionGroup', 'False', 'FileExis
tsError', 'FileNotFoundError', 'FloatingPointError', 'FutureWarning', 'GeneratorExit', 'IOError', 'ImportError', 'ImportWarning', 'Indenta
tionError', 'IndexError', 'InterruptedError', 'IsADirectoryError', 'KeyError', 'KeyboardInterrupt', 'LookupError', 'MemoryError', 'ModuleN
otFoundError', 'NameError', 'None', 'NotADirectoryError', 'NotImplemented', 'NotImplementedError', 'OSError', 'OverflowError', 'PendingDep
recationWarning', 'PermissionError', 'ProcessLookupError', 'RecursionError', 'ReferenceError', 'ResourceWarning', 'RuntimeError', 'Runtime
Warning', 'StopAsyncIteration', 'StopIteration', 'SyntaxError', 'SyntaxWarning', 'SystemError', 'SystemExit', 'TabError', 'TimeoutError',
'True', 'TypeError', 'UnboundLocalError', 'UnicodeDecodeError', 'UnicodeEncodeError', 'UnicodeError', 'UnicodeTranslateError', 'UnicodeWar
ning', 'UserWarning', 'ValueError', 'Warning', 'ZeroDivisionError', '__IPYTHON__', '__build_class__', '__debug__', '__doc__', '__import_
_', '__loader__', '__name__', '__package__', '__spec__', 'abs', 'aiter', 'all', 'anext', 'any', 'ascii', 'bin', 'bool', 'breakpoint', 'byt
earray', 'bytes', 'callable', 'chr', 'classmethod', 'compile', 'complex', 'copyright', 'credits', 'delattr', 'dict', 'dir', 'display', 'di
vmod', 'enumerate', 'eval', 'exec', 'execfile', 'filter', 'float', 'format', 'frozenset', 'get_ipython', 'getattr', 'globals', 'hasattr',
'hash', 'help', 'hex', 'id', 'input', 'int', 'isinstance', 'issubclass', 'iter', 'len', 'license', 'list', 'locals', 'map', 'max', 'memory
view', 'min', 'next', 'object', 'oct', 'open', 'ord', 'pow', 'print', 'property', 'range', 'repr', 'reversed', 'round', 'runfile', 'set',
'setattr', 'slice', 'sorted', 'staticmethod', 'str', 'sum', 'super', 'tuple', 'type', 'vars', 'zip']
```

# Enclosing scope (variables defined in enclosing functions)

- They refer to variables defined in enclosing functions
- Enclosing functions are functions that contain other functions (nested functions)
  - Sort of Example 3 from week 6's lecture.
- Enclosing scope is between local and global scopes in the LEGB rule
- They are easier to understand once you understand local and global scopes

# Enclosing scope

```
# Define a function that
# contains another function
def outer():
    x = "outer->x" # Local to outer()

    # Define a nested function
    def inner():
        x = "inner->x" # Local to inner()
        print(x) # Print local to inner()

    inner() # Run inner()
    print(x) # Print local to outer()
```

```
outer() # Run outer()
```

```
inner->x
outer->x
```

# Importing modules

- While .ipynb files are great file learning and teaching, they are not the best way for sharing code.
- When you write a lot of functions, you should save them in a .py file, which is a Python script.
- A Python script, or module, is just a file with Python code.
- The code can be functions, classes, or simply variables.

# Importing modules

- A folder containing python scripts is called a package.
  - NumPy, Pandas, Matplotlib, etc.
- You can import modules to use their code in your own code.
- You can import certain functions into the working environment from a file.
- You can import global variables into the working environment from a file.

# Example

- First time to open a .py script
- Will cover debugging later

week_08_main_1.py | week_08_main_2.py | week_08_example_fun.py

```python
# import packages if necessary

# define global constants
alpha = 1.0

# define your functions
def message_hello(input_str):  new *
    return 'Hi' + input_str


def fn_cubic(input_num):  4 usages (2 dynamic)  new *
    return input_num ** 3
```

week_08_main_1.py × week_08_main_2.py week_08_example_fun.py

```
1   # Import your packages, modules, global constants first
2   import self_py_fun.week_08_example_fun as ef
3   import numpy as np
4
5
6   # Call variables
7   print(ef.alpha)
8
9   # Call your functions
10  print(ef.message_hello('Tianwen'))
11  print(ef.fn_cubic(3))
12  print(np.array([1.0]))
13
```

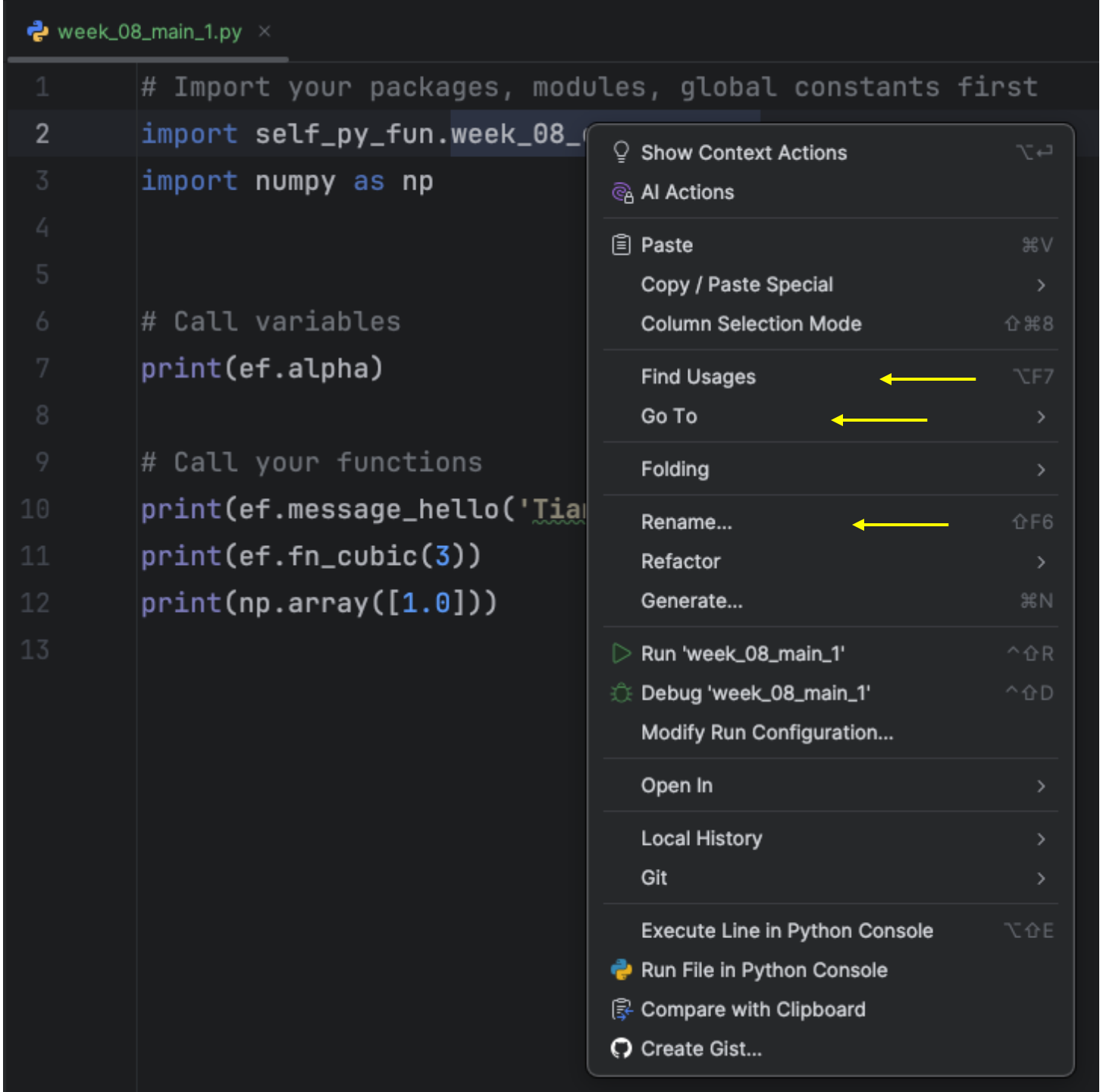
If you import NumPy in the week_08_example_fun.py, you do not need to import it again in the week_08_main_1.py if you have written line 2.

week_08_main_1.py | week_08_main_2.py | week_08_example_fun.py

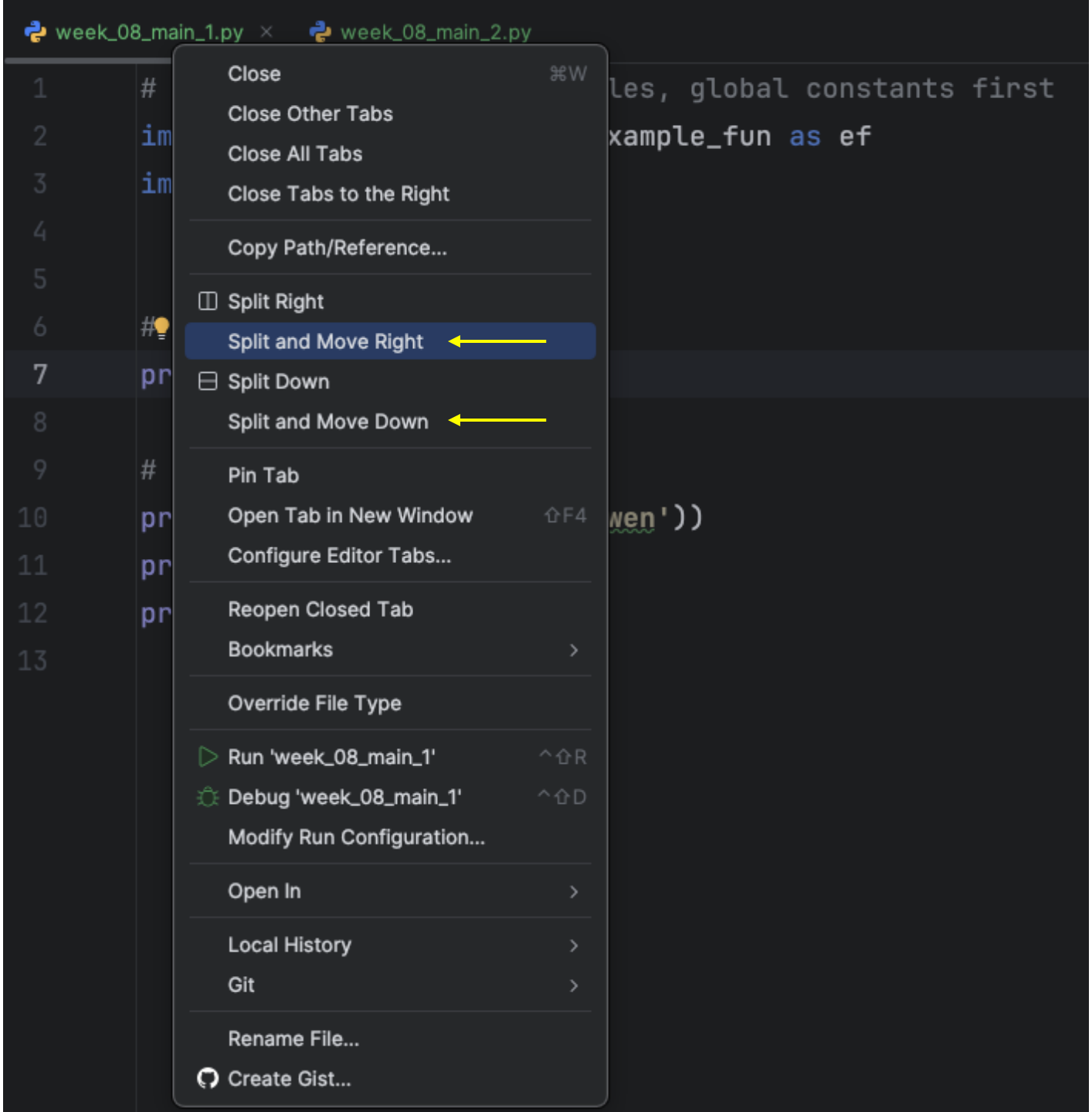
```python
# Import your packages, modules, global constants first
from self_py_fun.week_08_example_fun import *
import numpy as np

# Call your variable
print(alpha)

# Call your functions
# Since you import everything, we do not have to write ef.xxx
print(message_hello('Tianwen'))
print(fn_cubic(3))
print(np.array([1.0]))
```


week_08_main_1.py
1 # Import your packages, modules, global constants first
2 import self_py_fun.week_08_
3 import numpy as np
4
5
6 # Call variables
7 print(ef.alpha)
8
9 # Call your functions
10 print(ef.message_hello('Tia
11 print(ef.fn_cubic(3))
12 print(np.array([1.0]))
13
Show Context Actions ⌥↵
AI Actions
Paste ⌘V
Copy / Paste Special
Column Selection Mode ⇧⌘8
Find Usages ⌥F7
Go To
Folding
Rename... ⇧F6
Refactor
Generate... ⌘N
Run 'week_08_main_1' ^⇧R
Debug 'week_08_main_1' ^⇧D
Modify Run Configuration...
Open In
Local History
Git
Execute Line in Python Console ⌥⇧E
Run File in Python Console
Compare with Clipboard
Create Gist...

Right-click


week_08_main_1.py
week_08_main_2.py
Close ⌘W
Close Other Tabs
Close All Tabs
Close Tabs to the Right
Copy Path/Reference...
Split Right
Split and Move Right
Split Down
Split and Move Down
Pin Tab
Open Tab in New Window ⇧F4
Configure Editor Tabs...
Reopen Closed Tab
Bookmarks
Override File Type
Run 'week_08_main_1' ^⇧R
Debug 'week_08_main_1' ^⇧D
Modify Run Configuration...
Open In
Local History
Git
Rename File...
Create Gist...

Right-click

week_08_main_1.py

```python
# Import your packages, modules, gl
import self_py_fun.week_08_example_fun as
import numpy as np


# Call variables
print(ef.alpha)

# Call your functions
print(ef.message_hello('Tianwen'))
print(ef.fn_cubic(3))
print(np.array([1.0]))

```

week_08_main_2.py

```python
#Import your packages, modules, gl
from self_py_fun.week_08_example_fun impor
import numpy as np

# Call your variable
print(alpha)

# Call your functions
# Since you import everything, we do not h
print(message_hello('Tianwen'))
print(fn_cubic(3))
print(np.array([1.0]))
```

This is a cool function that I found very useful in practice.

# HW8

- You will convert your HW7.ipynb into a HW8.py and move relevant functions to a folder called “self_py_fun” under your Python project directory.
- You should call your self-written functions in the main file, i.e., HW8.py