More Functions

CS 1111
Introduction to Programming

Spring 2019

[*The Coder's Apprentice*, §5, §8-8.3]

Based in part on "Agnostic Programming: Learning to Design and Test Basic Programming Algorithms"
by Kinga Dobolyi, Kindle]

Function

Review

- Function = a block of code that can be called by other statements

- To define a function

```
def function_name(param1, param2, …):
```

- To call a function

```
function_name(arg1, arg2, …)
```

- Indent statements inside the function

Print versus Return

Review

Print

- All print statements reached by the function are executed
  - They are printed to the screen
- After a print statement is executed, the execution proceeds to the next statement

Return

- A return statement is optional
- Only the first return statement reached gets run
- If no return statement, function returns None
- A return ceases execution of a function and returns a value
- A return value is not printed, unless a function is printed

Review

Void vs. Value-Returning Functions

Void functions

- Does not return anything
  - *None* in Python
- Examples
  - `print(str)`
  - `random.seed(seed)`
  - `random.shuffle()`

Value-Return functions

- return something
- Examples
  - `abs(some_number)`
  - `random.randint(0, 100)`
  - `random.sample(some_list)`

(We will talk about random module later)

Pass by Value

```
my_value = 11

def change_a_value(some_value):
    print("Inside change_a_value(), some_value starts as: ", some_value)
    some_value *=2
    print("some_value now is: ", some_value)

print("Starting the program, my_value starts as: ", my_value)
change_a_value(my_value)
print("my_value now is still: ", my_value)
```

- Passing immutable types to a function.
- A copy of the variable (value and everything) is sent to the function.
- Changes made to the variable passed in are not reflected back where the function was called.

Pass by Reference

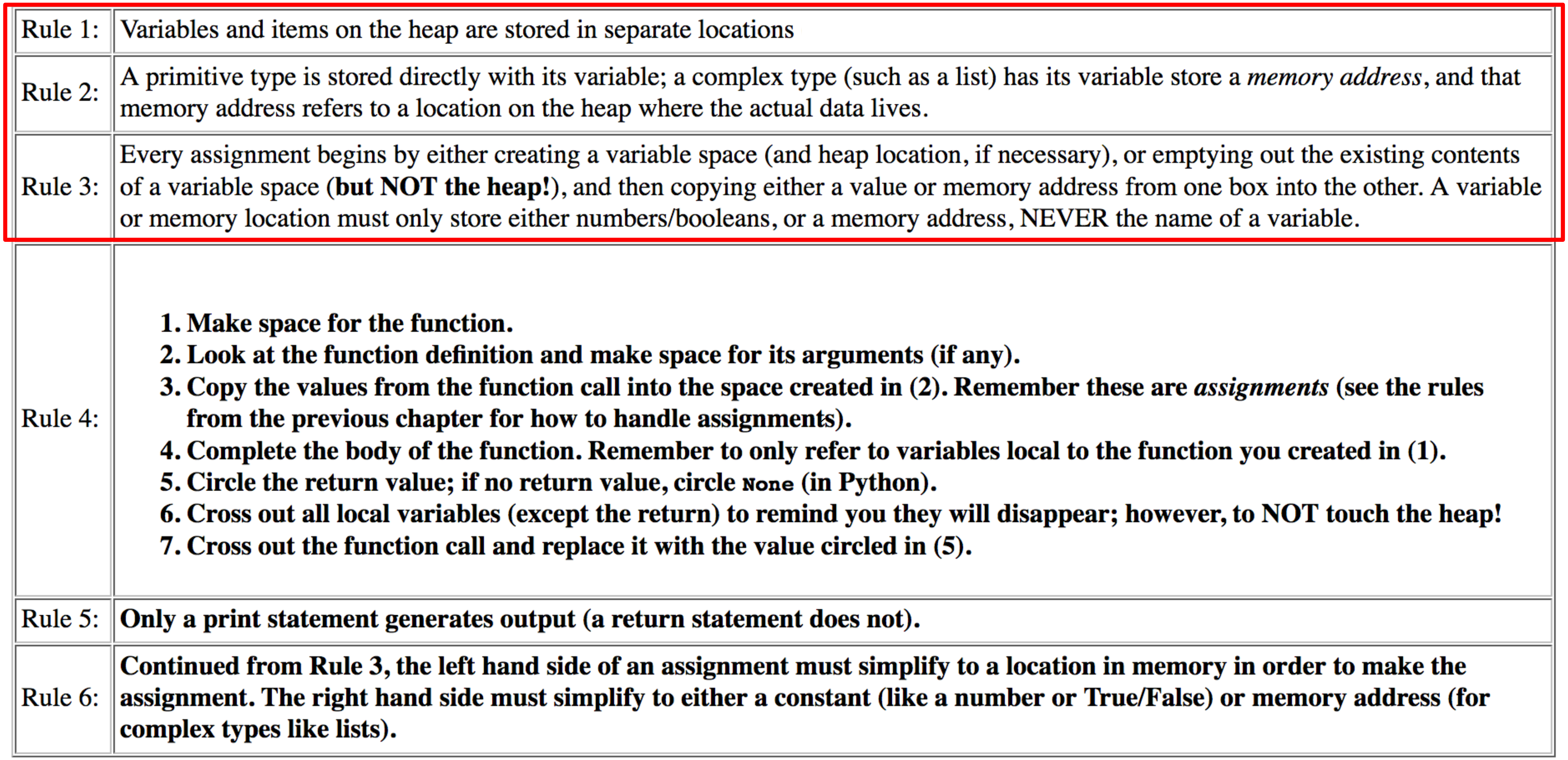
```
my_list = ['a', 'b', 'c', 'd']

def change_a_ref(some_list):
    print("Inside change_a_ref(), some_list starts as: ", some_list)
    some_list.append('x')
    print("some_list now is:", some_list)

print("Starting the program, my_list starts as: ", my_list)
change_a_ref(my_list)
print("my_list now is:", my_list)
```

- Passing mutable types to a function.
- A copy of the memory address of the object, is sent to the function.
- Changes made to the variable passed in are reflected back where the function was called.

Guideline: Tracing through code

Pass by value → copy of actual value

Pass by reference → copy of the memory address

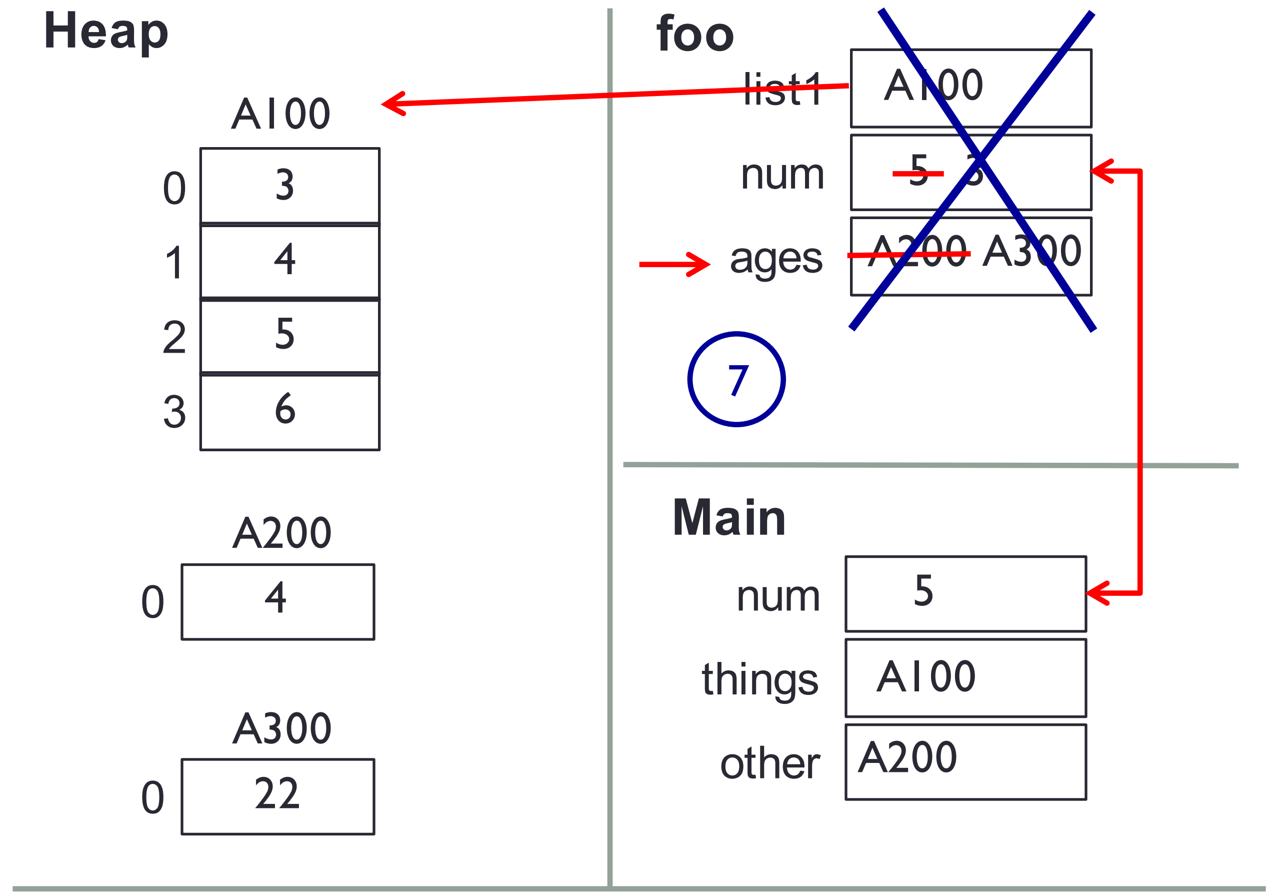


Rule 1:	Variables and items on the heap are stored in separate locations
Rule 2:	A primitive type is stored directly with its variable; a complex type (such as a list) has its variable store a *memory address*, and that memory address refers to a location on the heap where the actual data lives.
Rule 3:	Every assignment begins by either creating a variable space (and heap location, if necessary), or emptying out the existing contents of a variable space (**but NOT the heap!**), and then copying either a value or memory address from one box into the other. A variable or memory location must only store either numbers/booleans, or a memory address, NEVER the name of a variable.
Rule 4:	**1. Make space for the function.** **2. Look at the function definition and make space for its arguments (if any).** **3. Copy the values from the function call into the space created in (2). Remember these are *assignments* (see the rules from the previous chapter for how to handle assignments).** **4. Complete the body of the function. Remember to only refer to variables local to the function you created in (1).** **5. Circle the return value; if no return value, circle `None` (in Python).** **6. Cross out all local variables (except the return) to remind you they will disappear; however, to NOT touch the heap!** **7. Cross out the function call and replace it with the value circled in (5).**
Rule 5:	**Only a print statement generates output (a return statement does not).**
Rule 6:	**Continued from Rule 3, the left hand side of an assignment must simplify to a location in memory in order to make the assignment. The right hand side must simplify to either a constant (like a number or True/False) or memory address (for complex types like lists).**

Based in part on "Agnostic Programming: Learning to Design and Test Basic Programming Algorithms" by Kinga Dobolyi, Kindle]

Example: Tracing through Code

```
def foo(list1, num, ages):
    num = 3
    if len(list1) < 3:
        return 4
    list1.append(6)
    ages = [22]
    print(ages)
    return 7
Main
num = 5
things = [3, 4, 5]
other = [4]
~~foo(things, num, other)~~ 7
print(num)
print(things)
print(other)
things.remove(4)
things.remove(6)
foo(things, num, other)
result = foo(things, num, other)
print(result)
```



Heap

A100

0	3
1	4
2	5
3	6

A200

0	4

A300

0	22

foo

list1	A100
num	~~5~~ 3
ages	~~A200~~ A300

7

Main

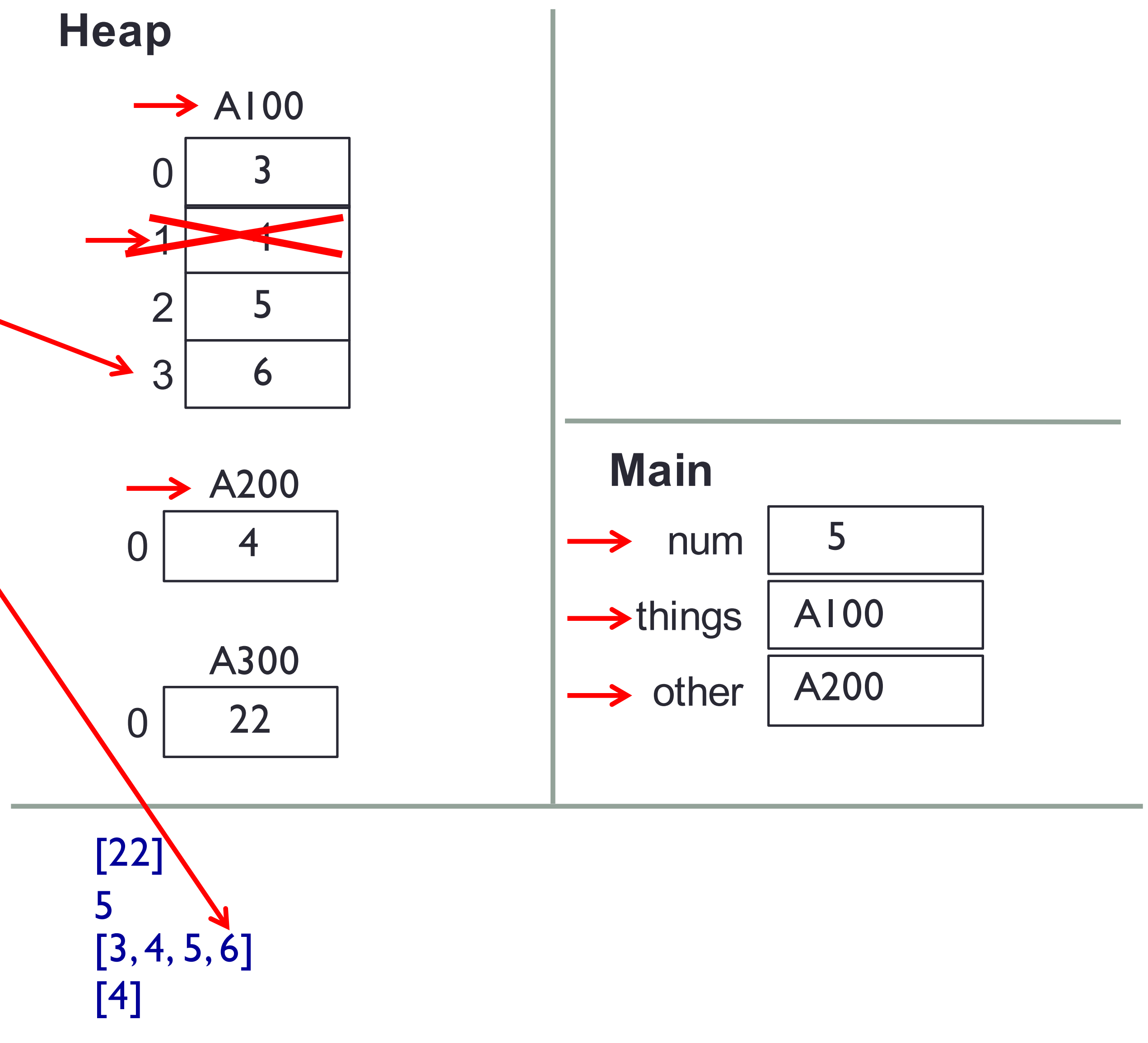
num	5
things	A100
other	A200

[22]

A list in this example is to demonstrate complex data type and passing by reference. Do not worry about list for now. We will discuss list in detail after exam1

Example: Tracing through Code (2)

```
def foo(list1, num, ages):
    num = 3
    if len(list1) < 3:
        return 4
    list1.append(6)
    ages = [22]
    print(ages)
    return 7

num = 5
things = [3, 4, 5]
other = [4]
~~foo(things, num, other)~~ 7
print(num)
print(things)
print(other)
things.remove(4)
things.remove(6)
foo(things, num, other)
result = foo(things, num, other)
print(result)
```



Heap

A100

0	3
~~1~~	~~4~~
2	5
3	6

A200

0	4

A300

0	22

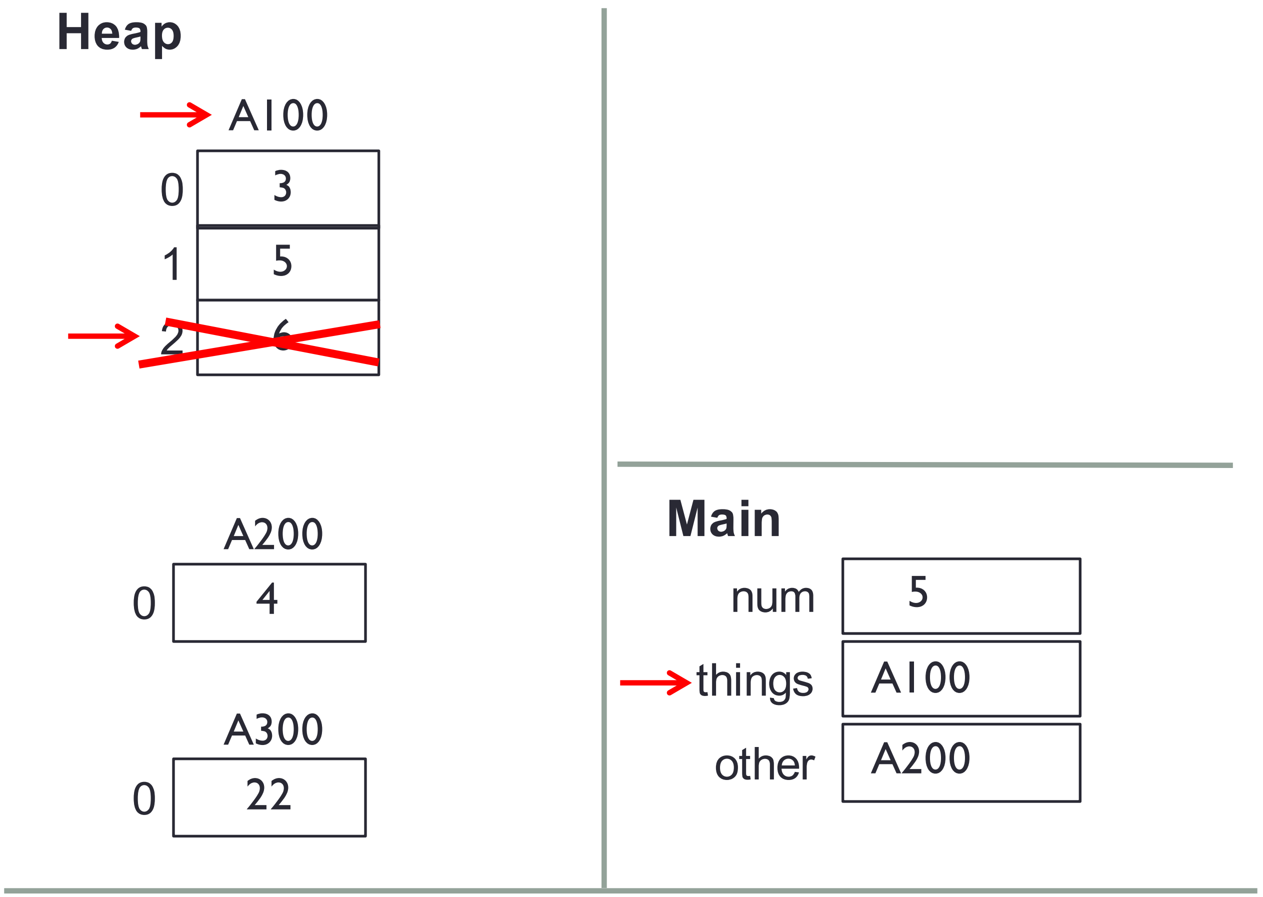
Main

num	5
things	A100
other	A200

```
[22]
5
[3, 4, 5, 6]
[4]
```

Example: Tracing through Code (3)

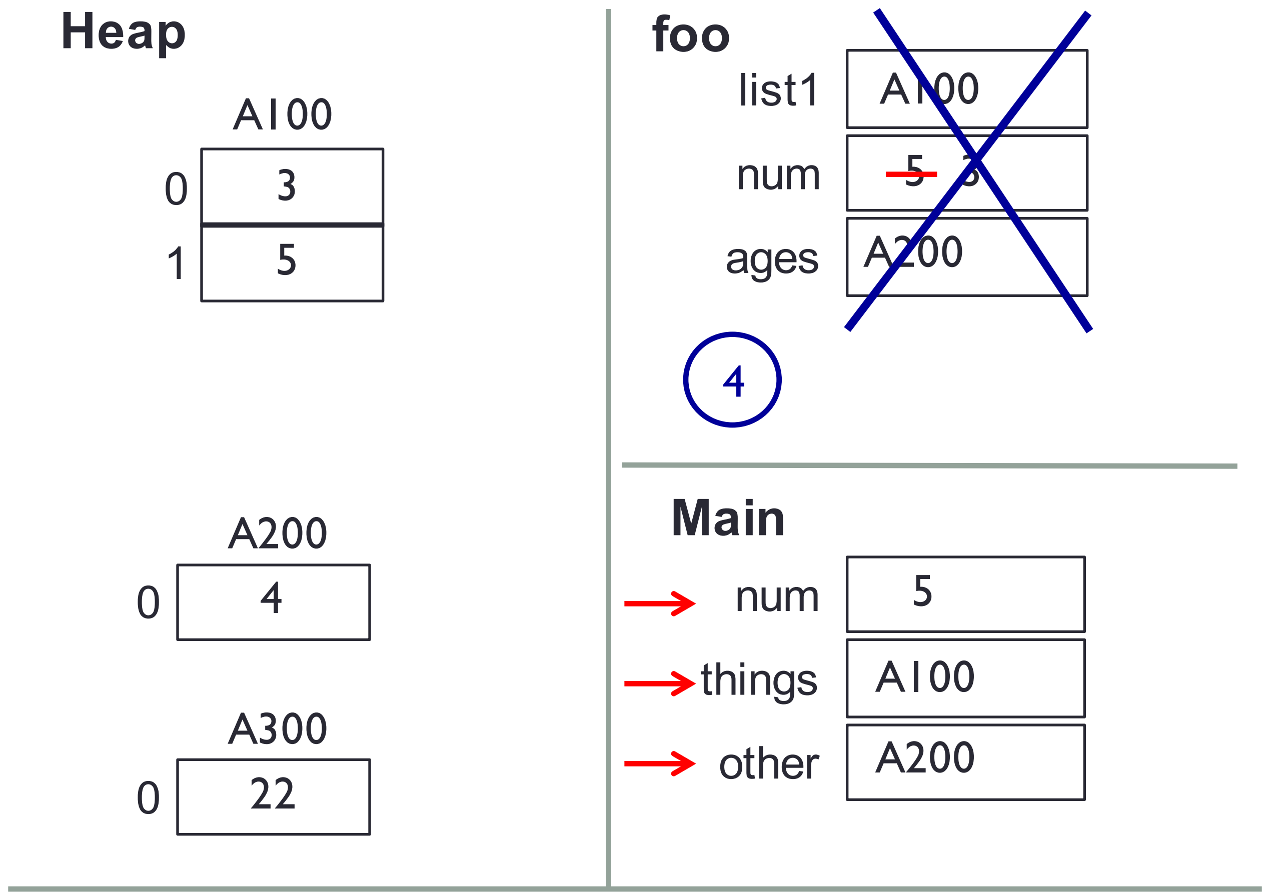
```
def foo(list1, num, ages):
    num = 3
    if len(list1) < 3:
        return 4
    list1.append(6)
    ages = [22]
    print(ages)
    return 7

num = 5
things = [3, 4, 5]
other = [4]
~~foo(things, num, other)~~ 7
print(num)
print(things)
print(other)
things.remove(4)
things.remove(6)
foo(things, num, other)
result = foo(things, num, other)
print(result)
```



```
[22]
5
[3,4,5,6]
[4]
```

Example: Tracing through Code (4)

```
def foo(list1, num, ages):
    num = 3
    if len(list1) < 3:
        return 4
    list1.append(6)
    ages = [22]
    print(ages)
    return 7

num = 5
things = [3, 4, 5]
other = [4]
~~foo(things, num, other)~~ 7
print(num)
print(things)
print(other)
things.remove(4)
things.remove(6)
~~foo(things, num, other)~~ 4
result = foo(things, num, other)
print(result)
```



Heap

A100

0	3
1	5

A200

0	4

A300

0	22

foo

list1	A100
num	~~5~~ 3
ages	A200

4

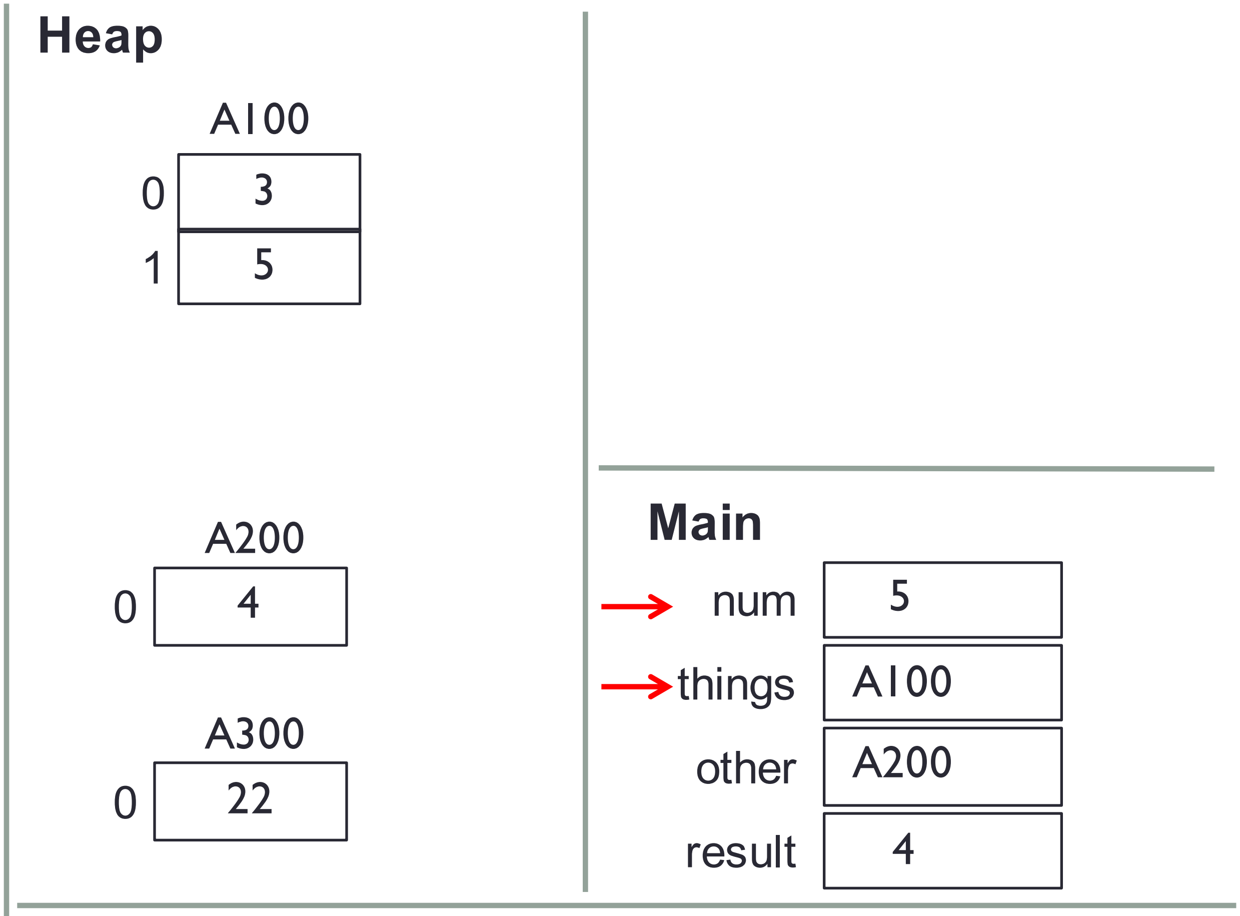
Main

num	5
things	A100
other	A200

```
[22]
5
[3,4,5,6]
[4]
```

Example: Tracing through Code (5)

```
def foo(list1, num, ages):
    num = 3
    if len(list1) < 3:
        return 4
    list1.append(6)
    ages = [22]
    print(ages)
    return 7

num = 5
things = [3, 4, 5]
other = [4]
~~foo(things, num, other)~~  7
print(num)
print(things)
print(other)
things.remove(4)
things.remove(6)
~~foo(things, num, other)~~  4
result = ~~foo(things, num, other)~~ 4
print(result)
```



Heap

A100
0 | 3
1 | 5

A200
0 | 4

A300
0 | 22

Main

num	5
things	A100
other	A200
result	4

```
[22]
5
[3,4,5,6]
[4]
4
```

Calling Functions from Functions

```
def main():
    print('The sum of 12 and 45 is ')
    show_sum(12, 45)

def show_sum(num1, num2):
    result = num1 + num2
    print(result)

main()
```

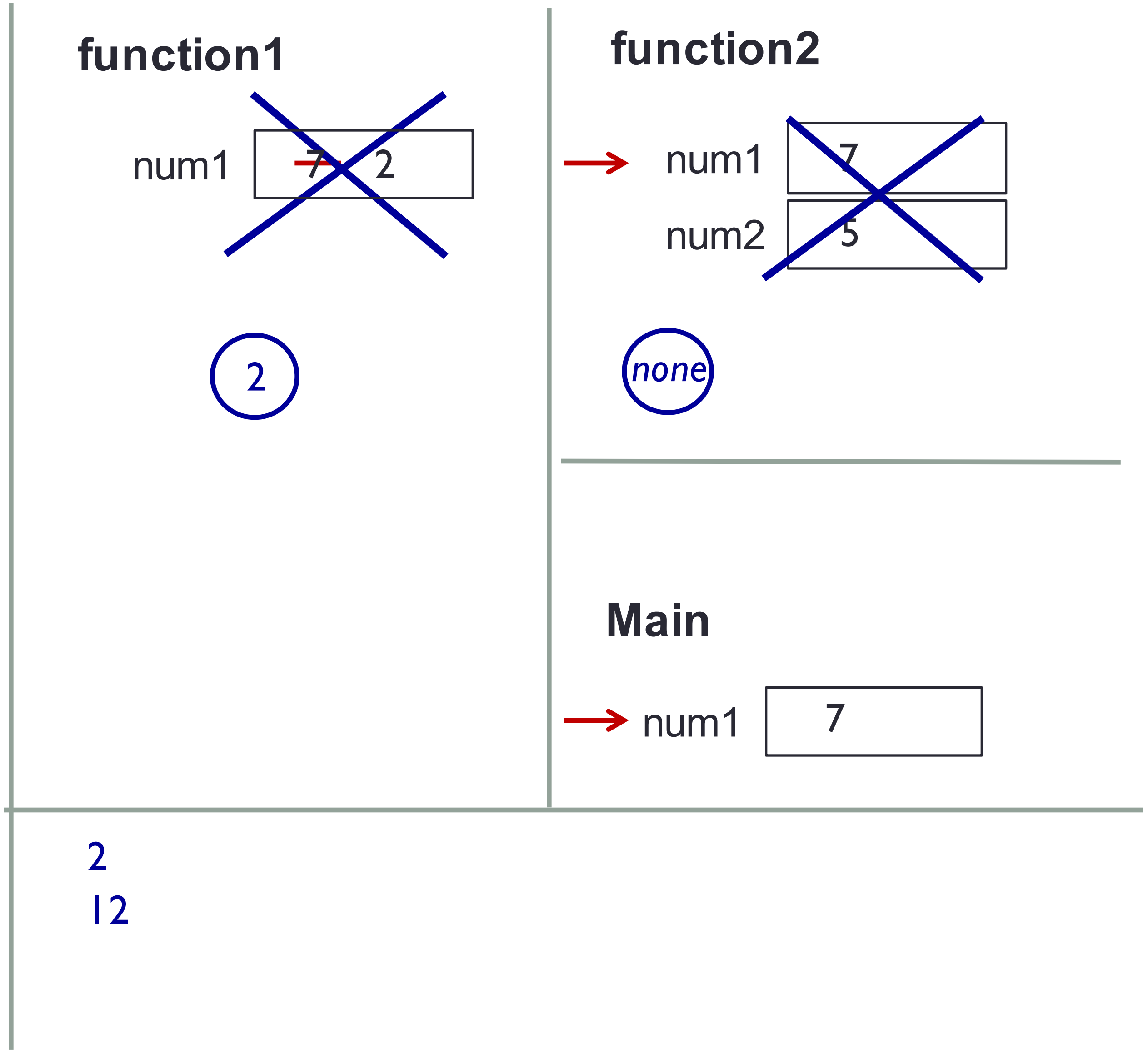
```
The sum of 12 and 45 is
57
```

More Example: Tracing through Code with multiple function

```
def function1(num1):
    num1 = 2
    print(num1)
    return num1

def function2(num1, num2):
    ~~function1(num1)~~  2
    print(num1 + num2)
          7    +  5

def other(num1, num2):
    num1 = function1(num1)
    print(num1)
```

Main

```
num1 = 7
~~function2(num1, 5)~~  none
print(other(num1, 5))
```



function1

num1 ~~7~~ 2

2

function2

num1 ~~7~~

num2 ~~5~~

none

Main

num1 7

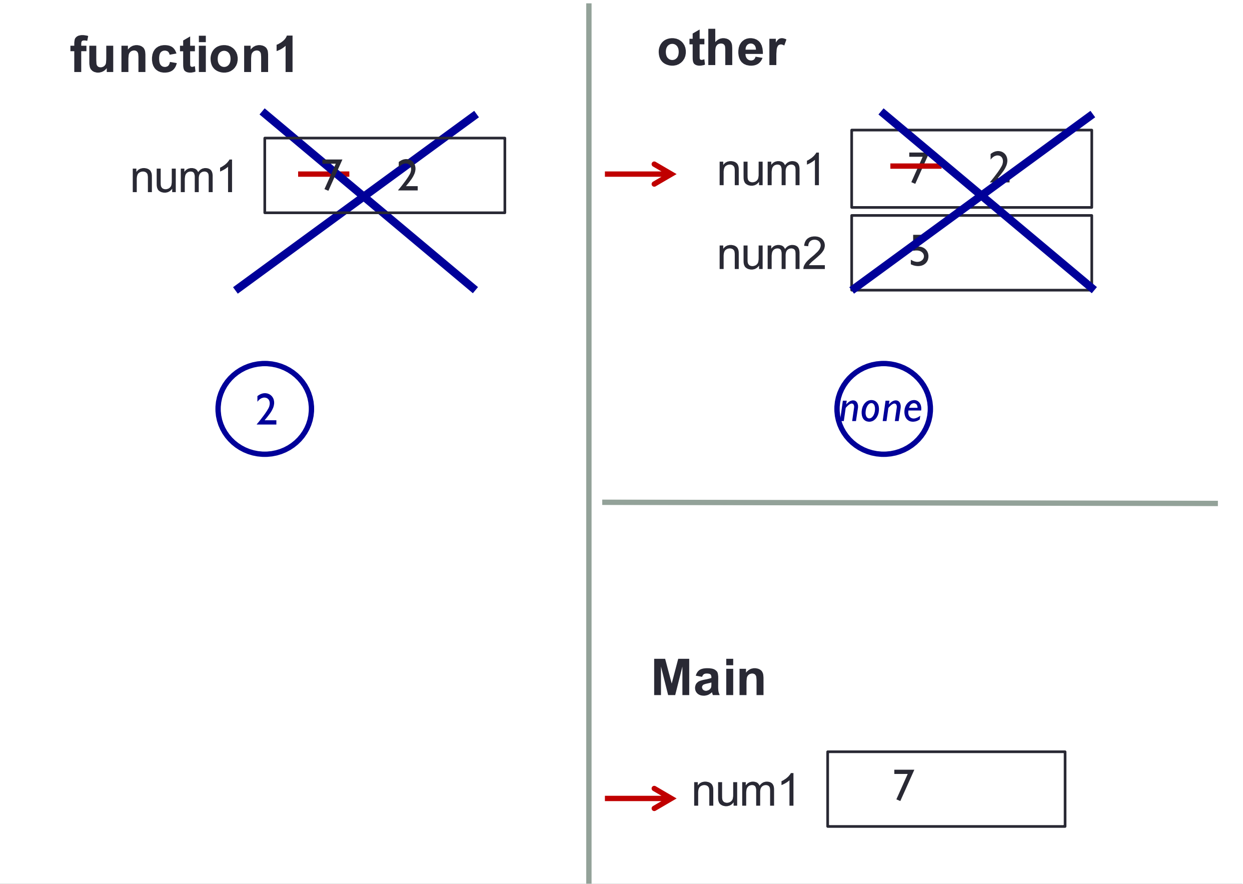
```
2
12
```

More Example: Tracing through Code with multiple function

```
def function1(num1):
    num1 = 2
    print(num1)
    return num1

def function2(num1, num2):
    function1(num1)
    print(num1 + num2)

def other(num1, num2):
    num1 = ~~function1(num1)~~ 2
    print(num1)
```

Main

```
num1 = 7
function2(num1, 5)
print(~~other(num1, 5)~~) none
```



function1

num1 ~~7~~ 2

2

other

num1 ~~7~~ 2

num2 5

none

Main

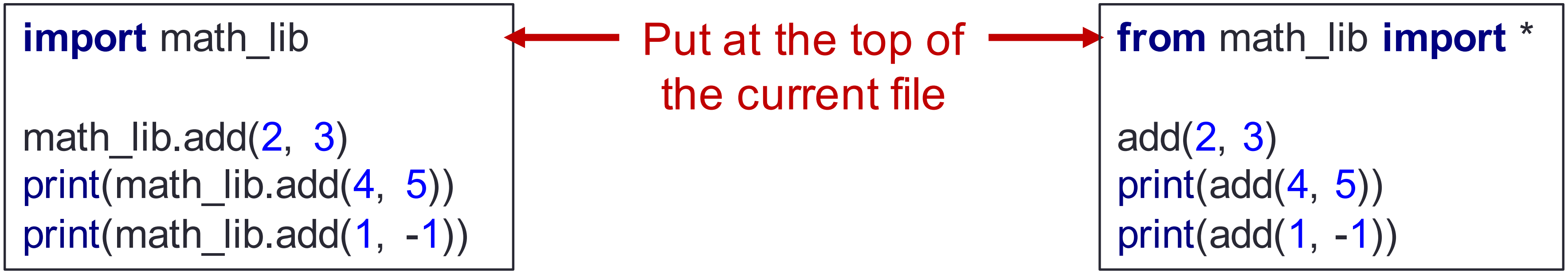
num1 7

```
2
12
2
2
none
```

Importing Existing Functions

- Assuming the `add()` function and other functions are saved in a file called `math_lib.py` (= module's name is `math_lib`)

- **`Import`** *`module_name`* or **`From`** *`module_name`* **`import *`** allows us to import all functions from `math_lib.py` into the current file
  - Call functions from other files
  - Can use `add()` without defining it here



```
import math_lib

math_lib.add(2, 3)
print(math_lib.add(4, 5))
print(math_lib.add(1, -1))
```

Put at the top of the current file

```
from math_lib import *

add(2, 3)
print(add(4, 5))
print(add(1, -1))
```

- Python imports some standard functions, such as `str()` and `len()` automatically; others need the import statement

Local and Global Variables

- Local variables
  - Arguments and any variables declared in the function
  - Cannot be seen by other functions or code
  - Even if they have the same name as variables outside the function, the computer treats them as different (think of two people both named Tom; they are different people though they happen to be named the same)
- Each function call has its own memory space and variables
- These local data disappear when the function finishes
- Arguments are assigned from the function call
- Global variables
  - Is accessible to all the functions in a program file

Local Variables

```
number = 0

def main():
    number = int(input('Enter a number: '))
    show_number()

def show_number():
    print('The number you entered is ', number)

main()
```

```
Enter a number: 7
The number you entered is 0
```

Global Variables

```
number = 0

def main():
    global number
    number = int(input('Enter a number: '))
    show_number()

def show_number():
    print('The number you entered is ', number)

main()
```

```
Enter a number: 7
The number you entered is 7
```