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]

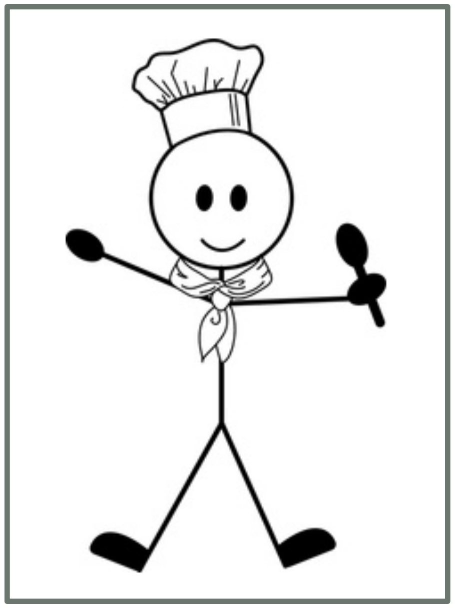
Let's order Big Mac

Call a function `make_bigmac()`

order Big Mac

return a Big Mac

Purpose, input, output



function `make_bigmac()`

Do action when it is called

Must exist before calling

How many times can we order a Big Mac?

Do we want a `make_bigmac()` to give us back a Big Mac or just show us?

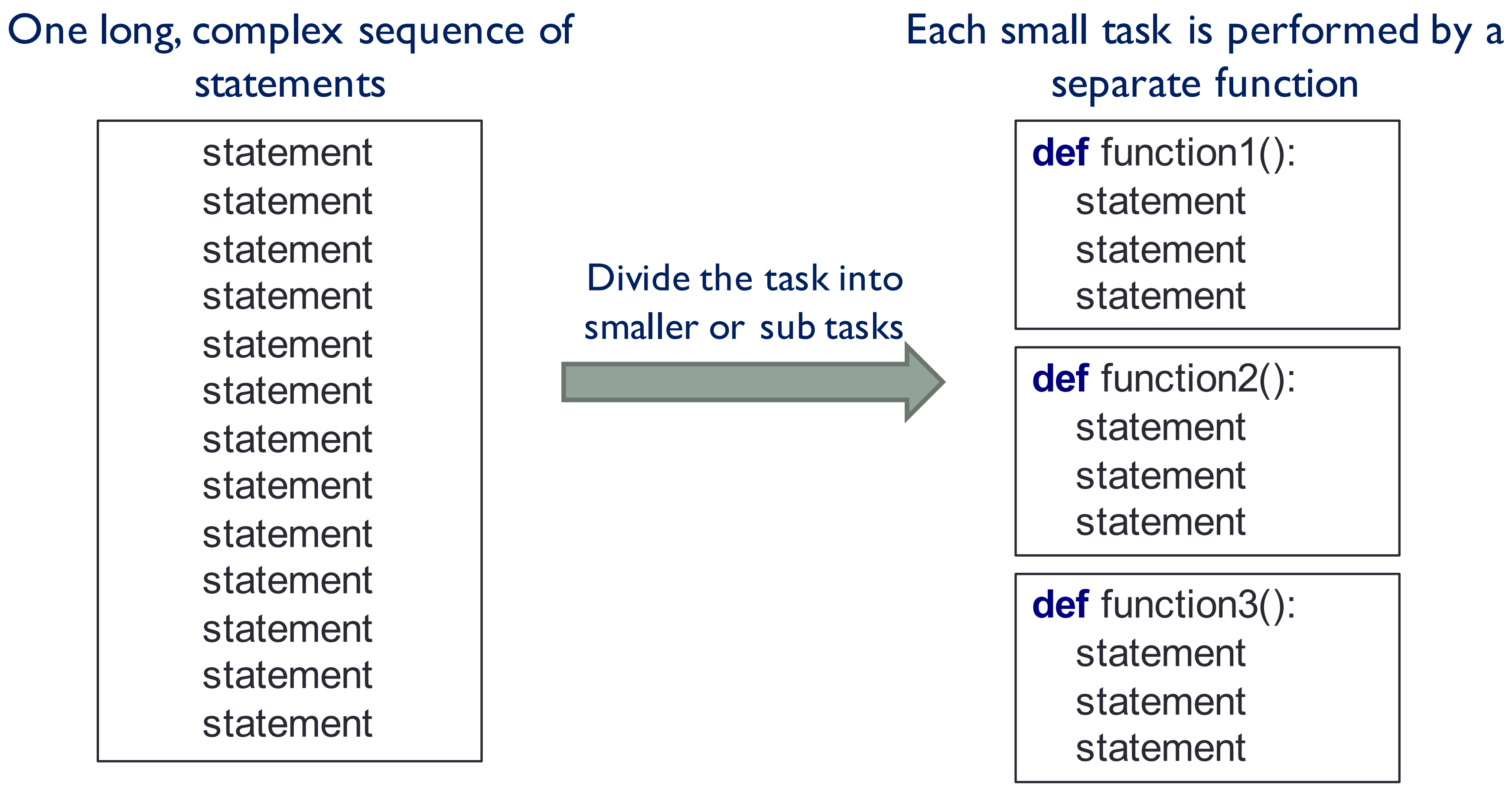
`return` `print`

Overview: Functions

- What are functions?
- Why use functions
- Defining and calling functions
- Void function and value-returning function
- return versus print in functions
- Tracing through functions

What are Functions?

- Groups of statements the exist within a program for the purpose of performing a specific task [Gaddis]



One long, complex sequence of statements
Divide the task into smaller or sub tasks
Each small task is performed by a separate function

```
statement
statement
statement
statement
statement
statement
statement
statement
statement
statement
statement
statement
statement
```

```
def function1():
    statement
    statement
    statement
```

```
def function2():
    statement
    statement
    statement
```

```
def function3():
    statement
    statement
    statement
```

Why use Functions

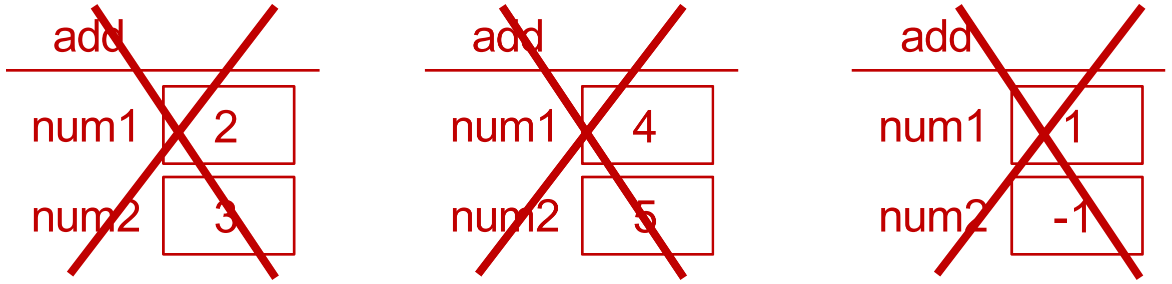
- Code reuse
  - Allow code to be reused with some modification through parameters
- Readability
  - Organize code based on "what" it does; e.g., make_bigmac(), load_file(file) and compute_gpa(scores)
  - Make code simpler and easier to understand
    - Note: use informative / descriptive names
- Maintainability
  - Make code easier to isolate, fix, and update
- Testing
  - Verify one functionality at a time, easier to isolate and fix errors

Defining and Calling Functions

```python
def add(num1, num2):
    print("I am adding " + str(num1) + " and " + str(num2))
    return (num1 + num2)
```

Main

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



- **def** is a keyword to define a function, ends in a colon
- Must name the function
- Specify arguments (optional)
- Provide the body of the function (everything indented belongs to the function)
- Function can be called, with arguments, after declared
- Call a previously defined function by its name
- Pass in values for the arguments

Void and Value-Returning Functions

Value-Returning function

```
def add(num1,  num2):
    print("I am adding " + str(num1) + " and " + str(num2))
    return (num1 + num2)
```

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

Main

Void function

```
def add(num1,  num2):
    print("I am adding " + str(num1) + " and " + str(num2))
```

Return versus Print

- **return** statement is optional
  - Only first return statement reached gets run
  - If no return statement, function returns **None**
- A return statement ceases execution of a function and returns a value
  - At most one (the first) return statement that is reached during a particular function call is executed
- A function can return value(s), specified by the first return statement that is executed
- All **print** statements reached by the function are executed; they are printed to the screen
- A return value is not printed, unless a function is printed
  - `print(add(2, 3))`

Tracing through Code with Functions

- 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 has its variable store a **memory address.**
    - A memory address refers to a location on the heap where the actual data is stored.
- 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!**).
  - 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.

Tracing through Code with Functions

- Rule 4:
  There are seven steps for every function call:
  1. Make space for the function.
  2. Look at the function definition and make space for its argument.
  3. Copy the values from the function call into the space created in (2). Remember these are *assignments*.
  4. Complete the body of the function. Remember to only refer to variables local to the function you crated in (1).
  5. Circle the return value; if no return value, circle **`None`** (to remind you there is no value to be sent back).
  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).

Tracing through Code with Functions

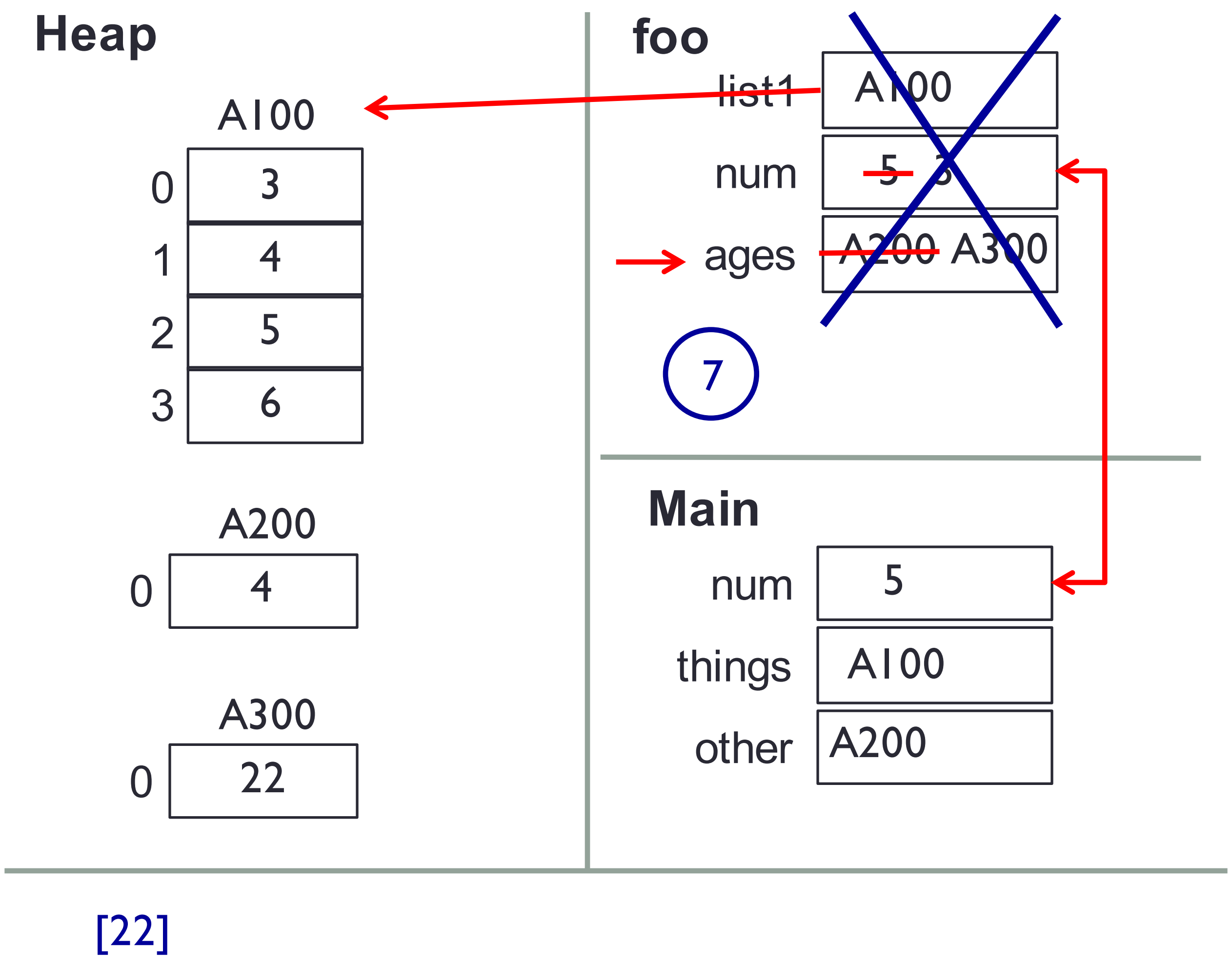
- Rule 5
  - Only a print statement generates output (a return statement does not).

- 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 of True/False) or memory address (for complex types like lists).

Extra slides

Example: Tracing through Code

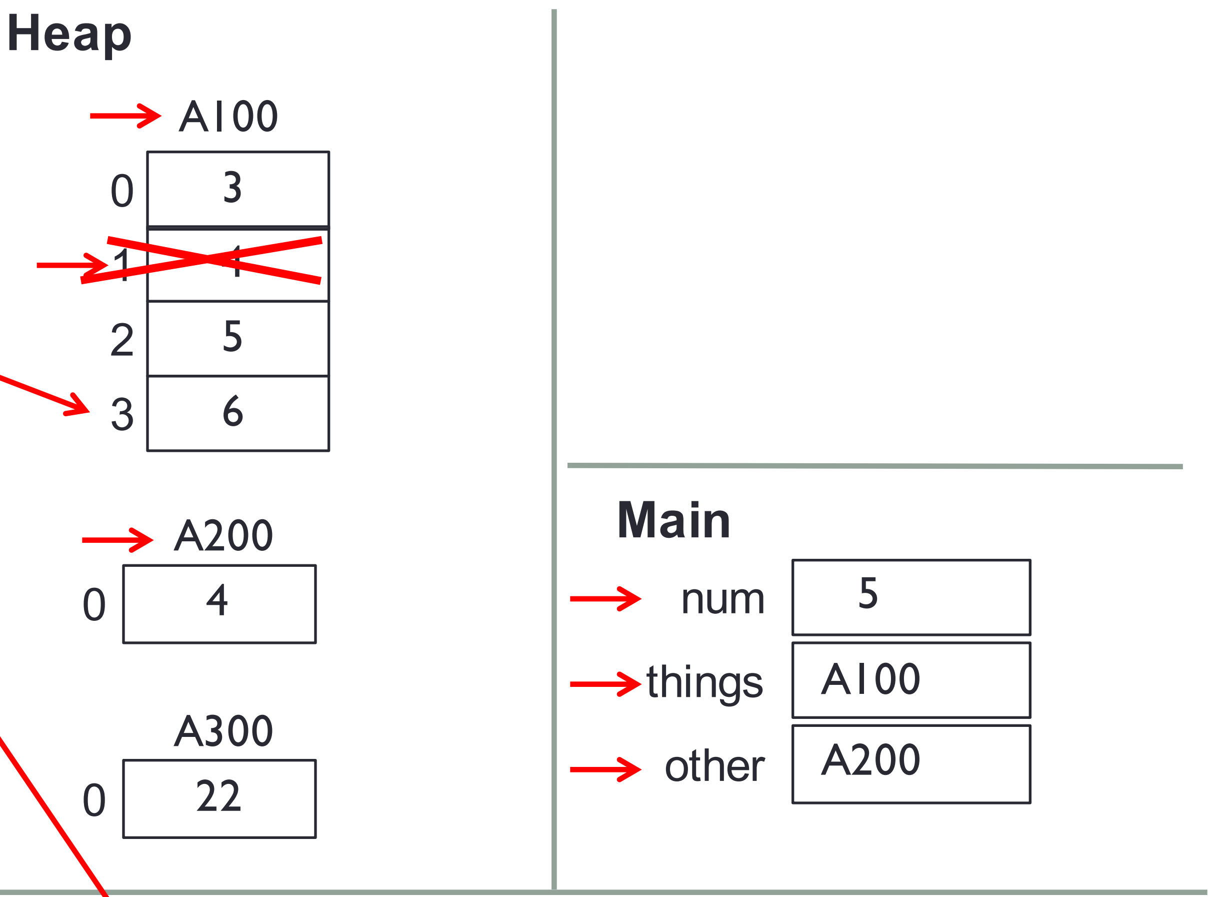
```
def foo(list1, num, ages):
    num = 3
    if len(list1) < 3:
        return 4
    list1.append(6)
    ages = [22]
    print(ages)
    return 7
Main
num = 5
things = [3, 4, 5]
other = [4]
~~foo(things, num, other)~~ 7
print(num)
print(things)
print(other)
things.remove(4)
things.remove(6)
foo(things,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



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)
```

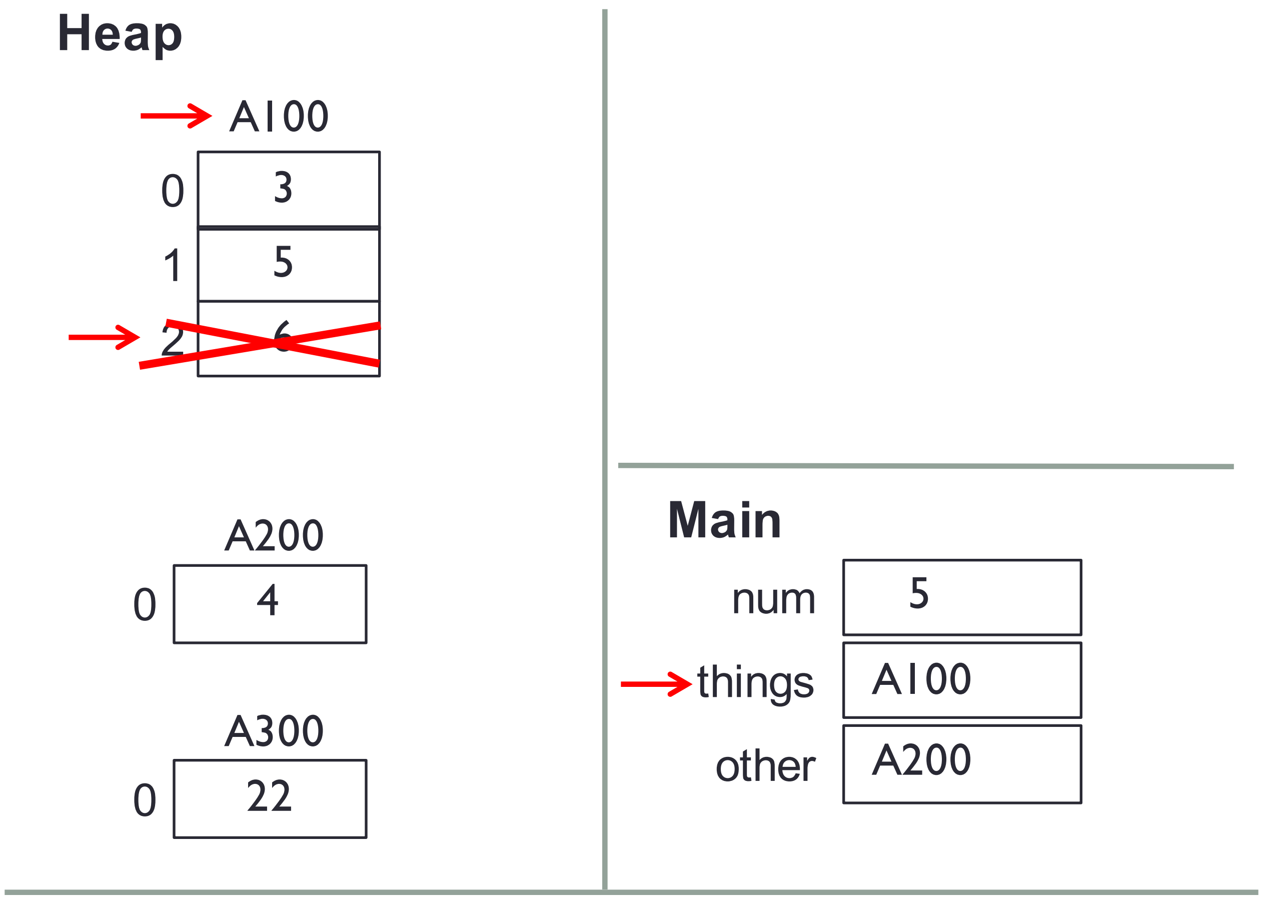


```
[22]
5
[3, 4, 5, 6]
[4]
```

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)
```

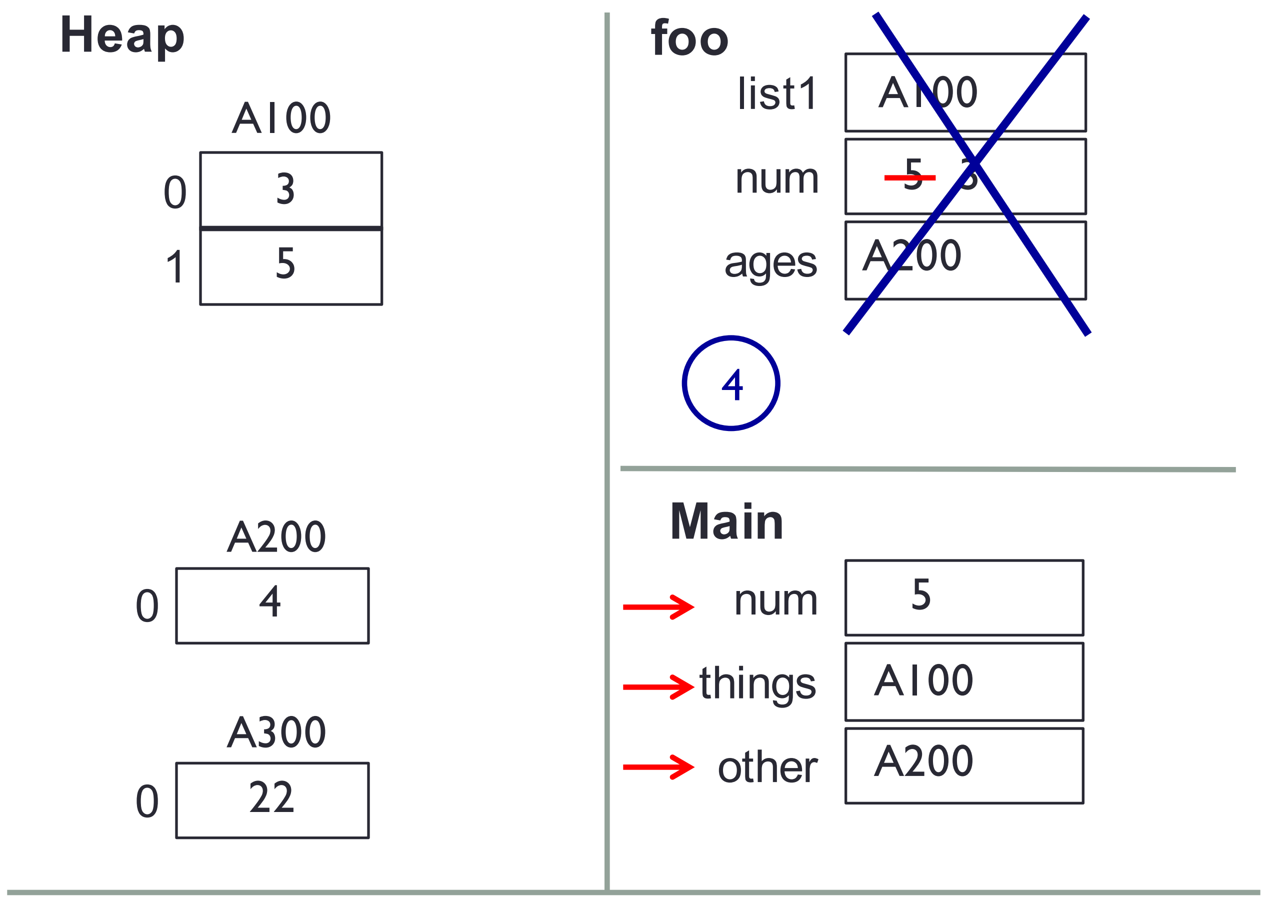


```
[22]
5
[3, 4, 5, 6]
[4]
```

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)
```

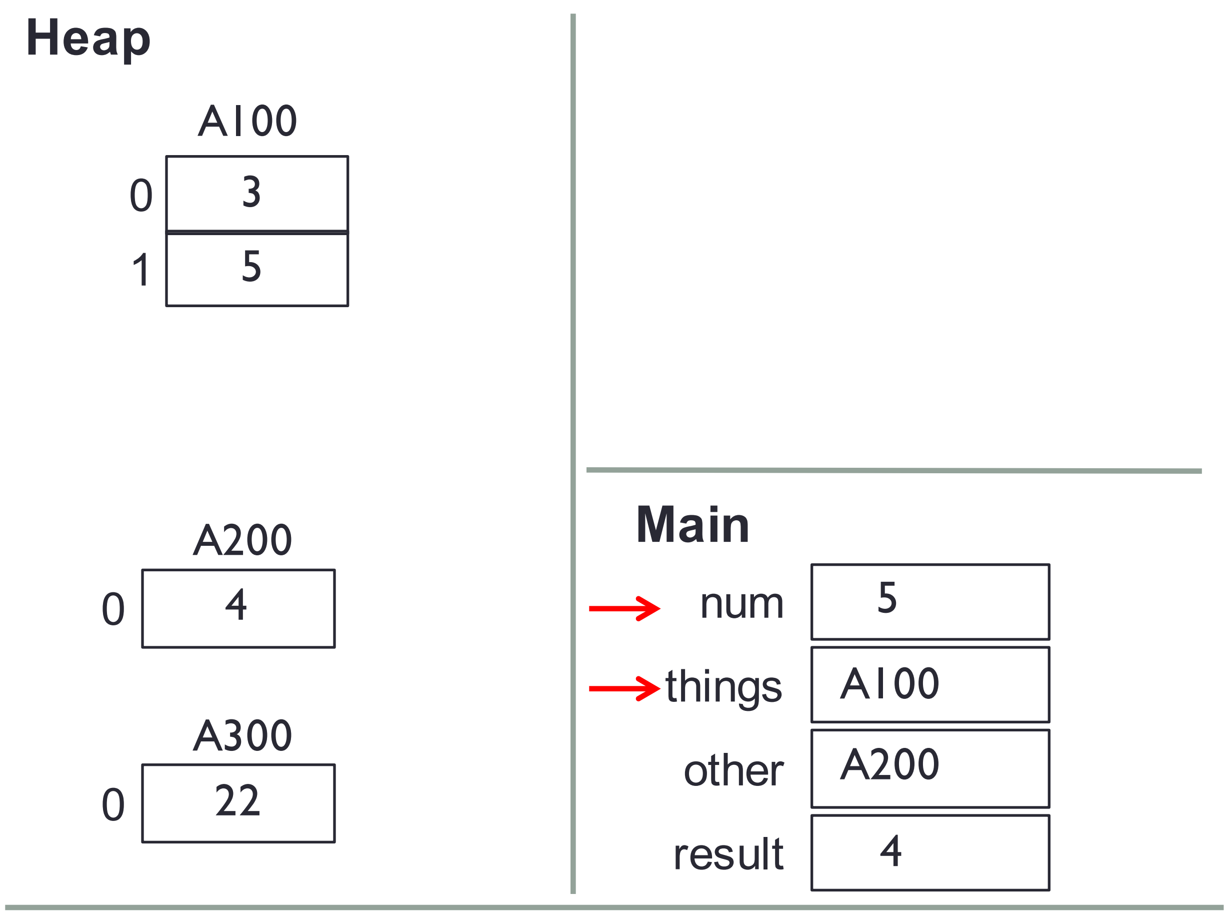


```
[22]
5
[3, 4, 5, 6]
[4]
```

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)
```



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