



THE UNIVERSITY OF
CHICAGO

**MASTERS IN
COMPUTATIONAL
SOCIAL SCIENCE**
THE UNIVERSITY OF CHICAGO

MACS 30111

Functions

Misc

- ▶ Final exam! DATE TBD:
 - ▶ <https://registrar.uchicago.edu/calendars/final-exams/>
- ▶ 1/24 Friday PA 1
- ▶ 1/27 Monday PA 1 **reflection**

Agenda

- ▶ **Deep / shallow copies**
- ▶ Introduction to functions
- ▶ Function call control flow
- ▶ Return statements
- ▶ Parameters
- ▶ Scoping
- ▶ Abstraction

Lists of lists: where it gets weird

↴ Test out the following:

↴ `m = [[0]*5]*5`

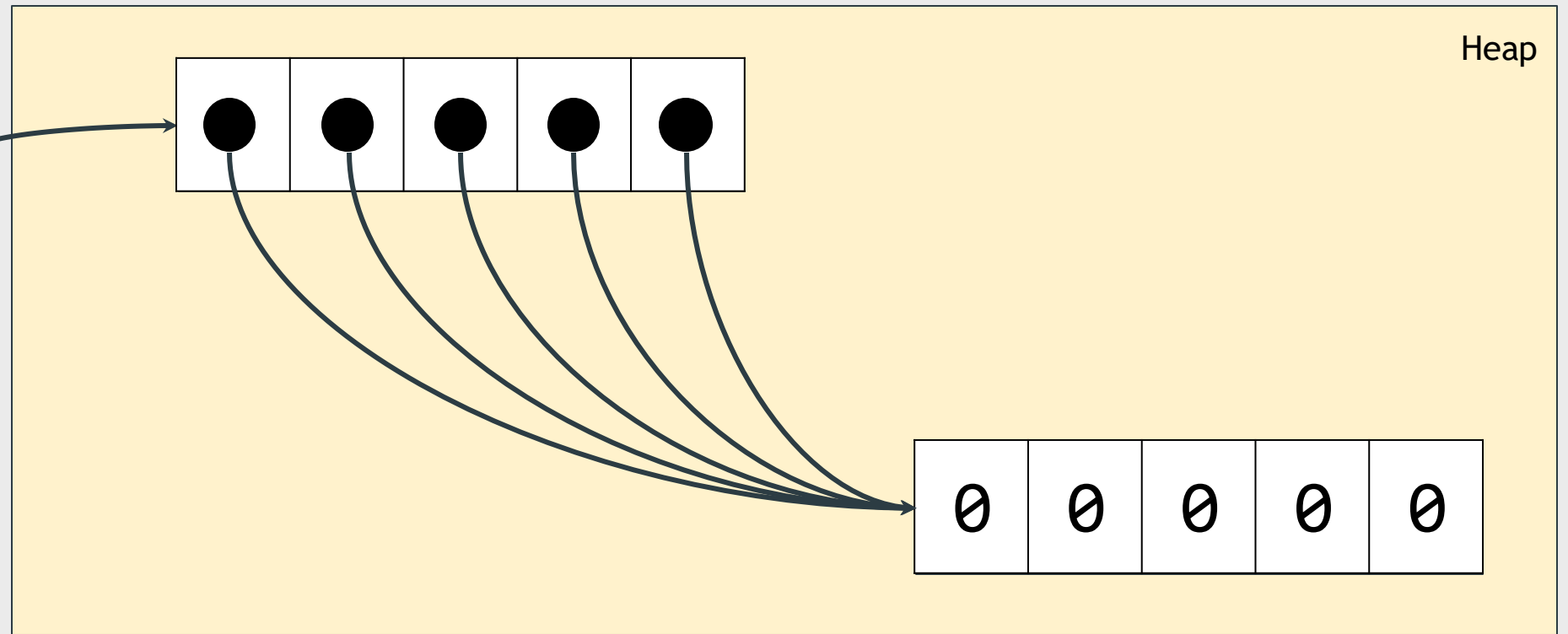
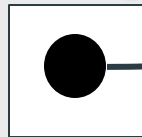
↴ `m1 = [[0]*5 for i in range(5)]`

↴ Now, try: `m1[2][3] = 1` vs `m[2][3] = 1`

Lists of

m

lists



```
m = [[0]*5]*5
```

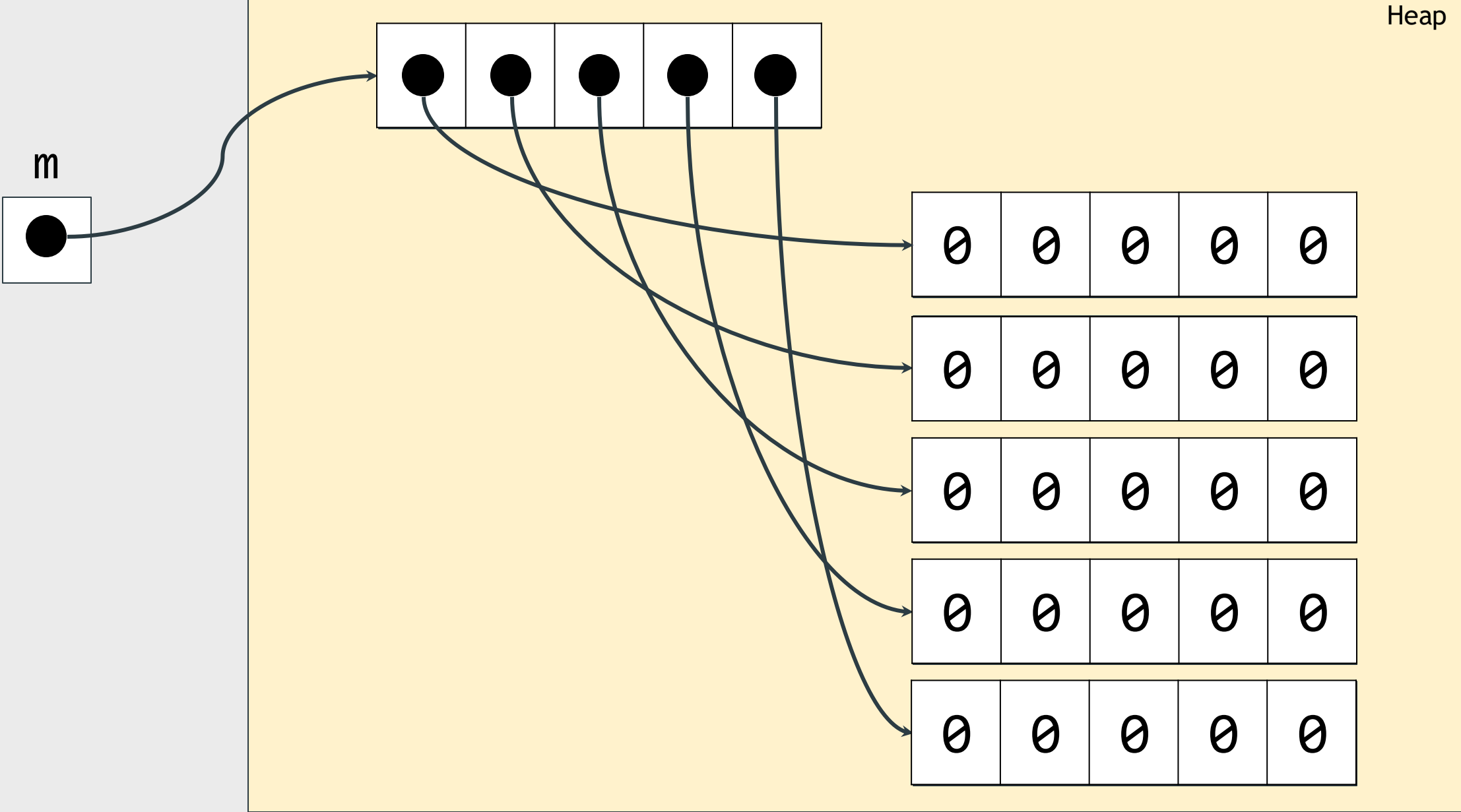
```
[[0, 0, 0, 0, 0],  
 [0, 0, 0, 0, 0],  
 [0, 0, 0, 0, 0],  
 [0, 0, 0, 0, 0],  
 [0, 0, 0, 0, 0]]
```

```
m[2][3] = 1
```

```
[[0, 0, 0, 1, 0],  
 [0, 0, 0, 1, 0],  
 [0, 0, 0, 1, 0],  
 [0, 0, 0, 1, 0],  
 [0, 0, 0, 1, 0]]
```

```
m = [[0]*5 for i in range(5)]
```

Memory

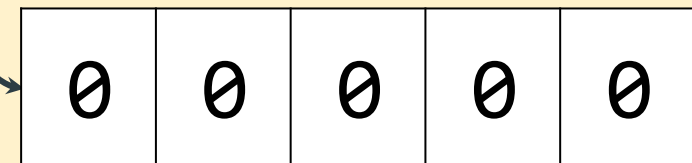
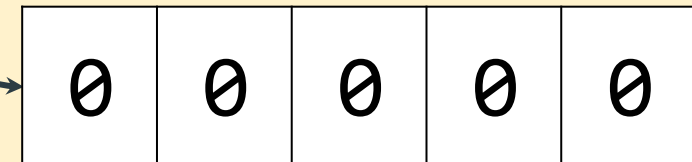
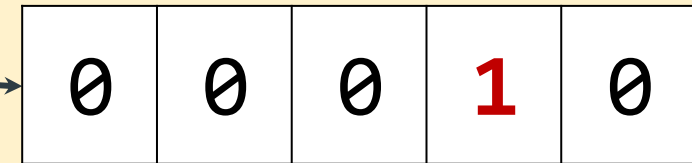
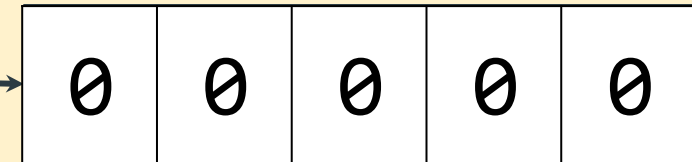
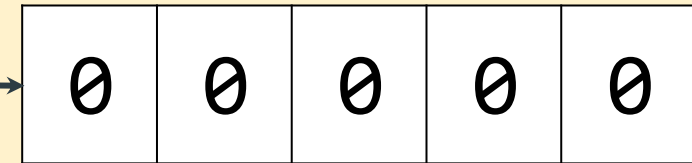
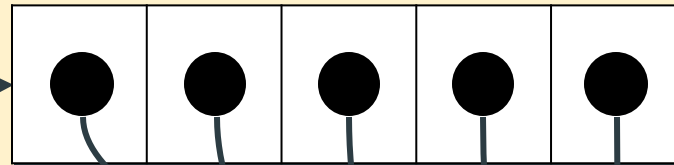
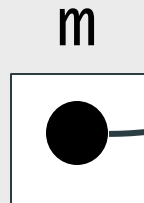


```
m = [[0]*5 for i in range(5)]
```

Memory

```
m[2][3] = 1
```

Heap

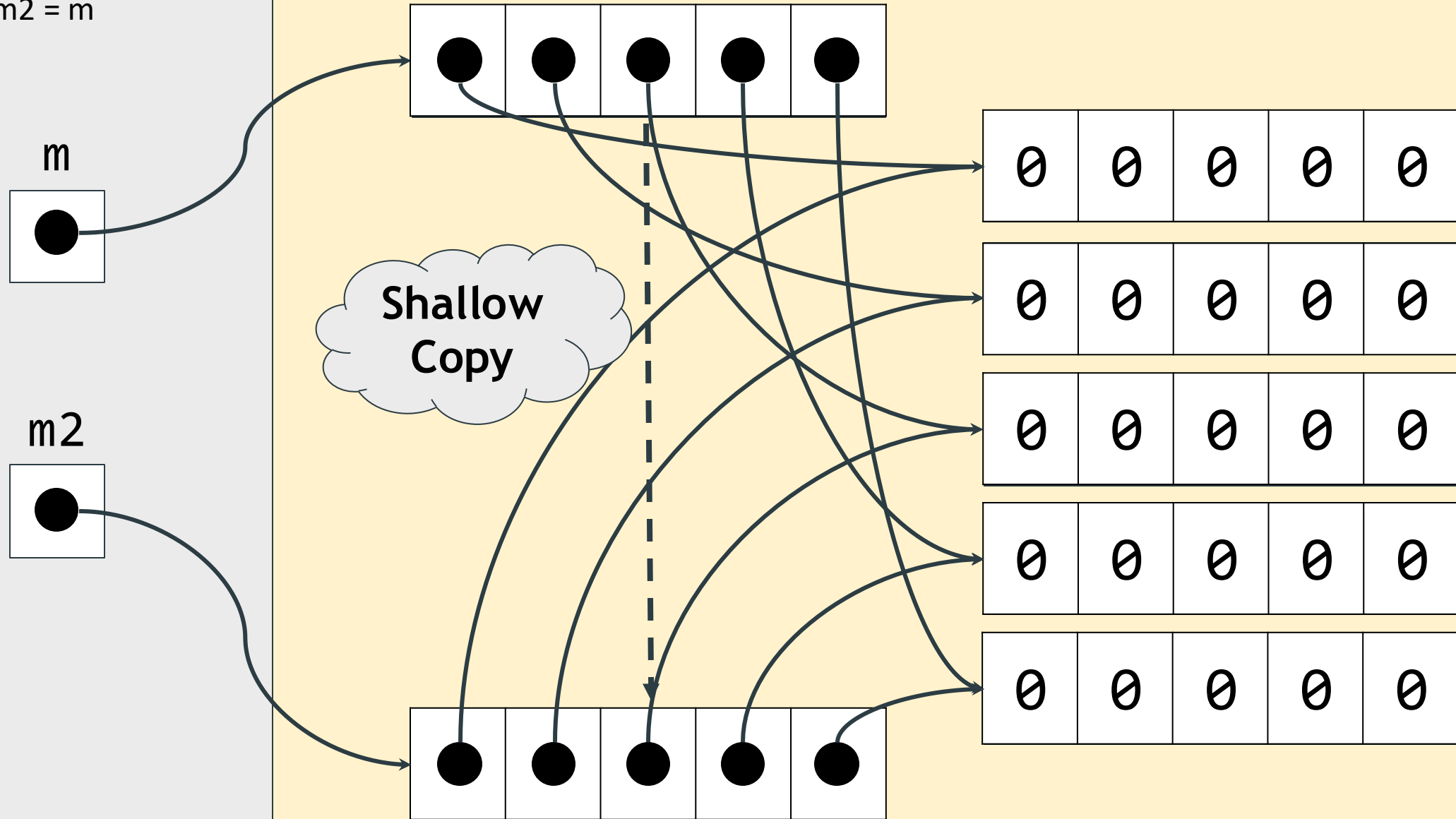


```
m= [[0]*5 for i in range(5)]
```

```
m2 = m
```

Memory

Heap

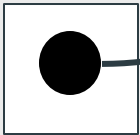



```
m = [[0]*5 for i in range(5)]
```

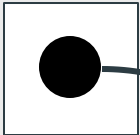
```
m2 = m
```

```
m[2][3] = 1
```

m



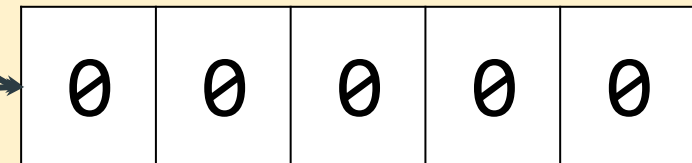
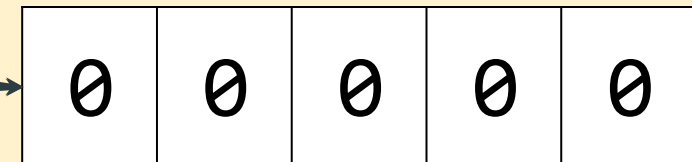
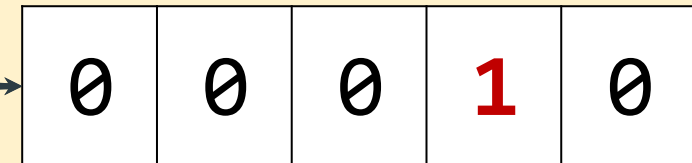
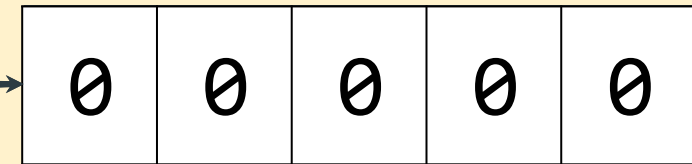
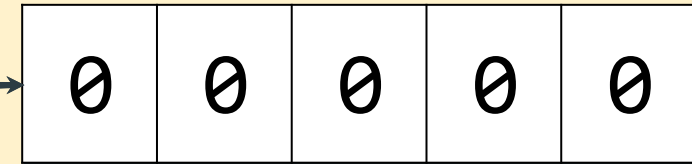
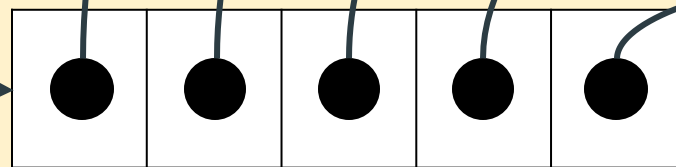
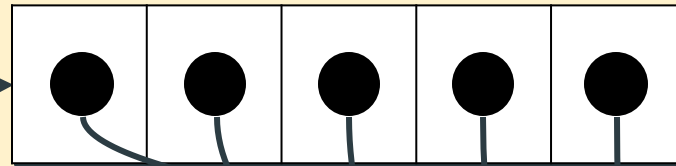
m2



Memory

Shallow Copy

Heap



Memory: TL;DR

- ↵ SHALLOW VS DEEP
- ↵ Modify with new vs in-place
- ↵ Challenge q: can I use `m[:]` to make a deep copy of `m`?

Functions!

Example function

keyword name parameters

```
def multiply(a, b):
```

Function header

```
'''
```

```
Compute the product of two values.
```

docstring

```
Inputs:
```

```
  a, b: the values to be multiplied.
```

```
Returns: the product of the inputs
```

```
'''
```

```
  n = a * b
```


body

```
  return n
```

Coding practice: 1.4.1

Function Call Control Flow

Calling a function alters the control flow of a program.



```
1  def multiply(a, b):
2      print("Start of multiply(a, b) function")
3      rv = a * b
4      print("End of multiply(a, b) function")
5      return rv
6
7  def main():
8      x = 5
9      y = 4
10     print("calling multiply(x, y)...")
11     z = multiply(x, y)
12     print("Returned from multiply(x, y)")
13     print("The value of z is", z)
```

return: specify the value to be returned to the caller and to transfer control back to the call site.

Testing: Zero, One, Many

- ↳ What does this mean?
- ↳ Why / how might it be meaningful?

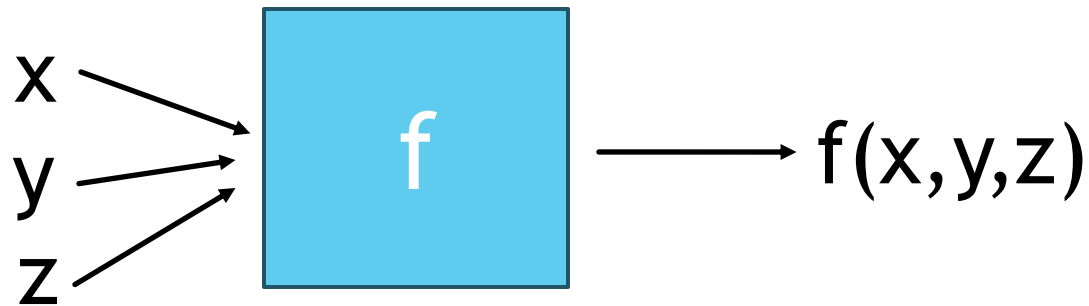
Topics:

- ❑ Shallow/deep copies
- ❑ Introduction to functions
- ❑ Function call control flow
- ❑ **Return statements**
- ❑ Parameters
- ❑ Scoping
- ❑ Abstraction

Mathematical Functions

Mathematical functions: take some values and produce a result.

Python functions: take some parameters as inputs and return some values as outputs.



Return Statements: multiple return statements

A return statement can appear **anywhere** in the function and can appear **multiple times**. Python computes the **return** value and leaves the function *immediately* upon encountering a **return** statement.

```
def absolute(x):  
    '''  
    Compute the absolute value of a number.  
  
    Inputs:  
        n (number): operand  
  
    Returns (number): the magnitude of the input  
    '''  
  
    if x < 0:  
        return -x  
    else:  
        return x
```

```
def is_prime(n):  
    '''  
    Determines whether the input is prime.  
  
    Inputs:  
        n (int): value to be checked  
  
    Returns (boolean): True, if the input is prime and False  
    otherwise  
    '''  
  
    if n == 1:  
        return False  
  
    for i in range(2, n):  
        if n % i == 0:  
            return False  
  
    return True
```

Return Statements: multiple return statements

A return statement can appear **anywhere** in the function and can appear **multiple times**. Python computes the **return** value and leaves the function *immediately* upon encountering a **return** statement.

```
def is_prime(n):  
    """  
    Determines whether the input is prime.  
  
    Inputs:  
        n (int): value to be checked  
  
    Returns (boolean): True, if the input is prime and False  
    otherwise  
    """  
  
    if n == 1:  
        return False  
  
    for i in range(2, n):  
        if n % i == 0:  
            return False  
  
    return True
```

Return Statements: return multiple values

A function can return **multiple values** using a tuple.

```
def bounds(lst):  
    min_val = min(lst)  
    max_val = max(lst)  
    return min_val, max_val
```

Parameters

Call-by-value

```
def multiply(a,b):
```

```
'''
```

Compute the product of two values.

Inputs: a, b: the values to be multiplied.

Returns: the product of the inputs.

```
'''
```

```
n = a * b
```

```
return n
```

Different ways to call the function:

```
multiply(3, 4)
```

```
2 + multiply(3, 4)
```

```
print("2 x 3 =", multiply(2,3))
```

```
x = 3
```

```
y = 5
```

```
multiply(x,y)
```

```
multiply(x-1, y+1)
```

```
multiply(4, multiply(3,2))
```

Parameters with Default Values

- specify a default value for parameters
- come at the end
- can be omitted or overwritten

```
def multiply(a, b = 10):
```

```
'''
```

```
    Compute the product of two values.
```

```
    Inputs: a, b: the values to be multiplied.
```

```
    Returns: the product of the inputs.
```

```
'''
```

```
    n = a * b
```

```
    return n
```

```
multiply(2,3)
```

```
multiply(2)
```

```
multiply(a=2, b=5)
```

Positional and Keyword Parameters

- Map to specific parameters depending on the position in the list of arguments.
- Specify the exact parameter using keyword arguments, position doesn't matter.
- Combining positional and keyword arguments, positional arguments must come first

```
def multiply(a,b):
```

```
'''
```

```
    Compute the product of two values.
```

```
    Inputs: a, b: the values to be multiplied.
```

```
    Returns: the product of the inputs.
```

```
'''
```

```
    n = a * b
```

```
    return n
```

```
multiply(2,3)
```

```
multiply(2, b = 3)
```

```
multiply(a=2, b=5)
```

Note: be careful if you only specify one var

Coding practice: 1.4.6.3

Topics:

- ❑ Introduction to functions
- ❑ Function call control flow
- ❑ Return statements
- ❑ Parameters
- ❑ **Scoping**
- ❑ Abstraction

Scoping

- ▶ Variables are only valid in a specific part of the code
- ▶ **Local** variables: variables defined inside a function, only valid within the function
- ▶ **Global** variables: variables defined outside of a function.
- ▶ When a global variable and a local variable have the same name, the local variable shadows the global variable.

```
c = 5
```

```
def add_c(x, c):
```

```
'''
```

```
Add x and c
```

```
'''
```

```
return x + c
```

```
add_c(3,2)
```

```
def update_c(new_c):
```

```
'''
```

```
Update value of c
```

```
'''
```

```
global c
```

```
c = new_c
```

Coding practice: 1.4.5 / 1.4.7

Functions as an abstraction mechanism

- **Organize** work into separate tasks
 - Every function should have a clear purpose.
- **Reuse** code (and avoid repeated code)
 - Whenever you find yourself cutting-and-pasting a block of code, ask yourself whether it would be better to create a function instead of repeating the block of code.
- **Test** the solution for a task in (relative) isolation

Functions: return vs print

- ↓ Print: displays in console
- ↓ Returns: what 'comes out' from the function

```
def multiply(a, b):  
    """ Print the product of two values. Inputs: a, b:  
    the values to be multiplied. Returns: None """
```

```
    n = a * b  
    print(n)
```

```
rv = multiply(5, 2)  
print("The return value is:", rv)
```

The return value is: None

```
def multiply(a, b):  
    """ Print the product of two values. Inputs: a, b:  
    the values to be multiplied. Returns: None """
```

```
    n = a * b  
    return n
```

```
rv = multiply(5, 2)  
print("The return value is:", rv)
```

The return value is: 10

Note: what happens when you call `rv = multiply(5,2)`?

Functions: how to think about them deeply

- What are you trying to do?
- How can you break it down?
- Where are the complex parts of your code?
- Where do you think you might run into issues?

Simulate a game with dice

- ▶ Going to Boston
 - ▶ <https://www.youtube.com/watch?v=MbBwiAU8SI8>
- ▶ Rules:
 - ▶ roll all three dice and set aside the largest one
 - ▶ roll the remaining two dice and set aside the largest one
 - ▶ roll the remaining die
 - ▶ sum of the above values is the score for the round
 - ▶ The players keep playing until one reaches 500 and wins.

Exercise: Steps

- ▶ In pairs, work through exercise 1.4.4:
 - ▶ Create a file that has multiple functions in it
 - ▶ ****SKETCH INDEPENDENTLY FIRST**** but then ****YES USE THE TEXT!!****
 - ▶ Be sure all is clean and clear
- ▶ Discuss **two innovations** / extensions / ‘twists’ on the game:
 - ▶ One that is difficult for the code as written:
 - ▶ Why is this difficult? Is there a way you could have written your code differently to address this?
 - ▶ One that will be easy to adapt:
 - ▶ Why is this easy? How does the structure of the code lend itself to this?

Work it out: What will you use

- ▶ Sketch your structure:
 - ▶ Big pieces first (main method)
 - ▶ Small pieces next (helper functions)
 - ▶ Details last (doc strings, etc)

Code ideas (1.4.4)

```
def play_round():
```

```
    """ Play a round of the game Going to Boston using three dice. Inputs: none Return (int): score earned """
```

```
    NUM_SIDES = 6  
    score = 0
```

```
    # roll 3 dice, choose largest
```

```
    die1 = random.randint(1, NUM_SIDES)
```

```
    die2 = random.randint(1, NUM_SIDES)
```

```
    die3 = random.randint(1, NUM_SIDES)
```

```
    largest = max(die1, max(die1, die2))
```

```
    score += largest
```

```
    # roll 2 dice, choose largest
```

```
    die1 = random.randint(1, NUM_SIDES)
```

```
    die2 = random.randint(1, NUM_SIDES)
```

```
    largest = max(die1, die2)  
    score += largest
```

```
    # roll 1 die, choose largest
```

```
    largest = random.randint(1, NUM_SIDES)
```

```
    score += largest
```

```
    return score
```

Code ideas (1.4.4)

```
def play_round():
```

```
    """ Play a round of the game Going to Boston using three dice. Inputs: none Return (int): score earned """
```

```
    NUM_SIDES = 6 score = 0
```

```
    # roll 3 dice, choose largest
```

```
    die1 = random.randint(1, NUM_SIDES)
```

```
    die2 = random.randint(1, NUM_SIDES)
```

```
    die3 = random.randint(1, NUM_SIDES)
```

```
    largest = max(die1, max(die1, die2))
```

```
    score += largest
```

```
    # roll 2 dice, choose largest
```

```
    die1 = random.randint(1, NUM_SIDES)
```

```
    die2 = random.randint(1, NUM_SIDES)
```

```
    largest = max(die1, die2) score += largest
```

```
    # roll 1 die, choose largest
```

```
    largest = random.randint(1, NUM_SIDES)
```

```
    score += largest
```

```
    return score
```

We're calling random.randint and taking the largest - maybe we want to specify a number of rolls and a max fn?

```
def get_largest_roll(num_dice)
```

```
def play_round()
```


TESTING (general steps)

- ▶ DOWNLOAD ALL FILES TO GOOD PLACE
- ▶ In terminal (either VS code or straight terminal), navigate to your folder where the files are.
- ▶ Call ipython
- ▶ Load autoreload

TESTING

- ▶ In terminal (either VS code or straight terminal), navigate to your folder where the files are.
- ▶ Call ipython
- ▶ Autoreload:

```
In [1]: %load_ext autoreload
```

- ▶ Import the 'base name' of your file (e.g. for boston.py, **import boston**)

```
In [2]: %autoreload 2
```

- ▶ Call the function using **name.function()**

```
In [3]: import boston
```

```
In [4]: boston.play_round_generalized(3)
```

```
Out[4]: 16
```

Innovations?

- ▶ What changes did you make?

Test files

- ▶ BE IN YOUR CONDA ENVIRONMENT (e.g. 111)
- ▶ `conda install pytest`
- ▶ `py.test <options> <method> test_assign.py`
 - ▶ Options: <https://docs.pytest.org/en/6.2.x/usage.html>

Successful code

```
cachedir: .pytest_cache
rootdir: /Users/jeanclipperton/Library/CloudStorage/Box-Box/Teaching/30111/mac30111/class_examples/SE2_sample_example
configfile: pytest.ini
plugins: django-4.9.0
collected 40 items

test_se2.py::test_peep_1 PASSED [ 2%]
test_se2.py::test_peep_2 PASSED [ 5%]
test_se2.py::test_peep_3 PASSED [ 7%]
test_se2.py::test_peep_4 PASSED [ 10%]
test_se2.py::test_peep_5 PASSED [ 12%]
test_se2.py::test_peep_6 PASSED [ 15%]
test_se2.py::test_has_more_1 PASSED [ 17%]
test_se2.py::test_has_more_2 PASSED [ 20%]
test_se2.py::test_has_more_3 PASSED [ 22%]
test_se2.py::test_has_more_4 PASSED [ 25%]
test_se2.py::test_has_more_5 PASSED [ 27%]
test_se2.py::test_has_more_6 PASSED [ 30%]
test_se2.py::test_has_more_7 PASSED [ 32%]
test_se2.py::test_has_more_8 PASSED [ 35%]
test_se2.py::test_make_star_strings_1 PASSED [ 37%]
test_se2.py::test_make_star_strings_2 PASSED [ 40%]
test_se2.py::test_make_star_strings_3 PASSED [ 42%]
test_se2.py::test_make_star_strings_4 PASSED [ 45%]
test_se2.py::test_make_star_strings_5 PASSED [ 47%]
test_se2.py::test_make_star_strings_6 PASSED [ 50%]
test_se2.py::test_replace_1 PASSED [ 52%]
test_se2.py::test_replace_2 PASSED [ 55%]
test_se2.py::test_replace_3 PASSED [ 57%]
test_se2.py::test_replace_4 PASSED [ 60%]
test_se2.py::test_replace_5 PASSED [ 62%]
test_se2.py::test_replace_6 PASSED [ 65%]
test_se2.py::test_replace_7 PASSED [ 67%]
test_se2.py::test_replace_8 PASSED [ 70%]
test_se2.py::test_rows_and_columns_contain_1 PASSED [ 72%]
test_se2.py::test_rows_and_columns_contain_2 PASSED [ 75%]
test_se2.py::test_rows_and_columns_contain_3 PASSED [ 77%]
test_se2.py::test_rows_and_columns_contain_4 PASSED [ 80%]
test_se2.py::test_rows_and_columns_contain_5 PASSED [ 82%]
test_se2.py::test_rows_and_columns_contain_6 PASSED [ 85%]
test_se2.py::test_rows_and_columns_contain_7 PASSED [ 87%]
test_se2.py::test_rows_and_columns_contain_8 PASSED [ 90%]
test_se2.py::test_rows_and_columns_contain_9 PASSED [ 92%]
test_se2.py::test_rows_and_columns_contain_10 PASSED [ 95%]
test_se2.py::test_rows_and_columns_contain_11 PASSED [ 97%]
test_se2.py::test_rows_and_columns_contain_12 PASSED [100%]

===== 40 passed in 0.05s =====
(111) jeanclipperton@MADPSS-250041 SE2_sample_example %
```

Recap

- ▶ Functions **underpin** a **LOT** of what we'll be doing
- ▶ **Function call control flow**: can move around in the file (potentially never call something!!)
- ▶ **Return statements**: can put where needed, generally good to have
- ▶ **Parameters**: what values the function takes
- ▶ **Scoping**: local vs global variables
- ▶ **Anatomy**: function and parameters, doc string, body (aka code), return statement