AAE 875 – Fundamentals of Object Oriented Programming and Data Analytics

Cornelia Ilin, PhD

Department of Ag & Applied Economics

UW-Madison

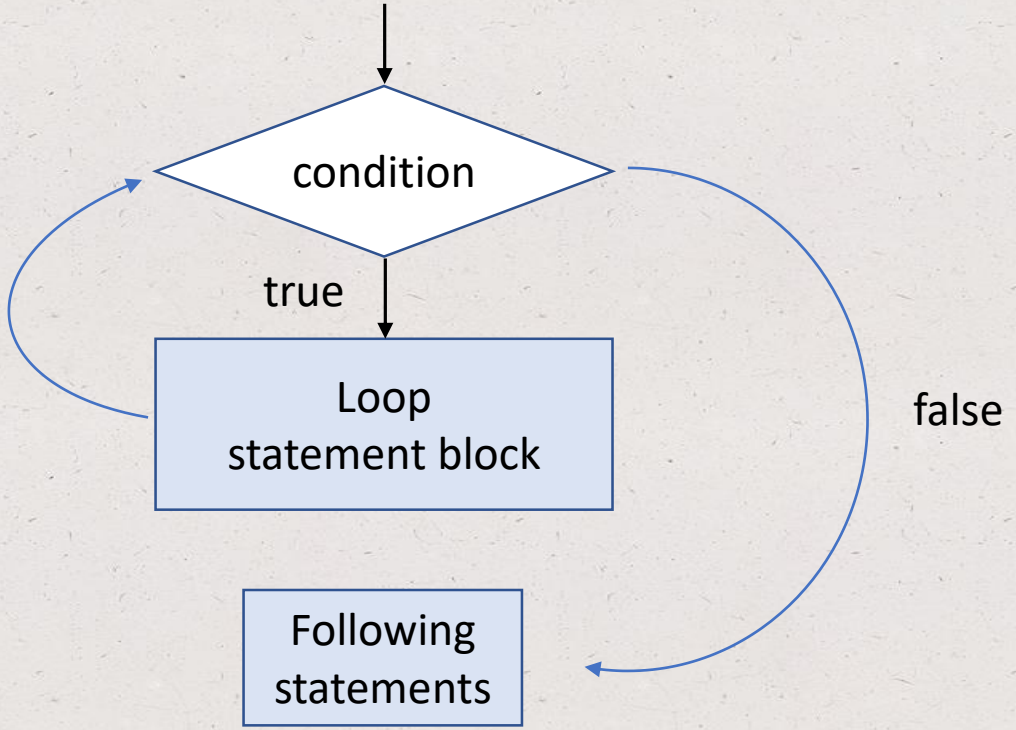
Week 2 - Summer 2019

Chapter 5: Loops

- While loops
- For loops
- Break and continue statements
- Nested loops

While loops

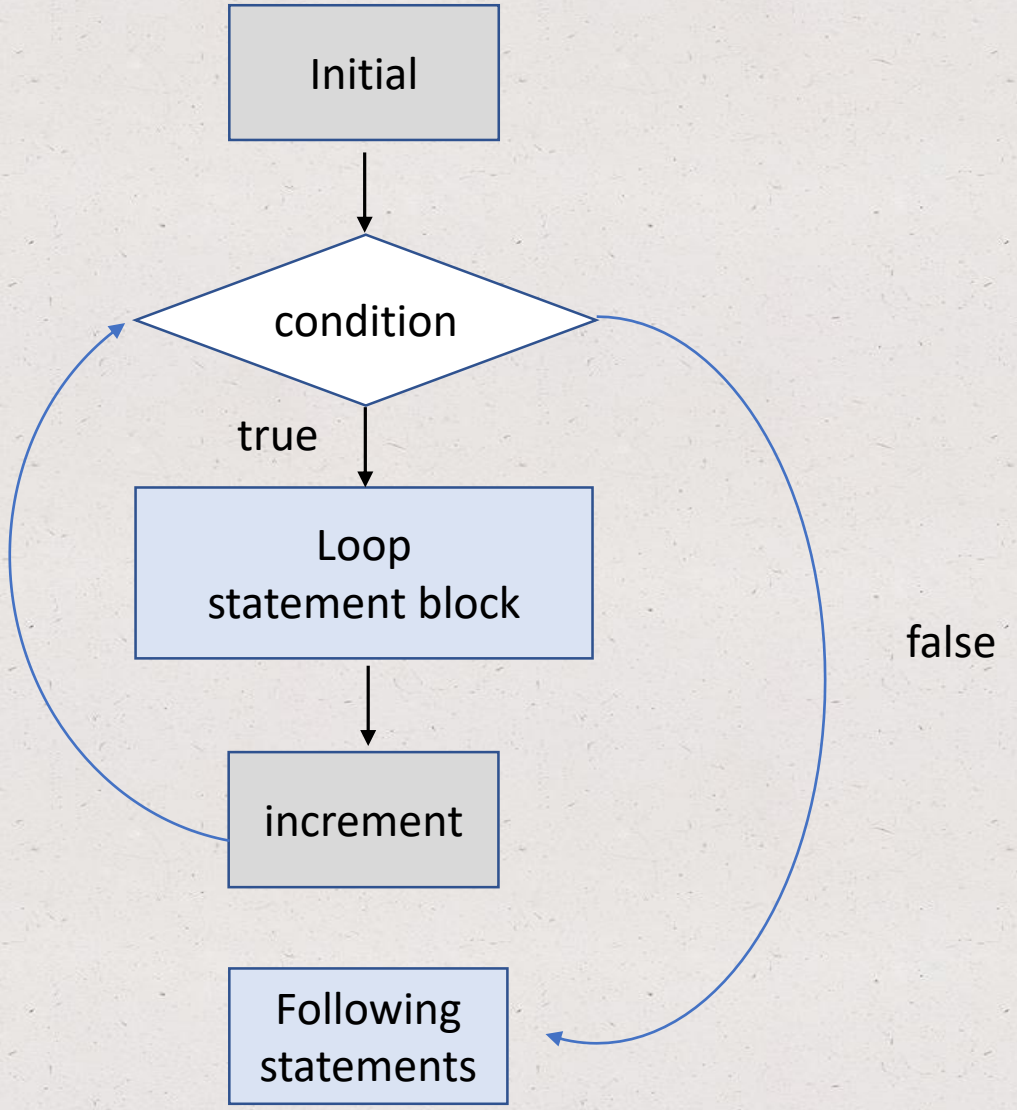
```
while condition:
    loop statement 1
    .
    .
    loop statement N
```



- Repeatedly executes the block of code (**loop body**) as long as the loop condition is True
- Each execution of the loop body is called an **iteration**
- Do not write infinite loops (!!): it's a loop that will always execute because the condition is always True

While loops

```
initial
while condition:
    loop statement 1
    .
    .
    loop statement N
    increment
```



- When a loop should iterate for a specific number of times, add an initial variable (outside the loop body) and an increment (in the loop body)
- Forgetting the increment statement will result in an infinite loop!
- **Note:** incrementation is performed manually using the increment statement!

Top Hat Question # 1

What is the output?

```
iter = 0

while iter <= 3:
    if(iter + 1 == 1):
        print('The', iter+1, '\'st iteration')
    elif(iter +1 == 2):
        print('The', iter+1, '\'nd iteration')
    elif(iter + 1 == 3):
        print('The', iter+1, '\'rd iteration')
    else:
        print('The', iter+1, '\'th iteration')
    iter += 1

print('goodbye!')
```

Top Hat Question # 2

What is the output?

```
iter = 0

while iter <= 3:
    if(iter + 1 == 1):
        print('The', iter+1, '\'st iteration')
    elif(iter +1 == 2):
        print('The', iter+1, '\'nd iteration')
    elif(iter + 1 == 3):
        print('The', iter+1, '\'rd iteration')
    else:
        print('The', iter+1, '\'th iteration')


print('the program has executed', iter, 'iterations')
```

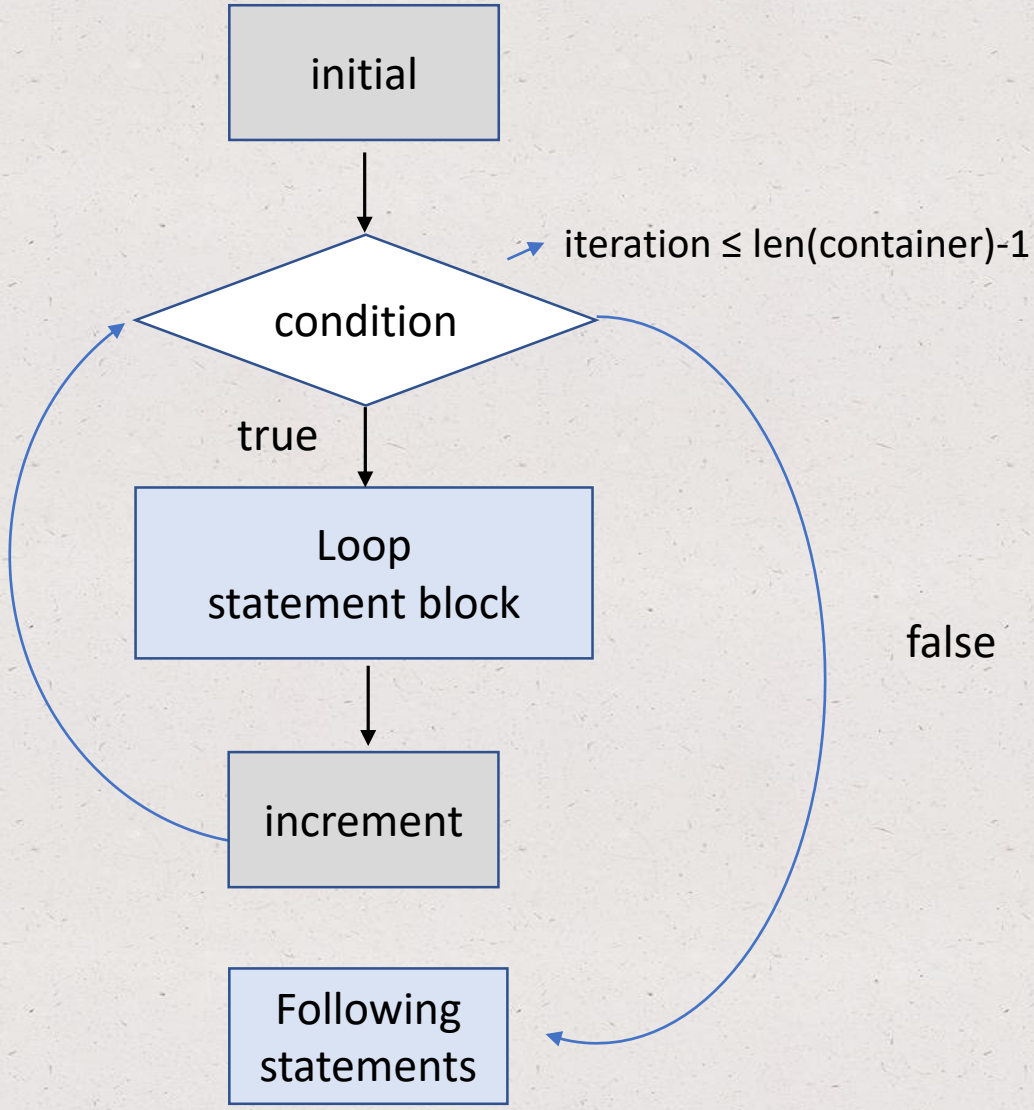
Top Hat Question # 3

What is the last number output?

```
iter = 0

while iter != 5:
    print(iter, end = ' ')
    iter -= 2

print('goodbye!')
```

For loops

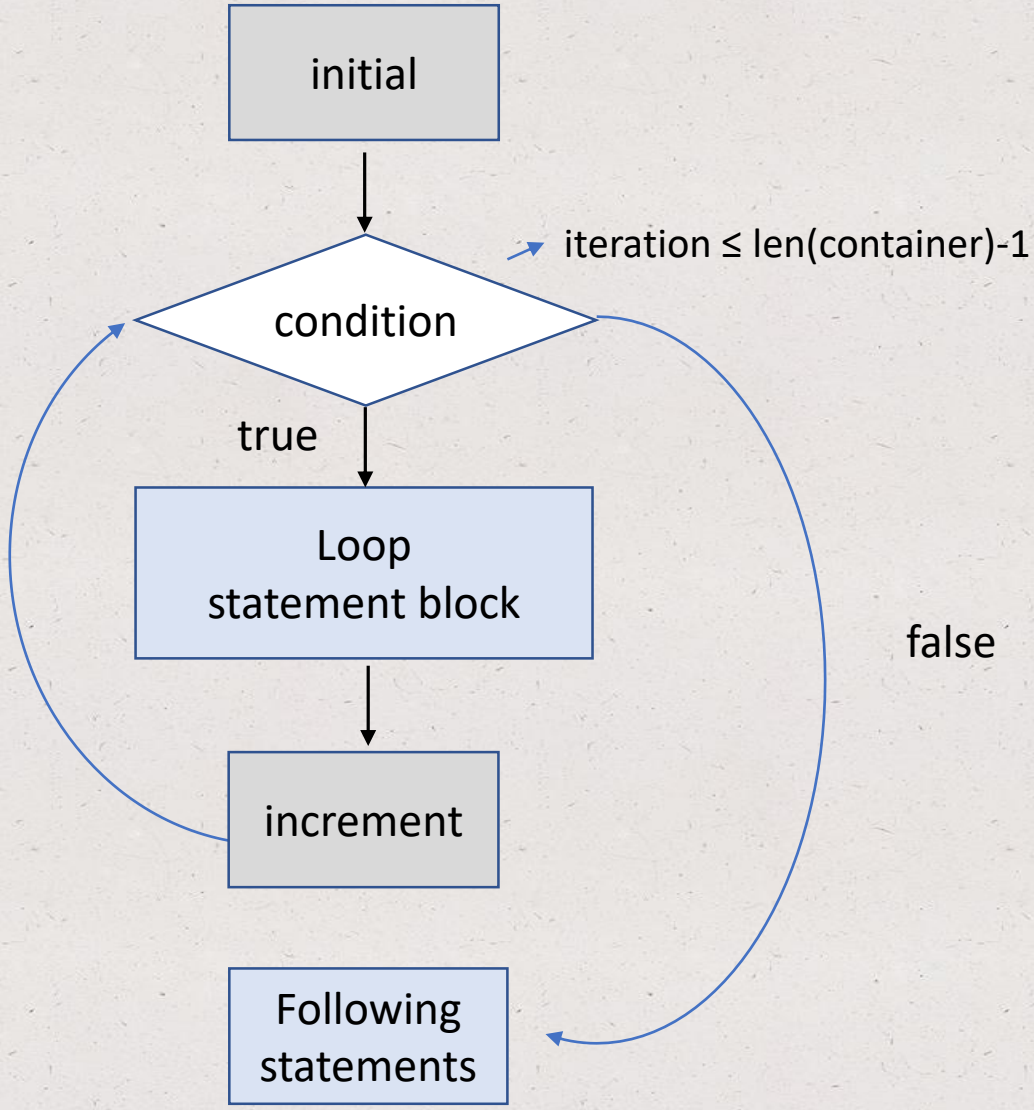
```
for variable in container:
    loop statement 1
    .
    .
    loop statement N
```



- A for loop statement loops over each element in a container one at a time
- With each iteration, the next element in the container is assigned to a variable
- The container in the for loop statement can be a list, tuple, string, or dictionary

For loops

```
for variable in container:
    loop statement 1
    .
    .
    loop statement N
```



- **Note 1**: no need to define the initial statement! Initial value automatically starts at 0 (i.e. index 0 in the container)
- **Note 2**: no need to define an increment statement! The loop automatically moves to the next increment until all elements in the container have been looped over.

For loops: Example 1 – tracing

```
aae_class = ['AAE 635', 'AAE 636', 'AAE 875']

for name in aae_class:
    print(name)
```

- What is the name of the container?
- What is the data type of the container?
- What is the value of variable *name* at iteration 1?

For loops: Example 1 – tracing

```
aae_class = ['AAE 635', 'AAE 636', 'AAE 875']

for name in aae_class:
    print(name)
```

Iteration	Index in container	Value of 'name'	Output
0	0	'AAE 635'	AAE 635

- What is the name of the container?
- What is the data type of the container?
- What is the value of variable *name* at iteration 1?

For loops: Example 1 – tracing

```
aae_class = ['AAE 635', 'AAE 636', 'AAE 875']

for name in aae_class:
    print(name)
```

Iteration	Index in container	Value of 'name'	Output
0	0	'AAE 635'	AAE 635
1	1	'AAE 636'	AAE 635 AAE 636

- What is the name of the container?
- What is the data type of the container?
- What is the value of variable *name* at iteration 1?

For loops: Example 1 – tracing

```
aae_class = ['AAE 635', 'AAE 636', 'AAE 875']

for name in aae_class:
    print(name)
```

- What is the name of the container?
- What is the data type of the container?
- What is the value of variable *name* at iteration 1?

Iteration	Index in container	Value of 'name'	Output
0	0	'AAE 635'	AAE 635
1	1	'AAE 636'	AAE 635 AAE 636
2	2	'AAE 875'	AAE 635 AAE 636 AAE 875

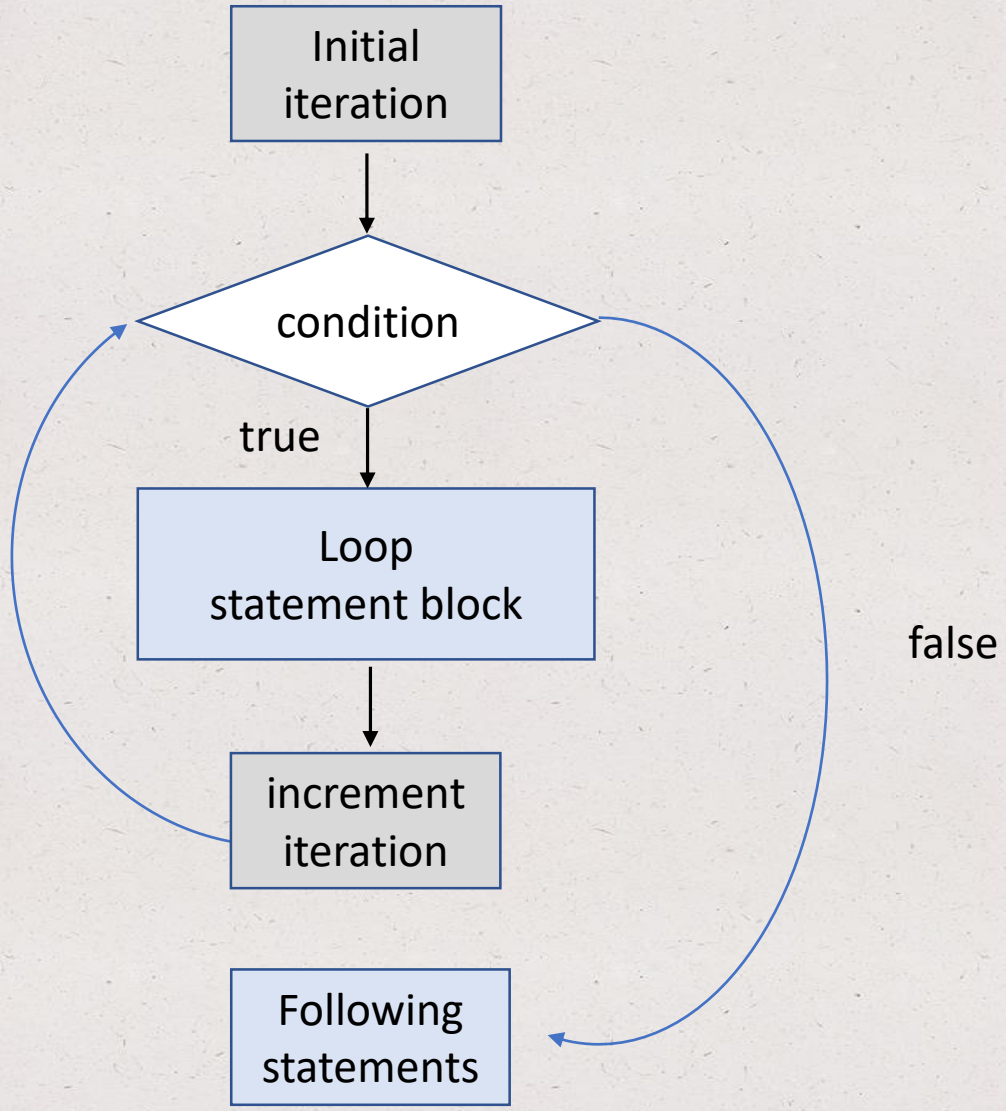
For loops

```
for variable in reversed(container):
    loop statement 1
    .
    .
    loop statement N
```

- One can use the built-in reversed() function to read the elements in the container in reverse order
- The loop starts from the last elements and ends with the first element

For loops and range()

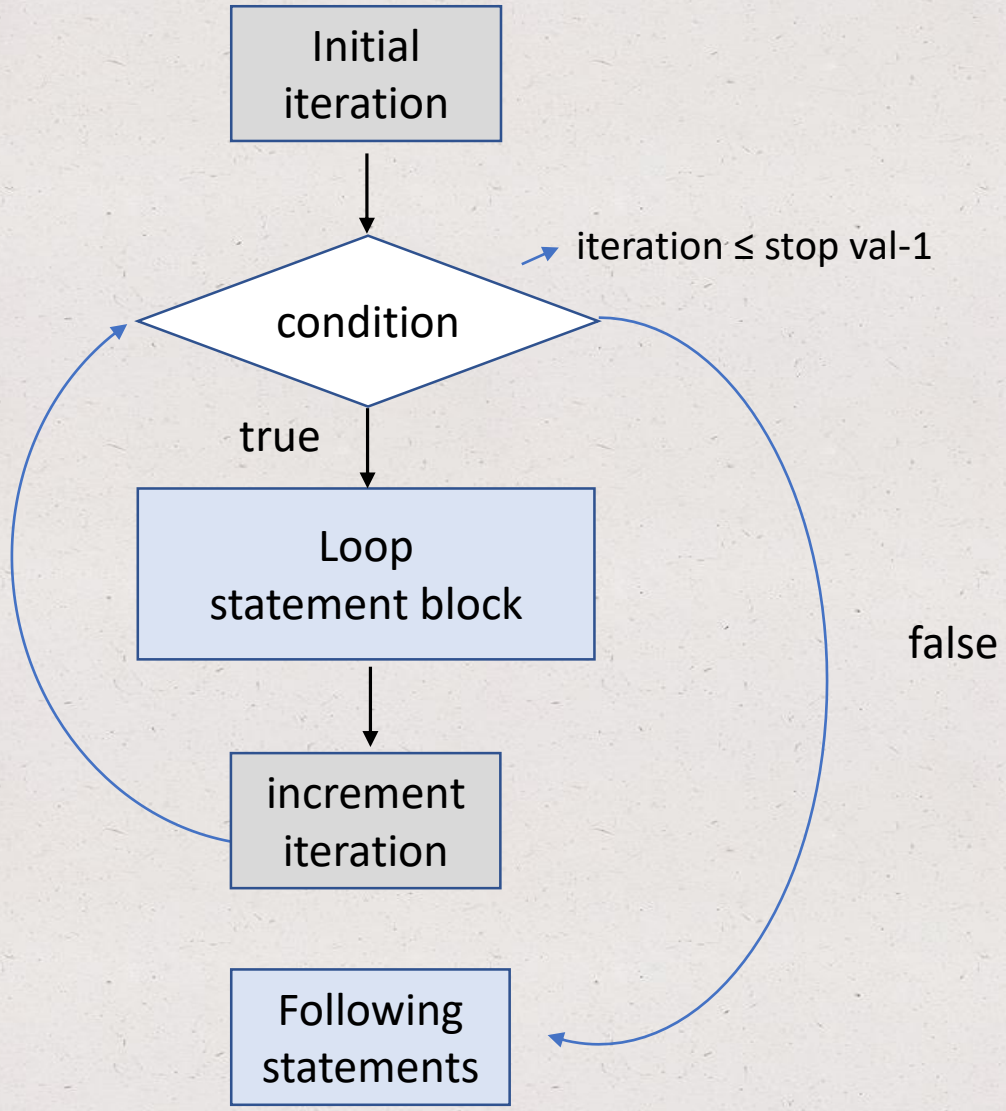
```
for i in range():
    loop statement 1
    .
    .
    loop statement N
```



- *While loops* are used to count for a specific number of iterations
- *For loops* are used to iterate over *all elements* of a container
- However, the built-in **range()** function allows for counting in for loops as well

For loops and range()

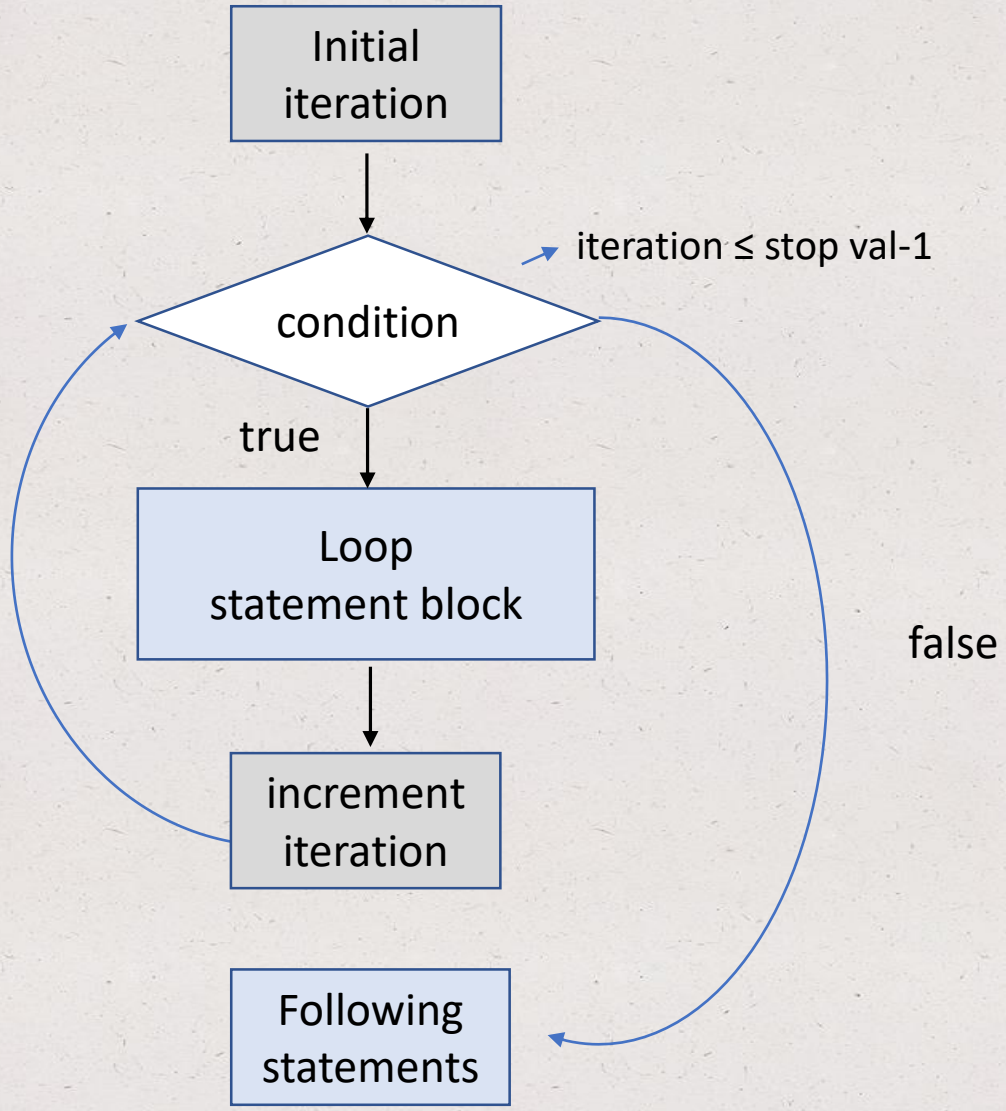
```
for i in range():
    loop statement 1
    .
    .
    loop statement N
```



- **range(stop)** -> the default start value is 0
- **range(start, stop [, step])** -> so range() can take up to 3 arguments
- **Note:** the stop value is not included in the generated sequence

For loops and range()

```
for i in range():
    loop statement 1
    .
    .
    loop statement N
```



- When range() is called a range type object is created
- The range type is an immutable sequence type
- Usually used as part of a *for loop* statement

Top Hat Question # 4

What is the output?

```
for i in range(10, 20, 2):
    print(i)
```

Top Hat Question # 5

What is the output?

```
for i in range(0, 20, 2):
    print(i)
```

While vs. For loops

```
for i in range():
    loop statement 1
    .
    .
    loop statement N
```

```
initial
while (condition):
    loop statement 1
    .
    .
    loop statement N
    increment
```

- Both while and for loops care be used to count a specific number of loop iterations
- A for loop combined with range() is preferred over while loops
- While loops can become stuck in an infinite loop if one forgets the increment statement

While vs. For loops

```
for i in range():
    loop statement 1
    .
    .
    loop statement N
```

```
initial
while (condition):
    loop statement 1
    .
    .
    loop statement N
    increment
```

- Both while and for loops care be used to count a specific number of loop iterations
- A for loop combined with range() is preferred over while loops
- While loops can become stuck in an infinite loop if one forgets the increment statement
- **How do we pick a loop?**

While vs. For loops

- While loops: when the number of iterations is not known in advance (e.g. depends on user input)
- For loops: when the number of iterations is known in advance (e.g. summing from *a* to *b*)

Loop statements

- break – exits the loop
- continue – jumps to the next iteration

Loop statements: Example 1 - tracing

```
for i in range(10):
    if i > 7:
        break
    if i % 2 == 0:
       print(i, end = ' ')
```

i	Output
0	0

Loop statements: Example 1 - tracing

```
for i in range(10):
    if i > 7:
        break
    if i % 2 == 0:
       print(i, end = ' ')
```

i	Output
0	0
1	

Loop statements: Example 1 - tracing

```
for i in range(10):
    if i > 7:
        break
    if i % 2 == 0:
       print(i, end = ' ')
```

i	Output
0	0
1	
2	0 2

Loop statements: Example 1 - tracing

```
for i in range(10):
    if i > 7:
        break
    if i % 2 == 0:
       print(i, end = ' ')
```

i	Output
0	0
1	
2	0 2
3	

Loop statements: Example 1 - tracing

```
for i in range(10):
    if i > 7:
        break
    if i % 2 == 0:
       print(i, end = ' ')
```

i	Output
0	0
1	
2	0 2
3	
4	0 2 4

Loop statements: Example 1 - tracing

```
for i in range(10):
    if i > 7:
        break
    if i % 2 == 0:
       print(i, end = ' ')
```

i	Output
0	0
1	
2	0 2
3	
4	0 2 4
5	

Loop statements: Example 1 - tracing

```
for i in range(10):
    if i > 7:
        break
    if i % 2 == 0:
       print(i, end = ' ')
```

i	Output
0	0
1	
2	0 2
3	
4	0 2 4
5	
6	0 2 4 6

Loop statements: Example 1 - tracing

```
for i in range(10):
    if i > 7:
        break
    if i % 2 == 0:
       print(i, end = ' ')
```

i	Output
0	0
1	
2	0 2
3	
4	0 2 4
5	
6	0 2 4 6
7	

Loop statements: Example 1 - tracing

```
for i in range(10):
    if i > 7:
        break
    if i % 2 == 0:
       print(i, end = ' ')
```

i	Output
0	0
1	
2	0 2
3	
4	0 2 4
5	
6	0 2 4 6
7	
8	

Top Hat Question # 6

What is the output?

```
for i in range(10):
    if i > 7:
        break
    if i % 2 == 0:
        continue
        print(i, end = ' ')
```

Nested loops

```
for n in range(num_col):
    for m in range(num_rows):
        inner loop statement 1
.
.
        inner loop statement N
outer loop statement 1
.
.
outer loop statement N
```

outer loop

inner loop

- Used for multidimensional problems
- Allow for repetitive tasks for every iteration
- **Example:** usually data comes in spreadsheet form; we should be able to read each line and column.

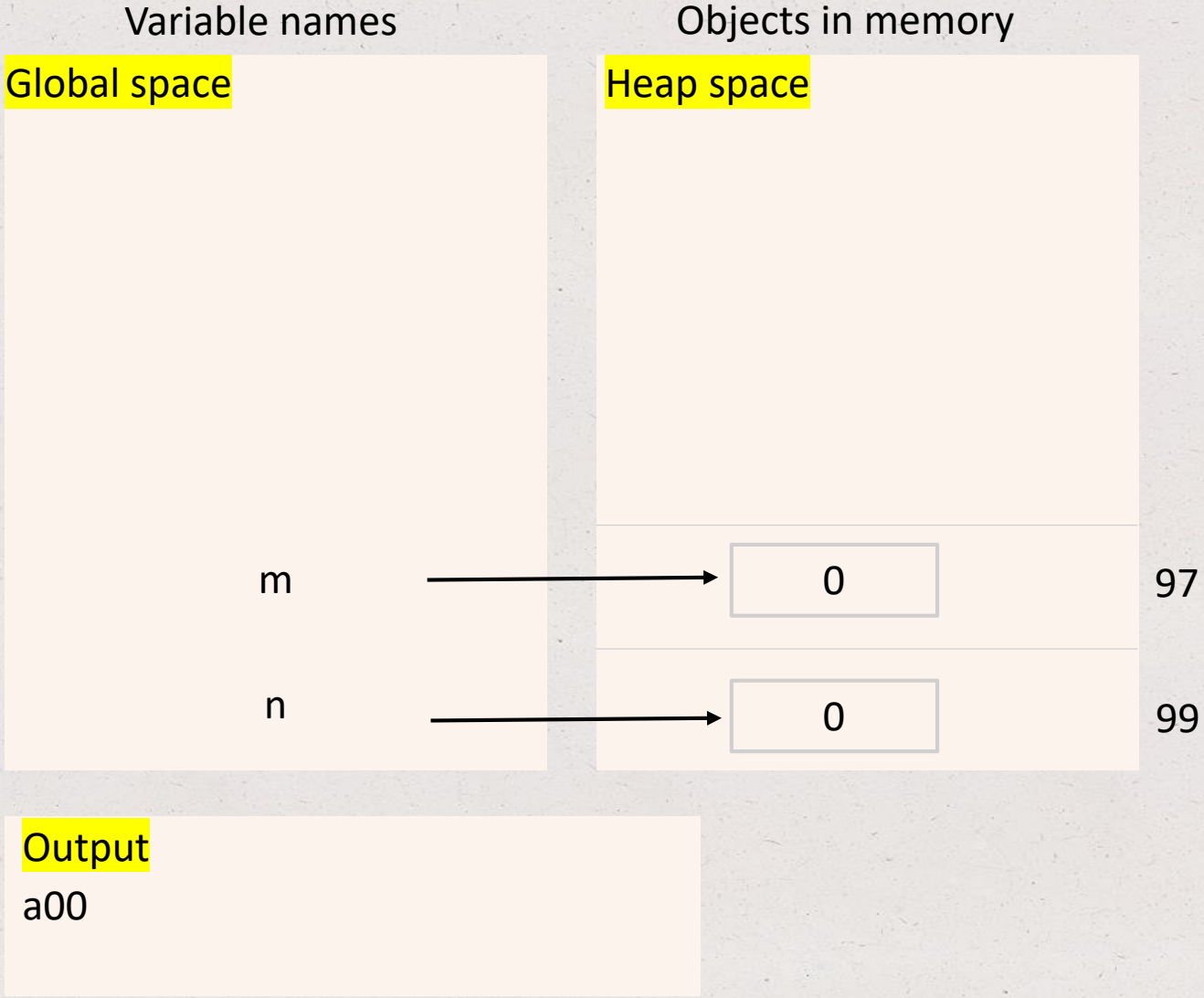
Nested loops: Example 1 - tracing

```
# print a 2 x 3 matrix
# desired output is:
# a00 a01 a02
# a10 a11 a12

for n in range(2):
    for m in range(3):
        print('a', end = '')
        print(n, end = '')
        print(m, end = ' ')
    print()
```

Iteration	n(column)	m(row)	Output
0	0	0	a00
1	0	1	a00 a01
2	0	2	a00 a01 a02
3	1	0	a00 a01 a02 a10
4	1	1	a00 a01 a02 a10 a11
5	1	2	a00 a01 a02 a10 a11 a12

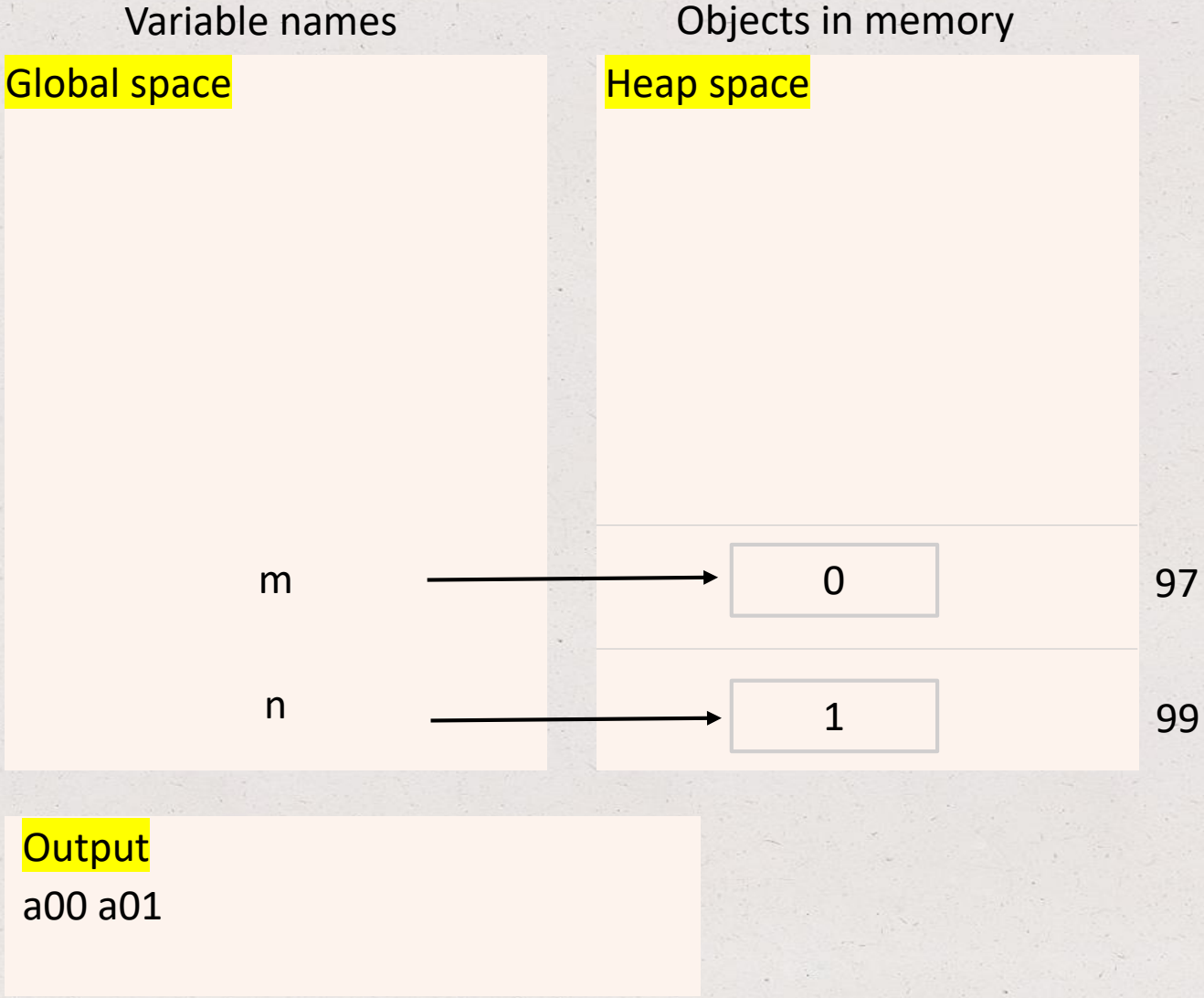
Nested loops: Example 1 – memory rep

```
# print a 2 x 3 matrix
# desired output is:
# a00 a01 a02
# a10 a11 a12

for n in range(2):
    for m in range(3):
        print('a', end = '')
        print(n, end = '')
        print(m, end = ' ')
    print()
```



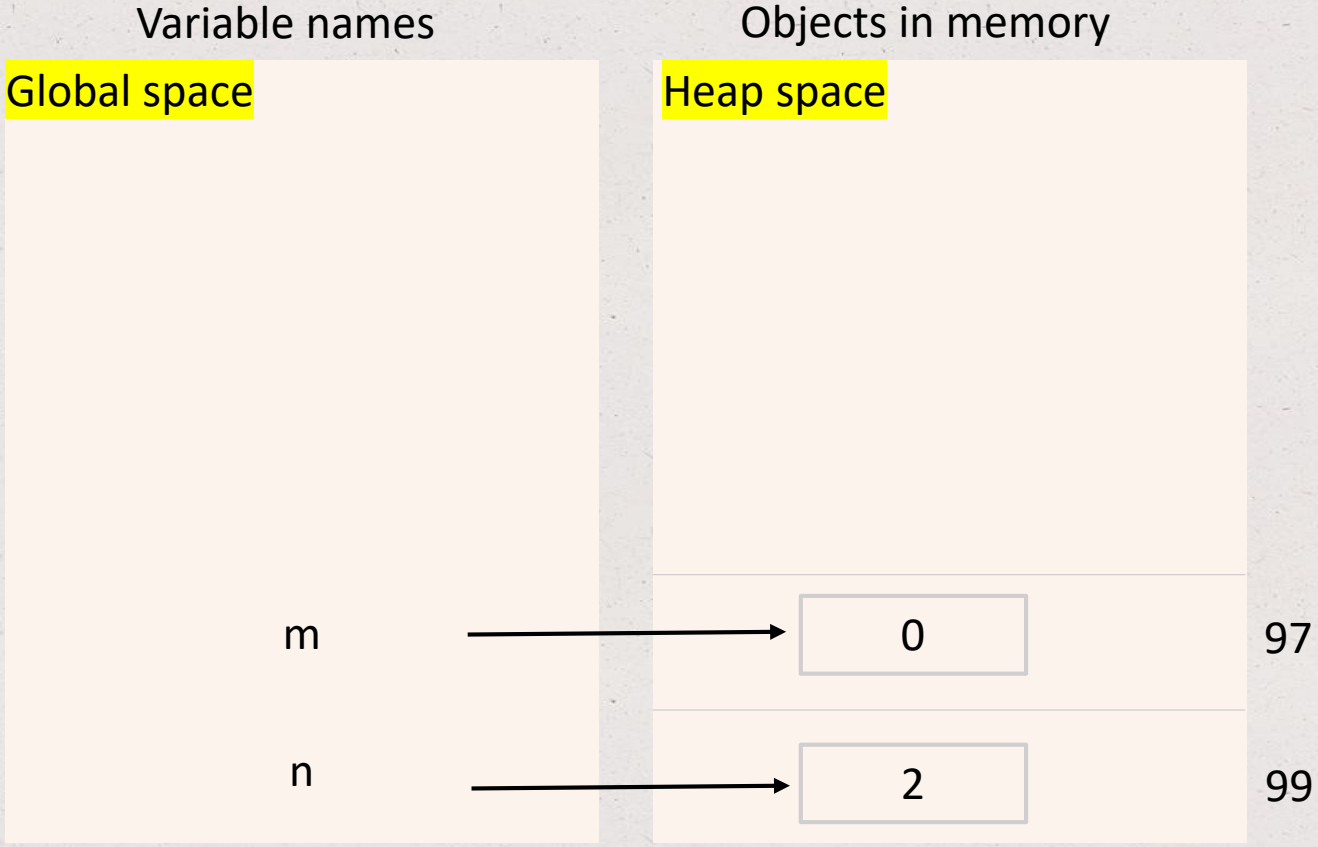
Nested loops: Example 1 – memory rep

```
# print a 2 x 3 matrix
# desired output it:
# a00 a01 a02
# a10 a11 a12

for n in range(2):
    for m in range(3):
        print('a', end = '')
        print(n, end = '')
        print(m, end = ' ')
    print()
```



Nested loops: Example 1 – memory rep

```
# print a 2 x 3 matrix
# desired output it:
# a00 a01 a02
# a10 a11 a12

for n in range(2):
    for m in range(3):
        print('a', end = '')
        print(n, end = '')
        print(m, end = ' ')
    print()
```

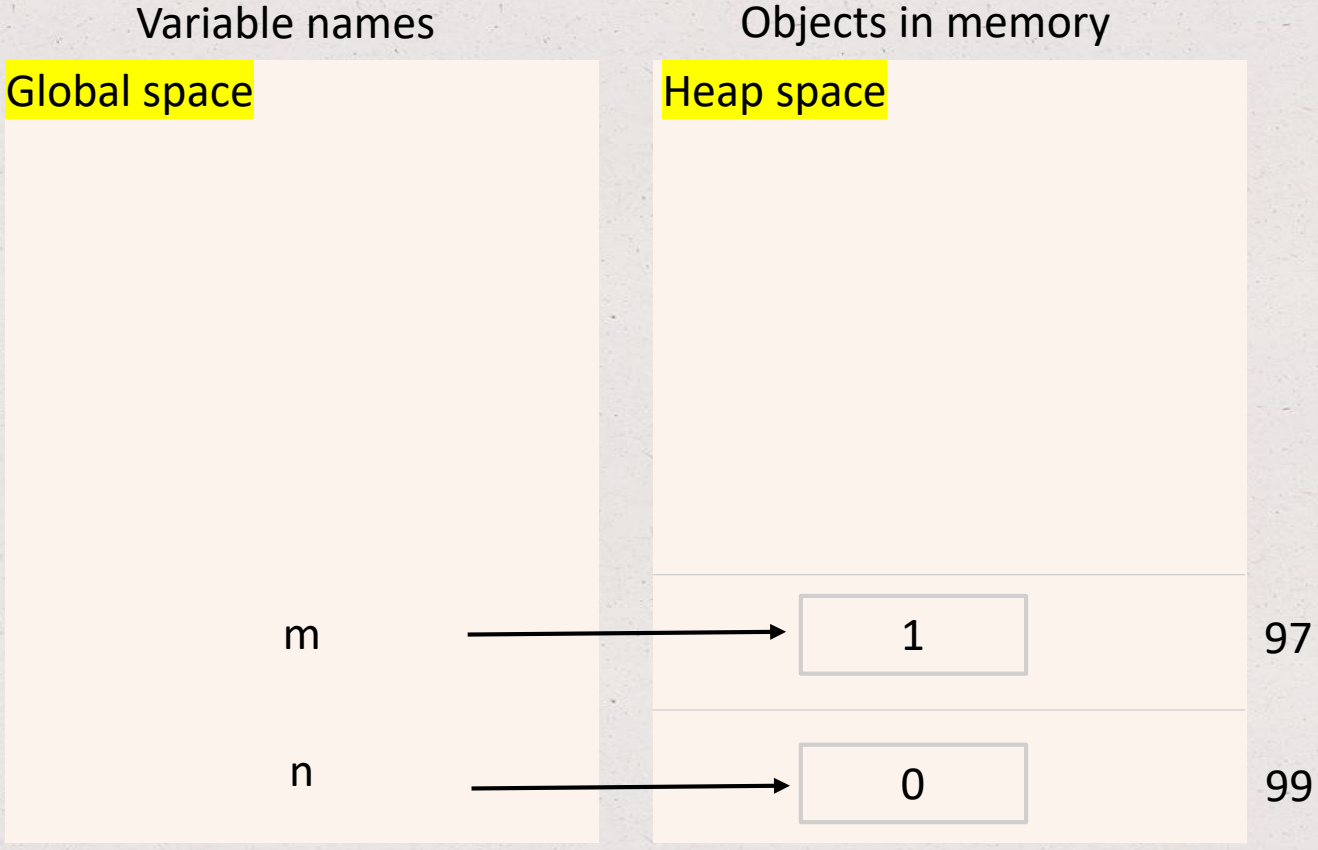


Output

a00 a01 a02

Nested loops: Example 1 – memory rep

```
# print a 2 x 3 matrix
# desired output it:
# a00 a01 a02
# a10 a11 a12

for n in range(2):
    for m in range(3):
        print('a', end = '')
        print(n, end = '')
        print(m, end = ' ')
    print()
```

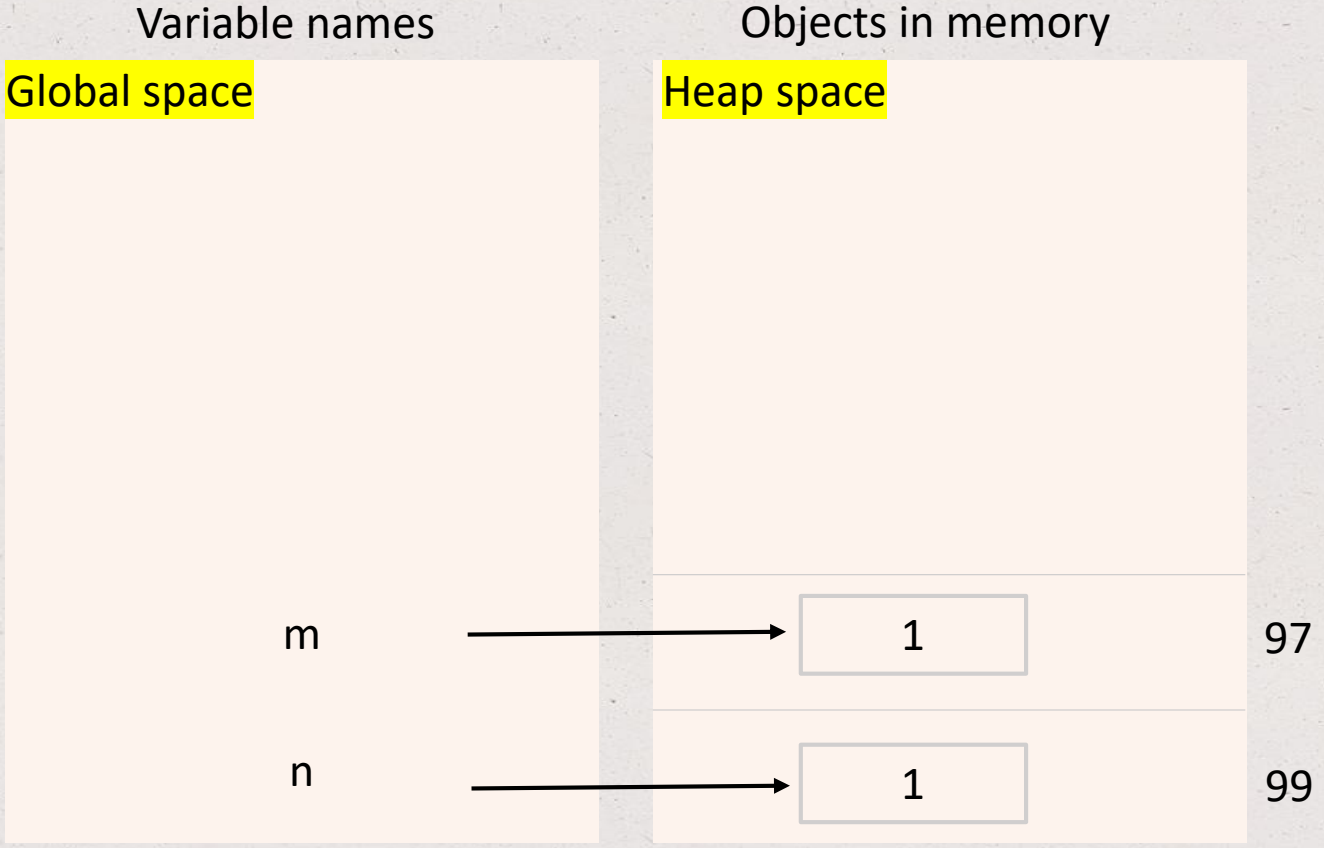


Output

a00 a01 a02
a10

Nested loops: Example 1 – memory rep

```
# print a 2 x 3 matrix
# desired output it:
# a00 a01 a02
# a10 a11 a12

for n in range(2):
    for m in range(3):
        print('a', end = '')
        print(n, end = '')
        print(m, end = ' ')
    print()
```

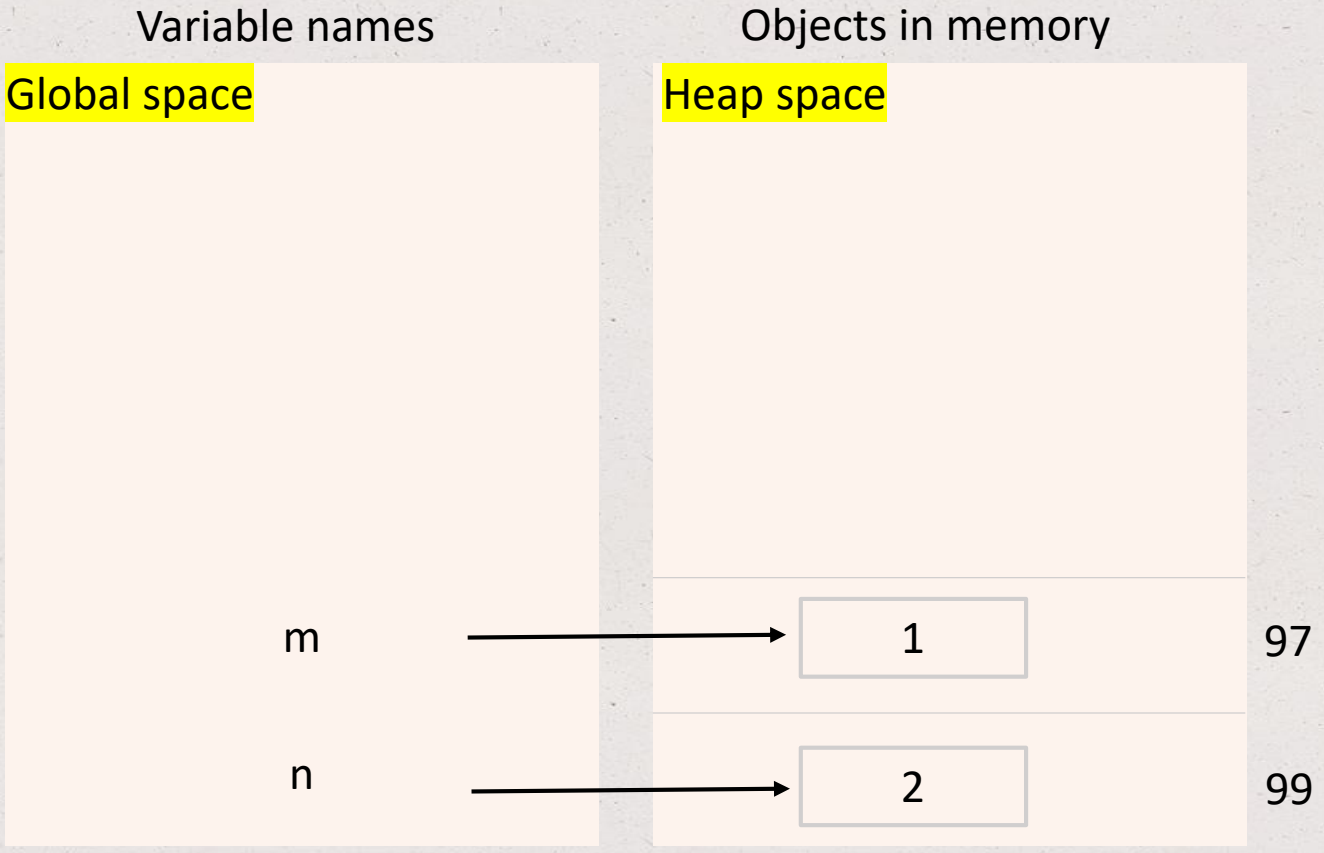


Output

a00 a01 a02
a10 a11

Nested loops: Example 1 – memory rep

```
# print a 2 x 3 matrix
# desired output it:
# a00 a01 a02
# a10 a11 a12

for n in range(2):
    for m in range(3):
        print('a', end = '')
        print(n, end = '')
        print(m, end = ' ')
    print()
```

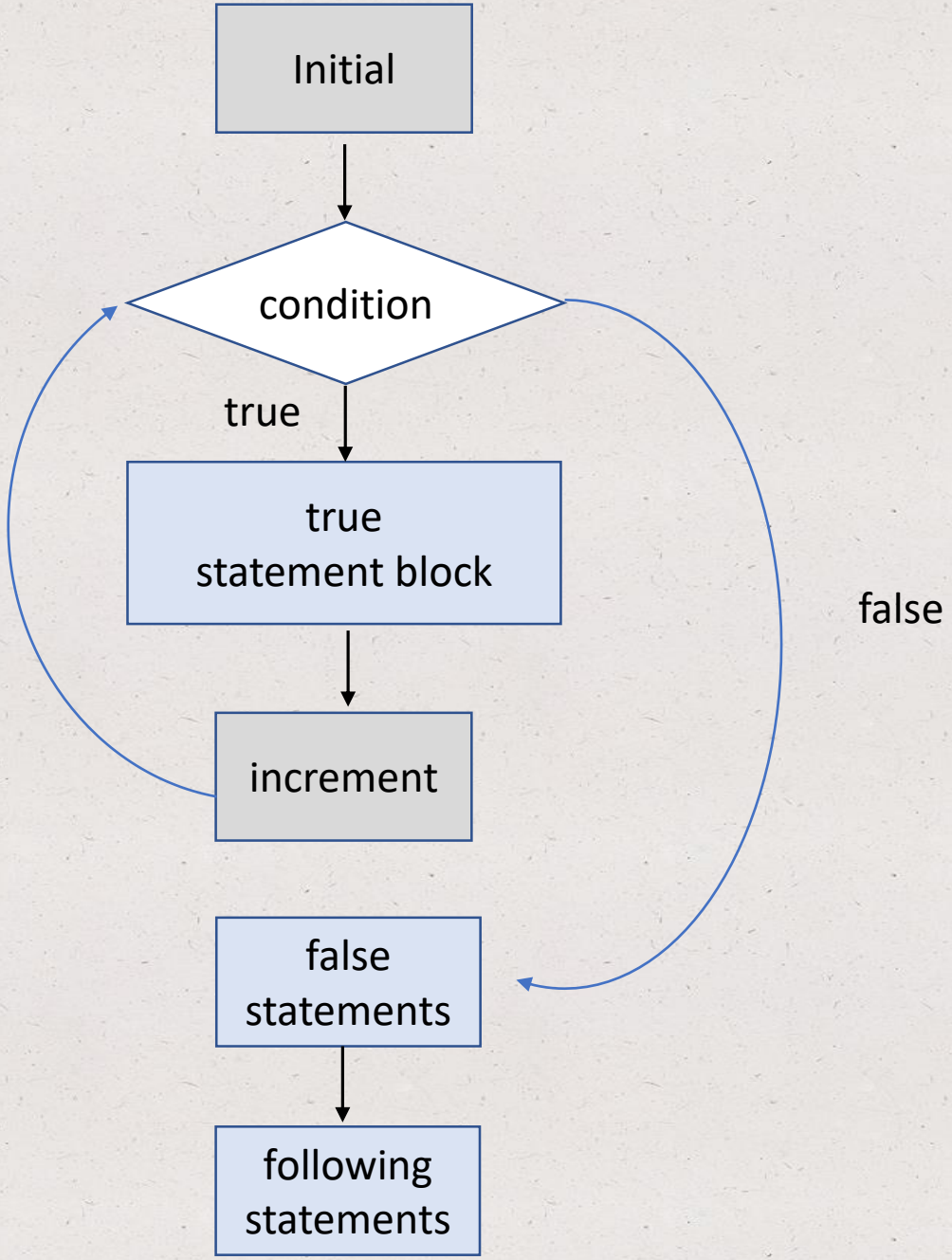


Output

a00 a01 a02
a10 a11 a12

While loops with else

```
while (condition):
    true statement 1
    .
    .
    true statement N
else:
    false statement 1
    .
    .
    false statement N
```

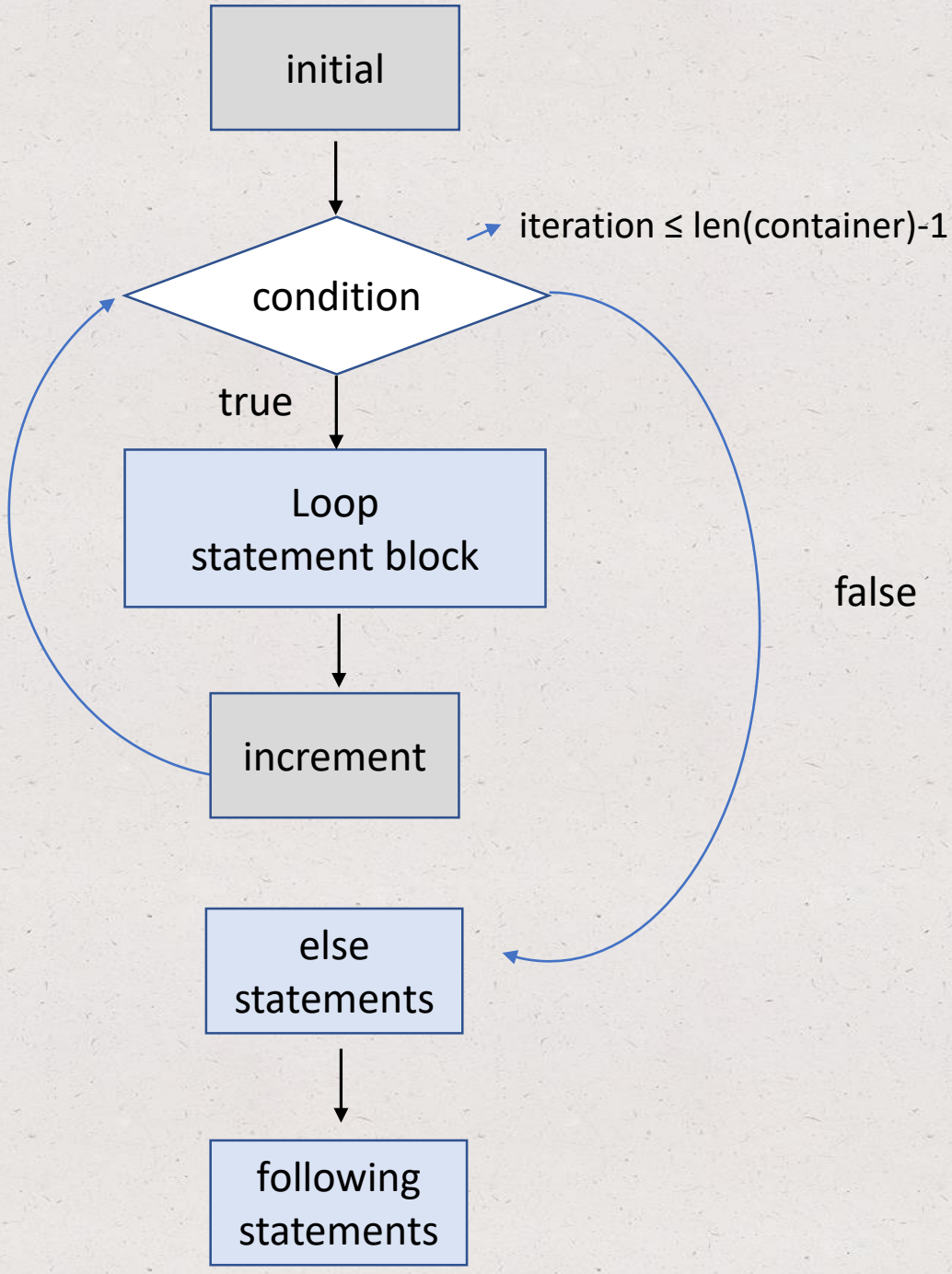


While loops with else: Example 1

```
iter = 0

while iter <= 5:
    print(iter, end = ' ')
    iter += 2
else:
   print ('goodbye')
```

For loops with else

```
for variable in container:
    loop statement 1
    .
    .
    loop statement N
else:
    loop statement 1
    .
    .
    loop statement N
```



For loops with else: Example 1

```
aae_class = ['AAE 635', 'AAE 636', 'AAE 875']

for name in aae_class:
    print(name)
else:
    print('you are done')
print('goodbye')
```

Multiple assignment

```
# the following statements:
AAE635 = 'Fall'
AAE636 = 'Fall'
AAE875 = 'Summer'

# can be written as:
AAE635, AAE636, AAE875 = ['Fall', 'Fall', 'Summer']
```

The built-in enumerate function

- **enumerate(*inerrable object, start = 0)*** – returns a tuple containing a **count** (from start which defaults to 0) and the **values** obtained from iterating over *inerrable object*

```
aae_classes = ['AAE635', 'AAE636', 'AAE875']
result = list(enumerate(aae_classes))
print(result)
```

```
[(0, 'AAE635'), (1, 'AAE636'), (2, 'AAE875')]
```

```
aae_classes = ['AAE635', 'AAE636', 'AAE875']
result = list(enumerate(aae_classes, start = 1))
print(result)
```

```
[(1, 'AAE635'), (2, 'AAE636'), (3, 'AAE875')]
```

The built-in enumerate function

- **enumerate(*inerrable object, start = 0)*** – returns a tuple containing a **count** (from start which defaults to 0) and the **values** obtained from iterating over *inerrable object*

```
aae_classes = ['AAE635', 'AAE636', 'AAE875']

for (index, classes) in enumerate(aae_classes):
    print('Class at position', index, 'is', classes)
```

```
Class at position 0 is AAE635
Class at position 1 is AAE636
Class at position 2 is AAE875
```

- In the example above, the for loop **unpacks** the tuple yielded by enumerate(aae_classes)

Top Hat Question # 7

What is the output?

```
aae_classes = ['AAE635', 'AAE636', 'AAE875']

for (value) in enumerate(aae_classes):
    print('Class at position x is', value)
```

Top Hat Question # 8

What is the output?

```
aae_classes = ['AAE635', 'AAE636', 'AAE875']

for (index, class) in enumerate(aae_classes):
    print('Class at position', index, 'is', class)
```

Chapter 6: Functions

- Definition
- Why use a function?
- Structure
- Calling functions
- More on functions
- Parameters
- Arguments
- Function comments

Definition

- Evolved from mathematical functions: g(x) = 3x + 3
- A function is defined by **(a)** name, and **(b)** a block of statements
- Can have multiple parameters, but only a **single** return object
- A couple of built-in Python functions (e.g. abs(), float(), int(), reversed(). Other examples?
- A function is also an object in Python

Why use a function?

- Abstraction: modular design (divide a code into several functions that can be tested separately)
- Improve code readability
- Avoids redundant code. Why copy-and-paste when you can create a function and call it multiple times?

Structure

```
def function_name(par1, par2, …):
    body
```


A **new object** of type function is created with the function_name bound to that object

Function terminology:

- **function_name:** use lowercase letters and underscores (e.g. average_price)
- **definition:** the name and the block of statements (body)
- **return statement:** the body can include a return statement
- **par1, par2,…:** input specified in a function definition

Structure: Example 1

```
def mean_prices(price1, price2):
    mean = (price1 + price2)/2
```

No return value

Function terminology:

- **function_name:** use lowercase letters and underscores (e.g. average_price)
- **definition:** the name and the block of statements
- **return statement:** the body can include a return statement
- **price1, price2,…:** input specified in a function definition

Structure: Example 2

```
def mean_prices(price1, price2):
    return (price1 + price2)/2
```

Function terminology:

- **function_name:** use lowercase letters and underscores (e.g. average_price)
- **definition:** the name and the block of statements
- **return statement:** the body can include a return statement.
- **price1, price2,…:** input specified in a function definition

More on **return statement**

- A function can only return a **single object!**
- That single object can be a variable or a container (a list or tuple) whose return value can be accessed by unpacking it

Top Hat Question # 9

What is the output?

```
def mean_age(age1, age2)
    mean = (age1 + age2)/2
```

Top Hat Question # 10

Is the following a valid function definition?

```
def mean_age(age1 + 5, age2):
    return (age1 + age2)/2
```

Top Hat Question # 10

Is the following a valid function definition?

```
def mean_age(age1 + 5, age2):
    return (age1 + age2)/2
```

A parameter cannot be an expression! Syntax error

Calling functions

```
function_name(arg1, arg2, …)
```


An invocation of the function name, causes the function object to execute a **call** operation

Function call terminology:

arg1, arg2,…: a value provided to a function parameter during a function call; arguments are assigned to function's parameters by *position* or *name*

Calling functions: Example 1

```
def mean_prices(price1, price2):
    return (price1 + price2)/2

mean_prices(50, 70)
```


Using the arguments provided, the *function is called* and evaluated to the return value of the function

A return statement can be placed anywhere in the body of the loop

Top Hat Question # 11

Is the following a valid function call?

```
def mean_age(age1, age2)
    return (age1 + age2)/2

mean_age(10 + 2, 5)
```

Top Hat Question # 11

Is the following a valid function call?

```
def mean_age(age1, age2)
    return (age1 + age2)/2

mean_age(10 + 2, 5)
```

An argument can be an expression!

Top Hat Question # 12

Is the following a valid function call?

```
def mean_age(age1, age2)
    return (age1 + age2)/2

mean_age(10 + 2)
```

Top Hat Question # 12

Is the following a valid function call?

```
def mean_age(age1, age2)
    return (age1 + age2)/2

mean_age(10 + 2)
```

An argument can be an expression, but when calling a function one needs to provide a value for each parameter. Syntax error.

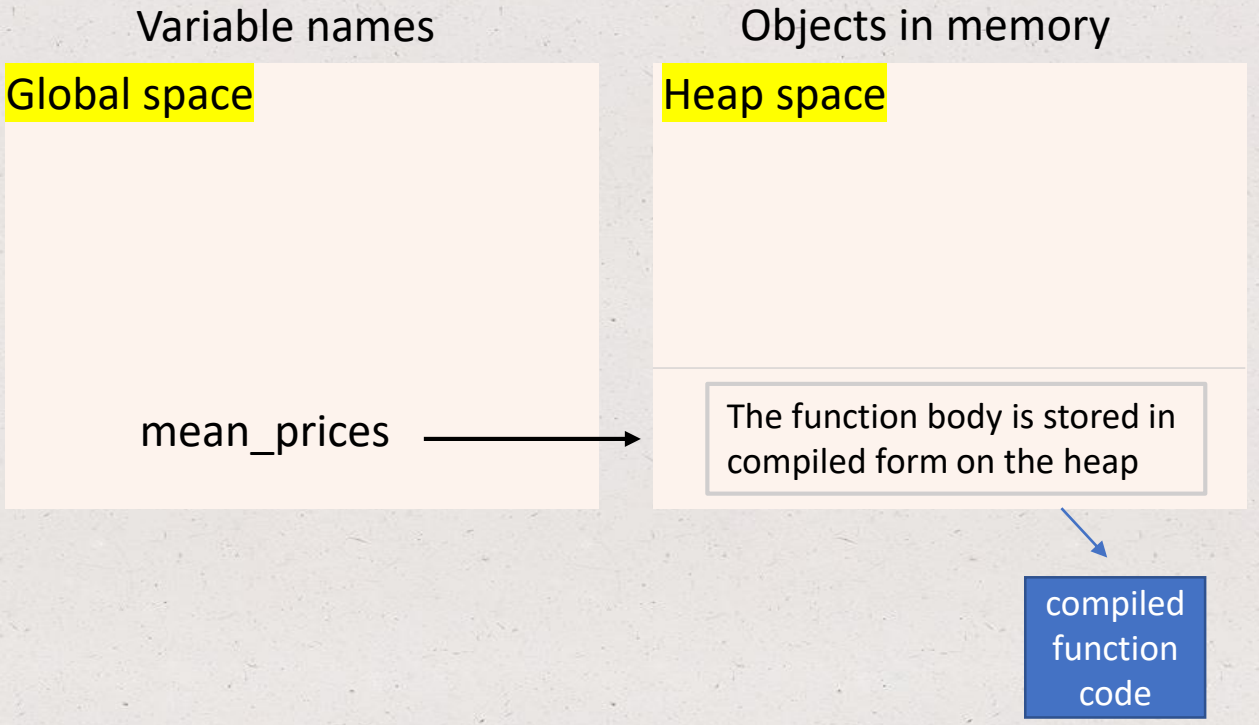
Calling functions: Example 2

```
def mean_prices(price1, price2):
    mean = (price1 + price2)/2

mean_prices(50, 70)
```


Using the arguments provided, the *function is called* and evaluated to the return value of the function. If **no return value** is specified then **None** is returned

Calling functions: Example 1 – memory rep

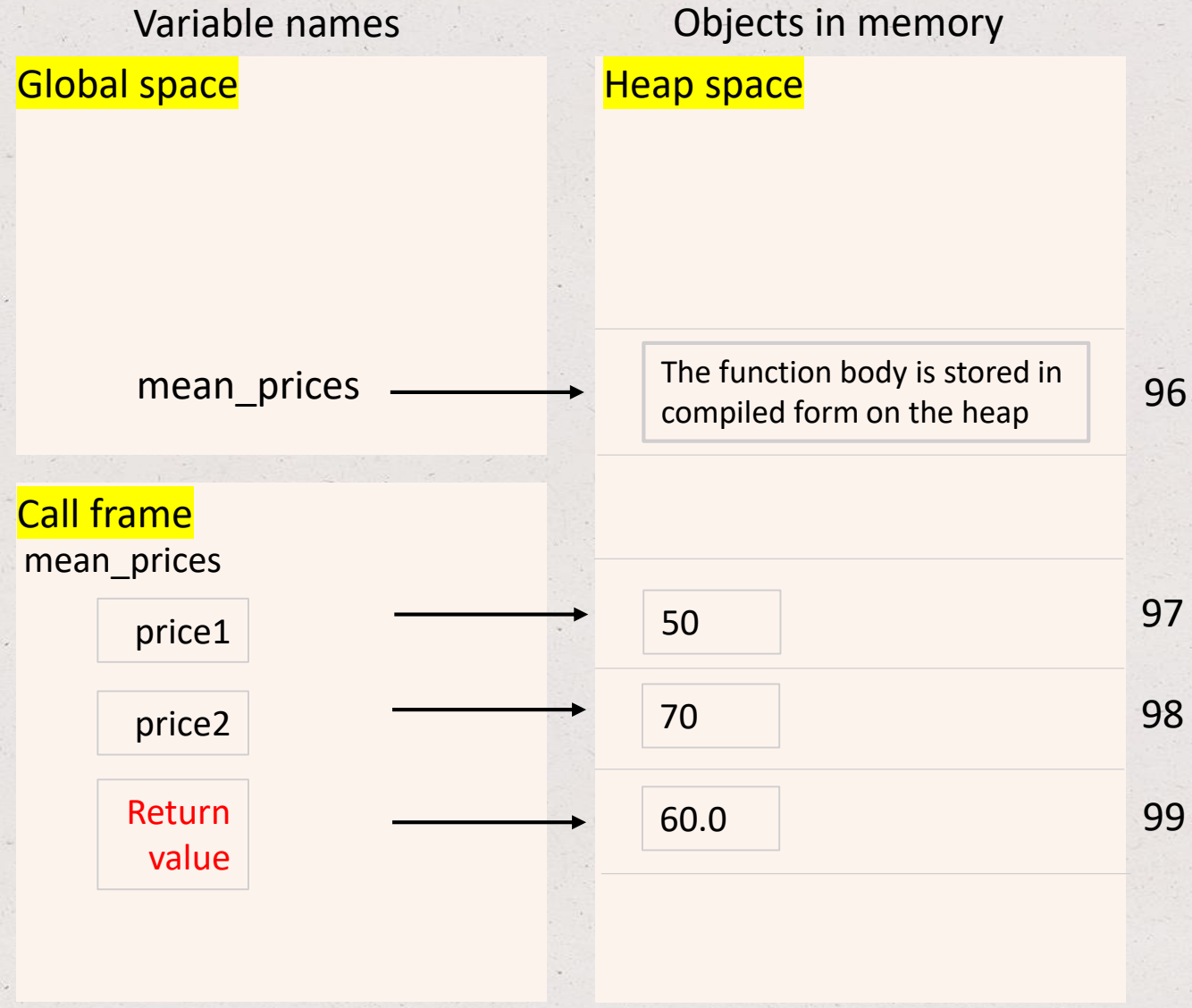
```
def mean_prices(price1, price2):
    return (price1 + price2)/2
```



Calling functions: Example 1 – memory rep

```
def mean_prices(price1, price2):
    return (price1 + price2)/2

mean_prices(50,70)
```

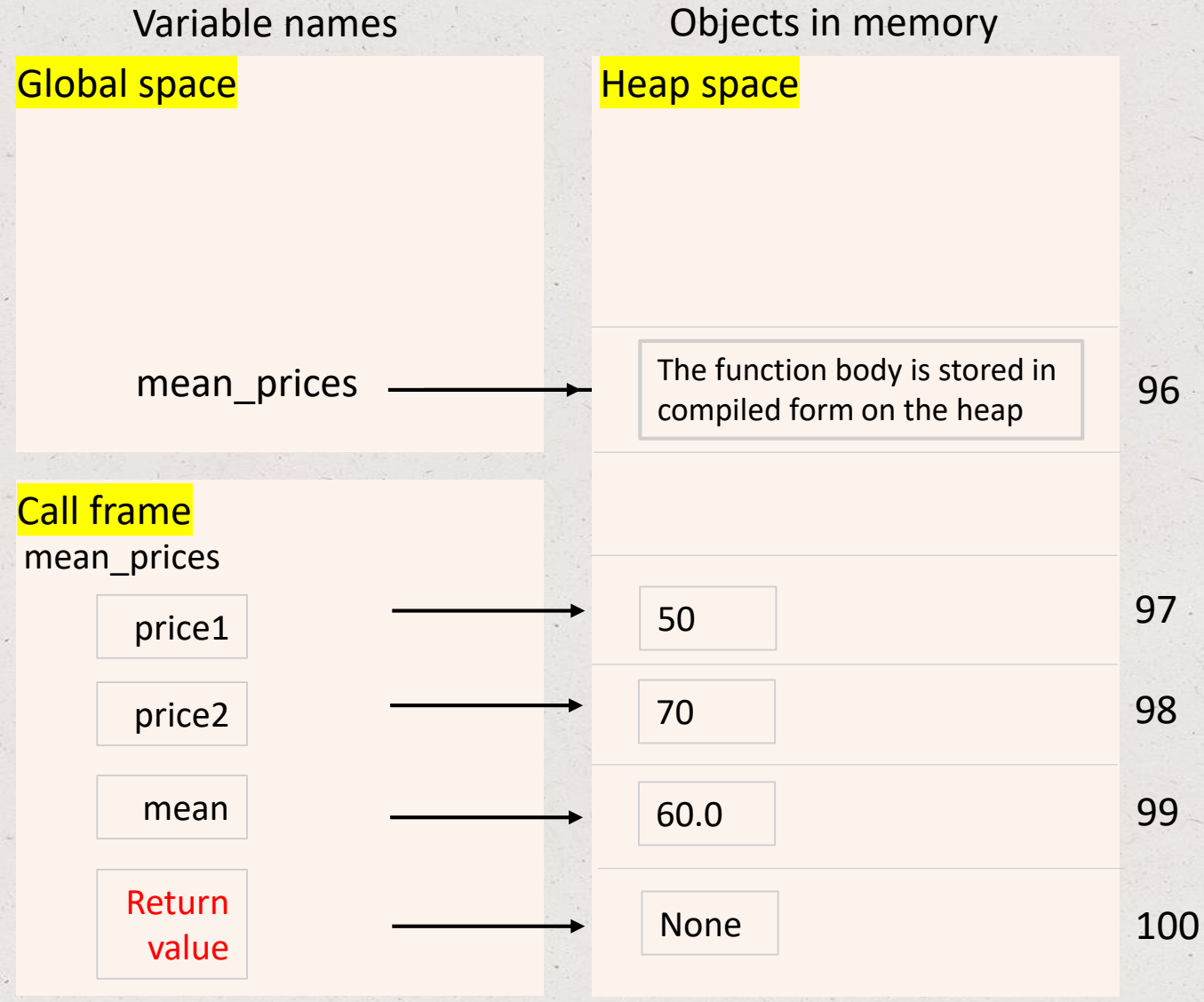
The **function call** jumps execution to the function statements



Calling functions: Example 1 – memory rep

```
def mean_prices(price1, price2):
    mean = (price1 + price2)/2

mean_prices(50,70)
```

The **function call** jumps execution to the function statements



Global vs. Heap space. Call frame

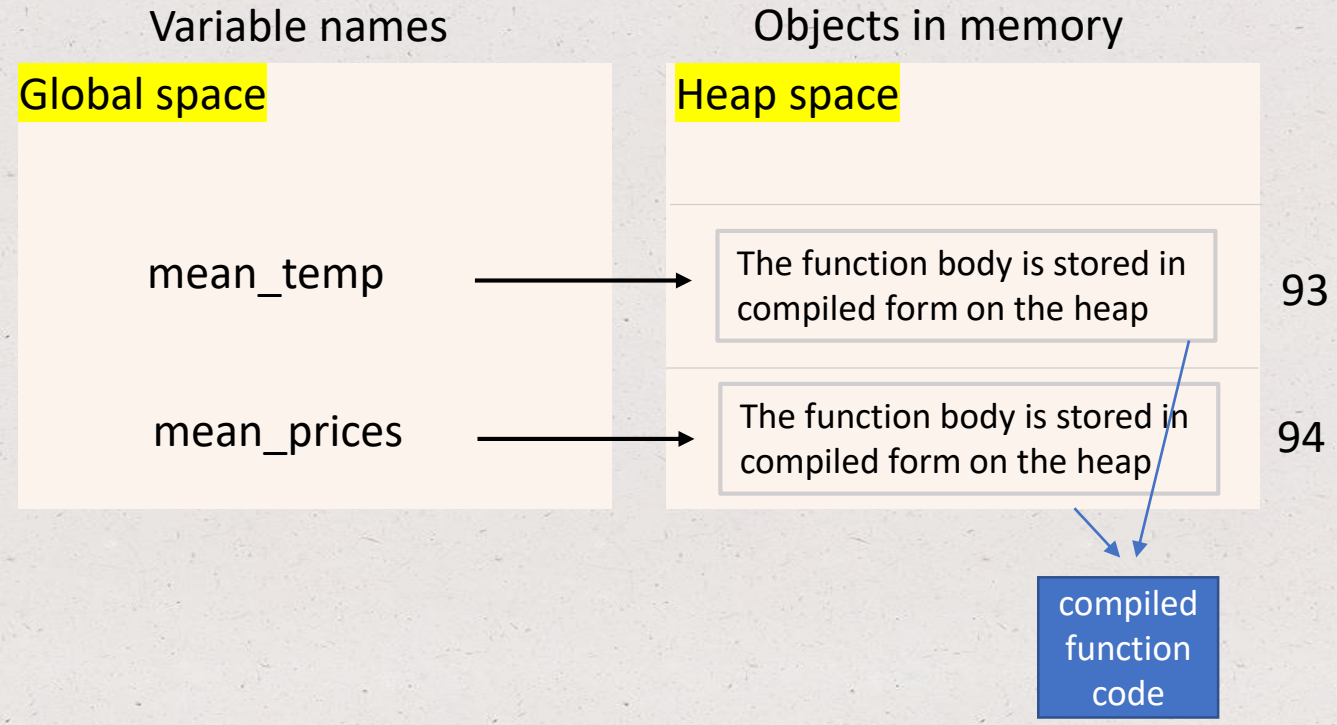
- **Global space:**
  - What you 'start with'
  - Stores global variables, modules, and functions
  - Lasts until you quit Python

- **Heap space:**
  - Where objects are stored
  - Have to access indirectly

- **Call frames:**
  - Stores the variables in function call (these variables are stored locally, aka local variables)
  - Deleted when call done!

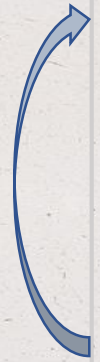
Calling functions: Example 3 – memory rep

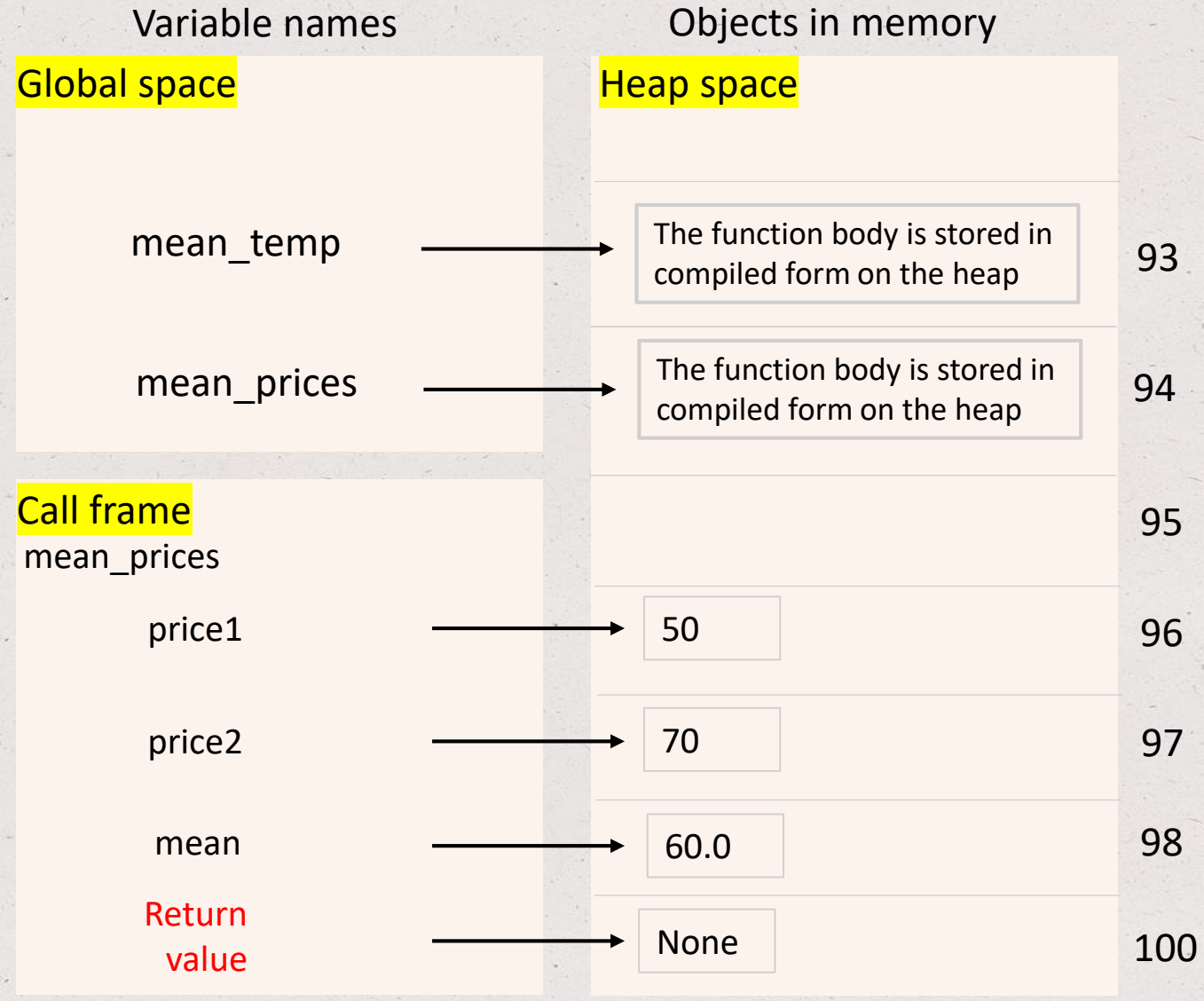
```
def mean_prices(price1, price2):
    mean = (price1 + price2)/2

def mean_temp(temp1, temp2):
    mean = (temp1 + temp2)/2
```



Calling functions: Example 3 – memory rep

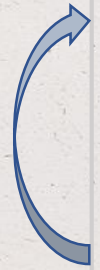
```
def mean_prices(price1, price2):
    mean = (price1 + price2)/2

def mean_temp(temp1, temp2):
    mean = (temp1 + temp2)/2

mean_prices(50,70)
mean_temp(20,40)
```

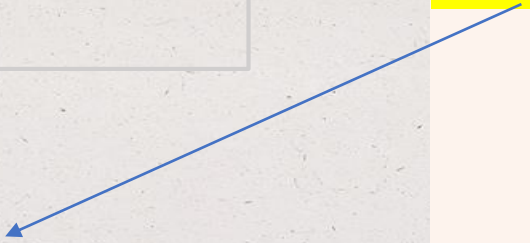


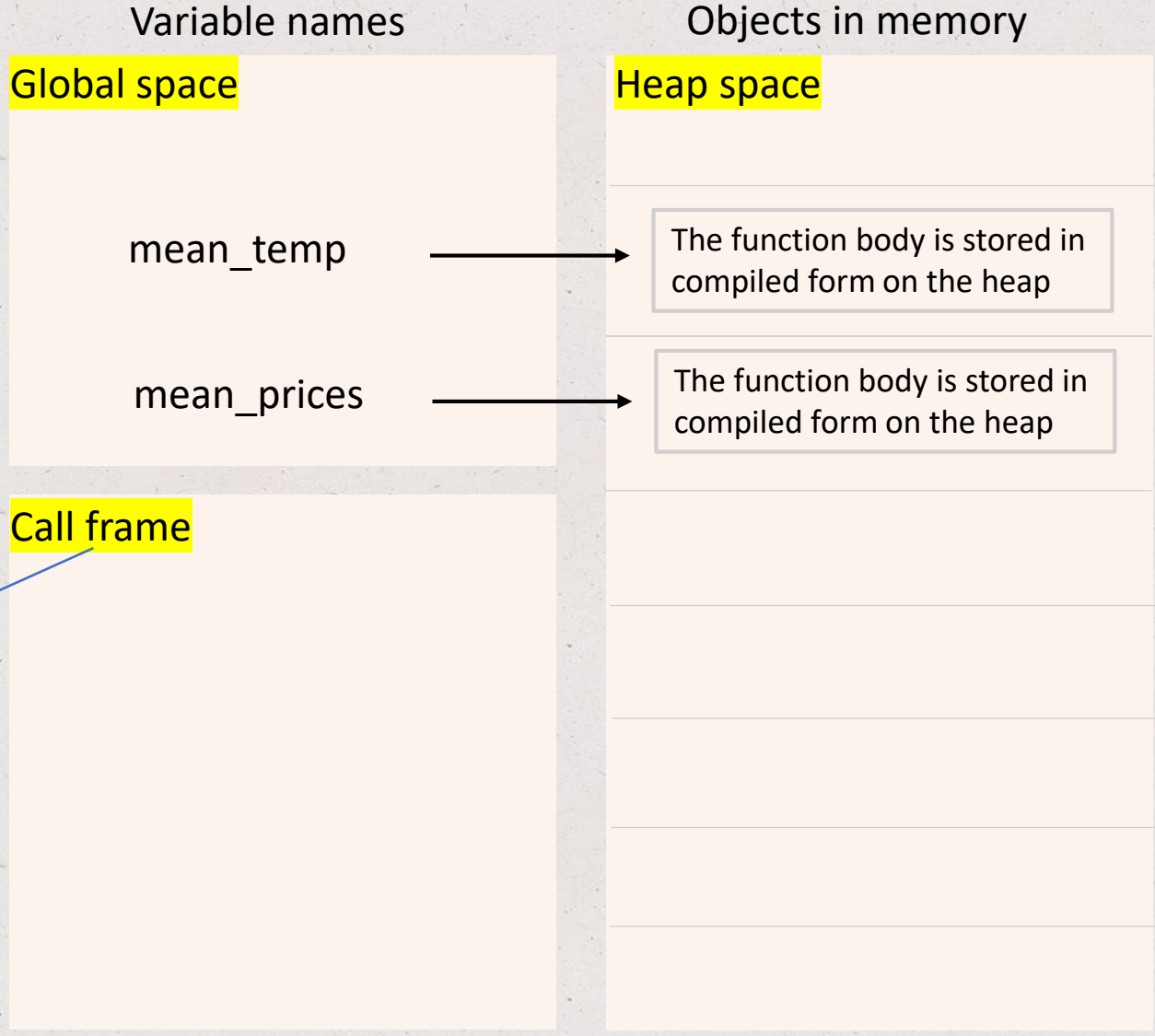


Calling functions: Example 3 – memory rep

```
def mean_prices(price1, price2):
    mean = (price1 + price2)/2

def mean_temp(temp1, temp2):
    mean = (temp1 + temp2)/2

mean_prices(50,70)
mean_temp(20,40)
```

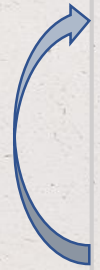


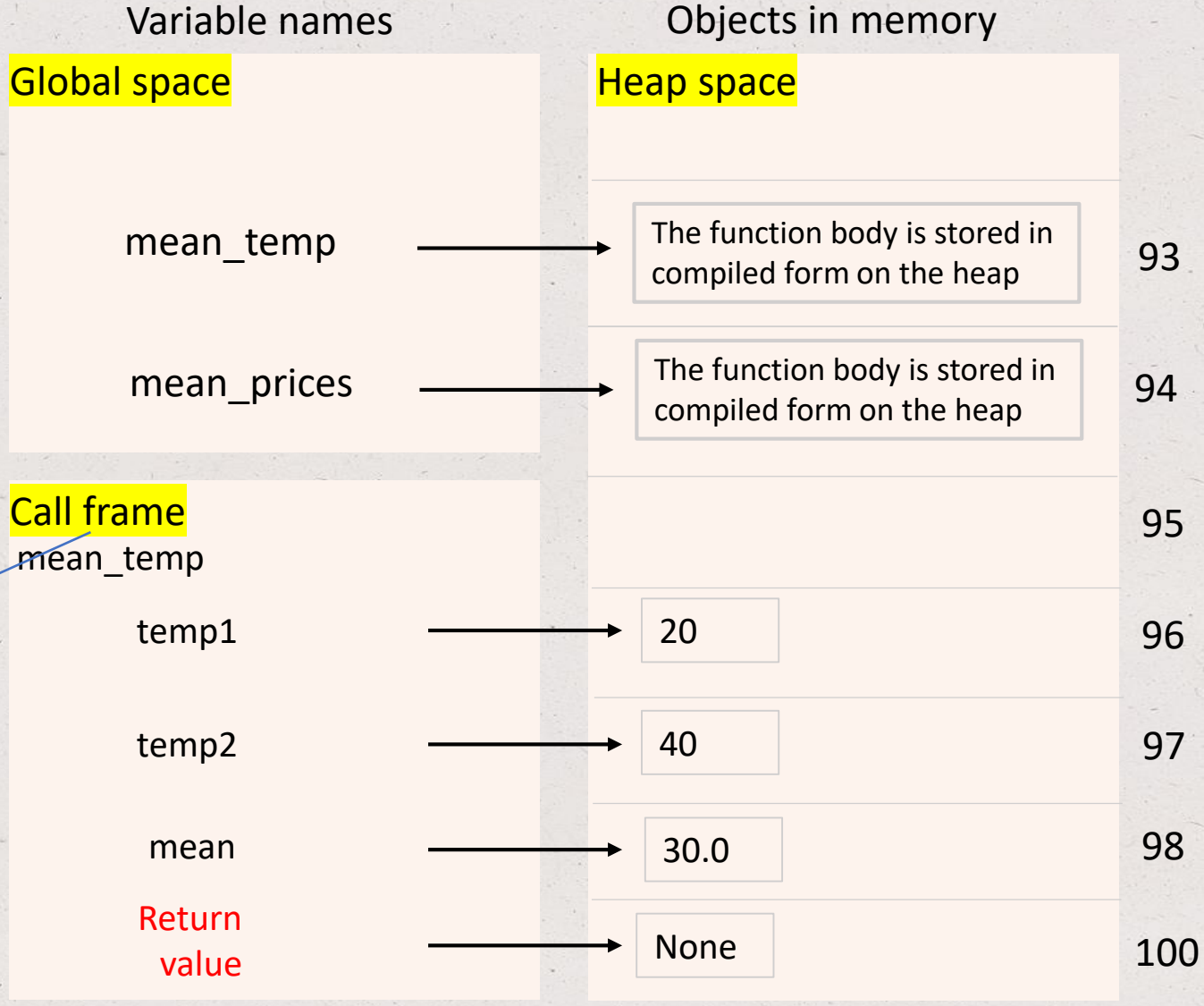




Calling functions: Example 3 – memory rep

```
def mean_prices(price1, price2):
    mean = (price1 + price2)/2

def mean_temp(temp1, temp2):
    mean = (temp1 + temp2)/2

mean_prices(50,70)
mean_temp(20,40)
```





Everything was deleted when the call to mean_prices() was done

More on functions

- **Can be used in assignment statements** (remember functions are objects!)

```
def mean_prices(price1, price2):
    mean = (price1 + price2)/2

mean_prices(50,70)

my_function = mean_prices
```


Calling my_function() is the same as calling mean_prices(). Both functions reference to the same object

More on functions

- **Can be used in assignment statements** (remember functions are objects!)

- The body of a function can include **nested function calls**

```
def mean_prices(price1, price2):
    mean = (price1 + price2)/2
    user_input = int(input('Introduce quantity:' ))
```

The input() function is nested in the mean_prices() function

More on functions

- **Can be used in assignment statements** (remember functions are objects!)
- The body of a function can include **nested function calls**
- **At least one statement is required in the body of a function**
- **Dynamic typing vs. static typing**
- **Function stubs: pass, raise NotImplementedError, print and return -1** (Adam to add more on this in the lab)

More on functions: Variable scope

- A **code block** in Python is any number of statements after a semicolon
- The scope of a variable is the region of code where it can be used
- A variable's **scope** is **limited** to the **code block** in which it's declared
- E.g. for a variable created inside a function, the variable scope is limited to inside that function
- A variable can only be used after it's declared

More on functions: Variable scope

- Two types of variables:
  - **Local:** defined inside a function; cannot be used outside the function
  - **Global:** defined outside a function; can be used at any time, including inside of functions

```
seed_type = 'GM'

def mean_prices(price1, price2):
    mean = (price1 + price2)/2

mean_prices(50,70)
```

seed_type is global variable

'price1', 'price2', 'mean' are local variables
(stored in the function's call frame)

More on functions: Variable scope

- Two types of variables:
  - **Local:** defined inside a function; cannot be used outside the function
  - **Global:** defined outside a function; can be used at any time, including **inside of functions**

```
seed_type = 'GM'

def mean_prices(price1, price2):
    mean = (price1 + price2)/2
    global seed_type
    seed_type = 'Conv'

mean_prices(50,70)
```

Must use a **global** statement to change the value of a global variable inside of a function; seed_type is not created locally in this case

Top Hat Question # 13

What happens in the memory of the computer when the following code is run?

```
seed_type = 'GM'

def mean_prices(price1, price2):
    mean = (price1 + price2)/2

mean_prices(50,70)
```

Top Hat Question # 14

What is the output?

```
seed_type = 'GM'

def mean_prices(price1, price2):
    mean = (price1 + price2)/2
    seed_type = 'Conv'

mean_prices(50,70)
print(seed_type)
```

Top Hat Question # 15

What is the output?

```
seed_type = 'GM'

def mean_prices(price1, price2):
    mean = (price1 + price2)/2
    global seed_type
    seed_type = 'Conv'

mean_prices(50,70)
print(seed_type)
```

Parameters

```
def function_name(par1, par2, …):
    body
```

More on function parameters:

- **Scope:** considered as variables within the body of the function (cannot be used outside the function)
- **Initialized:** at the moment of the function call

Arguments

```
function_name(arg1, arg2, …)
```

Passing *functions as arguments* can improve the readability of the code!

```
def mean_prices(price1, price2):
    return (price1 + price2)/2

def total_amount(price, quantity):
    return price * quantity

total_payment = total_amount(mean_prices(2, 4), 20)
print('Total payment is: ', total_payment)
```

Top Hat Question # 16

What is the output of the following function:

```
def mean_prices(price1, price2):
    return (price1 + price2)/2

def total_amount(price, quantity):
    amount = price * quantity

total_payment = total_amount(mean_prices(2, 4), 20)
print('Total payment is: ', total_payment)
```

Top Hat Question # 16

What is the output of the following function:

```
def mean_prices(price1, price2):
    return (price1 + price2)/2

def total_amount(price, quantity):
    amount = price * quantity

total_payment = total_amount(mean_prices(2, 4), 20)
print('Total payment is: ', total_payment)
```

No return value

Answer: the print() function prints *None*. Why? Use pythontutor to track the code

Arguments: mutability

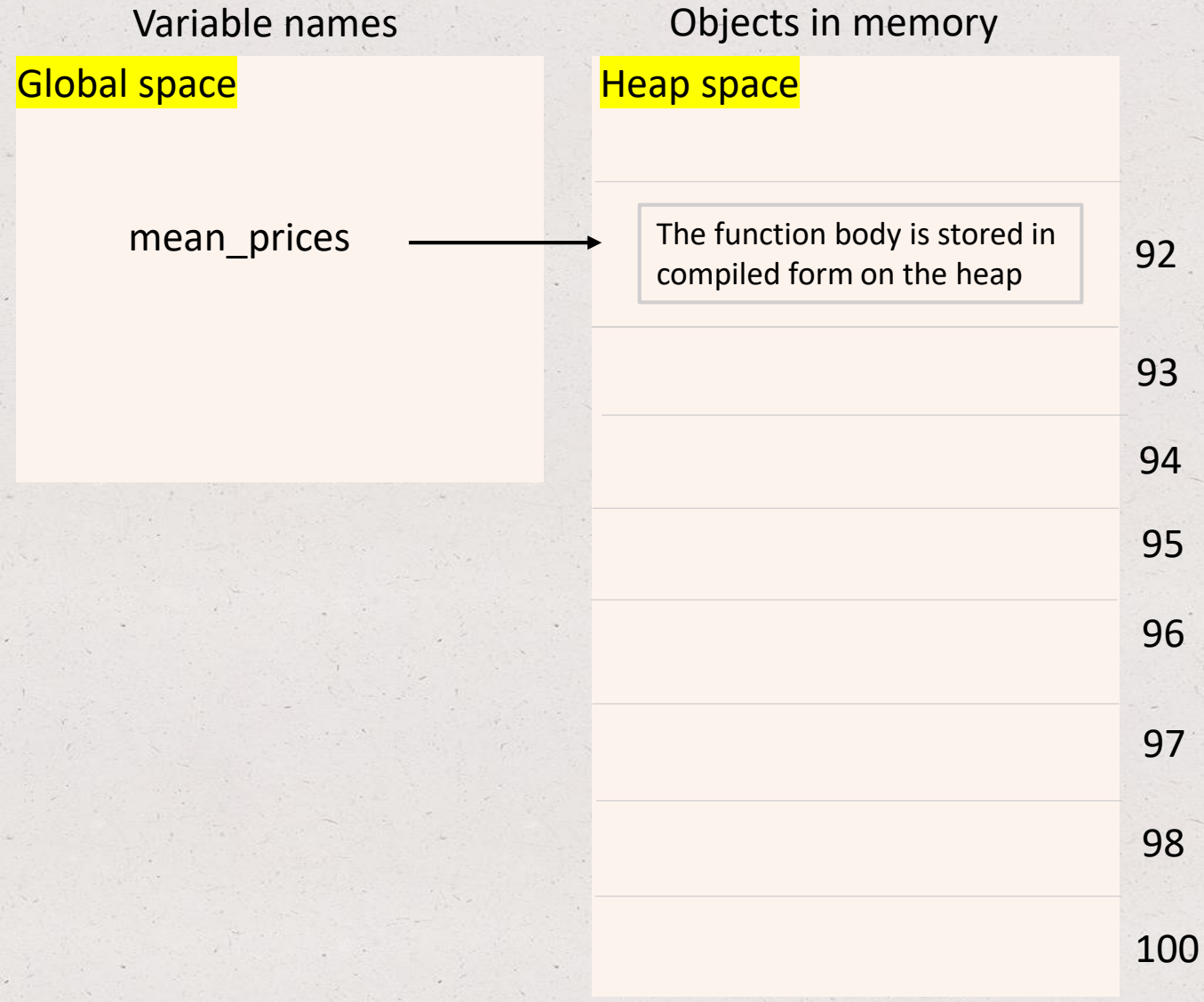
- What happens if we modify a function's argument that is referenced elsewhere in the program?

- Depends on the type of the object:
  - If **immutable** (e.g. string or integer): the modification **is** limited to **inside** of the function

  - If **mutable** (e.g. lists): the modification **is not** limited to **inside** the function; the modification will affect any other variables in the program that reference the same object

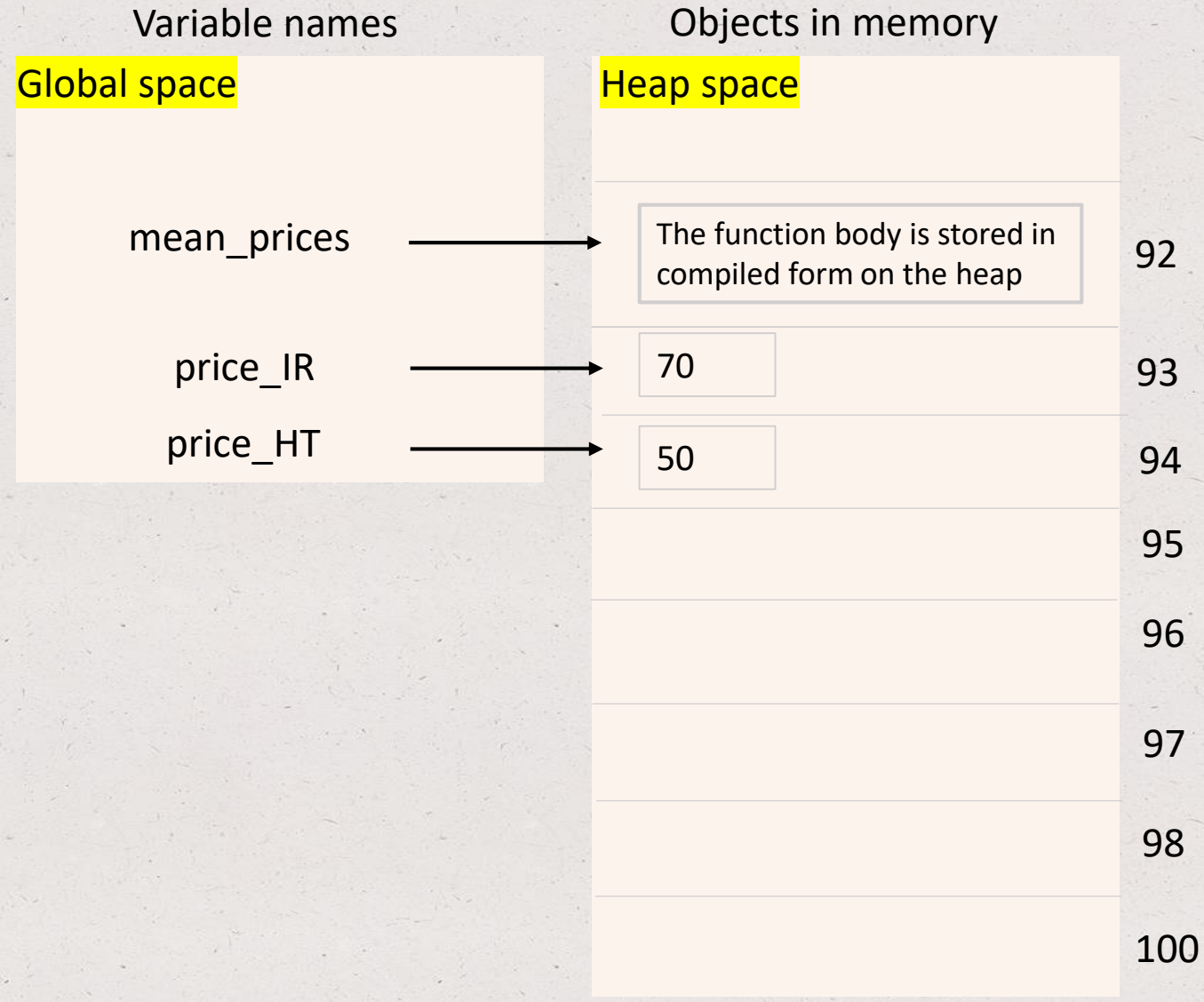
Arguments: mutability Ex 1 – Memory rep

```
def mean_prices(price1, price2):
    price1 += 5
    mean = (price1 + price2)/2


price_IR = 70
price_HT = 50

mean_prices(price_IR, price_HT)
print(price_IR)
```



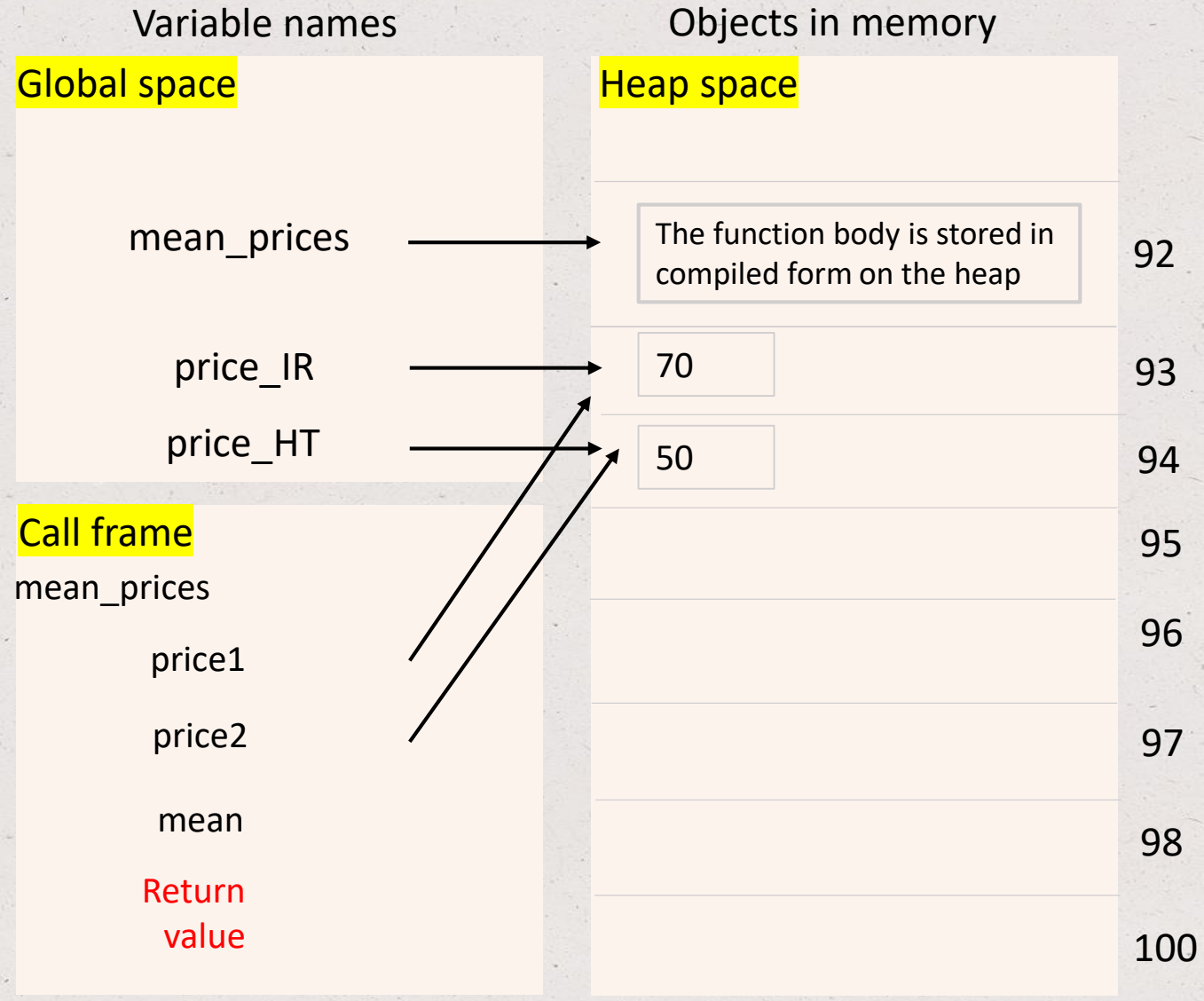
Arguments: mutability Ex 1 – Memory rep

```
def mean_prices(price1, price2):
    price1 += 5
    mean = (price1 + price2)/2


price_IR = 70
price_HT = 50

mean_prices(price_IR, price_HT)
print(price_IR)
```



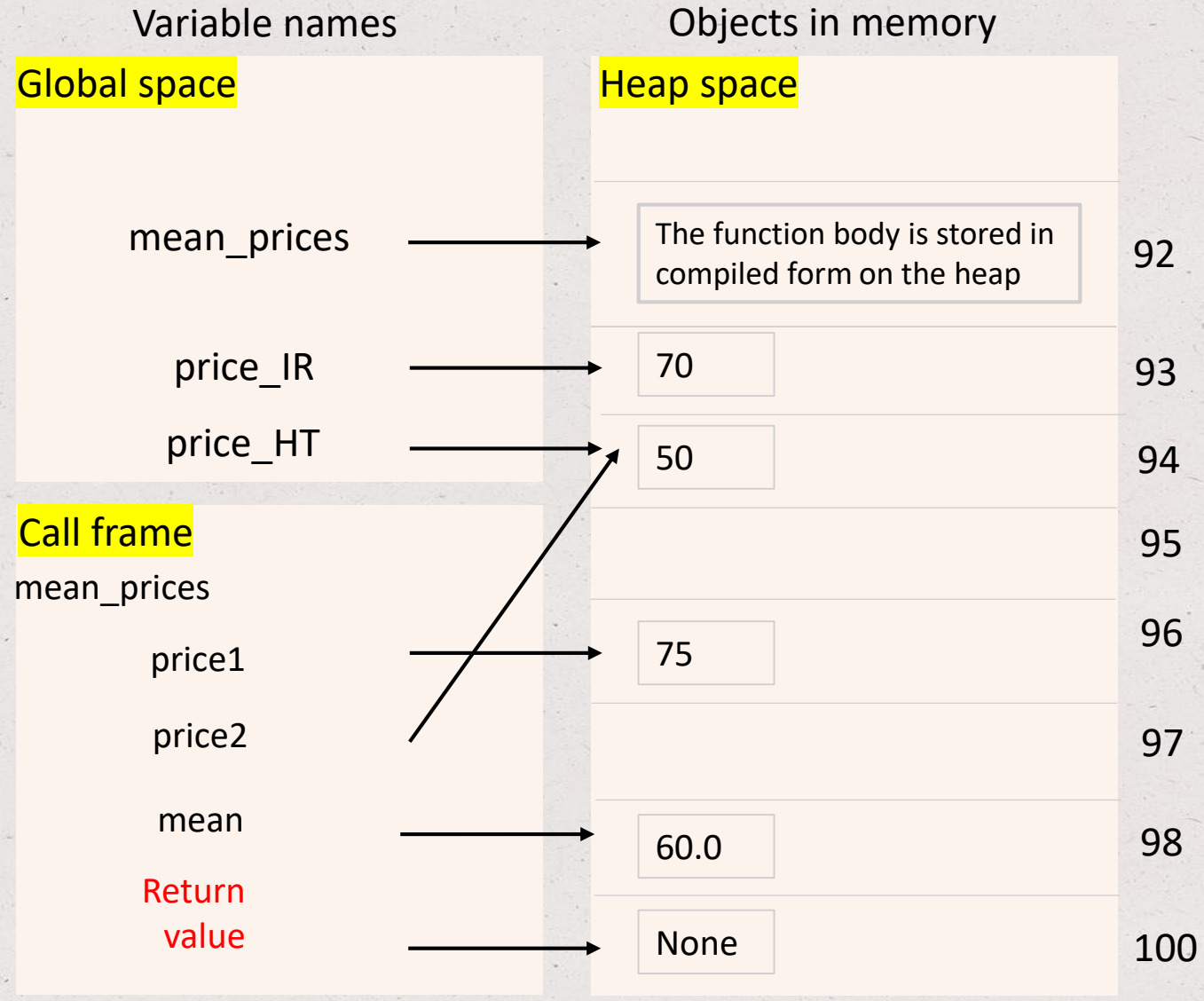
Arguments: mutability Ex 1 – Memory rep

```
def mean_prices(price1, price2):
    price1 += 5
    mean = (price1 + price2)/2


price_IR = 70
price_HT = 50

mean_prices(price_IR, price_HT)
print(price_IR)
```



Arguments: mutability Ex 1 – Memory rep

```
def mean_prices(price1, price2):
    price1 += 5
    mean = (price1 + price2)/2


price_IR = 70
price_HT = 50

mean_prices(price_IR, price_HT)
print(price_IR)
```

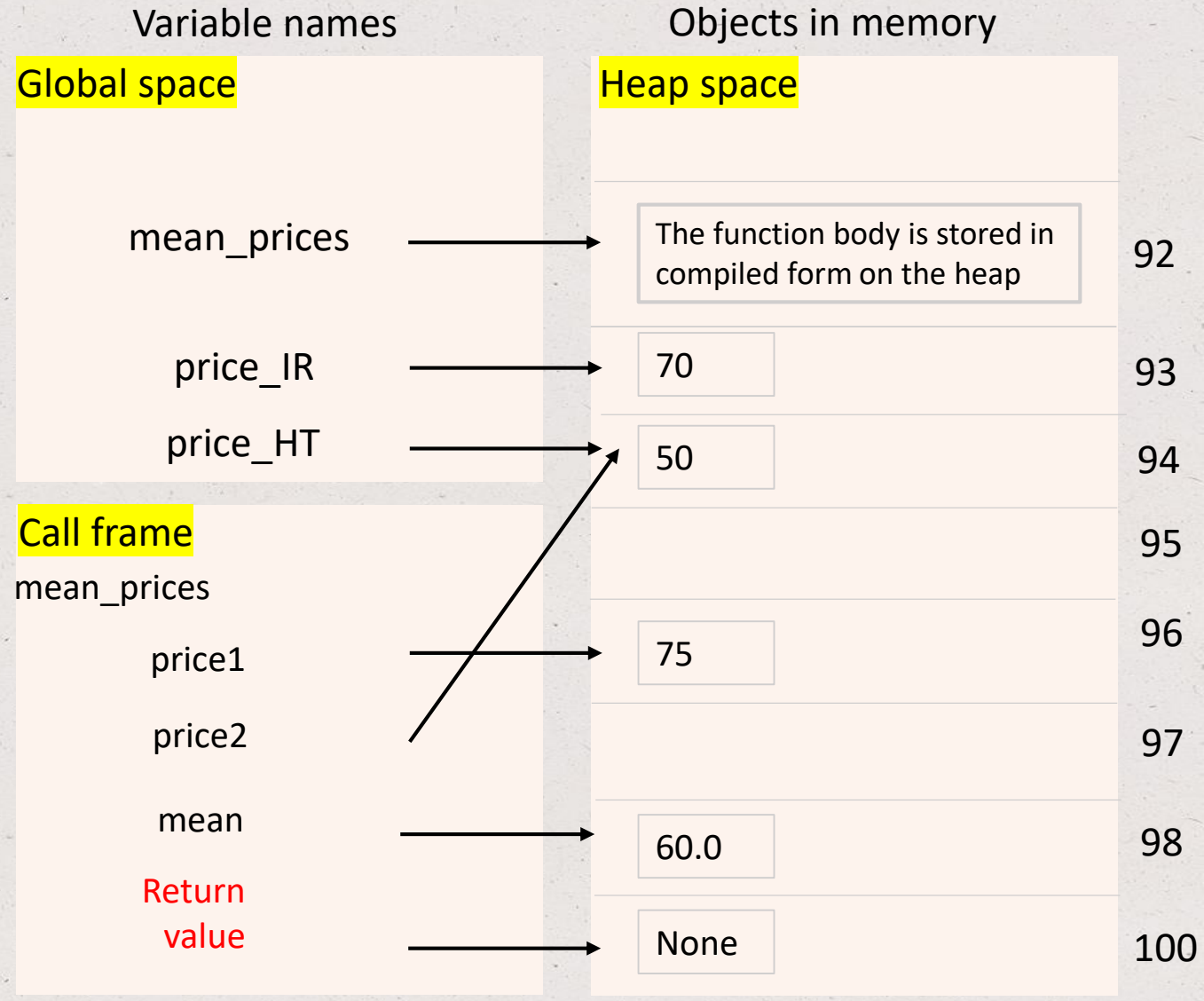


Arguments: mutability Ex 1 – Memory rep

```
def mean_prices(price1, price2):
    price1 += 5
    mean = (price1 + price2)/2


price_IR = 70
price_HT = 50

mean_prices(price_IR, price_HT)
print(price_IR)
```

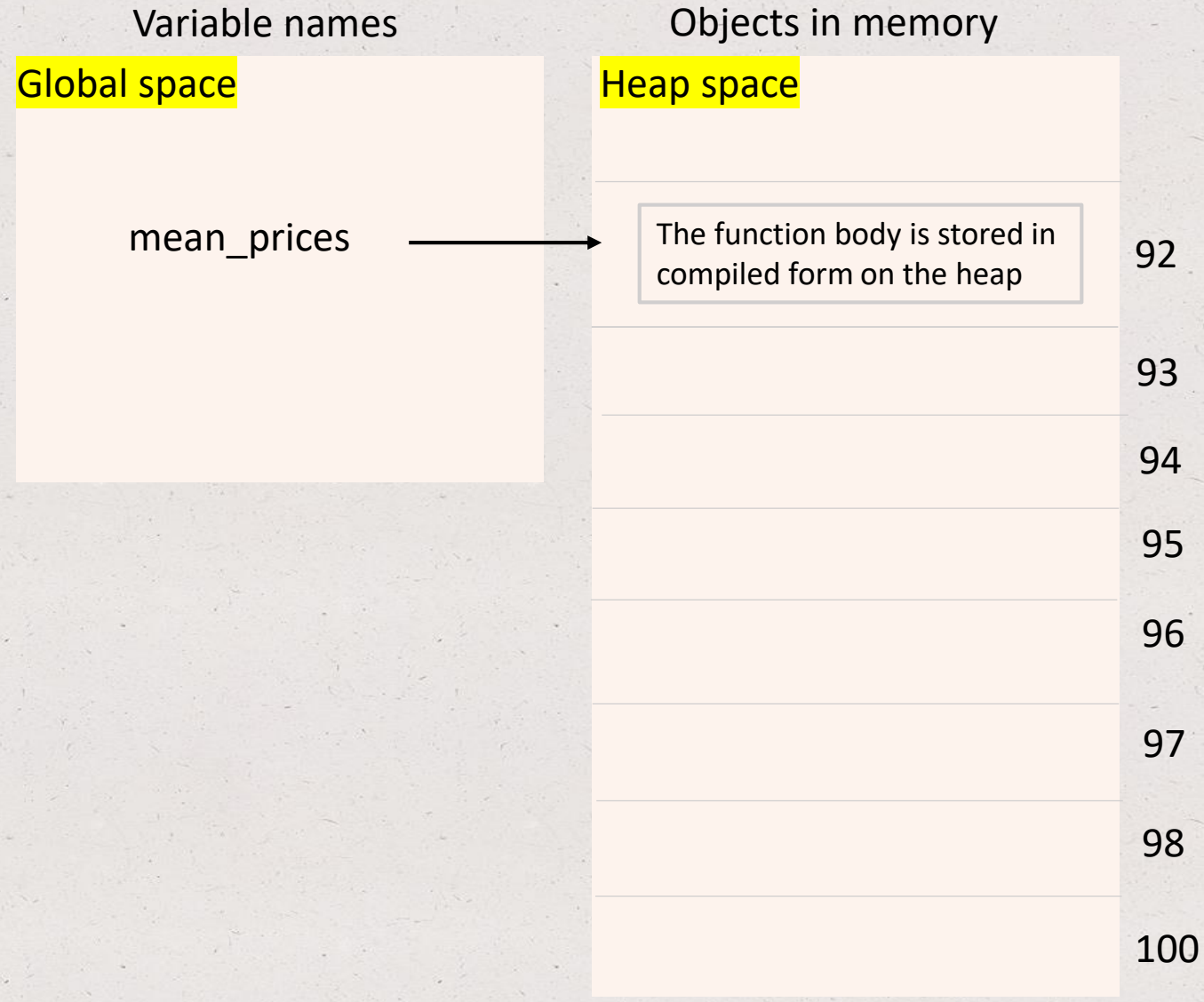
Output

70



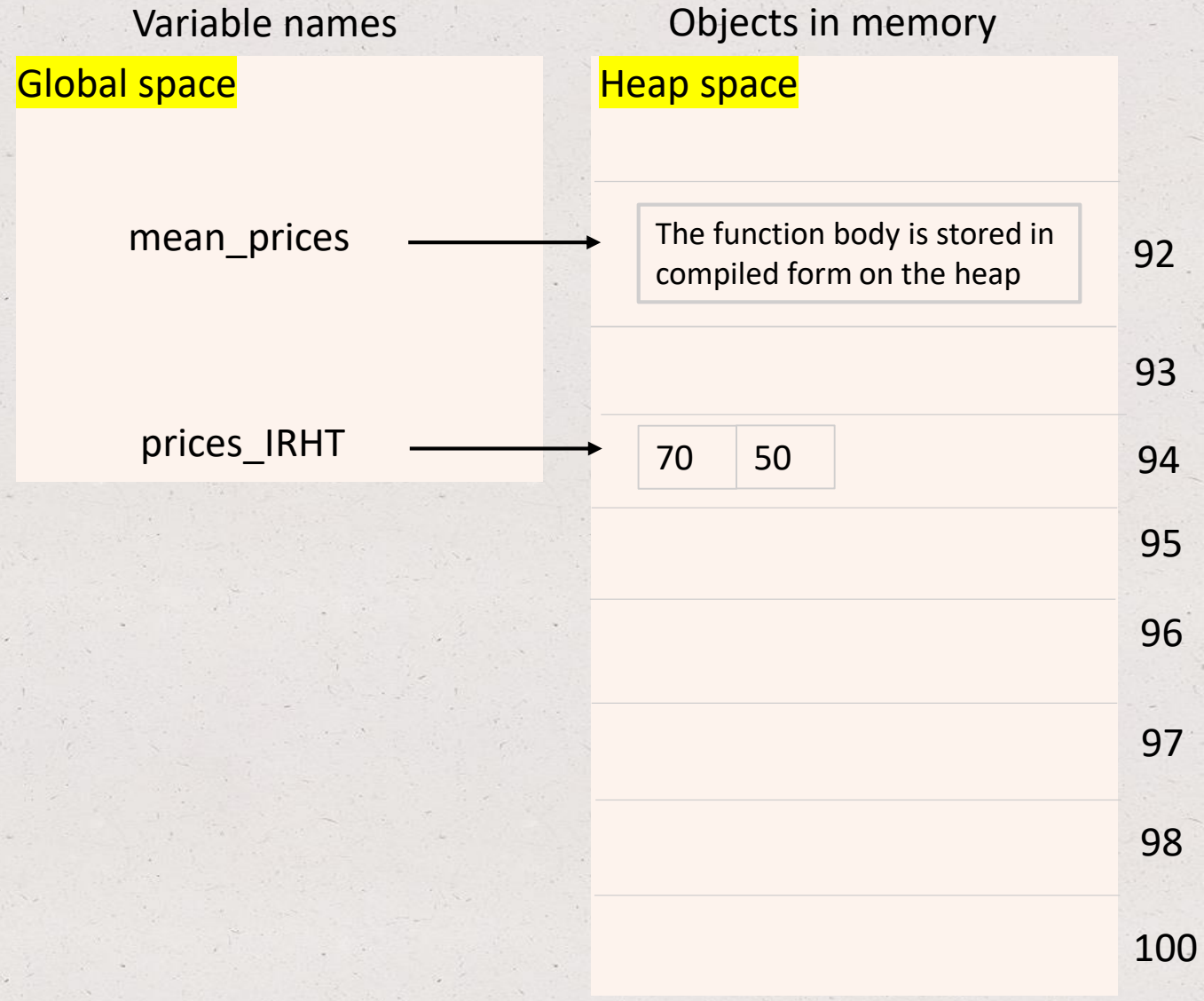
Arguments: mutability Ex 2 – Memory rep

```
def mean_prices(prices):
    prices[0] = 75
    mean = (prices[0] + prices[1])/2


prices_IRHT = [70, 50]

mean_prices(prices_IRHT)
print(prices_IRHT)
```



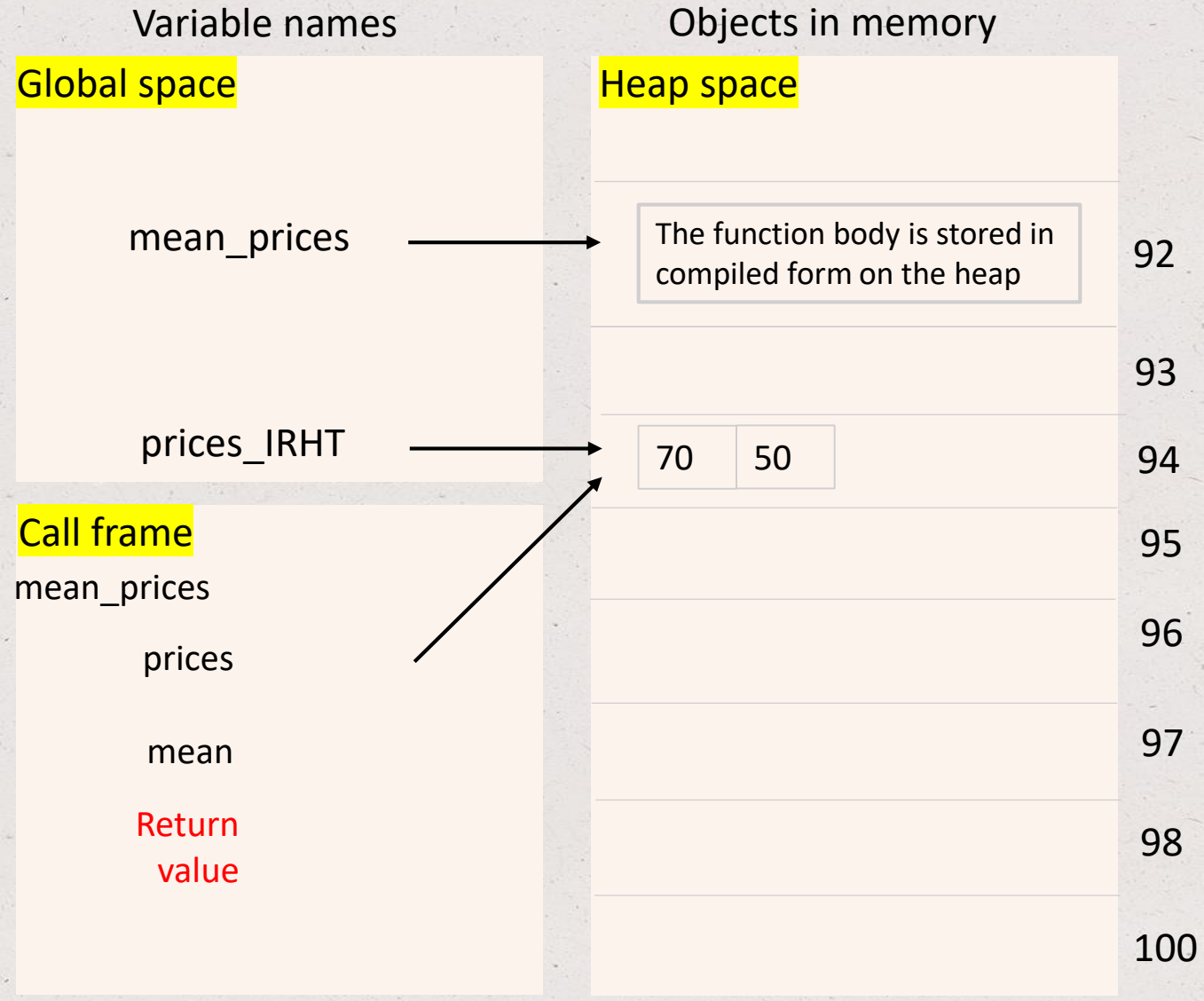
Arguments: mutability Ex 2 – Memory rep

```
def mean_prices(prices):
    prices[0] = 75
    mean = (prices[0] + prices[1])/2


prices_IRHT = [70, 50]

mean_prices(prices_IRHT)
print(prices_IRHT)
```



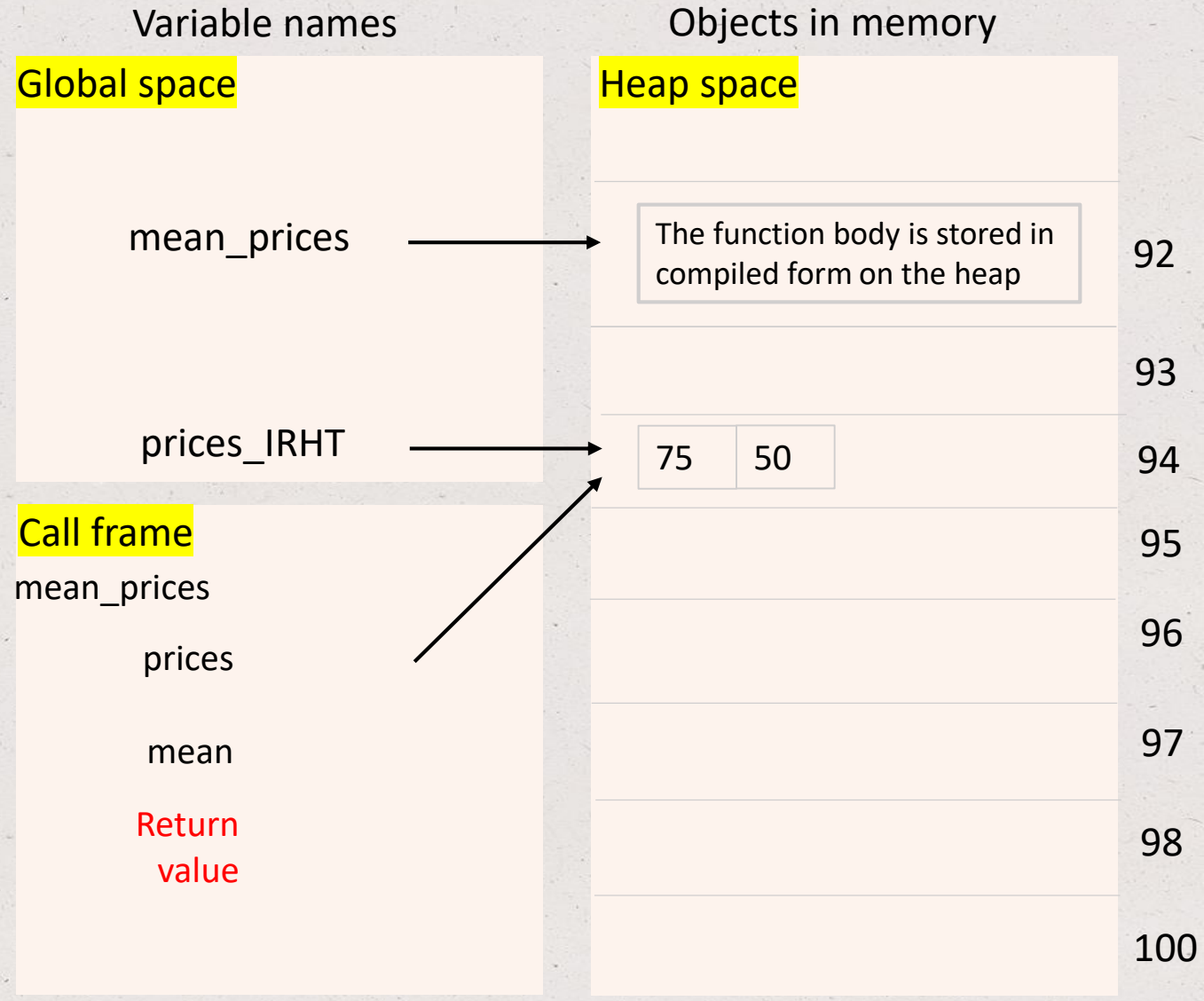
Arguments: mutability Ex 2 – Memory rep

```
def mean_prices(prices):
    prices[0] = 75
    mean = (prices[0] + prices[1])/2


prices_IRHT = [70, 50]

mean_prices(prices_IRHT)
print(prices_IRHT)
```



Arguments: mutability Ex 2 – Memory rep

```
def mean_prices(prices):
    prices[0] = 75
    mean = (prices[0] + prices[1])/2


prices_IRHT = [70, 50]

mean_prices(prices_IRHT)
print(prices_IRHT)
```

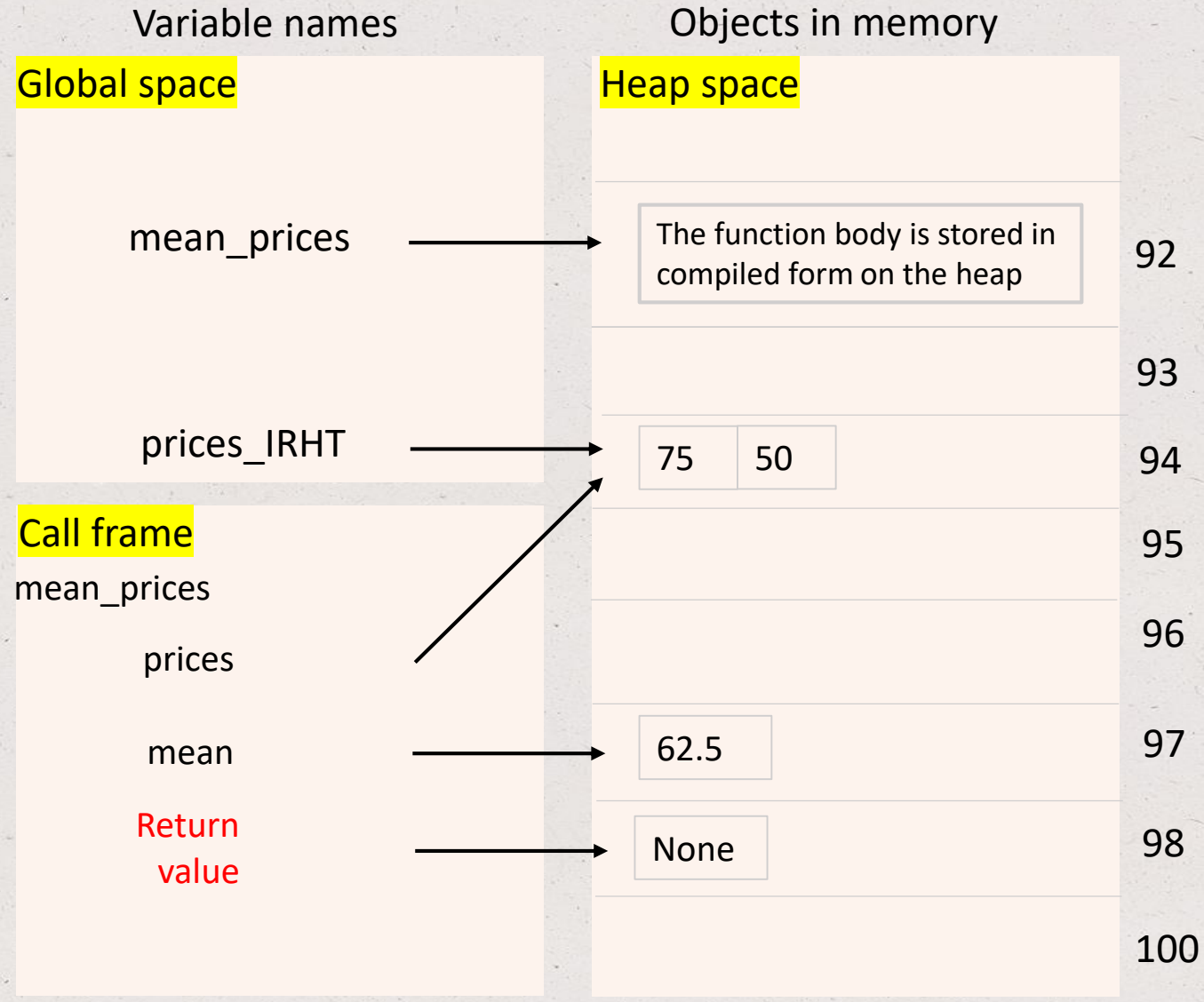


Arguments: mutability Ex 2 – Memory rep

```
def mean_prices(prices):
    prices[0] = 75
    mean = (prices[0] + prices[1])/2


prices_IRHT = [70, 50]

mean_prices(prices_IRHT)
print(prices_IRHT)
```

Output

[75, 50]



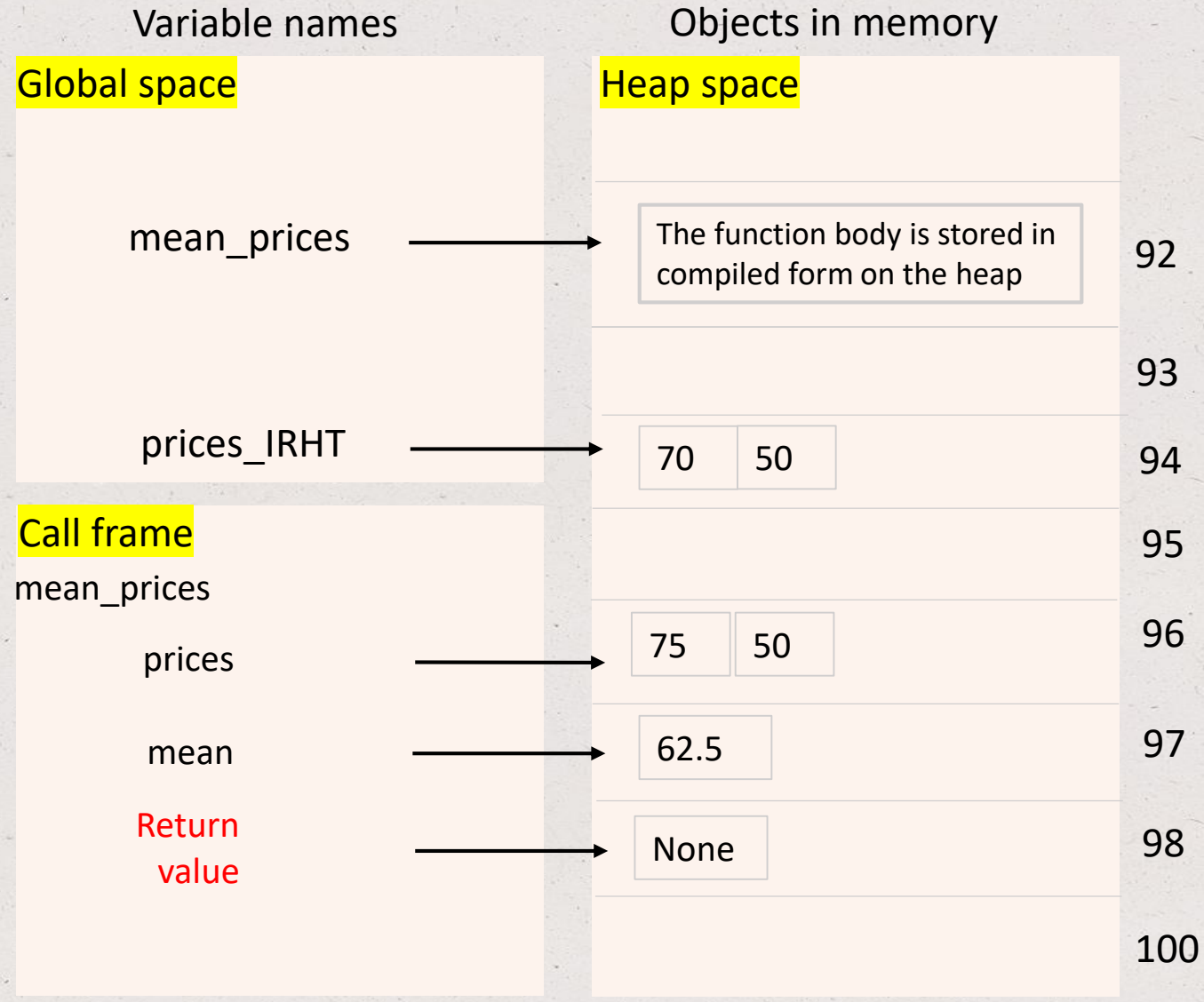
Arguments: mutability Ex 2 – Memory rep

```
def mean_prices(prices):
    prices[0] = 75
    mean = (prices[0] + prices[1])/2


prices_IRHT = [70, 50]

mean_prices(prices_IRHT[:])
print(prices_IRHT)
```

pass a copy of the object

Output

[75, 50]



Top Hat Question # 17

What is the output?

```
def mean_prices(prices):
    prices[1] = 75
    mean = (prices[0] + prices[1])/2


prices_IRHT = [70, 50]

mean_prices(pricesIRHT)
print(prices_IRHT)
```

Top Hat Question # 18

What is the output?

```
def mean_prices(prices):
    prices[1] = 75
    mean = (prices[1] + prices[2])/2


prices_IRHT = [70, 50]

mean_prices(prices_IRHT)
print(prices_IRHT)
```

Function comments

- AAE 875 and good practice in general

```
def mean_prices(price1, price2):
    """
    # Calculates the average of two prices
    # @param price1 First price value
    # @param price2 Second price value
    # @return average of price1 and price2
    """
    return (price1 + price2)/2
```

An explanation of the method
@param: an explanation for each parameter
@return: an explanation of the value returned

Chapter 7: More on Strings

- Slicing (with stride)
- Useful methods

String slicing

- my_string[start : end] – characters from indices **start to end – 1**
- my_string[start : - end] – all from indices **start to end** but the last –*end* characters
- my_string[: end] – characters from indices **0 to end – 1**
- my_string[start:] – characters from **start to end** of the string
- my_string[: -1] – all but the last character

Top Hat Question # 19

What is the output?

```
my_string = 'Hello world'

print(my_string[0 : 5])
print(my_string[0 : -5] + '!')
print(my_string[6: ])
```

Top Hat Question # 20

What is the output?

```
my_string = 'Hello world!'

print(my_string[0 , 5])
print(my_string[0 , -5] + '!')
print(my_string[6: ])
```

String slicing with stride

- The stride determines how much to increment the index after reading each element
- Defaults to 1 if no stride specified
- Syntax: my_string[start : end : **stride**]
- my_string[start : end : 2] – reads every other element between start and end – 1

String useful methods

- Finding and replacing
- Counting
- Comparison
- Splitting
- Joining
- Formatting

Useful string methods

- **Find and Replace**
  - str.replace(*old, new [, count]):* returns a copy of the string with all occurrences of substring *old* replaced by *new*. If the optional argument *count* is given, only the first *count* occurrences are replaced

```
my_string = 'Hello world, Hello!'
my_string = my_string.replace('Hello', 'Hey')
print(my_string)
```

```
Hey world, Hey!
```

Remember that strings are immutable objects. To update a string variable, a new string must be created.

```
my_string = 'Hello world, Hello!'
my_string = my_string.replace('Hello', 'Hey', 1)
print(my_string)
```

```
Hey world, Hello!
```

Useful string methods

- **Find and Replace**
  - str.replace(*old, new [, count]*): returns a copy of the string with all occurrences of substring *old* replaced by *new*. If the optional argument *count* is given, only the first *count* occurrences are replaced
  - str.find(*sub[, start[, end]]*): returns the lowest index in the string where substring *sub* is found, such that *sub* is contained in the range [*start, end*]. Optional arguments *start* and *end* are interpreted as in slice notation.

    Return -1 if *sub* is not found

```
my_string = 'Hello world, Hello!'
print(str.find('Hello'))
```

```
0
```

Useful string methods

- **Find and Replace**
  - str.replace(*old, new [, count]*): returns a copy of the string with all occurrences of substring *old* replaced by *new*. If the optional argument *count* is given, only the first *count* occurrences are replaced
  - str.find(*sub[, start[, end]]*): returns the lowest index in the string where substring *sub* is found, such that *sub* is contained in the range [*start, end*]. Optional arguments *start* and *end* are interpreted as in slice notation.

    Return -1 if *sub* is not found
  - str.rfind(): same as str.find(*sub[, start[, end]]*) but searches the string in reverse, returning the last occurrence of the string

Useful string methods

- **Find and Replace**
  - str.replace(*old, new [, count]*): returns a copy of the string with all occurrences of substring *old* replaced by *new*. If the optional argument *count* is given, only the first *count* occurrences are replaced
  - str.find(*sub[, start[, end]]*): returns the lowest index in the string where substring *sub* is found, such that *sub* is contained in the range [*start, end*]. Optional arguments *start* and *end* are interpreted as in slice notation.

    Return -1 if *sub* is not found
  - str.rfind(): same as str.find(*sub[, start[, end]]*) but searches the string in reverse, returning the last occurrence of the string
- **Counting**
  - str.count(sub [, start[, end]]): return the number of occurrences of substring *sub* in string S*[start : end].* Optional arguments *start* and *end* are interpreted as in slice notation

Useful string methods

- **Comparisons (evaluate to Boolean)**
  - Character by character using their ASCII values
    - Relational operators [*<, <=, >, >=*]
    - Equality operators [*==, !=*]
    - Membership operators [*in, not in*]
  - Determine if two variables are bound to the same object
    - Identity operators [*is, is not*]

```
string1 = 'Hello'
string2 = 'hello'
print(string1 == string2)
print(string1 > string2)
print(string1 != string2)
print(string1 is not string2)
```

```
False
False
True
True
```

Top Hat Question # 21

What is the output?

```
my_string = 'Hello world, Hello!'
my_string = my_string.replace('hello', 'hey')
print(my_string)
```

Top Hat Question # 22

What is the output?

```
my_string = 'Hello world, Hello!'
my_string = my_string.replace('hello', 'hey', 3)
print(my_string)
```

Top Hat Question # 23

What is the output?

```
my_string = 'Hello world, Hello!'
print(my_string.count('l'))
```

Top Hat Question # 24

What is the output if user_input is AAE 875?

```
user_input = input(“Enter a class number: \n“)

while user_input != “AAE 875”:
    print('Try again!')
    user_input = input(“Enter a class number: \n“)
else:
    print('Values match!')
```

Top Hat Question # 25

What is the output if user_input is AAE 870?

```
user_input = input(“Enter a class number: \n“)

while user_input != “AAE 875”:
    print('Try again!')
    user_input = input(“Enter a class number: \n“)
else:
    print('Values match!')
```

Top Hat Question # 26

What is the output if user_input is AAE 875?

```
user_input = input(“Enter a class number: \n“)

while user_input is not “AAE 875”:
    print('Try again!')
    user_input = input(“Enter a class number: \n“)
else:
    print('Values match!')
```

Useful string methods

- **Splitting** strings using the **split()** method
  - str.split (*sep = None [,maxsplit = -1]*): returns a **list** of the words (tokens) in the string, using *sep* as the delimiter string. If *maxsplit* splits are done
    - *sep* default value is whitespace characters
    - *sep* value can be changed by calling split() with a string argument

```
string1 = 'Hello, AAE 875 students!'
string2 = string1.split()
print(string2)
```

```
['Hello,' 'AAE', '875', 'students!']
```

```
string1 = 'Hello, AAE 875 students!'
string2 = string1.split(',')
print(string2)
```

```
['Hello', ' AAE 875 students!']
```

Useful string methods

- **Splitting** strings using the **split()** method
  - str.split (*sep = None [,maxsplit = -1]*): returns a **list** of the words (tokens) in the string, using *sep* as the delimiter string. If *maxsplit* splits are done
    - *sep* default value is whitespace characters
    - *sep* value can be changed by calling split() with a string argument
    - If the string starts or ends with the *sep,* or if two *sep* exist, then the resulting list will contain an empty string for each occurrence; no empty strings are generated if *sep* takes the default value

```
string1 = '   Hello, AAE 875 students!'
string2 = string1.split()
print(string2)
```

```
['Hello,' 'AAE', '875', 'students!']
```

Useful string methods

- **Splitting** strings using the **split()** method
  - str.split (*sep = None [,maxsplit = -1]*): returns a **list** of the words (tokens) in the string, using *sep* as the delimiter string. If *maxsplit* splits are done
    - *sep* default value is whitespace characters
    - *sep* value can be changed by calling split() with a string argument
    - If the string starts or ends with the *sep,* or if two *sep* exist, then the resulting list will contain an empty string for each occurrence; no empty strings are generated if *sep* takes the default value

```
string1 = '   Hello,, AAE 875 students!'
string2 = string1.split(',')
print(string2)
```

```
['   Hello,' '', ' AAE 875 students!']
```

Useful string methods

- **Splitting** strings using the **split()** method
  - str.split (*sep = None [,maxsplit = -1]*): returns a **list** of the words (tokens) in the string, using *sep* as the delimiter string. If *maxsplit* splits are done
    - *sep* default value is whitespace characters
    - *sep* value can be changed by calling split() with a string argument
    - If the string starts or ends with the *sep*, or if two *sep* exist, then the resulting list will contain an empty string for each occurrence; no empty strings are generated if *sep* takes the default value

```
string1 = '   Hello, AAE 875 students!\n'
string2 = string1.split('\n')
print(string2)
```

```
['Hello, AAE 875 students!', '']
```

Useful string methods

- **Join** strings using the **join()** method
  - str.join(*seq*): returns a string which is the concatenation of the strings in the sequence *seq*. The separator between elements is the string providing this method

```
classes = ['AAE875', 'AAE720']
print('You can take', ' or '.join(classes), 'this semester.')
```

```
You can take AAE875 or AAE720 this semester.
```

- More on String methods here: https://docs.python.org/3/library/stdtypes.html#string-methods