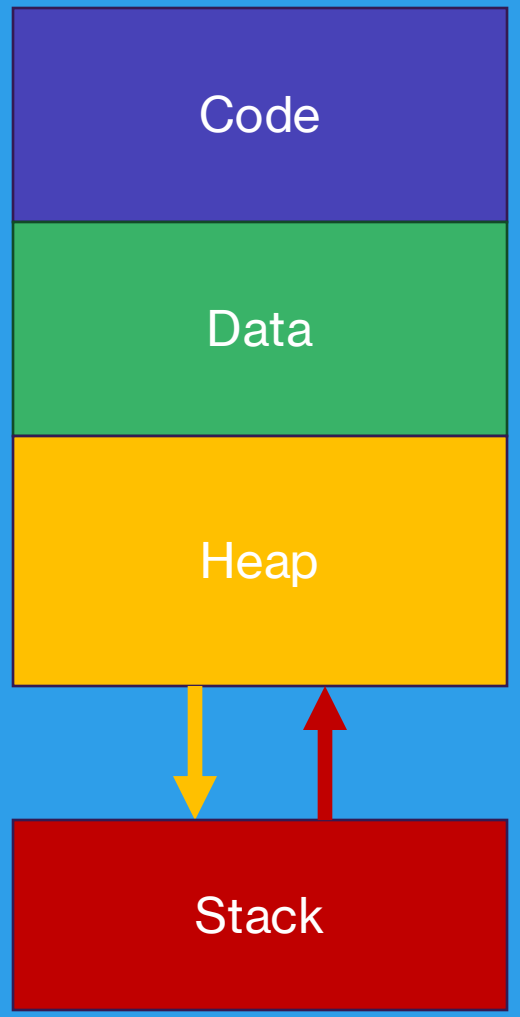
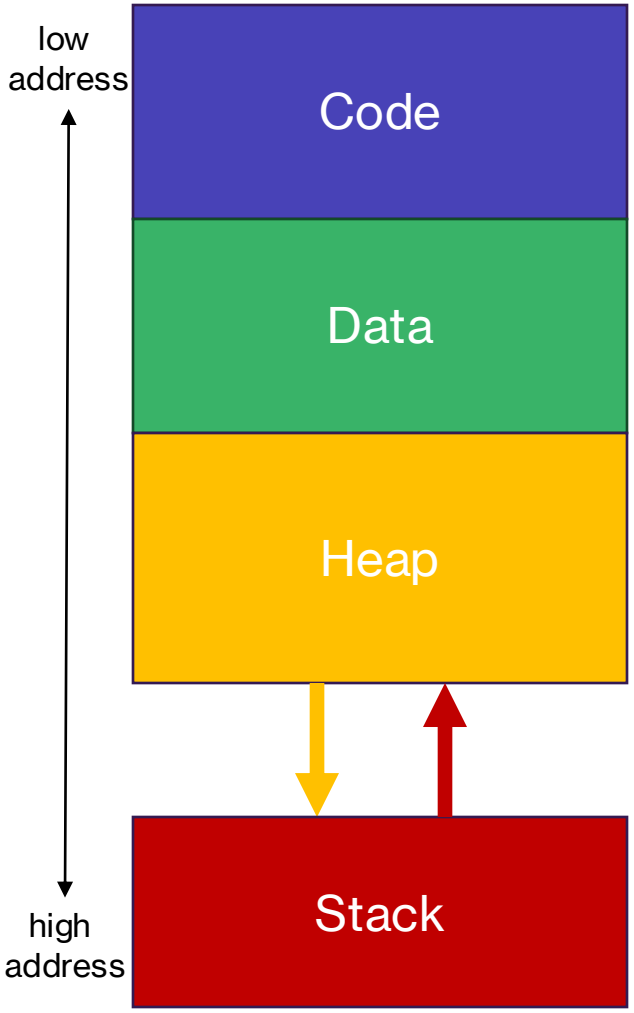
Computer Systems



Call Stack

CS51 – Fall 2025

The organization of memory



- Memory in modern computers is organized in four basic areas:
  - **Code**: Where the instructions of our program are stored.
  - **Data**: Where the global data are stored and live for the entire duration of our program. Think of global data as variables that are created outside a function. The size of this memory region is fixed.
  - **Heap**: Where dynamic user-defined objects, lists, dictionaries, etc are created. The size is automatically adjusted in Python.
    - We will see such data structures later in this course and you will also see how to work with objects (in Java) in CS62.
  - **Stack**: Devoted to managing subprogram calls and stores their local variables. Depending on the language, subprograms are also called *functions*, *methods,* or *procedures*. In Python, subprograms come in the form of functions (and methods if you work with classes).



The stack abstract data type

- A stack is a data structure (a way of organizing and storing data) that maintains a collection of elements.
- A stack is like a stack of dishes or cafeteria trays.
  - Dishes may be placed on the top of the stack or removed from the top of the stack, not the middle.
- If a stack is not empty, we may **pop** the top element.
  - Attempting to pop an element from an empty stack causes a **stack underflow**.
- To add an element to the stack, we **push** it to the top.
  - Pushing to a stack that has reached capacity would cause a **stack overflow.**
- The element popped is the one most recently pushed. This is known as last-in-first-out or **LIFO** principle.

The call stack

- The stack region in memory follows the same idea of the abstract data type of stack to manage function calls and their local variables.
- When our program calls a function, the system reserves memory space for the function needs.
  - The memory space allocated for a function call is called a **stack frame**. The newly-created stack frame is pushed onto the top of the call stack and becomes the active stack frame.
- When the work of the function is completed, the stack frame is popped from the call stack.
- When one function calls another, the work of the caller function pauses, and a new stack frame is being pushed into the call stack for the callee which now becomes the active stack frame.
- When the active stack frame at the top of the call stack is popped because the callee function completed its work, then the frame immediately below it becomes the active stack frame and the work of the caller function continues off where it was paused.

Look at the following Python program

```
def add(num1, num2):
    return num1 + num2

def double(num):
    return 2 * num

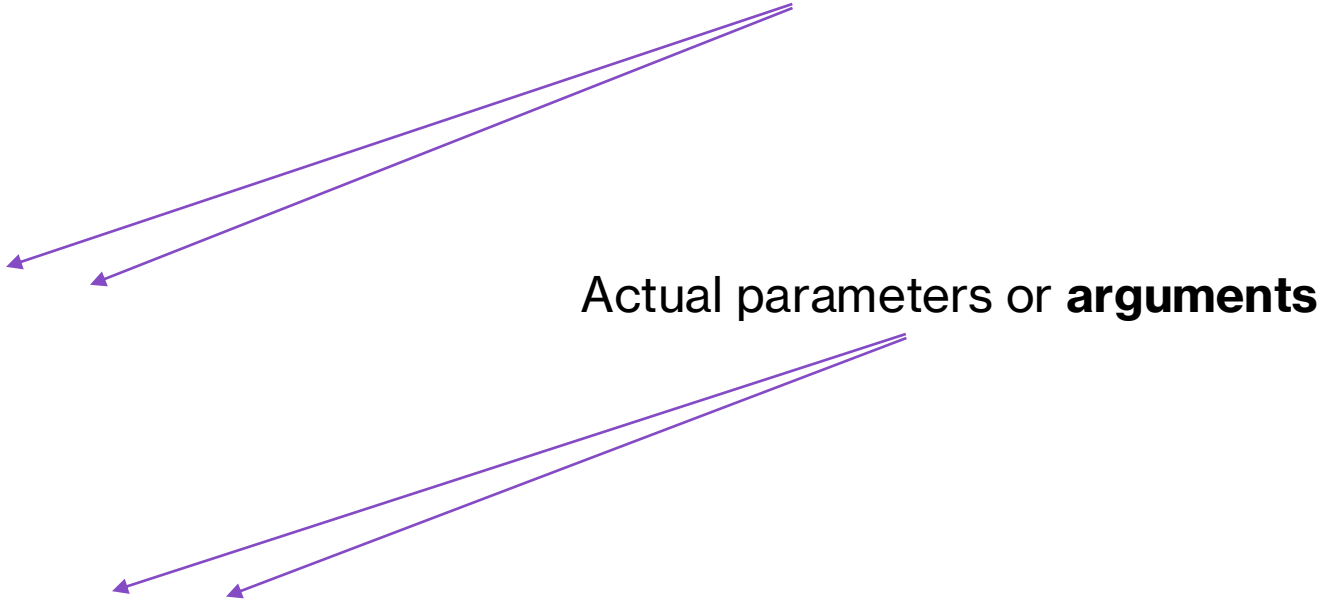
def add_then_double(a, b):
    added = add(a, b)
    return double(added)

def main():
    print(add_then_double(10, 20))

main()
```

Formal **parameters**

Actual parameters or **arguments**



The call stack

```
def add(num1, num2):
    return num1 + num2

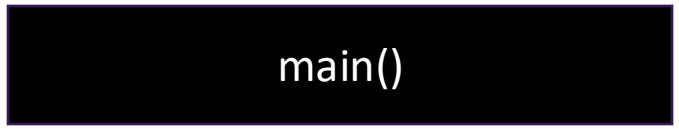
def double(num):
    return 2 * num

def add_then_double(a, b):
    added = add(a, b)
    return double(added)

def main():
    print(add_then_double(10, 20))

main()
```





The call stack

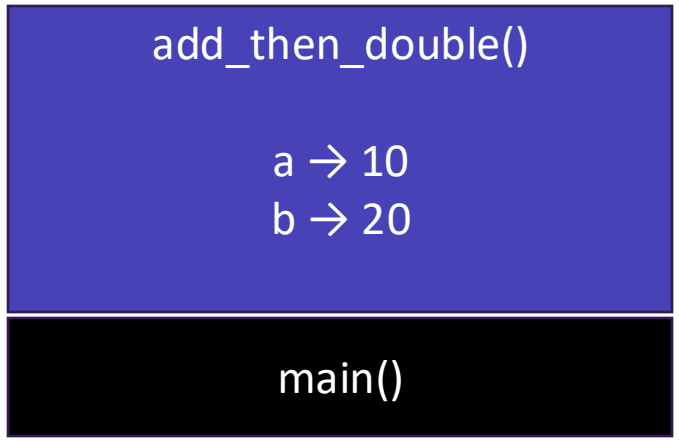
```
def add(num1, num2):
    return num1 + num2

def double(num):
    return 2 * num

def add_then_double(a, b):
    added = add(a, b)
    return double(added)

def main():
    print(add_then_double(10, 20))

main()
```



The call stack

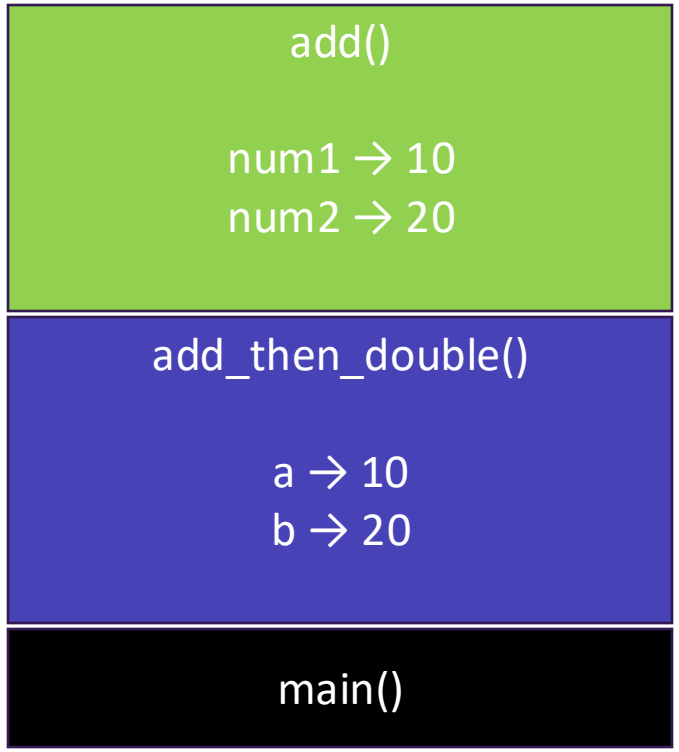
```
def add(num1, num2):
    return num1 + num2

def double(num):
    return 2 * num

def add_then_double(a, b):
    added = add(a, b)
    return double(added)

def main():
    print(add_then_double(10, 20))

main()
```



The call stack

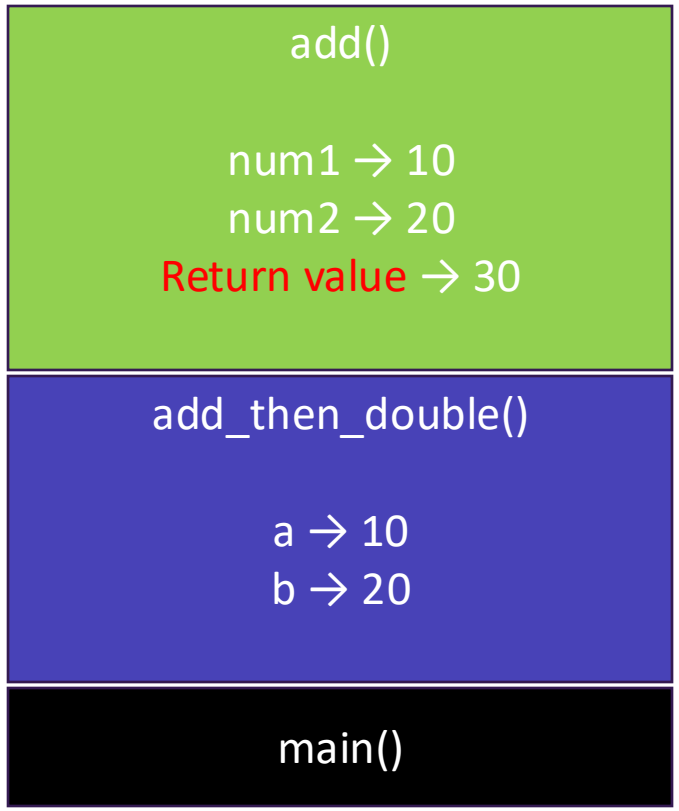
```
def add(num1, num2):
    return num1 + num2

def double(num):
    return 2 * num

def add_then_double(a, b):
    added = add(a, b)
    return double(added)

def main():
    print(add_then_double(10, 20))

main()
```



The call stack

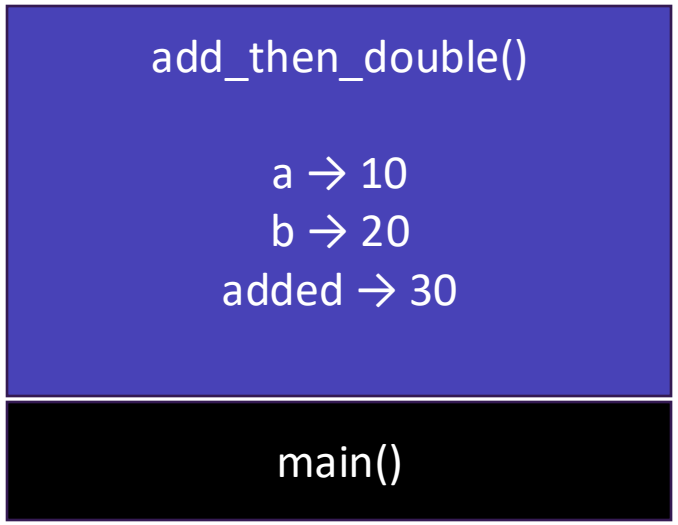
```
def add(num1, num2):
    return num1 + num2

def double(num):
    return 2 * num

def add_then_double(a, b):
    added = add(a, b)
    return double(added)

def main():
    print(add_then_double(10, 20))

main()
```



The call stack

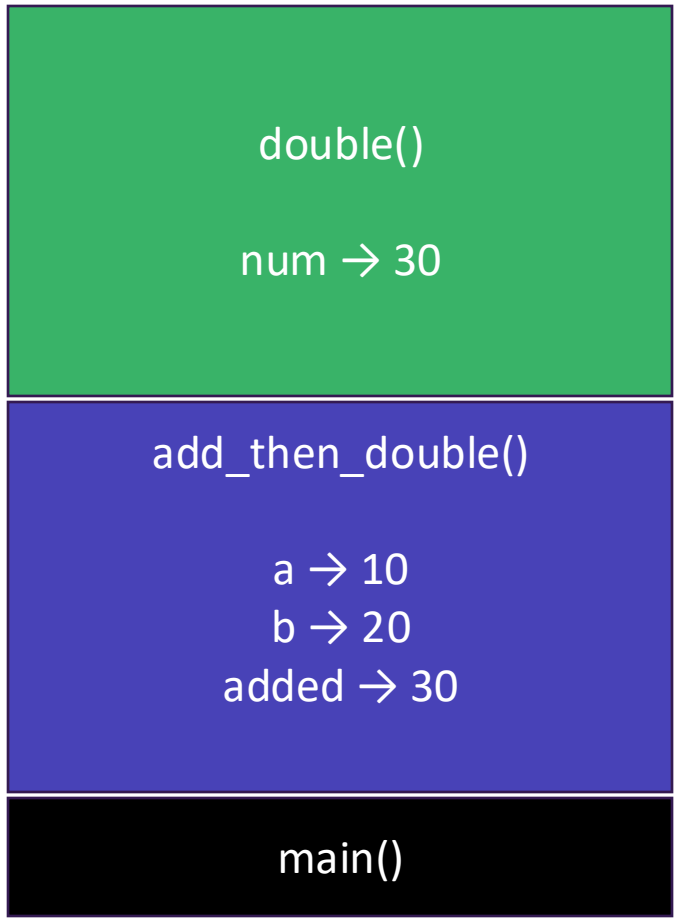
```
def add(num1, num2):
    return num1 + num2

def double(num):
    return 2 * num

def add_then_double(a, b):
    added = add(a, b)
    return double(added)

def main():
    print(add_then_double(10, 20))

main()
```



The call stack

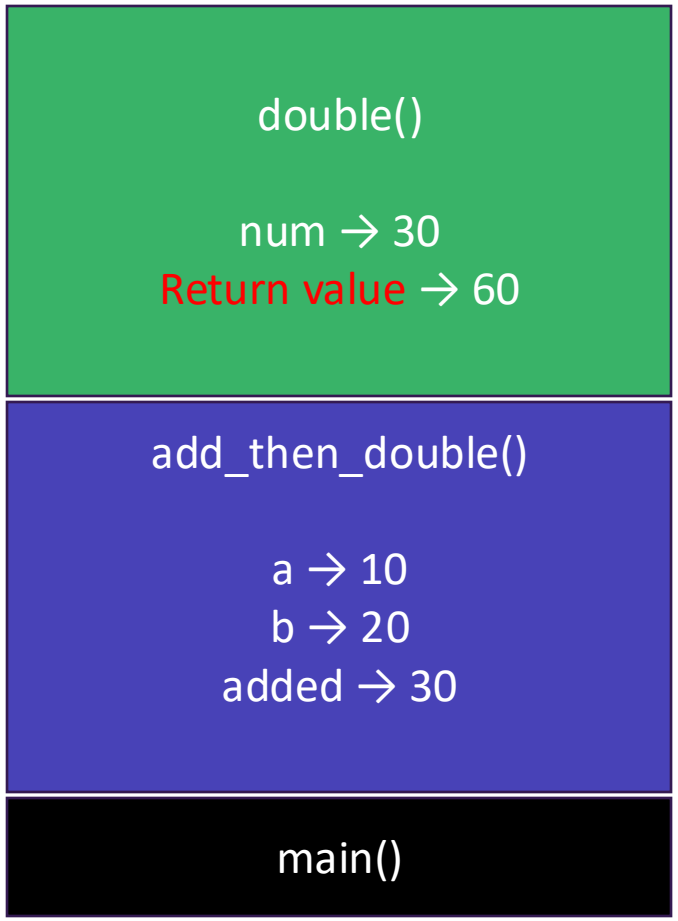
```
def add(num1, num2):
    return num1 + num2

def double(num):
    return 2 * num

def add_then_double(a, b):
    added = add(a, b)
    return double(added)

def main():
    print(add_then_double(10, 20))

main()
```



The call stack

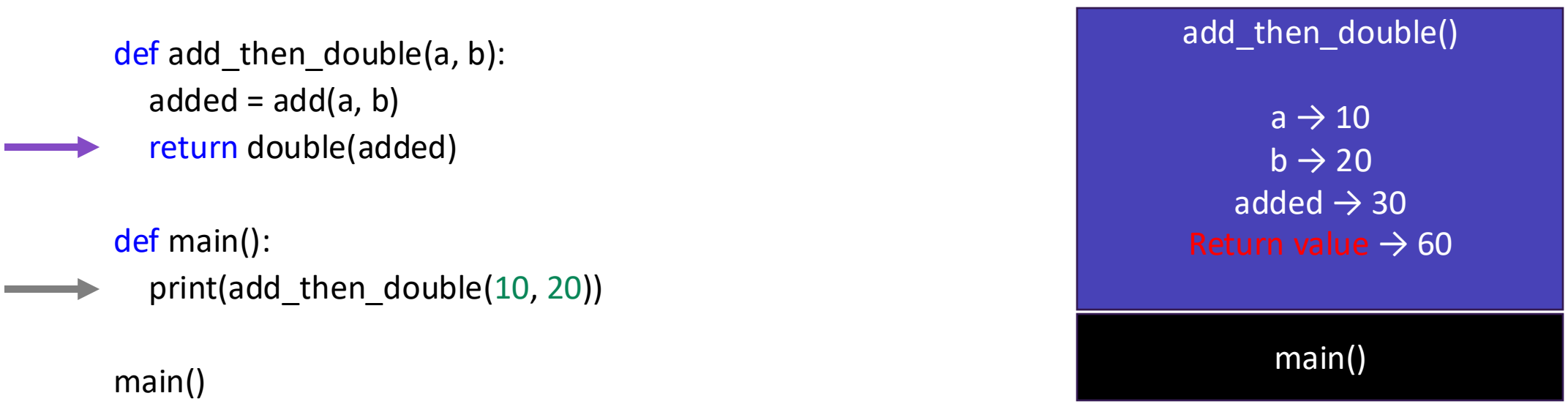
```
def add(num1, num2):
    return num1 + num2

def double(num):
    return 2 * num

def add_then_double(a, b):
    added = add(a, b)
    return double(added)

def main():
    print(add_then_double(10, 20))

main()
```



The call stack

```
def add(num1, num2):
    return num1 + num2

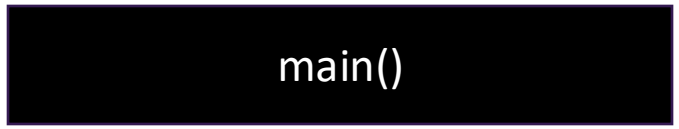
def double(num):
    return 2 * num

def add_then_double(a, b):
    added = add(a, b)
    return double(added)

def main():
    print(add_then_double(10, 20))

main()
```

“60” is printed



The call stack

```
def add(num1, num2):
    return num1 + num2

def double(num):
    return 2 * num

def add_then_double(a, b):
    added = add(a, b)
    return double(added)

def main():
    print(add_then_double(10, 20))

main()
```

The program ends

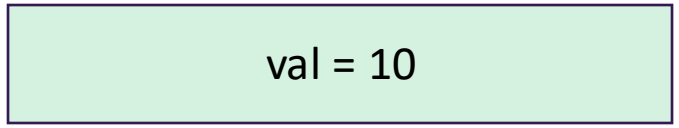
PRACTICE TIME – Visualize the call stack

```
def double_input(val):
    val = 2 * val
    return val


def triple_input(val):
    return val * 3


val = 10
print(double_input(val))
print(triple_input(val))
```

Data region with global variables



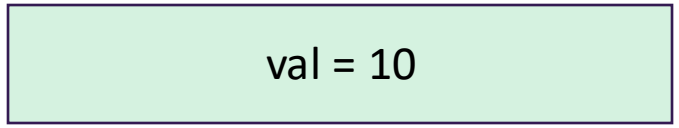
ANSWER – Visualize the call stack

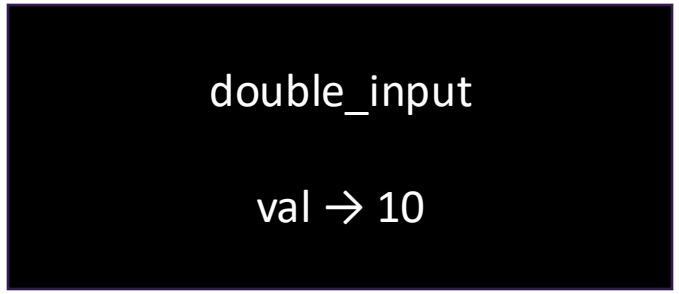
```
def double_input(val):
    val = 2 * val
    return val

def triple_input(val):
    return val * 3

val = 10
print(double_input(val))
print(triple_input(val))
```

Data region with global variables





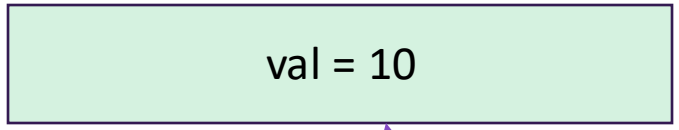
ANSWER – Visualize the call stack

```
def double_input(val):
    val = 2 * val
    return val

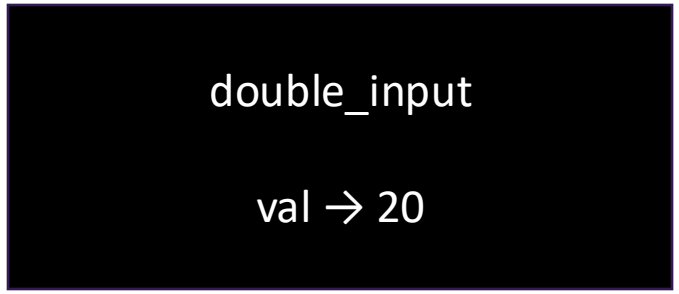
def triple_input(val):
    return val * 3

val = 10
print(double_input(val))
print(triple_input(val))
```

Data region with global variables



Note that the global variable val stays unaffected



ANSWER – Visualize the call stack

```
def double_input(val):
    val = 2 * val
    return val

def triple_input(val):
    return val * 3

val = 10
print(double_input(val))
print(triple_input(val))
```

Data region with global variables

val = 10

double_input
val → 20 Return value → 20

ANSWER – Visualize the call stack

```
def double_input(val):
    val = 2 * val
    return val


def triple_input(val):
    return val * 3


val = 10
print(double_input(val))
print(triple_input(val))
```

Data region with global variables

val = 10

“20” is printed

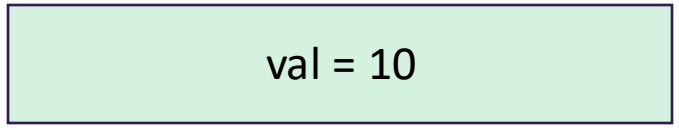
ANSWER – Visualize the call stack

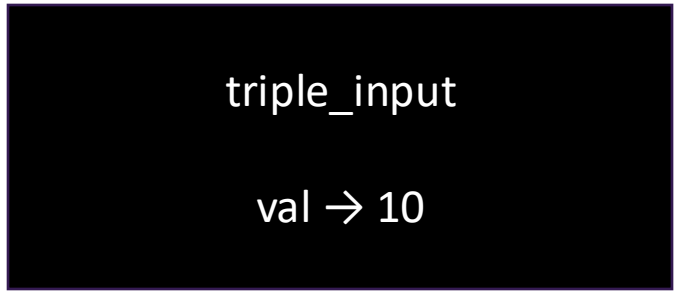
```
def double_input(val):
    val = 2 * val
    return val

def triple_input(val):
    return val * 3

val = 10
print(double_input(val))
print(triple_input(val))
```

Data region with global variables





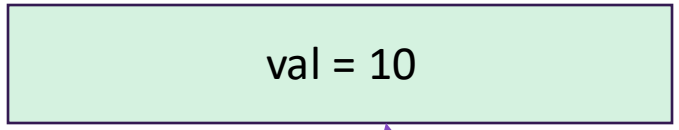
ANSWER – Visualize the call stack

```
def double_input(val):
    val = 2 * val
    return val

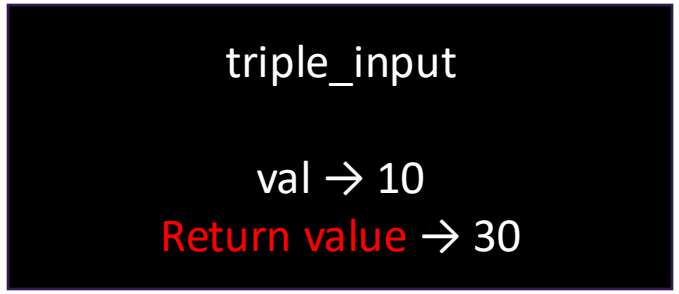
def triple_input(val):
    return val * 3

val = 10
print(double_input(val))
print(triple_input(val))
```

Data region with global variables



Note that the global variable val stays unaffected



ANSWER – Visualize the call stack

```
def double_input(val):
    val = 2 * val
    return val


def triple_input(val):
    return val * 3


val = 10
print(double_input(val))
print(triple_input(val))
```

Data region with global variables

val = 10

“30” is printed

Scope

- This example demonstrated the concept of **scope**. The scope of a variable is the portion of the code we can reference that variable in and have it be valid.
- When we declare a variable in the Python console/shell, e.g., x = 10, what is its scope?
  - The scope is any shell statements/expressions typed after it.
- When we declare a variable (outside a function) in a .py file, what is its scope?
  - anywhere below it in the file. We call such variables global and they reside in the data region of memory.
- When we declare a variable inside a function, what is its scope?
  - anywhere below it but **within the function**. Such local variables reside in the stack frame pushed into the stack for this specific function and cease to exist after the function completes its work. The same applies to function parameters.

Resources:

- Python Tutor can help you with visualizing the call stack and better understanding the scope of variables.

Code:

- function-calls.py
- scope.py

Practice Problem 1

- Assuming unsigned numbers, fill in the rest of the missing entries in the table for representations in other bases.

Binary	Decimal	Hex
100101		
	910	
		1AB

Answer Problem 1

Binary	Decimal	Hex
100101	37	25
1110001110	910	38E
000110101011	427	1AB

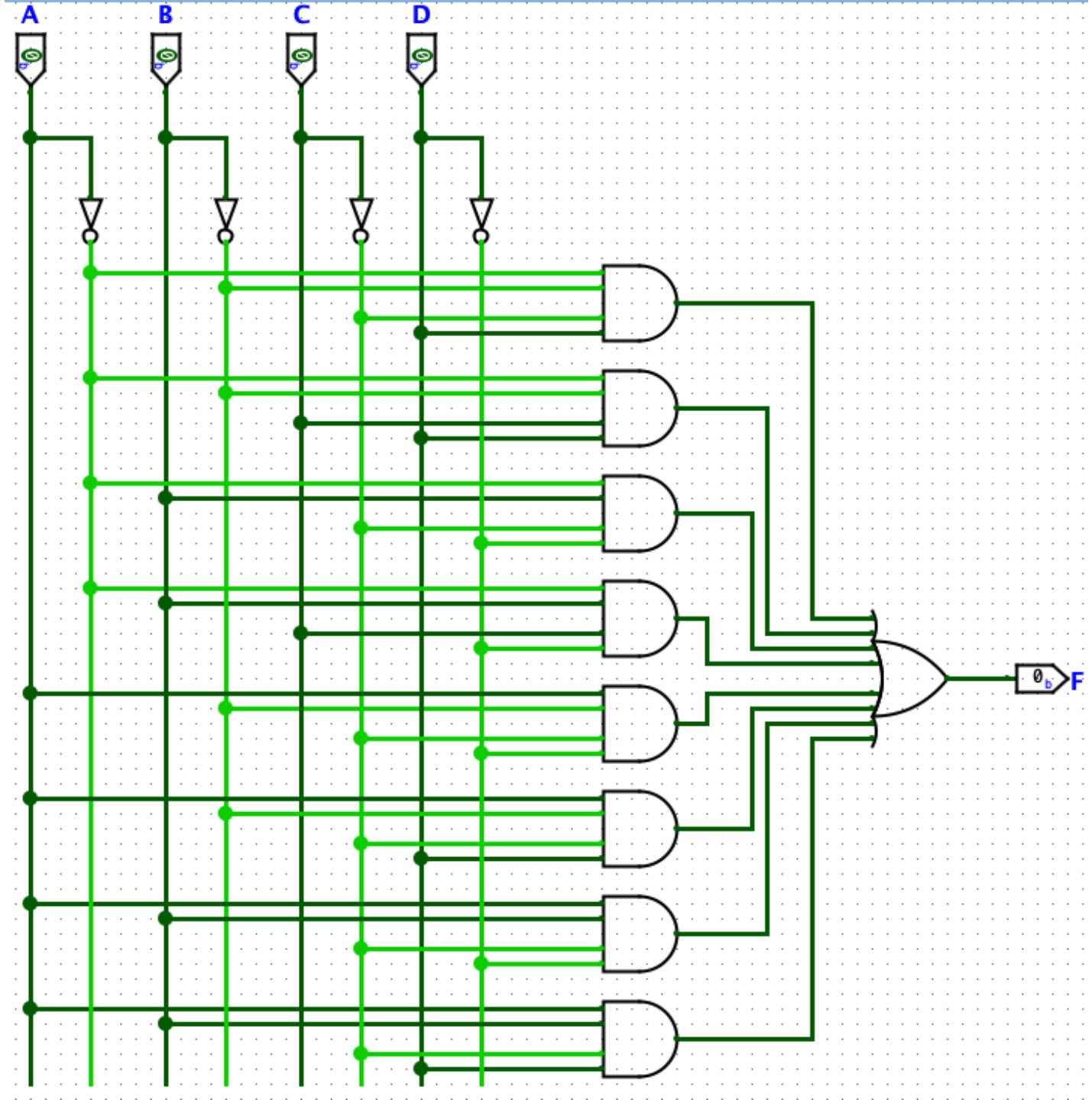
Practice Problem 2

- i) Given this 4-variable Boolean function, calculate is DNF and build the circuit that corresponds to this minterm expansion. ii) use a K-map to simplify it and build the corresponding circuit.

A	B	C	D	Output
0	0	0	0	0
0	0	0	1	1
0	0	1	0	0
0	0	1	1	1
0	1	0	0	1
0	1	0	1	0
0	1	1	0	1
0	1	1	1	0
1	0	0	0	1
1	0	0	1	1
1	0	1	0	0
1	0	1	1	0
1	1	0	0	1
1	1	0	1	1
1	1	1	0	0
1	1	1	1	0

Answer 2

- i) Given this 4-variable Boolean function, calculate is DNF and build the circuit that corresponds to this minterm expansion DNF:
- m1 v m3 v m4 v m6 v m8 v m9 v m12 v m13



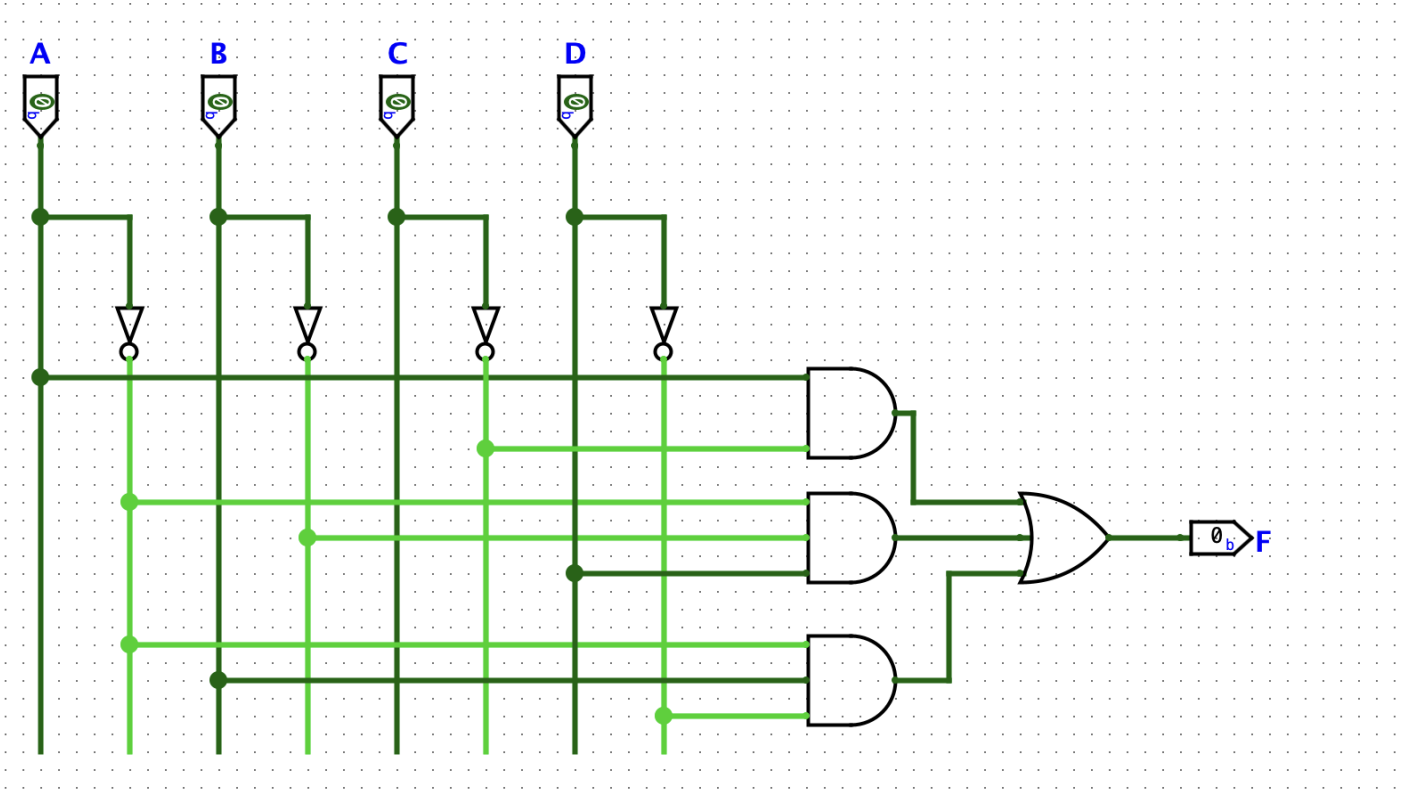
Answer 2

- ii) use a K-map to simplify it and build the corresponding circuit.

AB \ CD	00	01	11	10
00	0	1	1	0
01	1	0	0	1
11	1	1	0	0
10	1	1	0	0

red: $A \wedge \neg C$ green: $\neg A \wedge \neg B \wedge D$ yellow: $\neg A \wedge B \wedge \neg D$

Simplified: $(A \wedge \neg C) \vee (\neg A \wedge \neg B \wedge D) \vee (\neg A \wedge B \wedge \neg D)$



Practice Problem 3

- What would the results of the following bitwise masks be in decimal?
- ~8
- 9&3
- -2|3
- 7^9

Answer 3

- What would the results of the following bitwise masks be in decimal?
- ~8 = -9. Remember, ~x=-(x+1)
- 9&3 = 1
  - 01001 & 00011 = 0001
- -2|3 =-1
  - 110|001=111
- 7^9=14
  - 00111^01001= 01110

Practice Problem 4

- Assuming x is a positive number, what would the following function do?

```
def mystery(x):
    return x & (x - 1)
```

Practice Problem 4

- It resets (sets to 0) the least significant set bit (the right most bit that is 1) in x.