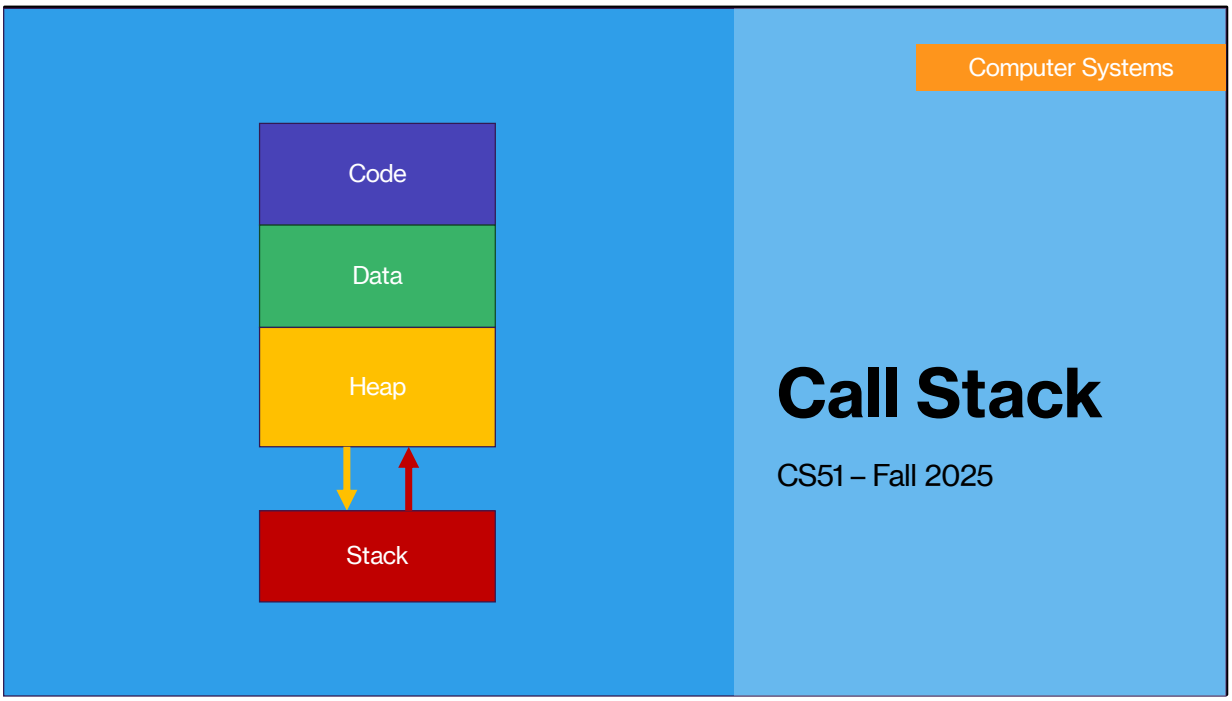
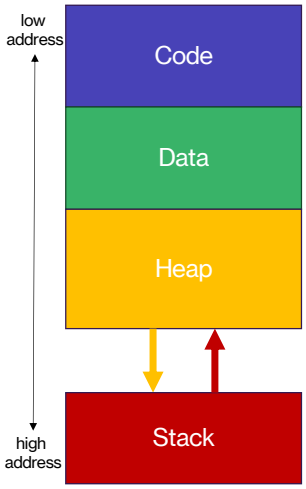


Today we will complete our Computer Systems unit of the class by learning about the call stack.

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).

2

Memory in modern computers is organized in four basic areas. code: where the instructions of program are stored, data, where global variables outside a function live, heap, where dynamic user-defined objects such as lists and dictionaries are created, and stack which is devoted to manage subprogram calls, that is calls to functions and store their local variables. Subprograms are known as functions, methods, or procedures.



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.

3

A stack is a data structure. that maintains a collection of elements like a stack of dishes or cafeteria trays. If a stack is not empty, we may pop, that is remove, the top element. If the stack had been empty we would have a stack underflow. When adding an element, we push it to the top. If we are working with a certain capacity, adding elements beyond the capacity would result to stack overflow. The key property of stacks is that they follow the LIFO, last in, first out principle. The last element we added to the stack is the first one to be removed.

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.

4

The stack region in memory follows the same idea of the abstract data type of stack we just saw, in order 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

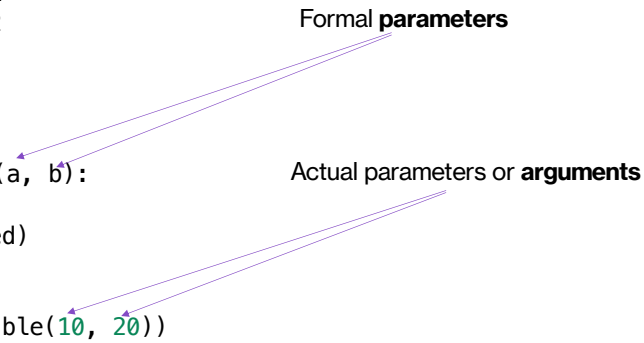
```
def add(num1, num2):
  return num1 + num2

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()
```



5

Look at the following Python program. We have four functions, main, who calls add_then_double, add_then_double which first calls add and then pass its result to double and which returns it. As a point of vocabulary here, a and b would be formal parameters. When invoking a function and passing actual parameters like 10 and 20, we call these arguments. OK, what should this program print?

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 begins with calling the main function which results in a stack frame for main being pushed into the call stack

The call stack

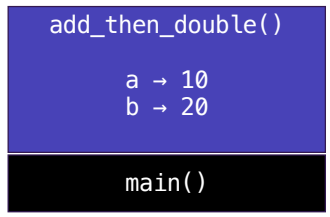
```
def add(num1, num2):
  return num1 + num2

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()
```



7

The main method is responsible for printing what the add_then_double function when called with arguments 10 and 20 will return. A new stack frame for add_then_double with the arguments 10 and 20 for the associated parameters a and b is being added to the top of the stack. The add_then_double stack frame becomes the active frame and main remembers where it paused.

The call stack

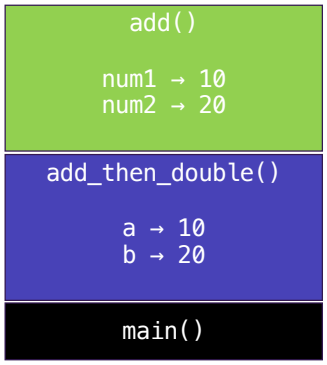
```
def add(num1, num2):
  return num1 + num2

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()
```



8

Add_then_double calls the function add so a new stack frame with the arguments 10 and 20 associated for the parameters num1 and num2 is being created for the add function.

The call stack

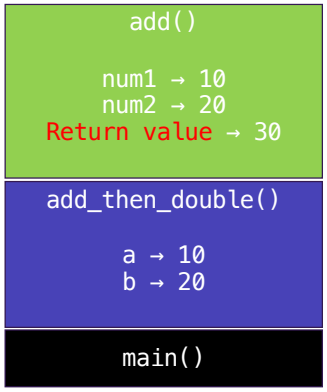
```
def add(num1, num2):
  return num1 + num2

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()
```



9

add completes its work and returns 30 to the function that called it, i.e. add_then_double.

The call stack

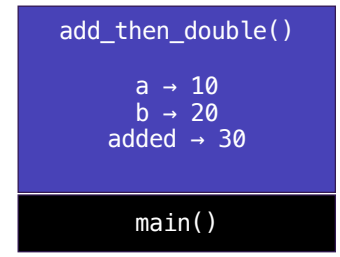
```
def add(num1, num2):
  return num1 + num2

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()
```



10

That results in the stack frame being popped. Now the top of the stack is add_then_double stack frame and the local variable added is initialized to 30.

The call stack

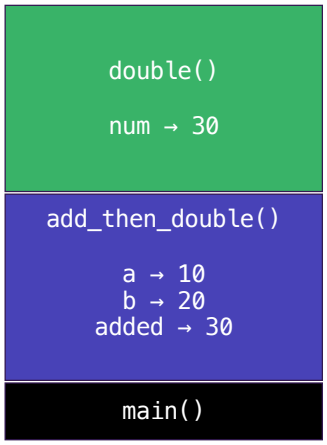
```
def add(num1, num2):
  return num1 + num2

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()
```



11

The last line of add_then_double calls the double function which results in a new stack frame being pushed, associating the parameter num with the argument 30.

The call stack

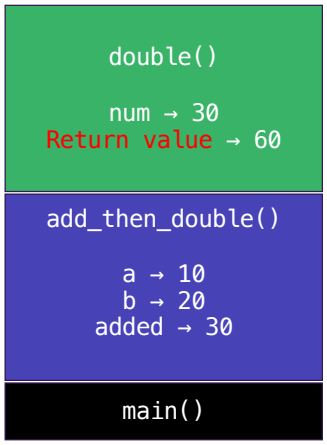
```
def add(num1, num2):
  return num1 + num2

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()
```



12

double returns 2*30=60

The call stack

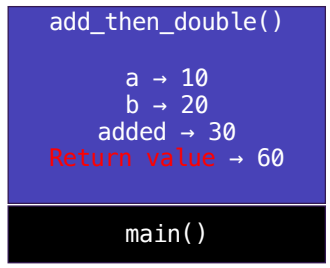
```
def add(num1, num2):
  return num1 + num2

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()
```



13

Its stack frame is popped and that results in add_then_double returning 60.

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



14

The stack frame is popped and main prints 60.

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

15

Finally, the stack frame for main is popped and the program ends.

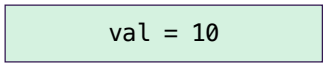
PRACTICE TIME – Visualize the call stack

```
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



16

Your turn now. Visualize the call stack. Note that as a first step, the data region with global variables holds that val=10. What would happen after that?

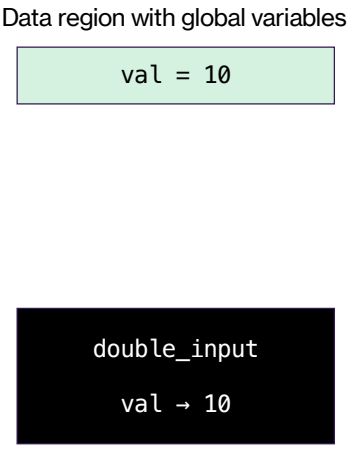
ANSWER – Visualize the call stack

```
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))
```





17

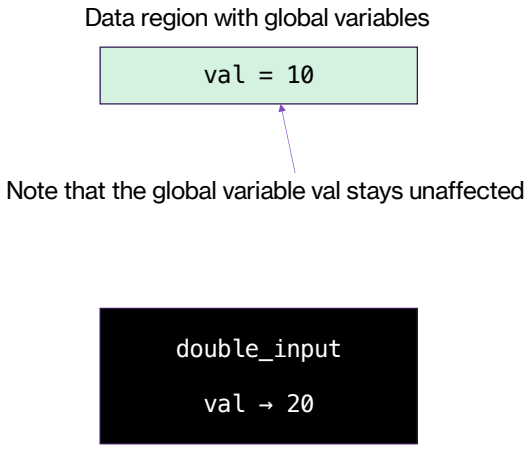
the print statement needs to first find what the return value for double_input is. This adds a stack frame to the call stack with the val being the parameter for the function double_input being associated with the argument 10.

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))
```



18

Note that the global variable val stays unaffected and is different that the local variable val in the stack frame double_input.

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
```

19

double_input updates val to 20 and returns that as a result to be 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

val = 10

"20" is printed

20

The stack frame of double_input is popped and 20 is printed.

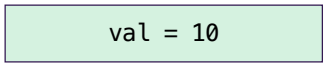
ANSWER – Visualize the call stack

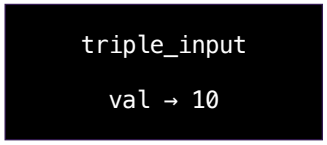
```
def double_input(val):
    val = 2 * val
    return val


def triple_input(val):
    return val * 3


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

Data region with global variables





21

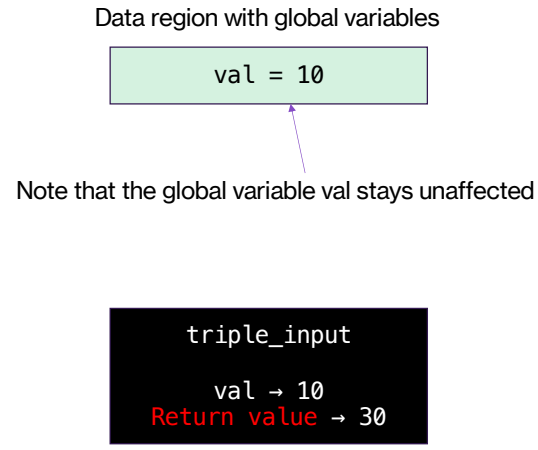
We continue with the next print statement which invokes the triple_input function which results in a new stack frame being added. The parameter val is associated with the argument 10

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))
```



22

Again notice that the global variable val stays unaffected. This is because triple input locally updated val to 30 and returned it.

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

23

This will result in 30 being printed! val remains 10.

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.

24

This example demonstrates the concept of scope which deals with what portion of the code can "see" a variable. When we declare a variable in the Python shell, its scope is any statements/expressions typed after it for the duration of that active console. When we declare a global variable in a py file, anywhere below it in the file we can reference it as it resides in the data region of the memory. When we declare a variable inside a function its scope is anything below it but within the function. These variables are only visible within the function and are removed from the stack when the work of the function is complete. Same for 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

25

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

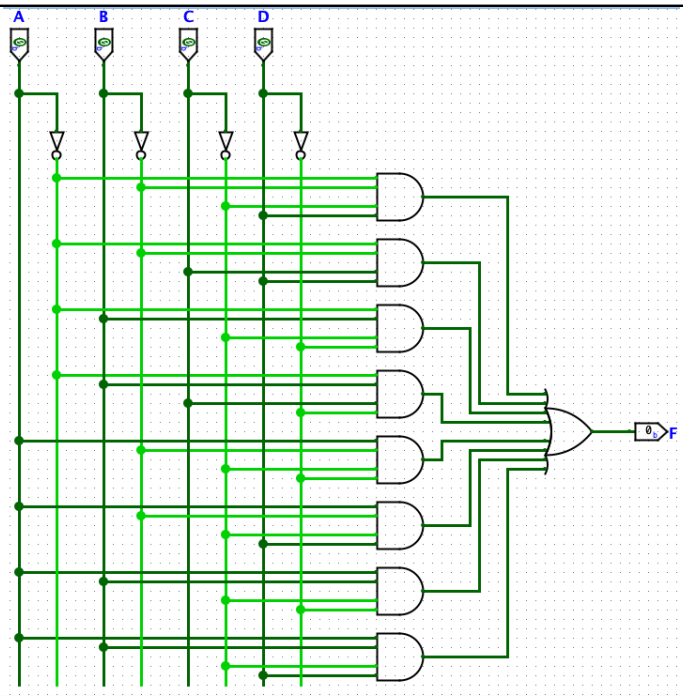
- i) Given this 4-variable Boolean function, calculate is DNF and build the circuit that corresponds to this minterm expansion. ii) use a K-map to simplify it and build the corresponding circuit.

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

28

Answer 2

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



29

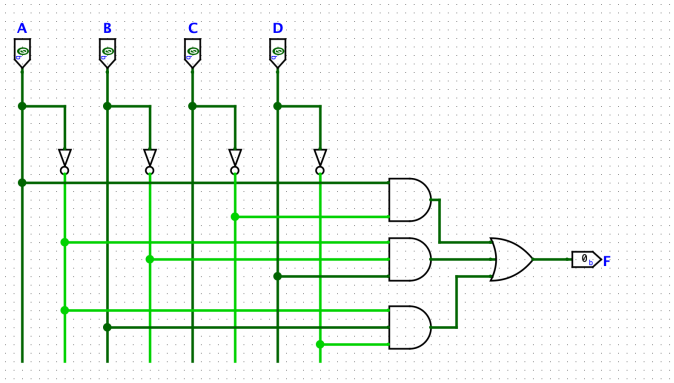
Answer 2

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



30

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.