Recursion

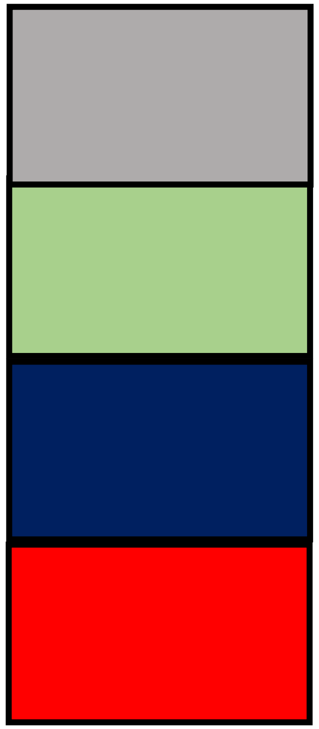
- Recursion is the process of repeating a rule or a definition <u>**by itself**</u> through a “task” **NOT USING LOOP**
- Extremely useful tool to solve large complicated problems
- Recursion works by breaking down a problem into smaller pieces and then solving the smaller pieces
- Benefits of breaking down a problem :

i. It is easier to solve smaller problems

ii. The solutions of the smaller problems can be joined to find the solution of the original problem

- Let the task given to you is build a 4 storied building.
- You are smart and you build one floor and asked your friend to do the same task, build, and this time 3 floors.
- Your friend is smart like you and he also built one floor and asked another friend to build 2 floors.
- The third did the same thing and gave to a 4th friend to build the one floor.
- 4th friend built the 1 floor and there is no floor remaining.

✔ It was easier to build one floor.
✔ Building one floor by friend actually solved the original problem of building 4 floors.



Analysis of the building problem

- If we recall the definition of recursion, it was repeating a rule by itself.
- Building a 4-storied building was the actual task
- The rule followed, was *building one floor and pass the rest of the floors to a friend* and this was repeated by all persons.
- “by itself” means this rule was forced within the task not using loops. [*This statement will be clear once you see a piece of code*]
- Recursion has 2 parts.

i. Recurrence Equation/Rule

ii. Base case

Parts of Recursion

- *Build one floor and if there are more floor to be build, pass the remaining to the next person* – is the recurrence equation, the rule that is repeated with the problem size decreasing at each step. Initially the problem size was 4 then at each step it was reduced by1 until the problem was 1.
- *Build one floor and if there are no more floors remaining, return it to the previous step or to the person who ordered it* – this is the base case. Base case is the very last step of the task where the problem size is in the smallest portion that cannot be further broken down.

More Formal Approach

- We need to convert the recurrence equation and the base case into some mathematical form.
- In the building problem we can write a method to do the task and call it build(n), where n is the problem size.
- Let us rewrite the rule as:

```
if (n>1):
    return (do 1 floor + build(n-1))
else
    do 1 floor and return
```

Basics of methods – Void Method

- Methods can be of 2 types: 1) That returns nothing which we call void and 2) that returns a value which we call a return type

```
def met1():
   x = 1
   print (“met 1”);
   met2(x);
   print (x);

def met2(y):
  print (“met 2”);
  y = y + 1
  met3();
  print(y);

def met3():
  print (“met 3”);
```

- Methods on the left do not return any values.
- If we call met1(), the output of the program would be

  met1 met2 met3 2 1

<u>? Question ?</u>

How does the program know that after completing met3() if must return to met2() not met1() ?

Answer is Stack!!

Basics of Methods – Use of Stack

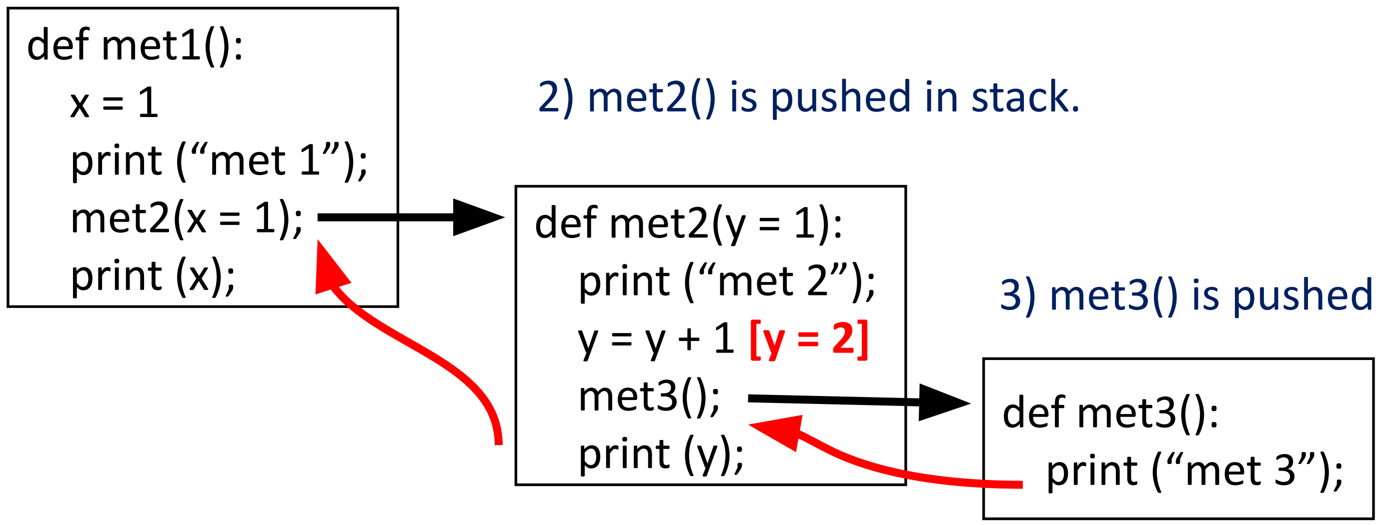
```
def met1():
   x = 1
   print ("met 1");
   met2(x);
   print (x);

def met2(y):
  print ("met 2");
  y = y + 1
  met3();
  print(y);

def met3():
  print ("met 3");
```

- Every method call is saved in stack. To be elaborate, the values of local variables, mathematical operations and so on.

1) met1() is called therefore met1() is pushed in stack.



```
def met1():
   x = 1
   print ("met 1");
   met2(x = 1);
   print (x);
```

2) met2() is pushed in stack.

```
def met2(y = 1):
   print ("met 2");
   y = y + 1 [y = 2]
   met3();
   print (y);
```

3) met3() is pushed in stack.

```
def met3():
   print ("met 3");
```

At the end of met3(), it is popped out of stack and according to LIFO the stack now points to met2() and exactly to the end of line number 4. Line 5 will be executed and met2() will be popped and stack now points to met1().

Basics of Methods – Use of Stack

?? Question ??

If a method call is a push what statement invokes the pop ??

Answer is return!!!

```
def met1():
    x = 1
    print ("met 1");
    met2(x);
    print (x);
```

```
def met2(y):
   print ("met 2");
   y = y + 1
   met3();
   print(y);
```

```
def met3():
   print ("met 3");
```

But nowhere in the code I have written the command return.
~ Because there is invisible return command at the end of every method that returns no value. As there is no value returned developers made it easier for us not to write return.
Try writing the command return at the end of a void method like the one I did it in met1(). See the output.

```
def met1():
    x = 1
    print ("met 1");
    met2(x);
    print (x);
    return
```

Return Type Methods

?? Question ??

How to know when to use a void method and a return type method ?

The answer is obvious, does methodA() need a value(s) to be completed by some other methods in order to fulfil methodA()'s task ? Look at the scenario below.

x = 5 + methodB();

In order to compute x, the second operand must be returned by methodB().

?? Question ??

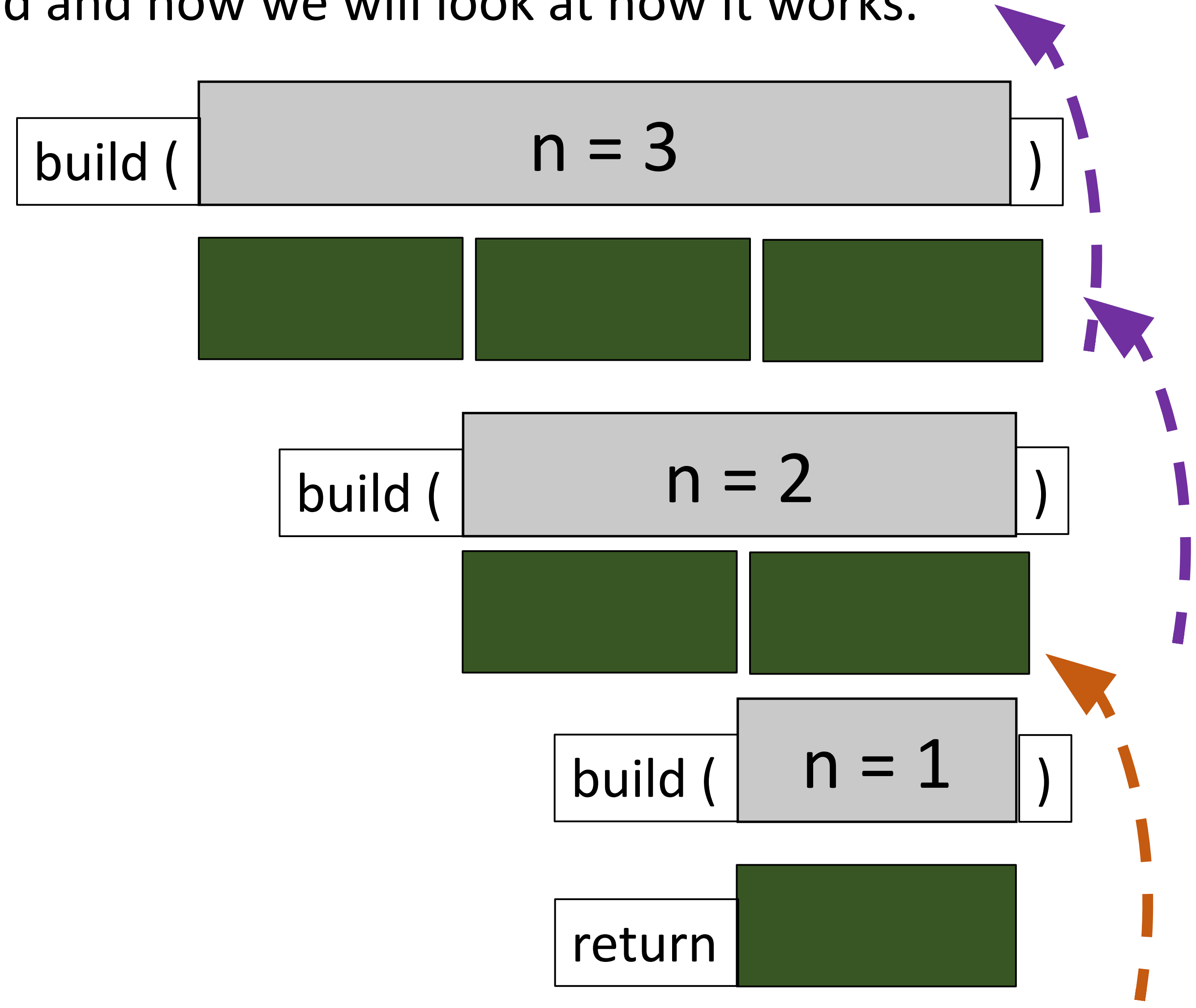
Should the method build(n) in our building problem be of void or return type?

If return, what value is expected to be returned at the end of the method?

The build(n) Method

- Recall what build(n) method did and now we will look at how it works.

```
if (n>1):
  return (do 1 floor + build(n-1))
else
  do 1 floor and return
```



Finally the build(n) Method

```
def build(n):
    if (n>1):
        return 1 + build(n-1)
    else:
        return 1
```

Giving our build(n) method a proper python syntax.