User-defined functions

- A predefined function is just a function someone else wrote and put in a module
- Programs are made up of multiple functions
  - Putting all your code in one file top-to-bottom is very hard to work with
  - Functions let you organize and reuse code

Parts of a function definition

```
def cube(x):
  # what the function does goes here as a
  # code block (all indented)
```

Function heading	`def cube(x):`
Name of the function	`cube`
List of parameters, with types	`(x)`
Function body	`The indented block`

Writing **cube(x)**

```
def cube(x):
  c = x * x * x
  return c
```

- In the function body you can put any statements
  - All the things we did outside functions work inside functions
- Functions work with their parameters
  - If you're not going to do something with x, why pass it in?
- The return statement:
  - Ends the function immediately
  - Returns the specified value (the function call evaluates to that value)

Alternative **`cube(x)`**

```
def cube(x):
  c = x * x * x
  return c

def cube(x):
  return x * x * x
```

Call and definition

- There are two distinct viewpoints on every function
  - The function call (outside)
    - Call by name
    - Provide (*pass in*) input parameters or *arguments*
    - Get back the return value and do something with it
  - The function definition (inside)
    - Receive the parameters
    - Do something with them (and also local variables)
    - Return (*pass out*) a value

Parameters

- *Formal parameters*
  - Used inside the function
  - Declared by name in the function heading
  - E.g. `x` in `def cube(x):`
- *Actual parameters*
  - Passed from outside in the function call
  - Must match the number and types of the formal parameters
  - E.g. `5` in `cube(5)`
- Each actual parameter provides a value for a formal parameter
  - `x` gets the value `5`

Functions, variables and memory

- Each function has its own *scope*
  - Space where all its variables exist
  - There is also a scope for the module
  - When a function is called, new space is created for its parameters and variables
  - When a function ends, those parameters and variables are discarded

Functions, variables and memory

- Consider this function definition and call:

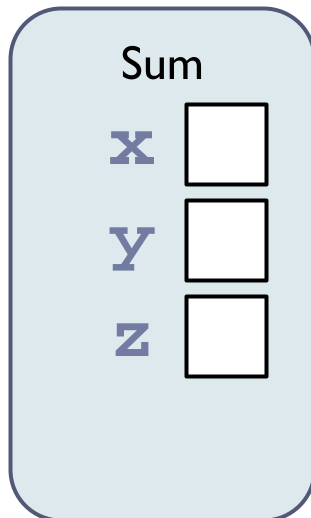
```
def sum_three(x, y, z):
      sum = x + y + z
      return sum

sum = sum_three(5, 6, 7)
```

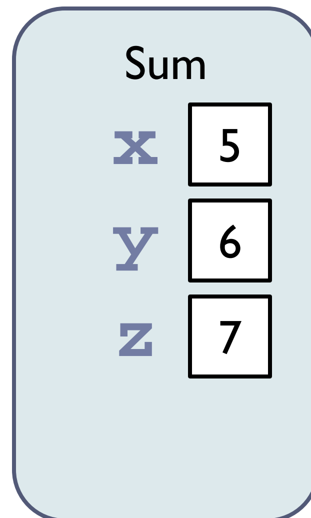
- x, y and z are the formal parameters inside the function
- 5, 6 and 7 are the values being passed in from outside

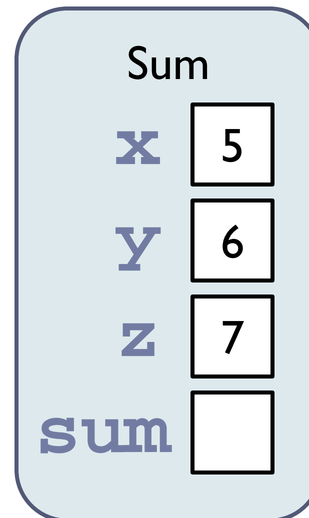
Functions, variables and memory

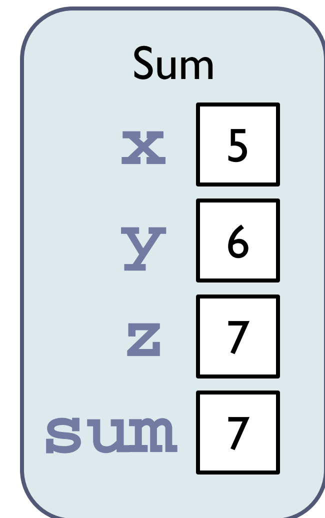
- When the function call is made:

1. Create the formal parameters
2. Assign actual parameter values
3. Create the local variable sum
4. Calculate and assign the sum
5. Return the sum (all variables discarded)









Functions, variables and memory

- Local variables and parameters inside a function are specific to that function!
  - They don't exist outside, which is why values must be passed in and returned
  - Functions cannot use variables declared in another function (even main)
    - We say that they are *out of scope*

- Variables with the same name in different functions are separate, distinct variables!

Using Functions

- Functions are like building blocks
- They allow complicated programs to be divided into manageable pieces
- Some advantages of functions:
  - Can be re-used (even in different programs)
  - A programmer can focus on just that part of the program and construct it, debug it, and perfect it
  - Different people can work on different functions simultaneously
  - Enhance program readability