

## writing your own function.

### Week 2 | Lecture 2 (2.2)

#### Upcoming

- Lab 0 Due 11:59 pm Friday.
- Lab 1 Released Thursday 6:00 pm.
- Reflection 2 Released Friday 6:00 pm.
- Tutorial and Practical sessions running all week.
- TUT0104 Recording posted on OCCS.

if nothing else, write **#cleancode**.

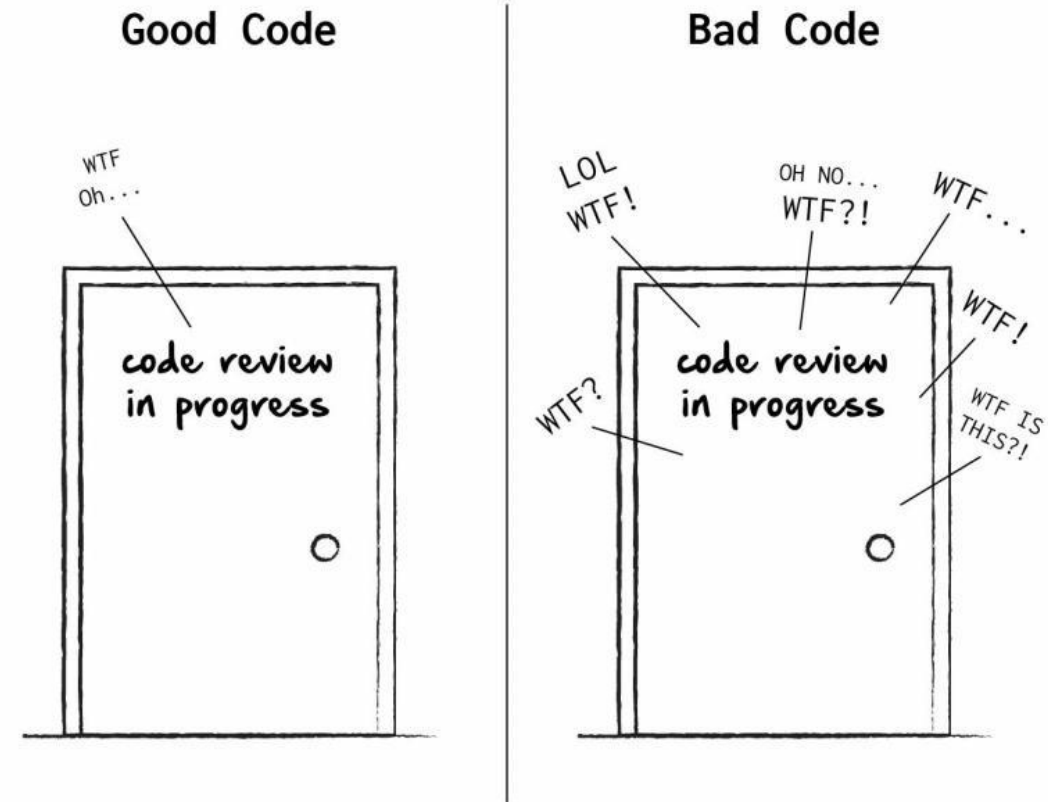
# This Week's Content

- **Lecture 2.1**
  - Functions, input & output, importing modules
- **Lecture 2.2**
  - **Defining your own function**
- **Lecture 2.3**
  - Design Problem: Forward Kinematics

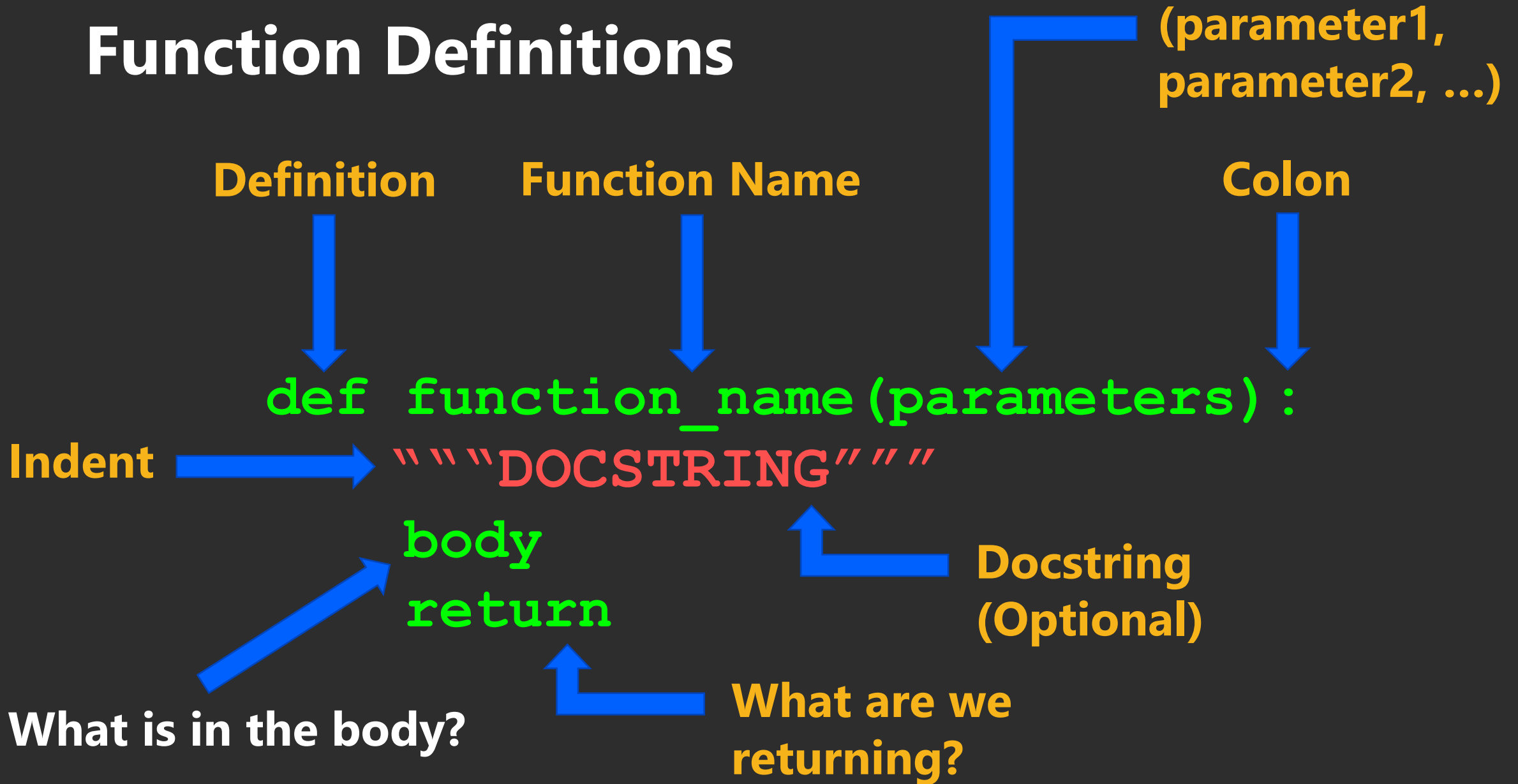
# Defining Your Own Functions

- The real power of functions is in defining your own.
- Good programs typically consist of many small functions that call each other.
- If you have a function that does **only one thing** (like calculate the sine of an angle), it is likely not too large.
- If its not too large, it will be easy to test and maintain.

**Code quality**  
is measured in WTFs/min



# Function Definitions



# Function Definitions

```
def function_name(parameters) :  
    body  
    return
```

- **def** - is a keyword, standing for "definition". All function definitions must begin with **def**. The **def** statement must end with a colon.
- **function\_name** - is the name you will use to call the function (like `sin`, `abs` but you need to create your own name).
- **parameters** - are the variables that get values when you call the function. You can have 0 or more parameters, separated by commas. Must be in parenthesis.
- **body** - body is a sequence of commands like we've already seen (assignment, multiplication, function calls).
- **return** - ends the function and returns data (like the sine of an angle).
- **Important**: all the lines of body must be indented. That is how Python knows that they are part of the function.

# Calling Functions

- The general form of a function call:

`function_name(arguments)`

`function_name()`

`function_name`

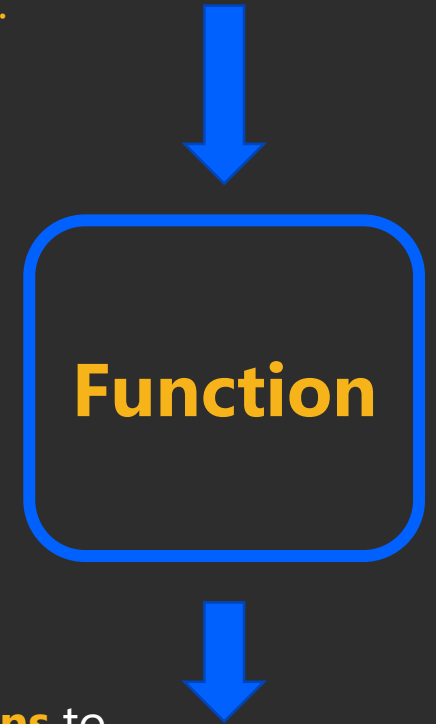
← Would not result in  
a function call.

- Terminology

- *argument*: a value given to a function.
- *pass*: to provide an argument to a function.
- *call*: ask Python to execute a function (by name).
- *return*: give a value back to where the function was called from.

The stuff we **pass** **Arguments**  
to the function.

Call  
Function  
`()`



The stuff the  
function **returns** to  
us after we **call** it. **Returns**

# Function Definitions

```
def function_name(parameters):  
    body  
    return
```

**x** is the parameter.

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

# Calling Functions

```
function_name(arguments)
```

**2** is the argument  
(data) passed to  
the **square**  
function.

```
square(2)
```

# Function Definitions

```
def function_name(parameters):
```

1. `"""DOCSTRING"""` (optional)

2. Code that does the thing

3. `return expression`

The `return` statement is optional and if it is not included, it's the same as writing `return None`

Open your  
notebook

Click Link:

1. Defining Your Own  
Functions



# Design Recipe

- How do we go about writing a function?
  - You should follow these six steps.
1. **Examples** (What do you want your function calls to look like?)
  2. **Type Contract** (Specify the type(s) of parameters and return values)
  3. **Header** (Decide on the name of the function)
  4. **Description** (Write a short description of what the function does)
  5. **Body** (Write the code that actually does the thing that you want)
  6. **Test** (Verify the function using examples)

# Design Recipe

- Write a function that converts from Fahrenheit to Celsius.

1. **Examples** (What do you want your function calls to look like?)

```
celsius = convert_to_celsius(32)
```

```
celsius = convert_to_celsius(212)
```

```
celsius = convert_to_celsius(98.6)
```

# Design Recipe

- Write a function that converts from Fahrenheit to Celsius.

## 2. Type Contract (Specify the type(s) of parameters and return values)

```
def convert_to_celsius(degrees_f):  
    """
```

```
    """
```

```
    ...
```

```
    """
```

```
    ... Do something
```

```
    return degrees_c
```

What types are  
passed in?

What types are returned?

# Design Recipe

- Write a function that converts from Fahrenheit to Celsius.

## 2. Type Contract (Specify the type(s) of parameters and return values)

```
def convert_to_celsius(degrees_f):  
    """
```

```
    (number) -> number  
    """
```

```
    ... Do something
```

```
    return degrees_c
```



What types are  
passed in? **Number**



What types are returned?  
**Number**

# Design Recipe

- Write a function that converts from Fahrenheit to Celsius.

**3. Header** (Decide on the name of the function and parameters)

```
def convert_to_celsius(degrees_f):  
    """  
    (number) -> number  
    """  
  
    ... Do something  
  
    return degrees_c
```

(you probably already did this in step 1)

# Design Recipe

- Write a function that converts from Fahrenheit to Celsius.

4. **Description** (Write a short description of what the function does)

```
def convert_to_celsius(degrees_f):  
    """  
    (number) -> number  
    Return the temperature in degrees Celsius corresponding to  
    the degrees Fahrenheit passed in.  
    """  
  
    ... Do something  
  
    return degrees_c
```

# Design Recipe

- Write a function that converts from Fahrenheit to Celsius.

**5. Body** (Write the code that actually does the thing that you want)

```
def convert_to_celsius(degrees_f):  
    """  
    (number) -> number  
    Return the temperature in degrees Celsius corresponding to  
    the degrees Fahrenheit passed in.  
    """  
  
    degrees_c = (degrees_f - 32) * 5 / 9  
  
    return degrees_c
```

# Design Recipe

- **Write a function that converts from Fahrenheit to Celsius.**

## 6. Test (Verify the function using examples)

- Run all the examples that you created in Step 1.
- Testing is so important.
- In industry, you'll be expected to provide tests for everything.

```
celsius = convert_to_celsius(32) # celsius should be 0
```

```
celsius = convert_to_celsius(212) # celsius should be 100
```

```
celsius = convert_to_celsius(98.6) # celsius should be 37.0
```



# Design Recipe

- How do we do about writing a function?
- You should follow these six steps.

1. **Type**
2. **Contract**
3. **Header**
4. **Description**
5. **Body**
6. **Test**

**Open your  
notebook**

**Click Link:**  
**2. Design Recipe**

# Docstring

- A Python documentation string, commonly known as **docstring**, helps you understand the capabilities of a function (or module, class).

```
def convert_to_celsius(degrees_f):
```

```
    """
```

```
    (number) -> number
```

```
    Return the temperature in degrees Celsius corresponding to  
    the degrees Fahrenheit passed in.
```

```
    """
```

```
    degrees_c = (degrees_f - 32) * 5 / 9
```

```
    return degrees_c
```

This is the  
**docstring**

# Docstring

- As we saw before, `help()` prints information about a function.
- The help function actually prints out the “**docstring**” that we write as part of a function definition.
- For the function we just wrote, we could type:

```
help(convert_to_celsius)
```

```
>>>
```

```
Help on function convert_to_celsius in module __main__:
```

```
convert_to_celsius(degrees_f)
```

```
    (number) -> number
```

```
    Return the temperature in degrees Celsius corresponding to the degrees  
    Fahrenheit passed in
```

# Docstring

- These are the most popular Docstrings format available.

Formatting Type	Description
<u>NumPy/SciPy docstrings</u>	Combination of reStructured and GoogleDocstrings and supported by Sphinx
<u>PyDoc</u>	Standard documentation module for Python and supported by Sphinx
<u>EpyDoc</u>	Render Epytext as series of HTML documents and a tool for generating API documentation for Python modules based on their Docstrings
<u>Google Docstrings</u>	Google's Style

# Docstring

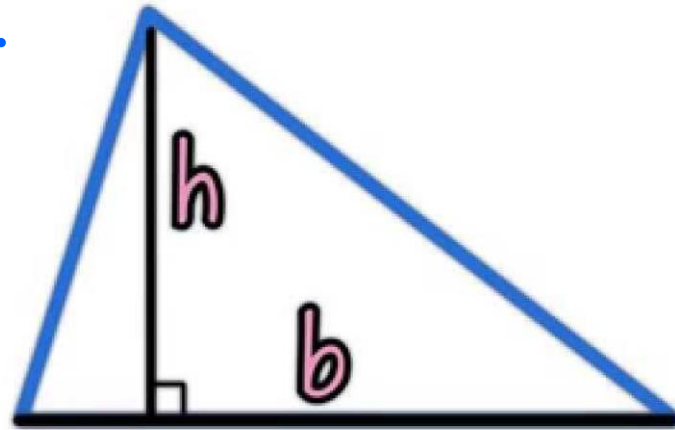
- This can be very valuable:
  - For other programmers to figure out what a function is supposed to do.
  - For you in the future when you have forgotten what you wrote (this happens a lot!).
- You should write a **docstring** for every function!
- Remember good vs bad code review.

**Open your  
notebook**

**Click Link:**  
**3. Docstring**

# Breakout Session 1

- Following the Design Recipe, write a function to calculate the area of a triangle.



$$\text{Area} = \frac{1}{2} \times b \times h = \frac{bh}{2}$$

**Open your notebook**


**Click Link:**

**6. Breakout Session 1**

# More Stuff You Can Do With Functions

## ▪ Nested Function Calls

```
print(3 + 7 + abs(-5))
```

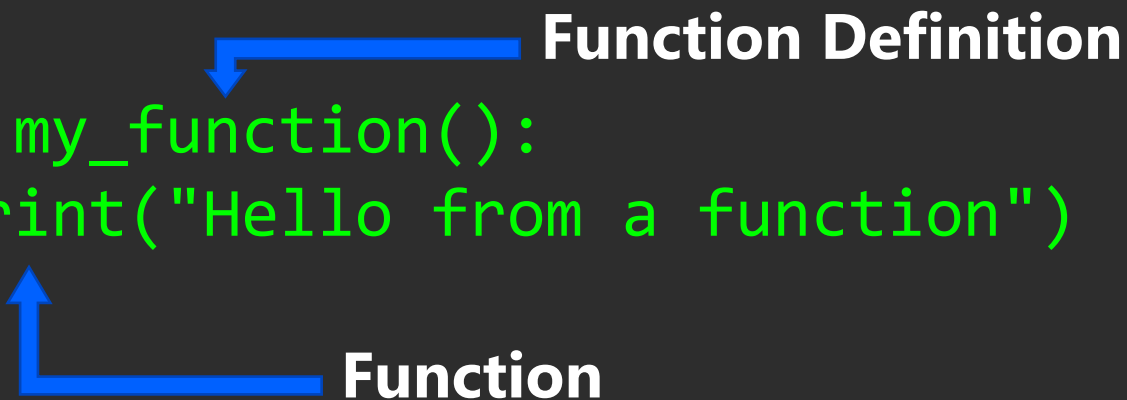


Function

Function

## ▪ Calling Functions within Functions

```
def my_function():  
    print("Hello from a function")
```



Function Definition

Function

Open your  
notebook

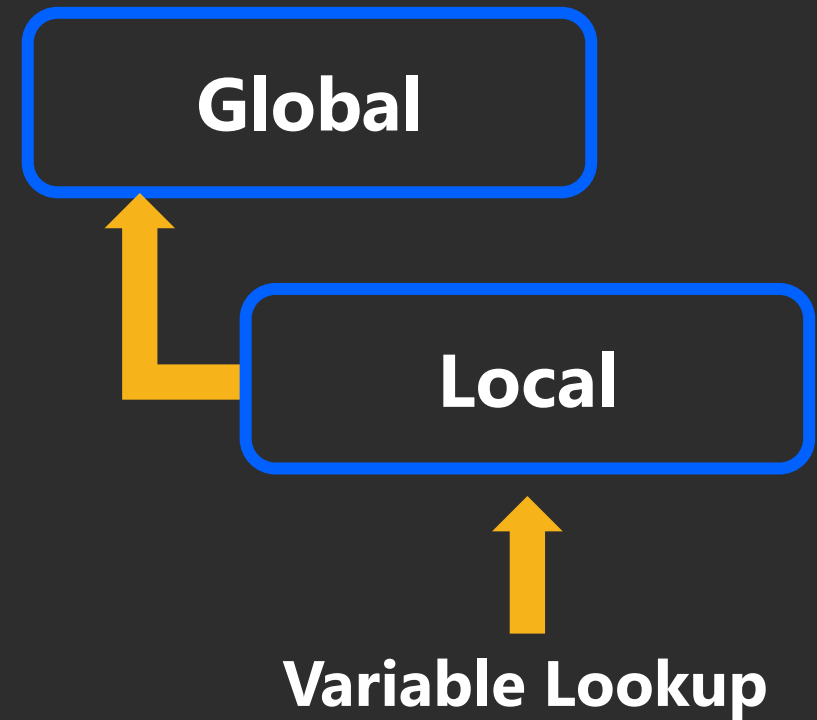
Click Link:

4. Nested Function  
Calls

5. Calling Functions  
within Functions

# Variable Scope

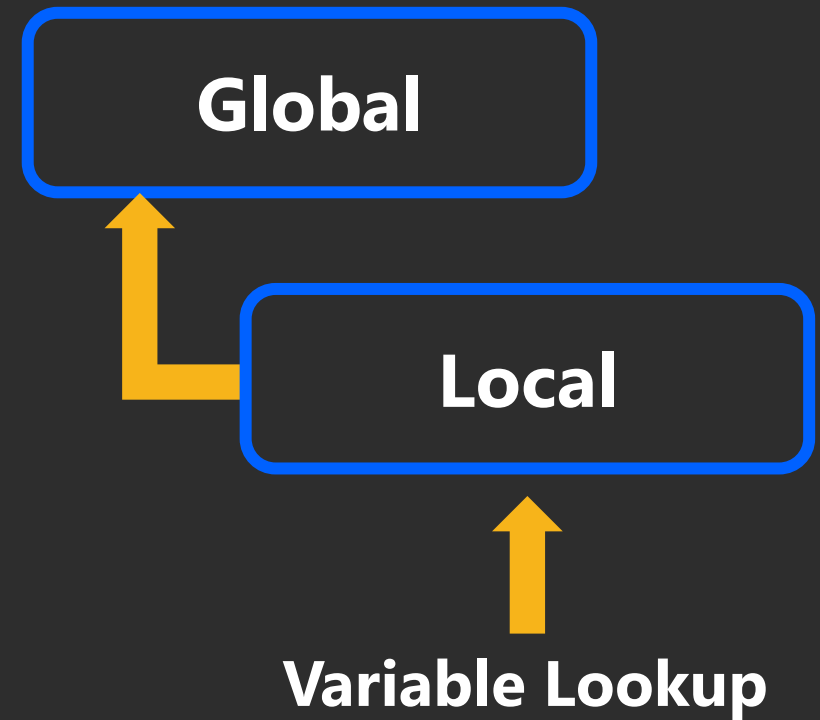
- A variable is only available from inside the region it is created, which is called the variable's scope.
- Python has four different scopes, and we will discuss the two most important for this course.
- Local Scope
- Global Scope





# Local Scope

- Whenever you define a variable within a function, its scope lies **ONLY** within the function.
- It is accessible from the point at which it is defined until the end of the function and exists for as long as the function is executing.
- This means its value cannot be changed or even accessed from outside the function.



# Local Scope

## Example 1

```
def my_function():  
    name = 'Sebastian'  
  
my_function()  
  
print(name)
```

What will  
happen when  
this code is  
run?

# Local Scope

## Example 1

```
def my_function():  
    name = 'Sebastian'
```

```
my_function()
```

```
print(name)
```

```
>>> Error
```

**name** is local to  
the function and  
not accessible  
outside in the  
global scope.

# Local Scope

## Example 1

```
def my_function():  
    name = 'Sebastian'  
  
my_function()  
  
print(name)
```

Global

# Local Scope

## Example 1

```
→ def my_function():  
    name = 'Sebastian'  
  
my_function()  
  
print(name)
```

Global

Local

(my\_function)

# Local Scope

## Example 1

```
def my_function():  
    name = 'Sebastian'
```

→ `my_function()`

```
print(name)
```

**Global**

**Local**

`(my_function)`  
`name = "Sebastian"`

# Local Scope

## Example 1

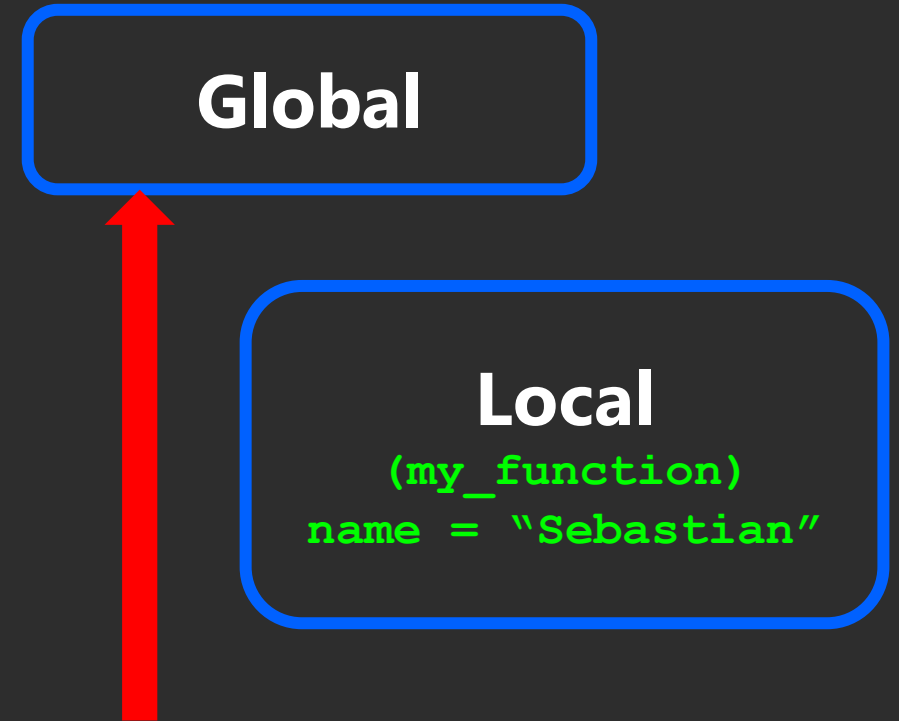
```
def my_function():  
    name = 'Sebastian'
```

```
my_function()
```

→ 

```
print(name)
```

```
>>> Error
```



### Variable Lookup

- Is **name** in global?
- No (Done)

# Local Scope

## Example 2

```
def my_function():  
    name = 'Sebastian'  
  
my_function()  
  
name = 'Ben'  
print(name)
```

What will  
happen when  
this code is  
run?



# Local Scope

## Example 2

```
def my_function():  
    name = 'Sebastian'
```

```
my_function()
```

```
name = 'Ben'  
print(name)
```

```
>>> Ben
```

What will  
happen when  
this code is  
run?

# Local Scope

## Example 2

Global

```
def my_function():  
    name = 'Sebastian'
```

```
my_function()
```

```
name = 'Ben'  
print(name)
```

# Local Scope

## Example 2

→ 

```
def my_function():  
    name = 'Sebastian'
```

```
my_function()
```

```
name = 'Ben'  
print(name)
```

Global

Local

(my\_function)

# Local Scope

## Example 2

```
def my_function():  
    name = 'Sebastian'
```

→ `my_function()`

```
name = 'Ben'  
print(name)
```

**Global**

**Local**

`(my_function)`  
`name = "Sebastian"`

# Local Scope

## Example 2

```
def my_function():  
    name = 'Sebastian'
```

```
my_function()
```

```
→ name = 'Ben'  
print(name)
```

**Global**

name = "Ben"

**Local**

(my\_function)  
name = "Sebastian"

# Local Scope

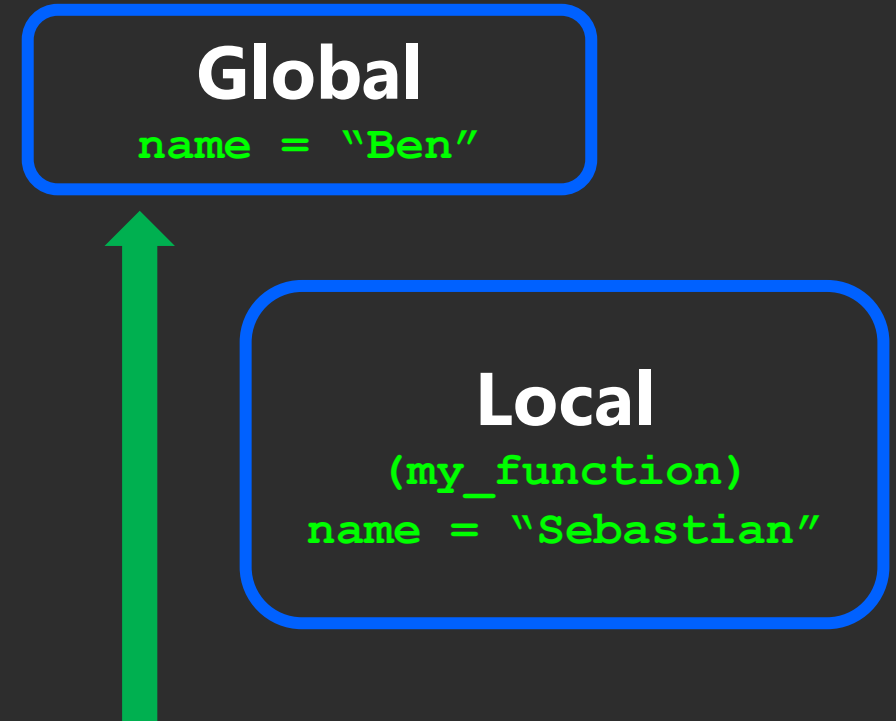
## Example 2

```
def my_function():  
    name = 'Sebastian'
```

```
my_function()
```

```
name = 'Ben'  
→ print(name)
```

```
>>> Ben
```



### Variable Lookup

- Is **name** in global?
- Yes (Done)

# Local Scope

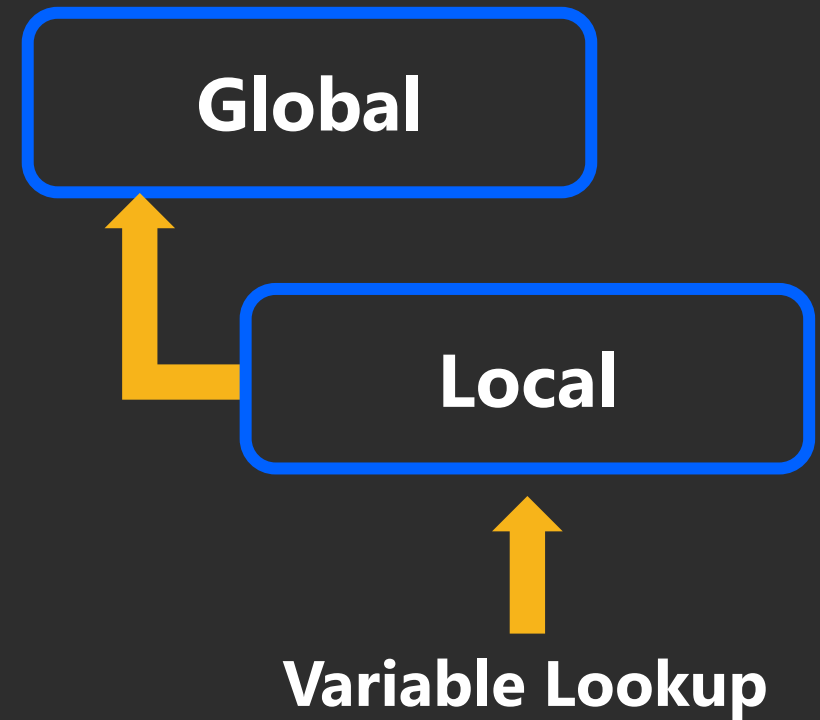
- Whenever you define a variable within a function, its scope lies **ONLY** within the function.
- It is accessible from the point at which it is defined until the end of the function and exists for as long as the function is executing.
- This means its value cannot be changed or even accessed from outside the function.

**Open your  
notebook**

**Click Link:**  
**7. Local Scope**

# Global Scope

- Whenever a variable is defined outside any function, it becomes a global variable, and its scope is anywhere within the program.
- This means that variables and functions defined outside of a function are accessible inside of a function.





# Global Scope

## Example 1

```
def my_function():  
    print(name)
```

```
name = 'Sebastian'
```

```
my_function()
```

What will  
happen when  
this code is  
run?

# Global Scope

Notice that **name** is not defined anywhere in the function.

## Example 1

```
def my_function():  
    print(name)
```

```
name = 'Sebastian'
```

```
my_function()
```

```
>>> Sebastian
```

**name** is in the global scope and is accessible inside the function.

**What will happen when this code is run?**

# Global Scope

## Example 1

```
def my_function():  
    print(name)
```

```
name = 'Sebastian'
```

```
my_function()
```



Global

# Global Scope

## Example 1

→ `def my_function():  
 print(name)`

`name = 'Sebastian'`

`my_function()`

Global

Local

`(my_function)`

# Global Scope

## Example 1

```
def my_function():  
    print(name)
```

→ `name = 'Sebastian'`

```
my_function()
```

**Global**

`name = "Sebastian"`

**Local**

`(my_function)`

# Global Scope

## Example 1

```
def my_function():  
    print(name)
```

```
name = 'Sebastian'
```

→ `my_function()`

**Global**

`name = "Sebastian"`

**Local**

`(my_function)`



**Variable Lookup**

- Is `name` in local?
- No

# Global Scope

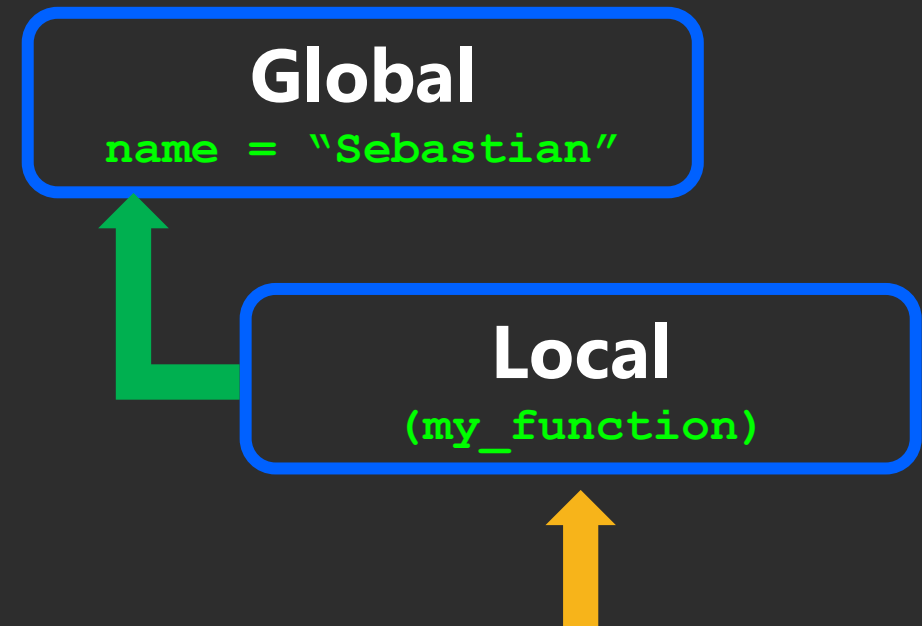
## Example 1

```
def my_function():  
    print(name)
```

```
name = 'Sebastian'
```

```
→ my_function()
```

```
>>> Sebastian
```



### Variable Lookup

- Is **name** in local?
- No
- Is **name** in global?
- Yes (Done)

# Global Scope

## Example 2

```
def my_function():  
    name = 'Ben'  
    print(name)
```

```
name = 'Sebastian'
```

```
my_function()
```

What will  
happen when  
this code is  
run?



# Global Scope

## Example 2

```
def my_function():  
    name = 'Ben'  
    print(name)
```

```
name = 'Sebastian'
```

```
my_function()
```

```
>>> Ben
```

What will  
happen when  
this code is  
run?

**name** is in the  
local and global  
scope. Python  
will use the local  
version.

# Global Scope

## Example 2

```
def my_function():  
    name = 'Ben'  
    print(name)
```

```
name = 'Sebastian'
```

```
my_function()
```

Global

# Global Scope

## Example 2

```
→ def my_function():  
    name = 'Ben'  
    print(name)
```

```
name = 'Sebastian'
```

```
my_function()
```

Global

Local

`my_function`

# Global Scope

## Example 2

```
def my_function():  
    name = 'Ben'  
    print(name)
```

→ `name = 'Sebastian'`

```
my_function()
```

**Global**

`name = 'Sebastian'`

**Local**

`my_function`

# Global Scope

## Example 2

```
def my_function():  
    name = 'Ben'  
    print(name)
```

```
name = 'Sebastian'
```

➔ 

```
my_function()
```

```
>>> Ben
```

### Global

```
name = 'Sebastian'
```

### Local

```
my_function  
name = 'Ben'
```



### Variable Lookup

- Is name in local?
- Yes (Done)

# Global Scope

## Example 3

```
def my_function():  
    print(name)
```

```
my_function()
```

What will  
happen when  
this code is  
run?

# Global Scope

## Example 3

```
def my_function():  
    print(name)
```

```
my_function()
```

```
>>> Error
```

`name` is not  
defined in the  
local or global  
scope.

What will  
happen when  
this code is  
run?

# Global Scope

```
def my_function():  
    print(name)
```

```
my_function()
```

Global



# Global Scope

→ `def my_function():`  
    `print(name)`

`my_function()`

**Global**

**Local**

`my_function`

# Global Scope

```
def my_function():  
    print(name)
```

→ `my_function()`

Global

Local

`my_function`

Variable Lookup

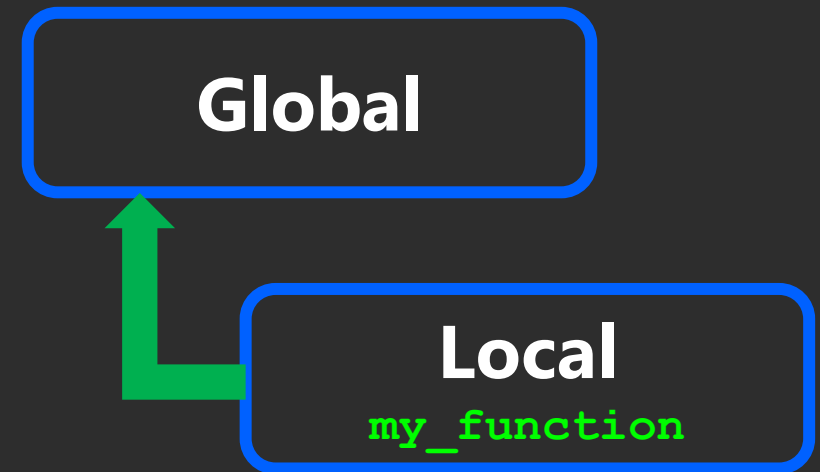
- Is `name` in local?
- No

# Global Scope

```
def my_function():  
    print(name)
```

→ `my_function()`

>>> **Error**



## Variable Lookup

- Is **name** in local?
- No
- Is name in global?
- No

# Global Scope

- Whenever a variable is defined outside any function, it becomes a global variable, and its scope is anywhere within the program.
- This means that variables and functions defined outside of a function are accessible inside of a function.

**Open your  
notebook**

**Click Link:**

**8. Global Scope**

# print v.s. return

- The difference between print and return is a point of confusion year after year.
- So, let's be proactive and address this.



Are we  
the same?

return



Eww, no.

print

## print

- **Use cases**
- Debugging.
- Displaying messages to users.

## return

- **Use cases**
- Used to end the execution of the function call and "return" the result.

# print

```
def square(x):  
    output = x * x  
    print(output)
```

```
>>> square(2)  
4
```

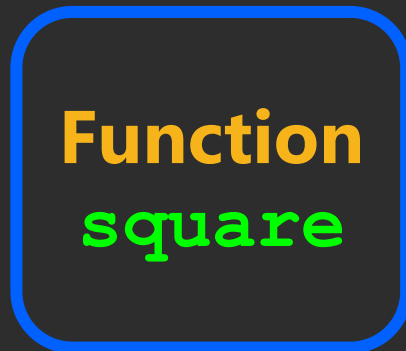
# return

```
def square(x):  
    output = x * x  
    return output
```

```
>>> square(2)  
4
```

# print

The stuff we **pass** **Arguments: 2**  
to the function.



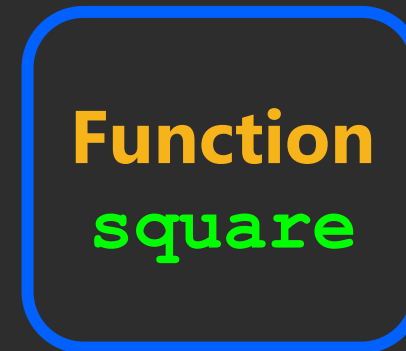
```
def square(x):  
    output = x * x  
    print(output)
```



The stuff the  
function **returns** to  
us after we **call** it. **Returns: None**

# return

The stuff we **pass** **Arguments: 2**  
to the function.



```
def square(x):  
    output = x * x  
    return output
```



The stuff the  
function **returns** to  
us after we **call** it. **Returns: 4**



# print v.s. return

```
def square(x):  
    output = x * x  
    print(output)
```

```
def square(x):  
    output = x * x  
    print(output)  
    return None
```

These two  
functions  
return the  
same thing.

**Open your  
notebook**

**Click Link:**

**9. print v.s. return**

# From Functions to Programs

- The recipe we discussed earlier highlights a few of the realities about programming whether for individual functions or for large pieces of software.
- 1. A formal design process (or even a recipe) can help.
  - Especially when you are writing a large program with many programmers, it is easy to get lost.
  - In fact, it is more often impossible to hold the entire program in your head.
  - Having a process helps you to figure out where you are and what you should do next.

# From Functions to Programs

- The recipe we discussed earlier highlights a few of the realities about programming whether for individual functions or for large pieces of software.
- 2. Functions can be written and then their insides can be forgotten about.
  - Do you know how Python calculates **sin()**?
  - Do you care?
  - You can successfully use functions without knowing how they are implemented if you know what they take in and what they return.
  - This is very important for large projects.

# From Functions to Programs

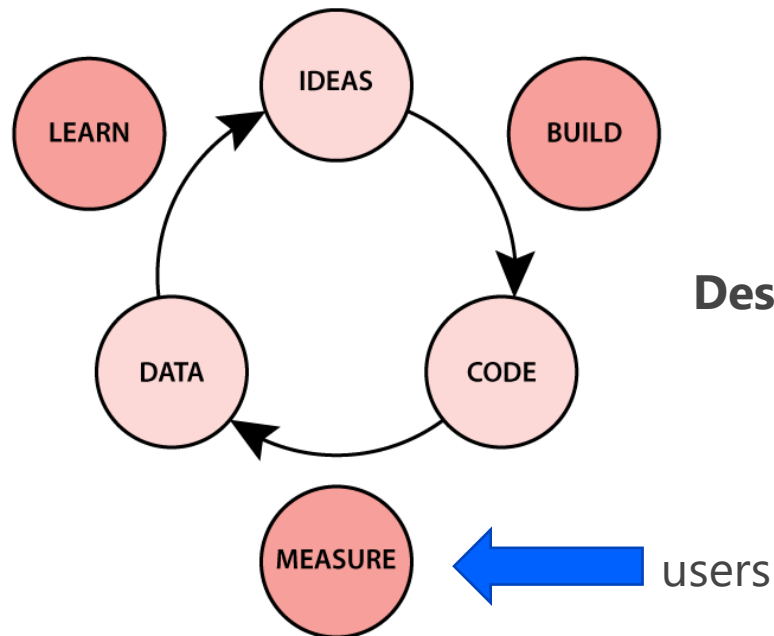
- The recipe we discussed earlier highlights a few of the realities about programming whether for individual functions or for large pieces of software.
- 3. Start with examples.
  - This helps in communication with the client, helps (a lot) to figure out what the problem really is, and is the core for testing your code.

# A Design Process for Programming

- APS111/112, a key part of engineering is the design of objects, processes, and systems.
- **Programming** is the design, implementation, testing, and documentation of a piece of software that solves a particular problem.
- The software might be part of a larger system (e.g., the avionics software of an aircraft, the accounting or human resources software of a business), but it represents the solution to a design problem (or part of a design problem).

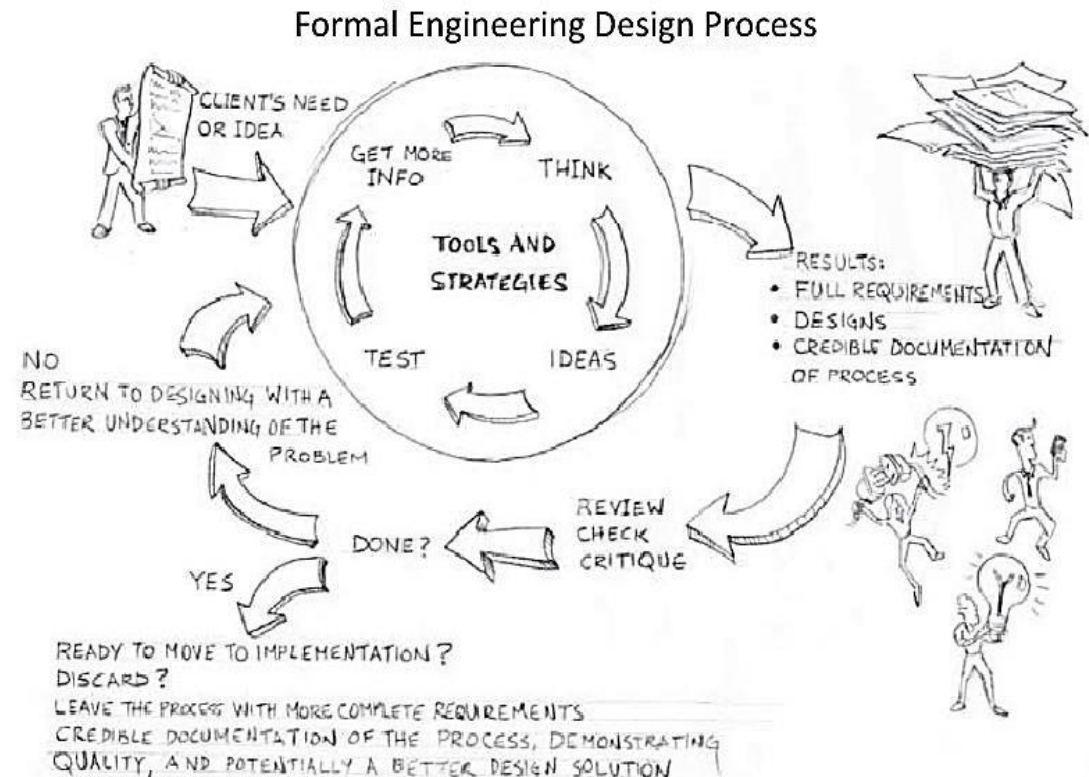
# A Design Process for Programming

- We will approach programming as an engineering design process and adapt the process you have already seen in APS111/112.



**Design is iterative.**

Taken from: [Designing Engineers: An Introductory Text](#)



# A Design Process for Programming

- In the next lecture, we are going to talk about a detailed design process for programming, based on the engineering design processes that are key to any engineering.
- The steps are as follows:
- **Define the Problem.**
- **Define Test Cases.**
- **Generate Multiple Solutions.**
- **Select a Solution.**
- **Implement the Solution.**
- **Perform Final Testing.**

# A Design Process for Programming

- **Define the Problem.**
- Write down what the problem actually is.



# A Design Process for Programming

- **Define Test Cases.**
- Create some examples that reflect your code solving the problem: input and output.

# A Design Process for Programming

- **Generate Multiple Solutions.**
- At this point a "solution" consists of an algorithm plan (the high-level sequence steps defining what your algorithm will do) and a programming plan (the high-level sequence of steps that you will take to code the algorithm).

# A Design Process for Programming

- **Select a Solution.**
- Based on the different algorithm and programming plans, decide which is the most promising.

# A Design Process for Programming

- **Implement the Solution.**
- Start to execute your programming plan.
- Test as you go!
- You may realize that your algorithm plan doesn't solve the problem, or even that you do not understand the problem.
- If so, go back to earlier steps.

# A Design Process for Programming

- **Perform Final Testing.**
- Make sure that your original test cases as well as any others that you have thought up work.

# A Design Process for Programming

- It is critical to realize that programming is:
  - Iterative: you will go back and change your algorithm/programming plan. You will write some code during Step 3: you might not be able to define a solution without writing some code to solve part of the problem. You will move back-and-forth in this process.
  - This process is a lot about finding your own mistakes: even for good programmers, most of their time is spent testing and debugging!

# PRACTICE!

writing your own function.

**Week 2** | Lecture 2 (2.2)

if nothing else, write **#cleancode**.