Lect 8: More on Functions

@Copyright Samaneh Gholami

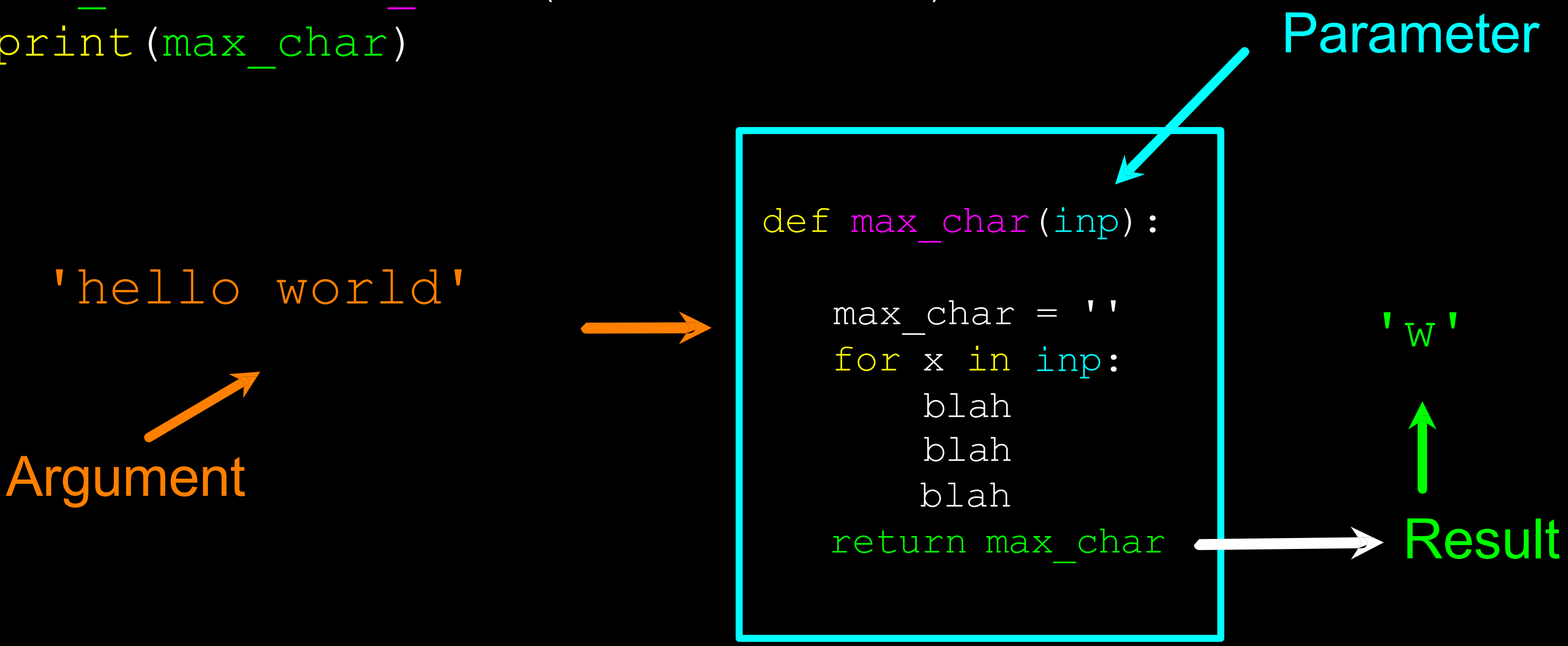
Learning Objectives:

- ❑ Calling and Enhancing Functions
- ❑ Functions Without Arguments
- ❑ Default Argument Values
- ❑ Using Named Arguments
- ❑ Testing Functions
- ❑ Understanding Scope
- ❑ Documenting Functions
- ❑ Calling Functions Within Functions

@Copyright Samaneh Gholami

Review:
Arguments, Parameters, and Results

```
>>> max_char = max_char('hello world')
>>> print(max_char)
w
```



```
def max_char(inp):

    max_char = ''
    for x in inp:
        blah
        blah
        blah
    return max_char
```

@Copyright Samaneh Gholami

Review:

Let's create our get_at_content function. For this function, the input is going to be a single DNA sequence, and the output is going to be a decimal number. To translate these into Python terms: the function will take a single argument of type *string*, and will return a value of type *number*.

```
def get_at_content(dna):
    length = len(dna)
    a_count = dna.count('A')
    t_count = dna.count('T')
    at_content = (a_count + t_count) / length
    return at_content


get_at_content("ATGACTGGACCA")
```

```
at_content = get_at_content("ATGACTGGACCA")
print("AT content is " + str(get_at_content("ATGACTGGACCA")))
```

@Copyright Samaneh Gholami

Calling and improving our function

```
def get_at_content(dna):
    length = len(dna)
    a_count = dna.count('A')
    t_count = dna.count('T')
    at_content = (a_count + t_count) / length
    return at_content

my_at_content =
get_at_content("ATGCGCGATCGATCGAATCG")
print(str(my_at_content))
print(get_at_content("ATGCATGCAACTGTAGC"))
print(get_at_content("aactgtagctagctagcagcgta"))
```

- The first call correctly calculates AT content as 0.45.
- The second call shows too many decimal places.
- The third call returns 0.0 incorrectly for lowercase input.
- Issues with decimal formatting and case sensitivity need fixing.

```
0.45
0.5294117647058824
0.0
```

@Copyright Samaneh Gholami

Calling and improving our function (cont.)

```python
def get_at_content(dna):
    length = len(dna)
    a_count = dna.upper().count('A')
    t_count = dna.upper().count('T')
    at_content = (a_count + t_count) / length
    return round(at_content, 2)

my_at_content =
get_at_content("ATGCGCGATCGATCGAATCG")
print(str(my_at_content))
print(get_at_content("ATGCATGCAACTGTAGC"))
print(get_at_content("aactgtagctagctagcagcgta"))
```

```
0.45
0.53
0.52
```

- Improve `get_at_content` by adding a rounding step to control significant figures in the result.
- Use Python’s built-in `round` function, specifying the number to round and the desired precision.
- Call `round` on the result before returning it.
- Convert the input sequence to uppercase to handle any lowercase issues before calculation.

@Copyright Samaneh Gholami

Functions Without Arguments: When and Why to Use Them

- Python allows functions to be defined without any arguments.
- Functions without arguments are valid and follow Python's syntax rules.
- These functions don't require any input to execute.
- However, functions without arguments are often less practical or versatile.

```
def get_at_content():
    length = len(dna)
    a_count = dna.upper().count('A')
    t_count = dna.upper().count('T')
    at_content = (a_count + t_count) / length
    return round(at_content, 2)

dna = "ACTGATGCTAGCTA"
my_at_content = get_at_content()
print(str(my_at_content))
```

@Copyright Samaneh Gholami

Using Named Arguments to Call Functions

```python
def get_at_content(dna, sig_figs):
   length = len(dna)
   a_count = dna.upper().count('A')
   t_count = dna.upper().count('T')
   at_content = (a_count + t_count) / length
   return round(at_content, sig_figs)

test_dna = "ATGCATGCAACTGTAGC"
print(get_at_content(test_dna, 3))

print(get_at_content(dna="ATCGTGACTCG", sig_figs=2))
print(get_at_content(sig_figs=2, dna="ATCGTGACTCG"))
print(get_at_content("ATCGTGACTCG", sig_figs=2))
```

- The two-argument `get_at_content` requires the DNA sequence first, then the number of significant figures.
- **Keyword arguments** in Python let us specify arguments by name, not order.
- Using keywords, we pass arguments as `name=value` pairs.
- This method adds flexibility, allowing any order.

@Copyright Samaneh Gholami

Using Default Values for Function Arguments

```python
def get_at_content(dna, sig_figs=2):
    length = len(dna)
    a_count = dna.upper().count('A')
    t_count = dna.upper().count('T')
    at_content = (a_count + t_count) / length
    return round(at_content, sig_figs)

print(get_at_content("ATCGTGACTCG"))
print(get_at_content("ATCGTGACTCG", 3))
print(get_at_content("ATCGTGACTCG", sig_figs=4))
```

```
0.45
0.455
0.4545
```

@Copyright Samaneh Gholami

Testing Functions

Assertion

An *assertion statement* tests whether an expression is true or false, causing an error if it is false.

assert *expression*

```
def get_at_content(dna, sig_figs=2):
    length = len(dna)
    a_count = dna.upper().count('A')
    t_count = dna.upper().count('T')
    at_content = (a_count + t_count) / length
    return round(at_content, sig_figs)

assert get_at_content("ATGC") > 0.6
assert get_at_content("ATGC") == 0.6
```

@Copyright Samaneh Gholami

To function or not to function...

- How do you get started? Well first you have to start implementing functions (there will be a hesitancy at first, but push through this hesitancy)
- Then try and organize your code into "paragraphs" - capture a complete thought and "name it"
- Don’t repeat yourself - make it work once and then reuse it
- If something gets too long or complex, break it up into logical chunks and put those chunks in functions (refactoring)
- Make a module of **common** stuff that you do over and over - perhaps share this with your classmates

@Copyright Samaneh Gholami

How Do I Use Functions?

- One very simple concept I try and follow:
  - Functions should encapsulate something useful
  - Or sometimes functions are useful to break up long stretches of code that are hard to follow
  - And should be able to be tested (more on this later)

- **Follow this simple rule**:
  - Functions should do one thing (force yourself to adhere to this principle)
  - Try to limit the length, and the number of arguments
  - If you function does “this” **AND** “that”, its probably doing too much (you want it to do a single job)

- These points will help you divide your code into manageable chunks, suitable for functions

@Copyright Samaneh Gholami

So, When Should I Use Functions in Python?

Uses cases when a piece of code should be put into a function

- 1st Case
- You know it will be used to perform a **calculation**
  - With **calculations, you have a good idea** the action is going to happen **more then once**
    - Putting string into a specific format
    - Printing the header or footer of a report
    - Conversions
    - etc.

@Copyright Samaneh Gholami

So, When Should I Use Functions in Python?

Uses cases when a piece of code should be put into a function

- 2nd **Case**
- **Logical units** of your program want to **break up**
    - Helps make your program **easier to understand**
    - Later we'll see how these get wrapped up in a module/package

@Copyright Samaneh Gholami

So, When Should I Use Functions in Python?

Uses cases when a piece of code should be put into a function

- 3rd **Case**
  - Bundle a set of instructions that you want to use repeatedly or that, because of their complexity, are better self-contained in a sub-program and called when needed

@Copyright Samaneh Gholami

Scope

@Copyright Samaneh Gholami

What's Scope

Scope is the space within which a variable label (name) exists and can be used. Python talks of **names** being **bound** to the code in an area, and that determines its scope. **Binding** in Python is **usually** through **assignment**

- So far, we haven't had to deal much with scope, i.e. we've just assumed scope
- Loops/if compound statements don't effect scope

```
a = [1]                      # "a" can be  declared  here
for i in 1,2,3,4:                                          [1]
   a = [1]                   # or here.                    [2]
   a[0] = i                                                [3]
   print (a)                 # 1 ,2 ,3 ,4                  [4]
print("end of loop")                                       end of loop
print (a)                    # 4                           [4]
print (i)                                                  4
```

- All the "**a**" variables here are treated as same object wherever they are first used, and "**i**" can be seen everywhere after it has been first declared
- Functions complicate this a bit..

@Copyright Samaneh Gholami

Functions And Scoping

The declaration of a function looks similar to a compound statement, **the function declaration generates a block**, which has scoping rules. **Starting with variables labels (names) made inside a function:**

```
def function1():
     b = 10
     print(b)

function1()
print(b)
```

```
10
Traceback (most recent call last):
  File "scope3.py", line 13, in <module>
    print(b)
NameError: name 'b' is not defined
```

- This will run the print statement within the function
- **But then fails** because `b` has been first allocated inside the function
- **This makes `b` a local variable**, only available within the function and functions within it
- This is a very useful feature for functions

@Copyright Samaneh Gholami

Local Variables. Why's Are They Useful?

- Because all the variables defined in a function are local to that function
- For example, **`b`** in our **`function1`** function
- The locals only exist while the function is running
  - Lifetime is the time during which a variable takes up memory
  - Variable is “born” when it is initialized…
  - … and “dies” when the function the variable(s) is in, **`return`**s

```
def function1():
      b = 10
      print(b)
function1()
print(b)
```

@Copyright Samaneh Gholami

Using Global Variable in a Function

One solution is to define the variable outside the function:

```
b = 10                  # b here is a "global variable"
                        # it can be seen everywhere there
                        # isn't a variable of the same
                        # name assigned.

def function1():
    print(b)            # b here is a "free variable" i.e.
                        # defined outside the current block, i.e.
                        # the variable is used in a code block but not defined there

function1()  ⟶  10
print(b)     ⟶  10
```

I try to avoid the use of globals!

Remember your pylint error:
C0103: Constant name "x" doesn't conform to UPPER_CASE naming style (invalid-name)

@Copyright Samaneh Gholami

Using Global Variable in a Function

As soon as you declare a variable inside a block, it is a new variable label (name). Contrast:

```
b = 10
def function1():
      b = 20
      print(b)       → # Prints 20.
print(b)             → # Prints 10.
function1()
print(b)             → # Prints 10.
```

New assignment! The scope of a local variable is the body of the function it is in, starting from its initialization

with:

```
b = 10
def function1():
      print(b)       → # Prints 10;
print(b)             → # Prints 10.
function1()
print(b)             → # Prints 10
```

Using global in the function

@Copyright Samaneh Gholami

One Strange Python Behavior

However, you have to watch for strange behaviour:

```
b = 10
def function1():
     print(b)
     b = 20          # Adding this line makes
                     # the line above fail, hmmm... That's odd?


print(b)             # This would print 10.
function1()
print(b)
```

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 2, in a
UnboundLocalError: local variable 'b' referenced before assignment
```

Python scans blocks before running **and assumes any variable label (name) assigned within a block is local to that block. This effects the whole block before and after**, so the `b` in the function is different from that outside**. At that point, the `print(b)` is trying to use a variable before a value has been assigned to it.**

@Copyright Samaneh Gholami

Global Variables

Variables outside of functions in scripts are global: in theory they can be seen anywhere.
However, the rule about local assignments creating local variables undermines this (previous slide).
To force a local assignment to a global variable, use the `global` keyword, thus:

```
b = 10
def function1():
    global b
    b = 20
    print(b)  ──────→ # Prints 20.
print(b)──────────→ # Prints 10.
function1()
print(b)──────────→ # Now prints 20 as the function
                    # changes the global b.
```

But remember:

I try to avoid the use of `global` variables!
Especially in public functions

@Copyright Samaneh Gholami

Documenting Functions

@Copyright Samaneh Gholami

Python Docstrings

- You 're familiar with “#” to indicate comments explaining what’s going on in a Python file
- Python has a special kind of commenting convention: documentation strings or "docstrings" to document a function
- These go inside the function definition, as the very first thing
- So it must be indented!
  - It starts with """ (three double quotes)
  - Continues across multiple lines until another """

@Copyright Samaneh Gholami

Example Document String

```
def triangle(size, letter):
    """
    NoneType : triangle(size, letter)
    Takes: int in the first argument (size of the triangle)
           string (character) in the second argument
    Returns: NoneType
    e.g. triangle(2, "+")
    """
    for i in range(1, size+1):
        print(letter * i)

triangle(2, "+")

help(triangle)
```

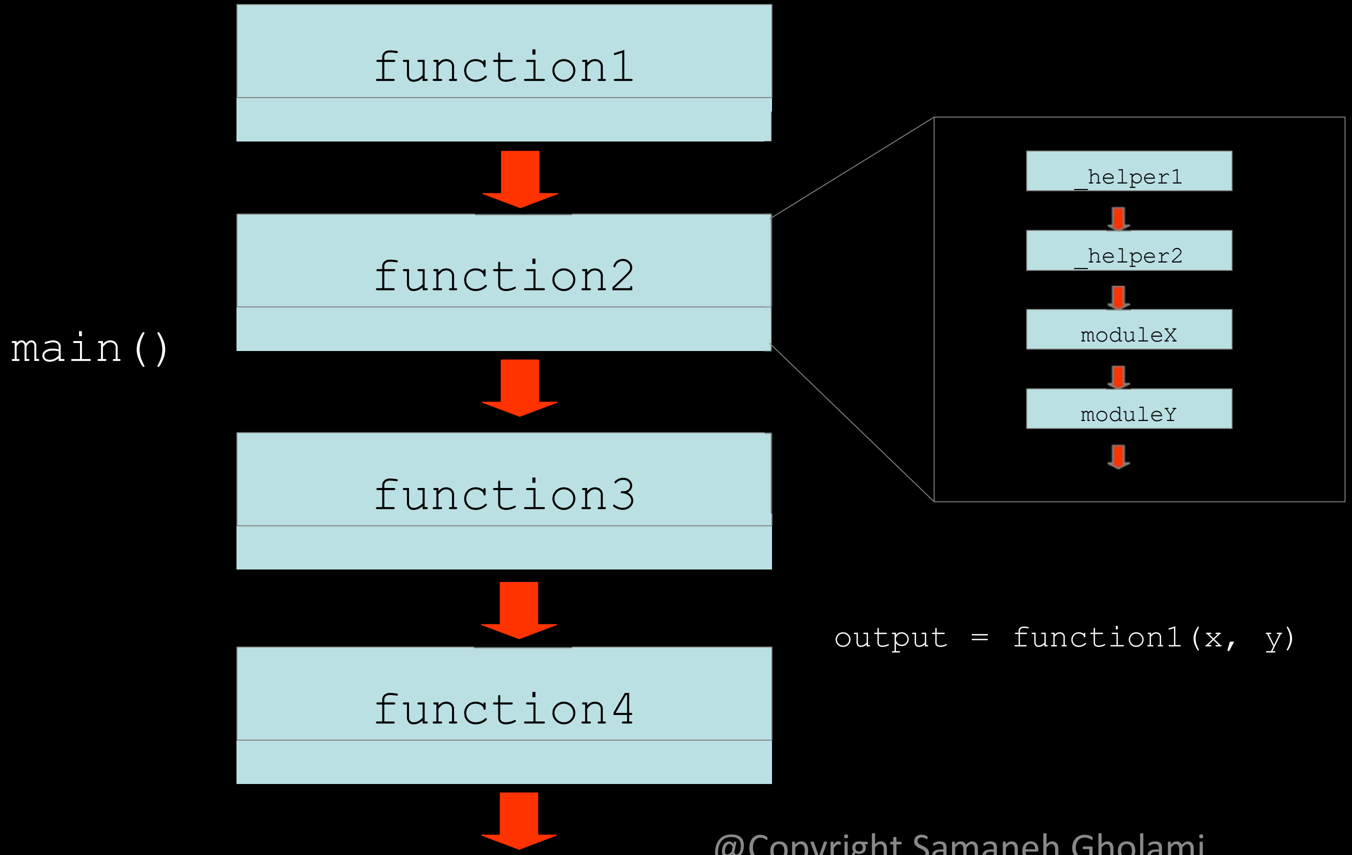
```
help(triangle)
Help on function triangle in module __main__:

triangle(size, letter)
    NoneType : triangle(size, letter)
    Takes: int in the first argument (size of the triangle)
    string (character) in the second argument
    returns: NoneType
    e.g. triangle(2, "+")
```

@Copyright Samaneh Gholami

Function Calls Within Functions

@Copyright Samaneh Gholami

Program Design Using Functions



```
output = function1(x, y)
```

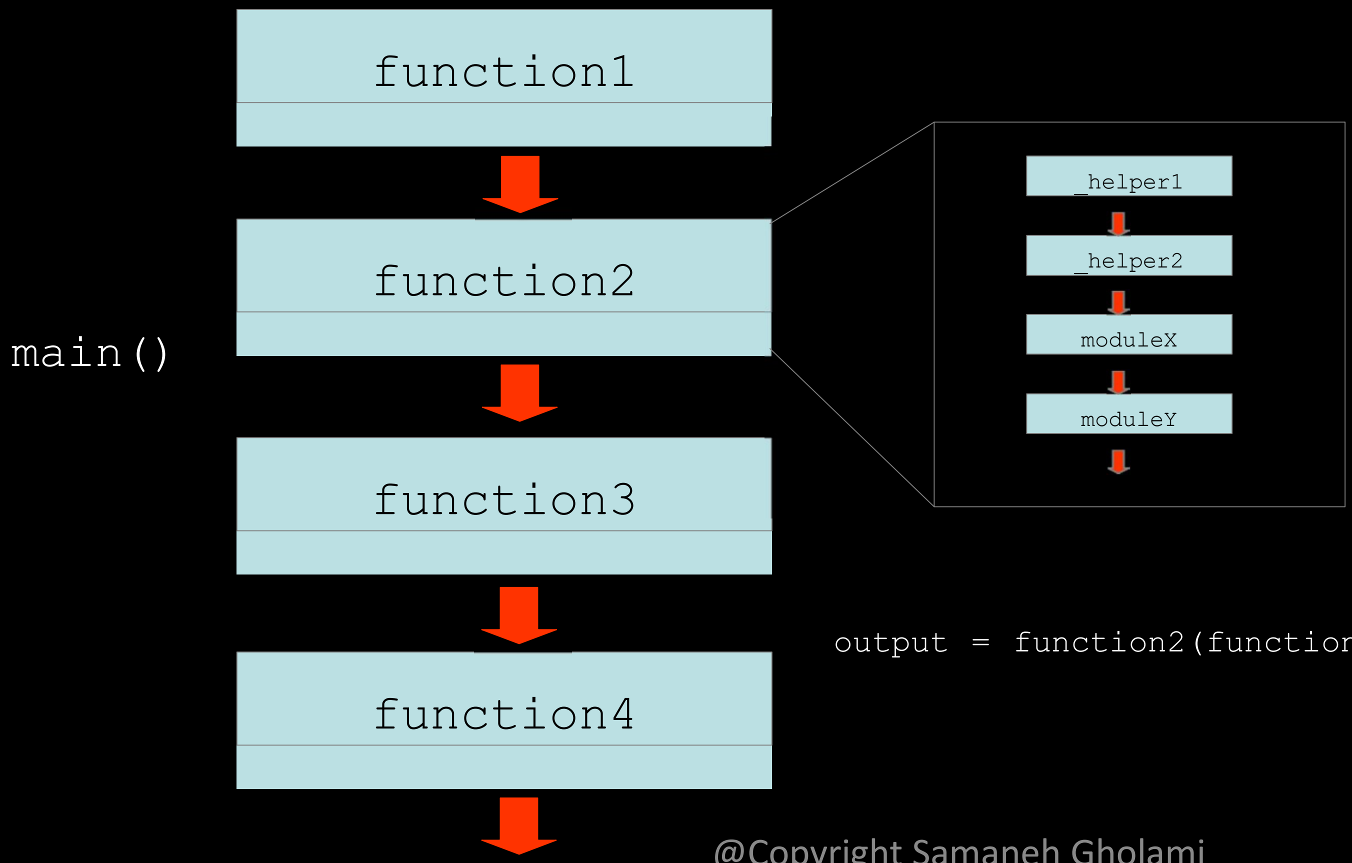
Functions can use **other** functions to make more complex and flexible programs

Functions allow complex processes to be broken up into smaller steps

@Copyright Samaneh Gholami

CL

Program Design Using Functions

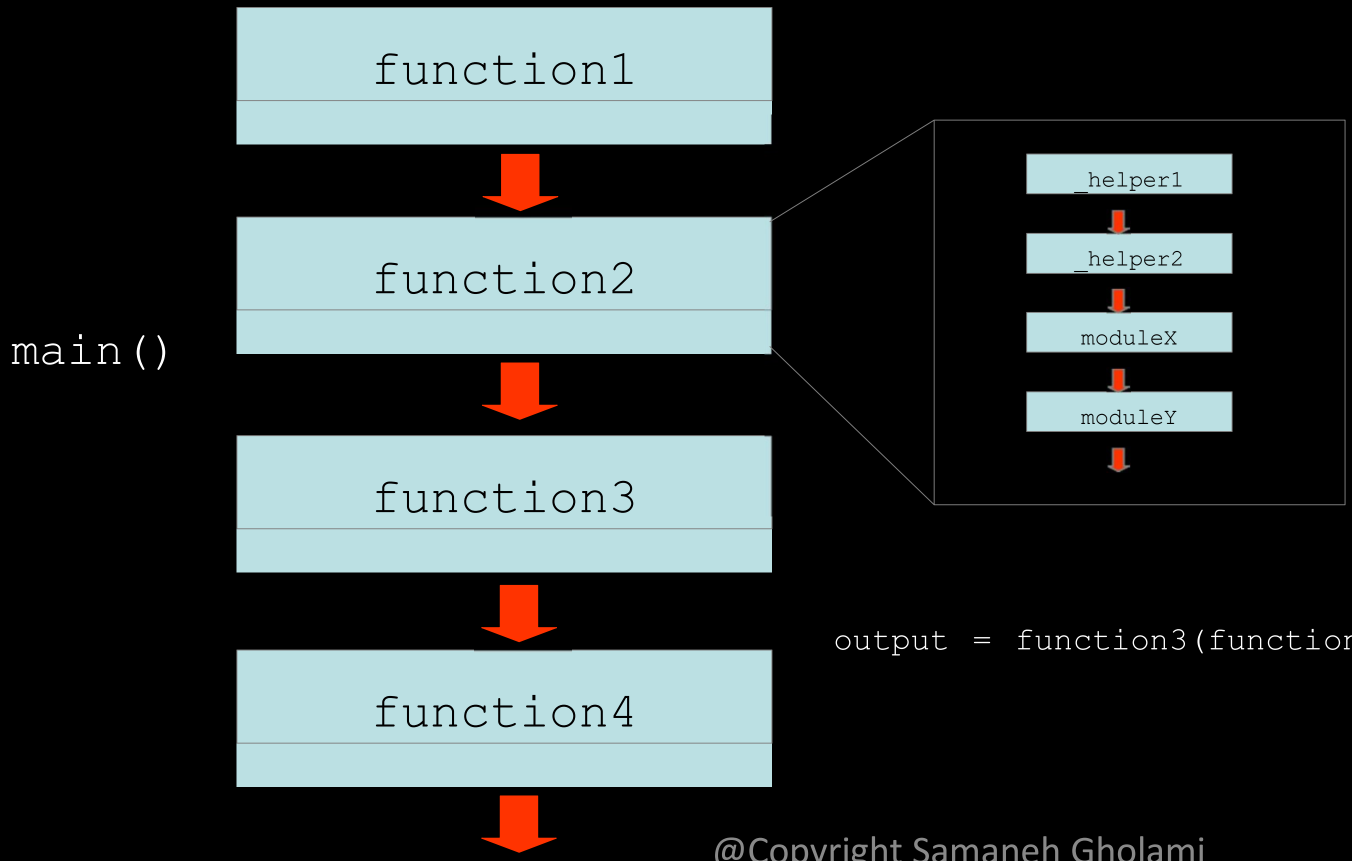


Functions can use **other** functions to make more complex and flexible programs

```
output = function2(function1(x, y))
```

@Copyright Samaneh Gholami

Program Design Using Functions



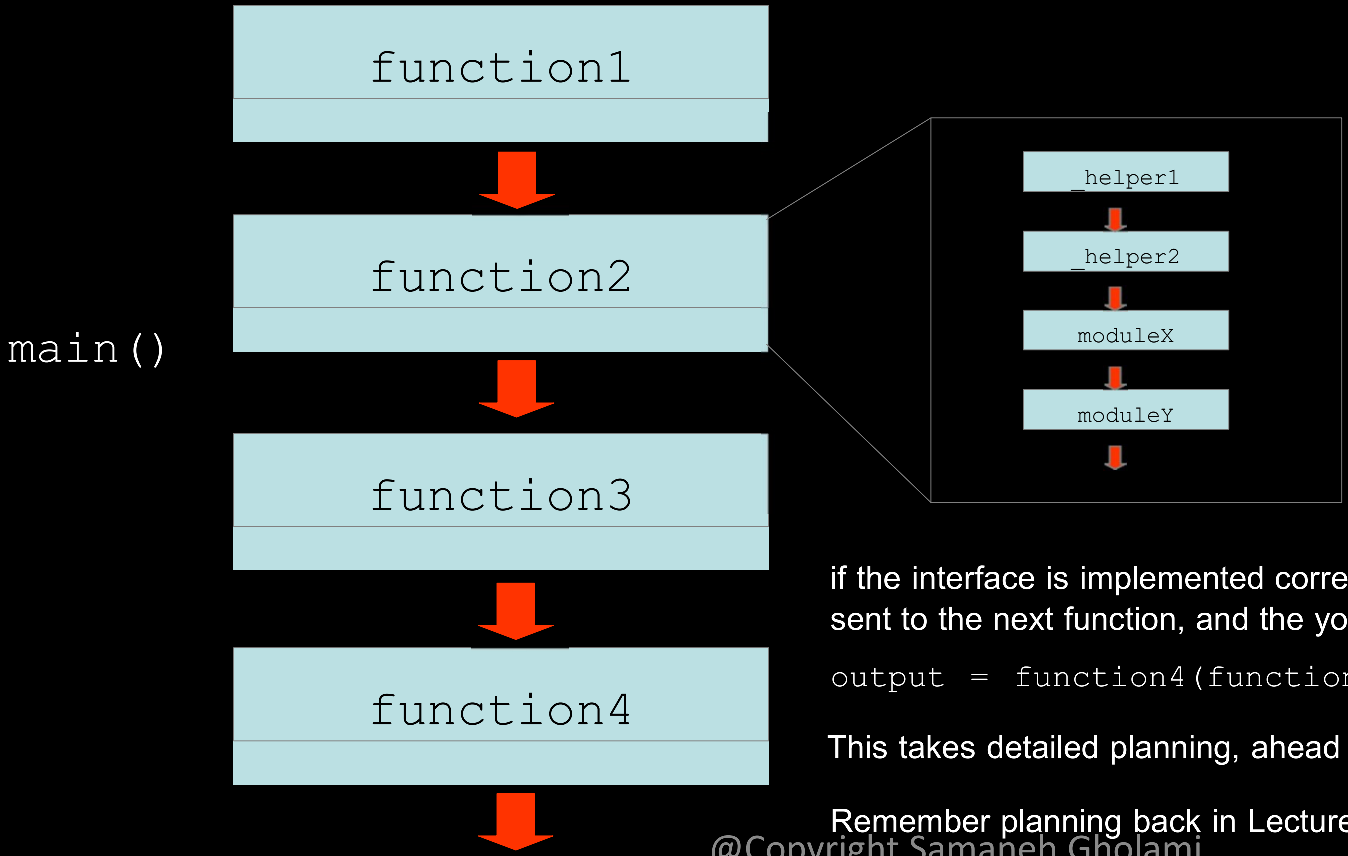
Functions can use **other** functions to make more complex and flexible programs

```
output = function3(function2(function1(x, y)))
```

@Copyright Samaneh Gholami

CL

Program Design Using Functions



Functions can use **other** functions to make more complex and flexible programs

if the interface is implemented correctly the output from one function can be sent to the next function, and the you can do the following:

```
output = function4(function3(function2(function1(x, y))))
```

This takes detailed planning, ahead of time…

Remember planning back in Lecture 4…

@Copyright Samaneh Gholami

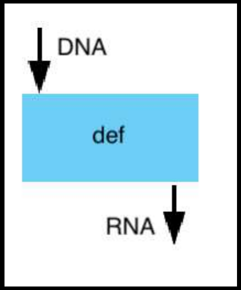
Function Calls Within Functions

```
def validate_base_sequene(base_sequence, RNAflag=False):
   """Return True if the string base_sequence contains only upper- or lowercase
   T (or U, if RNAflag), C, A, and G characters, otherwise False"""
   seq = base_sequence.upper()
   return len(seq) == (seq.count('U' if RNAflag else 'T') +
                  seq.count('C') +
                  seq.count('A') +
                  seq.count('G'))


def gc_content(base_seq):
   """Return the percentage of G and C characters in base_seq"""
   assert validate_base_sequene(base_seq), 'argument has invalid characters'
   seq = base_seq.upper()
   return ((base_seq.count('G') + base_seq.count('C')) / len(base_seq))*100

print(gc_content('ATCG'))
```

@Copyright Samaneh Gholami

Example: DNAtoRNA(dna)

Write a function that takes a DNA segment as input and returns the corresponding RNA segment as output .



function.py

@Copyright Samaneh Gholami