APS106

UNIVERSITY OF TORONTO

Advanced Functions and Aliasing

Week 5 | Lecture 5 (5.3.2)

While waiting for class to start:

Download and open the Jupyter Notebook (.ipynb) for Lecture 5.2.1

You may also use this lecture's JupyterHub link instead (although opening it locally is encouraged).

Upcoming (Today!):

- Reflection 5 released Friday @ 11 AM
- Lab 4 due this Friday @ 12 PM
- Lab 5 out already, due next Friday
- Behrang's Coffee Break / Office Hours Friday @ 1 PM
- Lab on Friday @ 2PM

if nothing else, write #cleancode

Today's Content

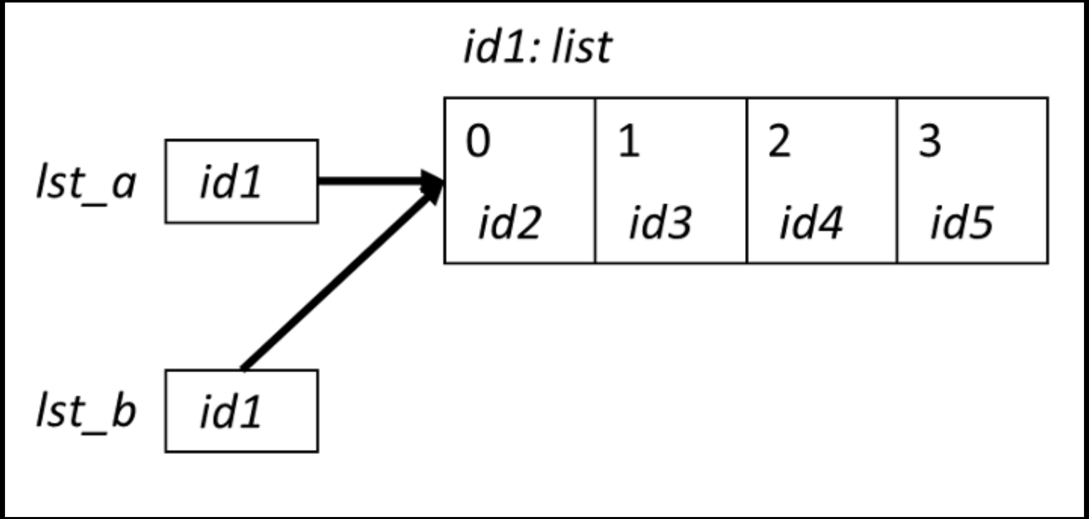
- **Lecture 5.3**
  - **Dictionaries**
  - Advanced functions

Taking functions to the next level!

- So far we have only covered the essential concepts related to functions
- In this lecture, we will discuss:
  - Aliasing
  - Creating and using default values

Aliasing

- When two variable names refer to the same object, they are aliases.
- When we modify one variable, we are modifying the object it refers to, hence also modifying the second variable.



- This is common source of error when working with list objects.

Avoiding Aliasing

```
>>> lst1 =  [11, 12, 13, 14, 15, 16, 27]
>>> lst2 = lst1
>>> lst1[-1] = 17
>>> lst2
[11, 12, 13, 14, 15, 16, 17]

>>> id(lst1)
49012568
>>> id(lst2)
49012568
```

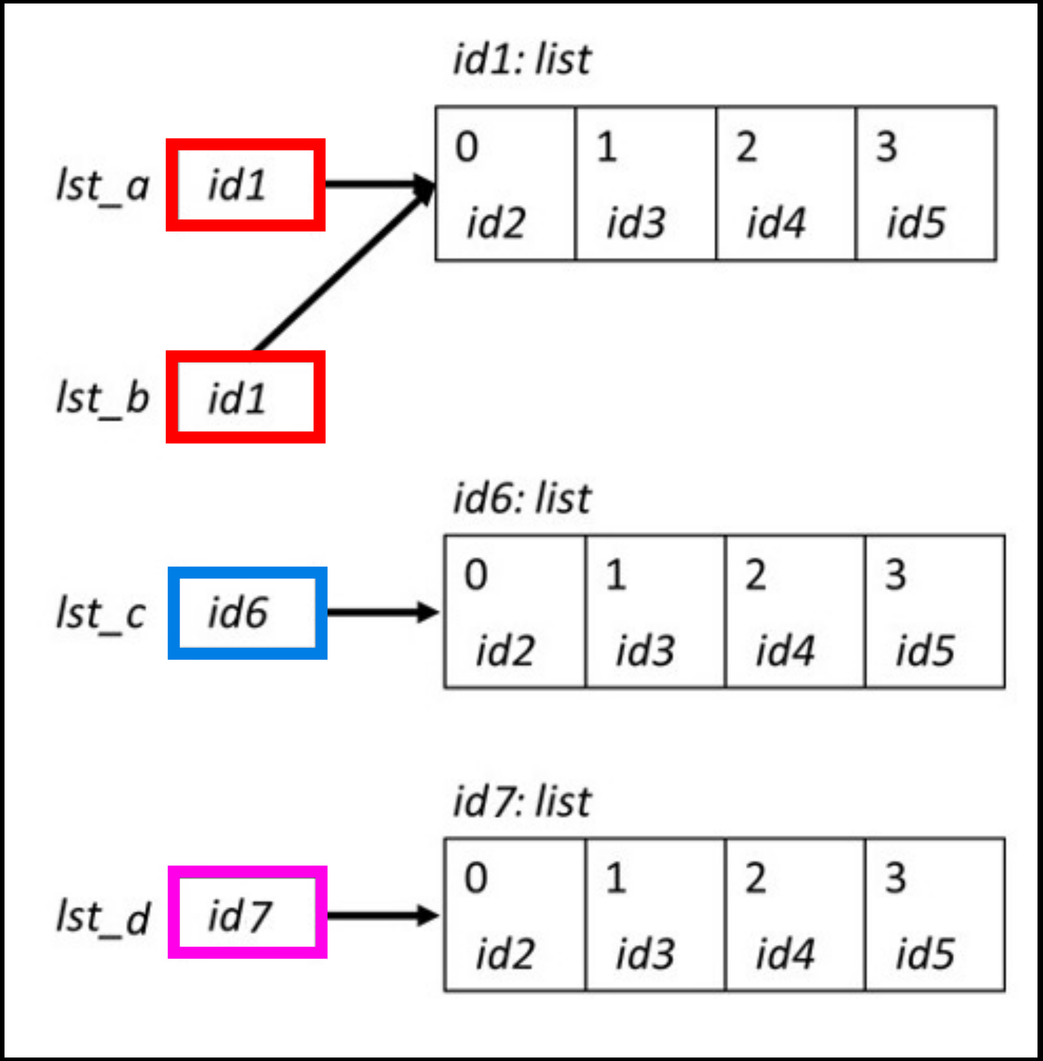
- How can we copy lst1 into another list without aliasing?

Copying Lists and Avoiding Aliasing

- There are two simple ways to copy lists:
  - Using the `list( )` function
  - Completely slice the list `[:]`

```
>>> lst_a =  [0, 1, 2, 3]
>>> lst_b = lst_a
>>> lst_c = list(lst_a)
>>> lst_d = lst_a[:]

>>> id(lst_a)
39012510
>>> id(lst_b)
39012510
>>> id(lst_c)
54514112
>>> id(lst_d)
24514139
```



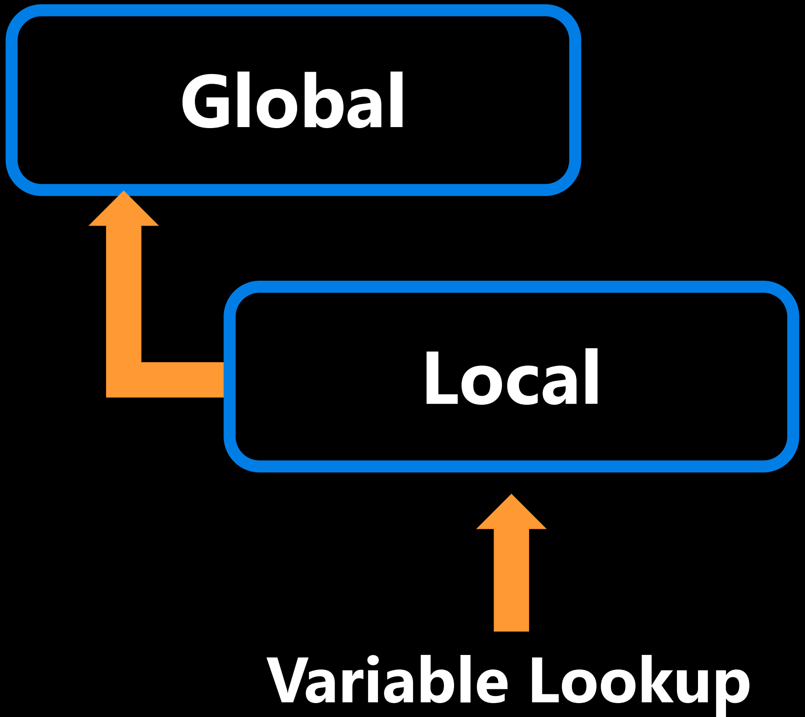
Summarizing and Revisiting Aliasing

- Python passes parameters by object references
  - An object is not copied, its reference is passed
- If the object being referenced is immutable (number, string, tuple), it is not possible to modify that object
- If the object being referenced is mutable (lists, sets, dictionaries), then a change made in the function is also reflected in the referenced object
  - This is called aliasing

- Catonio Banderas
- Kitty
- McHandsomePants
- Munchkin
- Fatso
- Little Predator
- Baby
- Furball

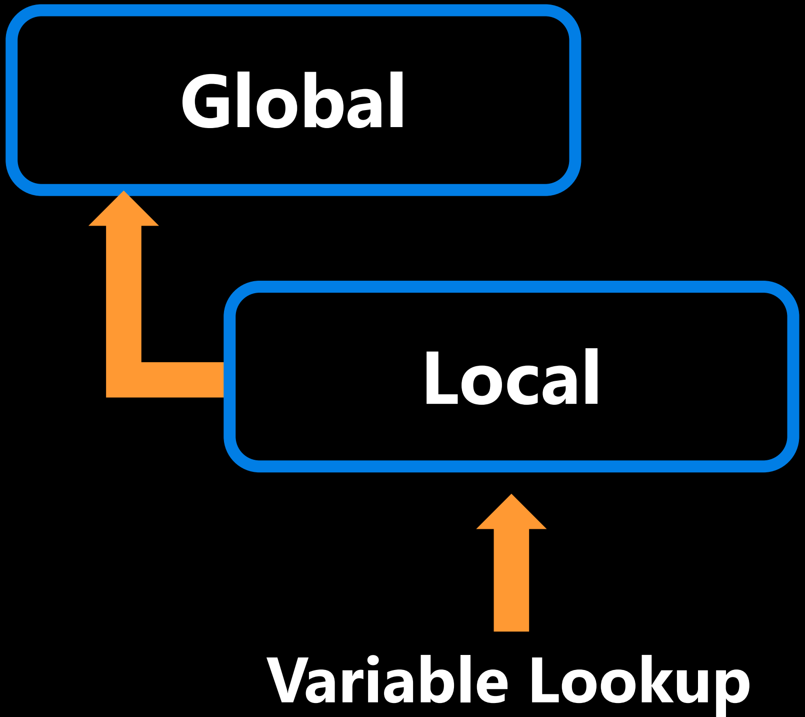
Recap: Variable Scope

- 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



Variable 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.



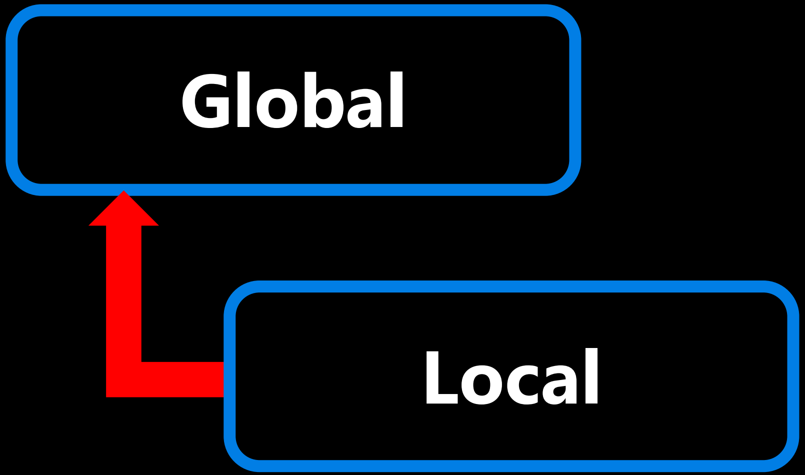
Variable Scope

- Local Scope

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


my_function()

print(name)
```

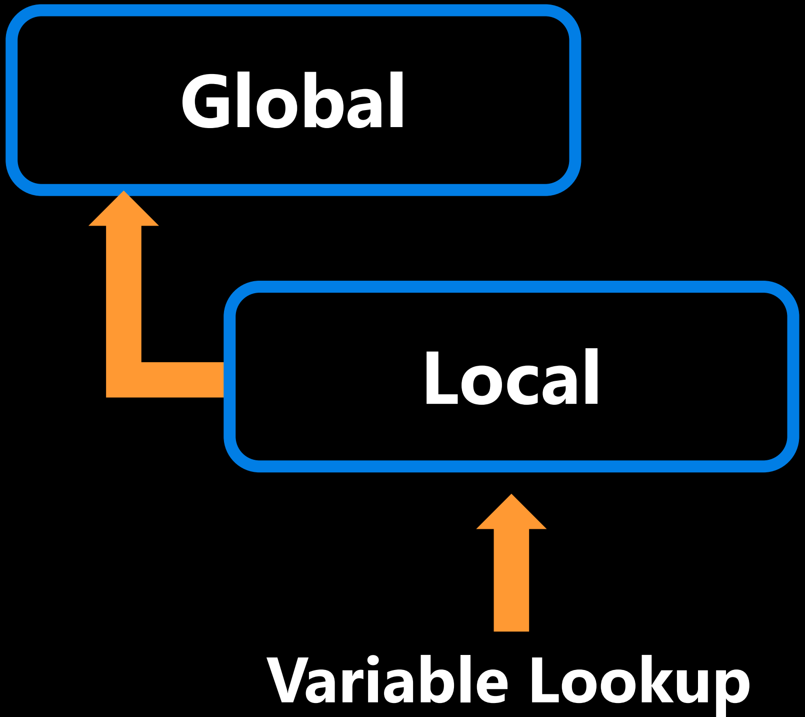


name is local to the function and not accessible outside.

```
Error: builtins.NameError: name 'name' is not defined
```

Variable 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.



Variable Scope

Notice that **name** is not defined anywhere when we define the function.

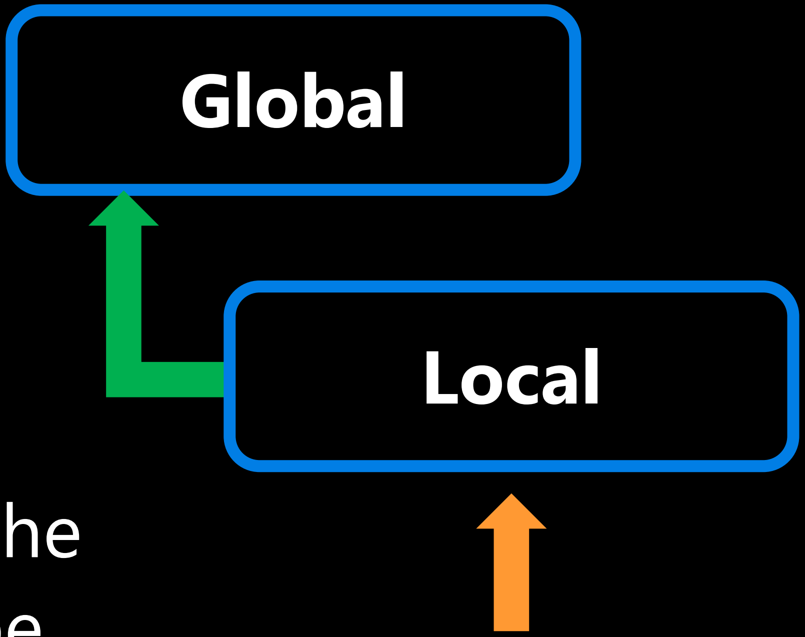
- Global Scope

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


name = 'Sebastian'

my_function()
```

OUTPUT: Sebastian



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

Variable Lookup

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

Variable Scope

- Global Scope

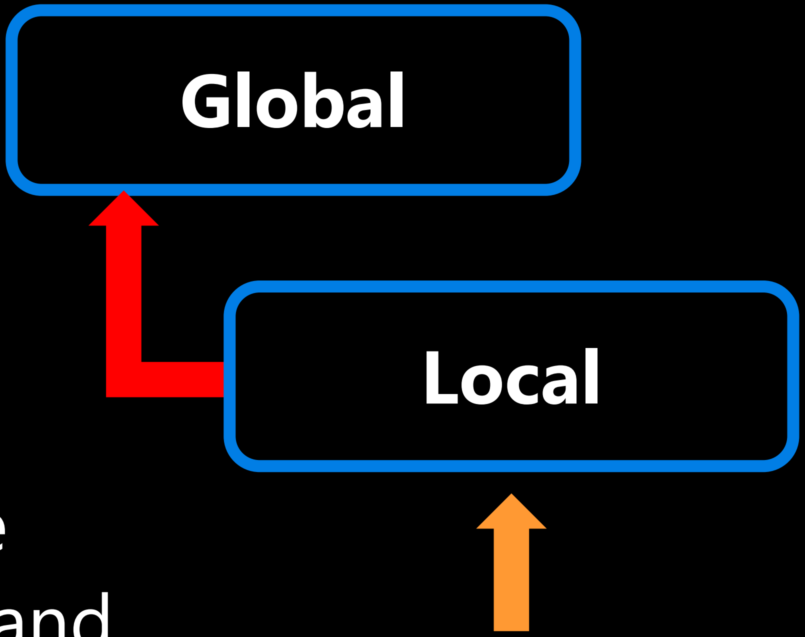
```
def my_function():
    name = 'Ben'
    print(name)


name = 'Sebastian'

my_function()
```

OUTPUT: Ben

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



Variable Lookup

- Is name in local?
- Yes (Done)

Example: Immutable Type

- When you pass an int to a function, the function gets a reference to the int object
- If the function modifies the int object, then the change is not reflected at the global scope level

```
def zero(x):
    x = 0
    return x

>>> x = 1
>>> x_new = zero(x+5)
>>> print(x_new)
0
>>> print(x)
1
```

Example: Mutable Type (Aliasing)

- When you pass a list to a function, the function gets a reference to the list
- If the function modifies the list parameter, then that change is reflected at the global scope level

```
def zero(some_list):
    '''
    (list)-> None
    changes all elements of some_list to zero
    '''
    for i in range(len(some_list)):
        some_list[i] = 0

>>> my_list = [0, 1, 2, 3, 4]
>>> zero(my_list)
>>> print(my_list)
[0, 0, 0, 0, 0]
```

Example: Mutable Type (Aliasing)

- When you pass a list to a function, the function gets a reference to the list
- If the function modifies the list parameter, then that change is reflected at the global scope level

```
def zero(some_list):
    '''
    (list)-> None
    changes all elements of some_list to zero
    '''
    new_list = some_list …
    for i in range(len(some_list)):
        new_list[i] = 0

>>> my_list = [0, 1, 2, 3, 4]
>>> zero(my_list)
>>> print(my_list)
[0, 0, 0, 0, 0]
```

This can be corrected by ensuring the `new_list` does not refer to the original `some_list` object

How can we do this?
new_list = some_list[:]
new_list = list(some_list)

Python Visualizer

Let’s see how it looks in a visualizer!

- https://tinyurl.com/aps106aliaslist

Let’s Code!

- Let’s take a look at how this works in Python!
  - Mutation
    - Modification of an object
  - Mutable vs Immutable
    - Can be mutated/modified
  - Aliasing
    - Two variables are referring to the same object in memory
    - Mutation through one variable affects the other variables

Open your notebook

Click Link:
1. More on Mutability and Aliasing

UNIVERSITY OF TORONTO

Let’s Recap Functions

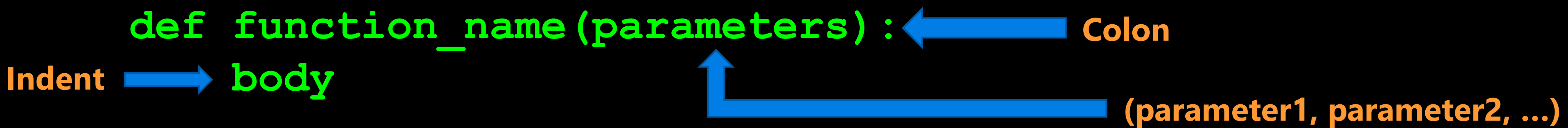
- The general form of a function call:

```
function_name(arguments)
```

- 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.

Function Definitions

- The general form of a function definition is:

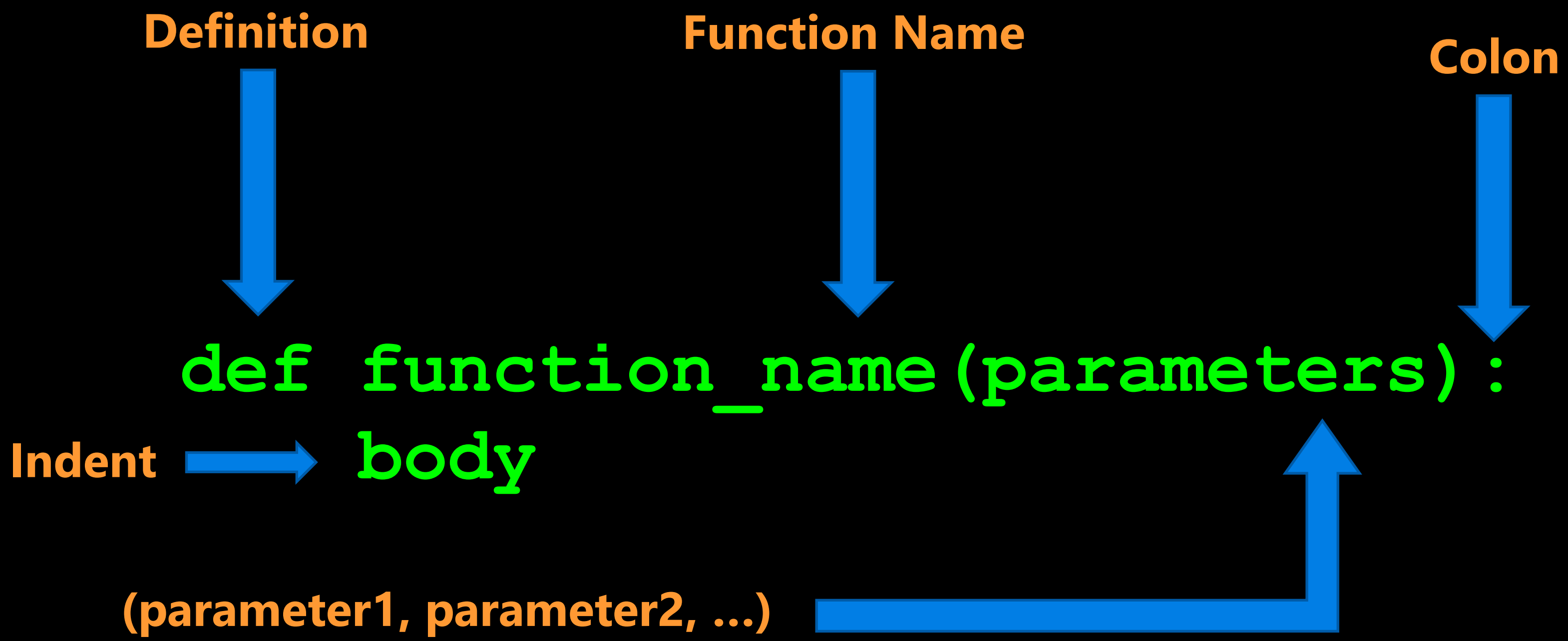


```
def function_name(parameters):
    body
```

- **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** is a sequence of commands like we've already seen (assignment, multiplication, function calls).
- **Important:** all the lines of body must be indented. That is how Python knows that they are part of the function.

Function Definitions

- The general form of a function definition is:



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

```
function_name(arguments)
```

Default Values

- When working with certain functions, such as range and print, you do not need to pass an argument for every parameter
  - If no parameter is passed, the default parameter values will be used
- For example:

```
>>> range(2, 5)
range(2, 5)

>>> range(10)
range(0, 10)
```

```
>>> print("hello", "world", sep='')
helloworld

>>> print("hello", "world")
hello world
```

UNIVERSITY OF TORONTO

Default Values of print

- Take a closer look at the print function

```
>>> help(print)
Help on built-in function print in module builtins:
print(...)
    print(value, ..., sep=' ', end='\n', file=sys.stdout)
    ...
```

- Here we see that print has several parameters
  - **value, ...,**: the values to be printed.
  - **sep=' '**: an optional argument that by default will be a space. When multiple values are printed, this string will be printed between pairs of values.
  - **end='\n'**: an optional argument that by default will be a newline character. This string is printed after the last value.
  - **file**: an optional argument that by default is sys.stdout, which specifies where to print.

Examples: Default Values of print

- Let’s look at some examples of the print function behaviour:

```
.py FILE:    print(123)                    OUTPUT:        123
             print(456)                                   456
```

- We see that the newline character ‘\n’ is automatically printed after ‘123’. We can also provide multiple values to the print function:

```
             print(123,456)                OUTPUT:      123 456
```

- Let’s use print again, but this time pass arguments to override the default parameter values:

```
             print(123,456, sep=‘’, end=‘!’)              OUTPUT:      123456!cats
             print(‘cats’)
```

- Notice the order does not matter:

```
             print(123,456, end=‘!’, sep=‘’)
             print(‘cats’)                                OUTPUT:      123456!cats
```

Python Visualizer

- Print examples visualized!
  - https://tinyurl.com/2p9c443y

What if we want default parameters in our custom functions?

Function Definitions One More Time…

- The general form of a function definition is:

Definition

Function Name

Colon

```
def function_name(param1, param2, …):
    body
```

Indent

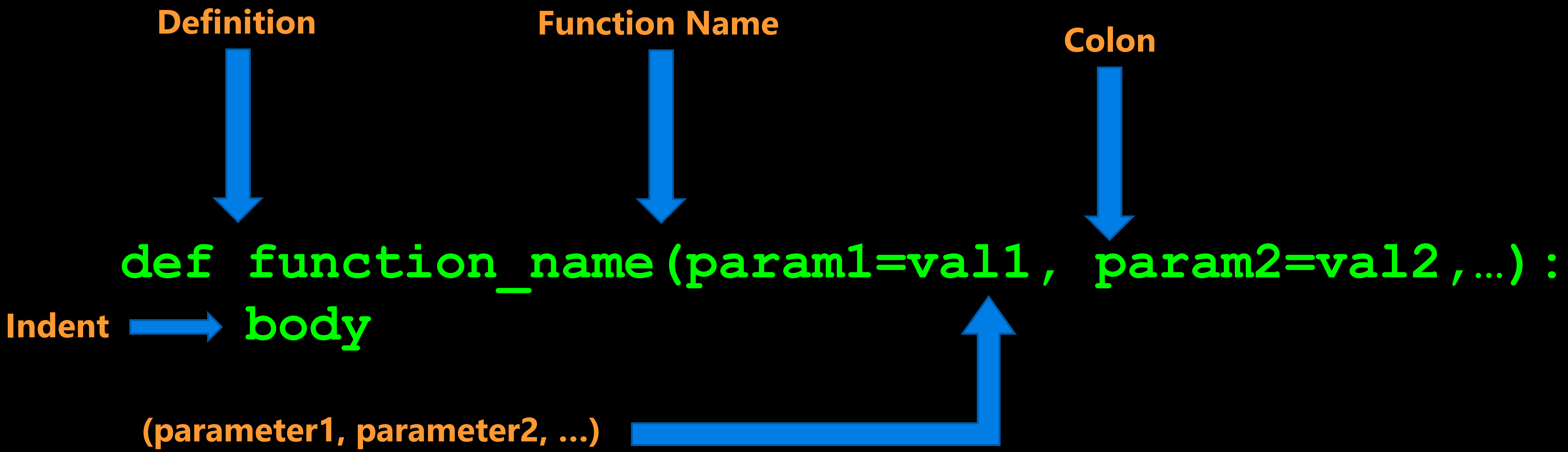
(parameter1, parameter2, …)

```
function_name(arg1, arg2)
```

"Why is my function not outputting anything"?
"Oh I never called the function"

Function Definitions One More Time...

- The general form of a function definition with default values is:



```
def function_name(param1=val1, param2=val2,...):
    body
```

```
function_name(arg1, arg2)
```

Function Definitions with Default Parameters

- The general form of a function definition with default values is:

```
def function_name(parameter1=val1, parameter2=val2,…):
    body
```

- Assigning a value to a **parameter** in the function definition indicates the default value (i.e. the value to use when no argument is provided.
- Using the above example, I could call `function_name()` or `function_name(val1, val2)` and it would be identical

A Greeting Example

- The general form of a function definition with default values is:

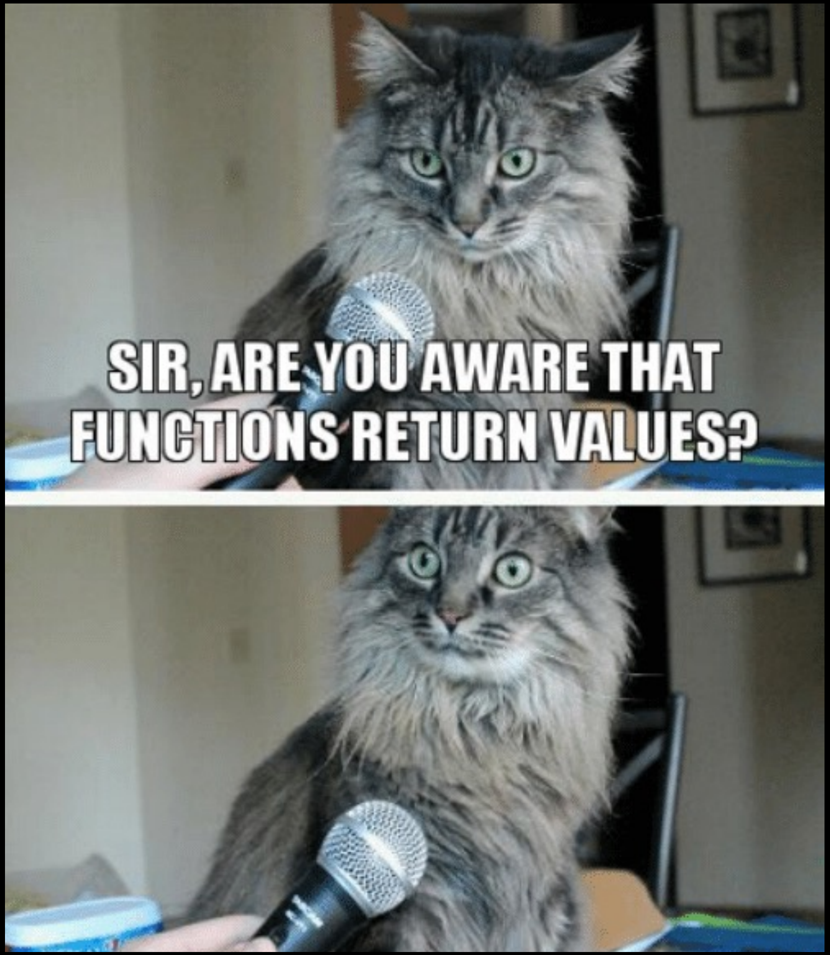
```
def make_greeting(title, name, surname, formal=True):
    if formal:
        return ("Hello " + title + " " + surname)
    return ("Hey " + name)

>>> print(make_greeting("Mr.", “Neo”, ”Anderson"))
Hello Mr. Anderson

>>> print(make_greeting("Mr.", “Neo”, ”Anderson", False)
Hey Neo
```

Python Visualizer

- https://tinyurl.com/7x79adcw



Let's Code!

- Let's take a look at how this works in Python!
  - Creating default parameters

Open your notebook

Click Link:
2. Default Function Values

Mentimeter Checkpoint

- Join at www.menti.com:
  - Code: 9198 0341
- Link
  - https://www.menti.com/bl9boggpujis

APS106

UNIVERSITY OF TORONTO

Advanced Functions and Aliasing

Week 5 | Lecture 5 (5.3.2)

if nothing else, write #cleancode