COL1000
Introduction to Programming

Priyanka Golia

Most (if not all) of the content is borrowed from Prof. Subodh Kumar's slides

Quiz

```python
def lists():
    return [1,2,3]

a = lists()
b = lists()
print(a is b)
```

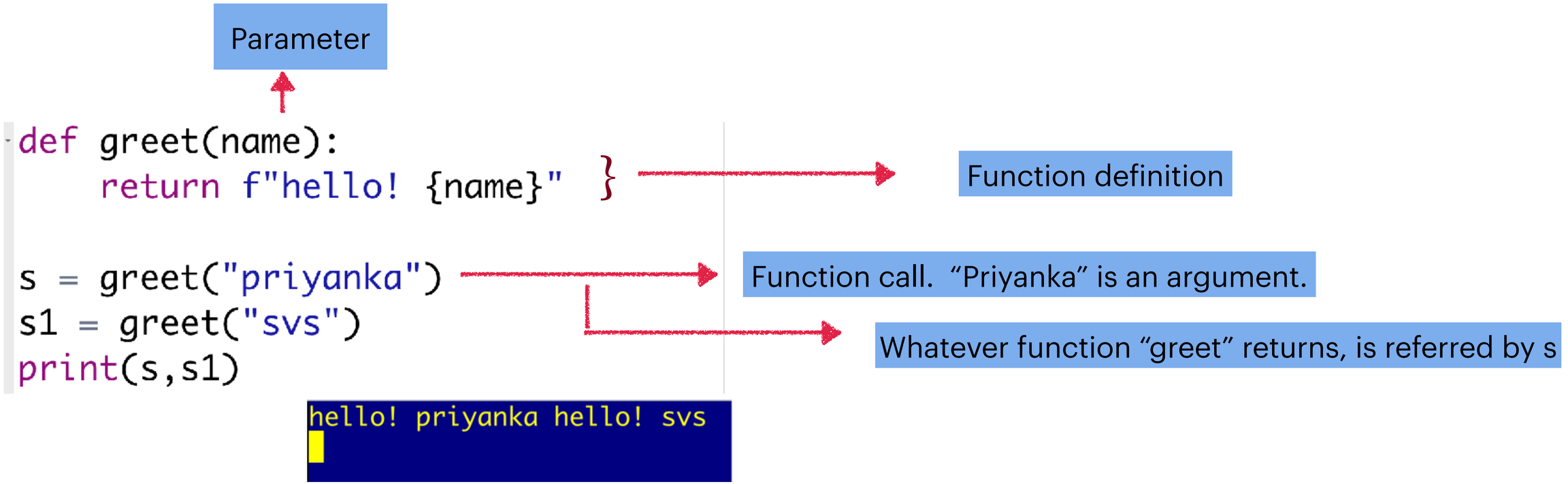
False

```python
def greet(name):
    print(f"hello! {name}")

s = greet("priyanka") + " how are you?"
print(s)
```

```
hello! priyanka
Traceback (most recent call last):
  File "run.py", line 1, in <module>
    import lec_main
  File "/home/p11653/lec_main.py", line 3, in <module>
    import lec18
  File "/home/p11653/lec18.py", line 4, in <module>
    s = greet("priyanka") + " how are you?"
TypeError: unsupported operand type(s) for +: 'NoneType' and 'str'
```

Functions



- **def** starts a function definition.
- greet is the **function name** (same rules as variable names).
- (name) are the **parameters**.
- **:** ends the header line; the indented block is the function body.
- **return** sends a value back to the caller (optional).

If no return, then it returns None

Parameters vs. Arguments

Parameter (at definition time) — A parameter is a placeholder you write in a function definition. It says: “This function expects a value here.”

```
def greet(name):    # ← "name" is a parameter
return f”hello {name}”
```

Here name is just a symbol (variable) inside the function — no actual value yet

An argument is the actual value (object) you pass when calling the function. It “fills” the parameter.

```
S = greet(“priyanka”)    # ← "priyanka" is an argument.
```

Definition doesn’t execute until unless it is called.

More on Arguments!!

```
def rectangle_area(width, height):
    return width * height

print(rectangle_area(3, 4))                    # positional
print(rectangle_area(width=3, height=4))       # keyword
print(rectangle_area(height=4, width=3))       # keyword can be reordered
```

Positional arguments

Keyword arguments

Notice if you are using keywords, you can reorder them

More on Arguments!!

```
1 def f (a, b=1, c=2):
2     print(f"a:{a}, b:{b}, c:{c}")
3     return a + b + c
4
5 print(f"sum is {f(3)}")
6 print(f"sum is {f(3,7)}")
7 print(f"sum is {f(3,5,8)}")
```

Will use default values of b and c.

Will use default values of c.

```
a:3, b:1, c:2
sum is 6
a:3, b:7, c:2
sum is 12
a:3, b:5, c:8
sum is 16
```

```
1 def f (a, b=1,c):
2     print(f"a:{a}, b:{b}, c:{c}")
3     return a + b + c
4
5 print(f"sum is {f(3,7)}")
6 print(f"sum is {f(3,5,8)}")
```

```
Traceback (most recent call last):
  File "run.py", line 1, in <module>
    import lec_main
  File "/home/p10944/lec_main.py", line 3, in <module>
    import lec19
  File "/home/p10944/lec19.py", line 1
    def f (a, b=1,c):
            ^
SyntaxError: non-default argument follows default argument
```

Does 7 go into b or into c?
Ambiguity! Python avoids that by forbidding such definitions.

More on Arguments!!

```
1 def f (a, *,b=1,c):
2     print(f"a:{a}, b:{b}, c:{c}")
3     return a + b + c
4
5 print(f"sum is {f(3,c=7)}")
```

Notice in the def. * in parameters allows a non default parameter to follow default one.

But, there is a catch — when calling parameters after * has be called via keyword arguments.

```
a:3, b:1, c:7
sum is 11
```

A bare asterisk (*) in the parameter list, without a variable name following it, indicates that all subsequent parameters when calling must be passed as keyword arguments

More on Arguments!!

```
1 def f (a, *,b=1,c):
2     print(f"a:{a}, b:{b}, c:{c}")
3     return a + b + c
4
5 print(f"sum is {f(3)}")
```

```
Traceback (most recent call last):
  File "run.py", line 1, in <module>
    import lec_main
  File "/home/p11225/lec_main.py", line 3, in <module>
    import lec19
  File "/home/p11225/lec19.py", line 5, in <module>
    print(f"sum is {f(3)}")
TypeError: f() missing 1 required keyword-only argument: 'c'
```

Notice the error! It say keyword only argument

```
1 def f (a,b, * ,c):
2     print(f"a:{a}, b:{b}, c:{c}")
3     return a + b + c
4
5 print(f"sum is {f(1,c=1)}")
```

```
Traceback (most recent call last):
  File "run.py", line 1, in <module>
    import lec_main
  File "/home/p10729/lec_main.py", line 3, in <module>
    import lec19
  File "/home/p10729/lec19.py", line 5, in <module>
    print(f"sum is {f(1,c=1)}")
TypeError: f() missing 1 required positional argument: 'b'
```

Notice the error!

More on Arguments!!

```python
def f (*num):
    print(f"inside function, num:{num}")
    sum = 0
    for i in num:
        sum += i
    return sum

print(f"sum is {f(1,2,3,4)}")
```

To accept an arbitrary number of positional arguments (`*args`):
When you define a parameter with an asterisk before its name, like `*args`, it allows the function to accept any number of positional arguments. These arguments are then collected into a tuple.

```
inside function, num:(1, 2, 3, 4)
sum is 10
```

More on Arguments!!

```
1 def f (a, *b, c):
2     print(f"inside function")
3     print(f"a:{a}, b:{b}, c:{c}")
4     return a + sum(b) + c
5
6 print(f"sum is {f(1,2,3,4)}")
```

```
    print(f"sum is {f(1,2,3,4)}")
TypeError: f() missing 1 required keyword-only argument: 'c'
```

Again, Ambiguity! a= 1, but then what about b and c, hence the error.

```
1 def f (a, *b, c):
2     print(f"inside function")
3     print(f"a:{a}, b:{b}, c:{c}")
4     return a + sum(b) + c
5
6 print(f"sum is {f(1,2,3,c=4)}")
```

```
inside function
a:1, b:(2, 3), c:4
sum is 10
```

“Return” in Functions

```
def greet(name):
    return f"hello! {name}"

s = greet("priyanka") + " how are you?"
print(s)
```

```
hello! priyanka how are you?
```

```
1 def greet(name):
2     print(f"hello! {name}")
3
4 s = greet("priyanka") + " how are you?"
5 print(s)
```

```
hello! priyanka
Traceback (most recent call last):
  File "run.py", line 1, in <module>
    import lec_main
  File "/home/p11653/lec_main.py", line 3, in <module>
    import lec18
  File "/home/p11653/lec18.py", line 4, in <module>
    s = greet("priyanka") + " how are you?"
TypeError: unsupported operand type(s) for +: 'NoneType' and 'str'
```

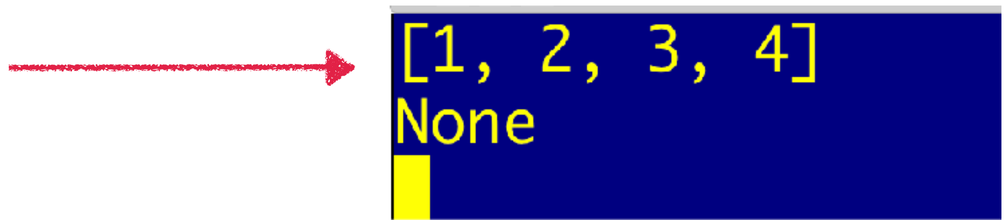
You can use explicit return statements with or without a return value. If you build a return statement without specifying a return value, then you’ll be implicitly returning None.

“Return” in Functions

```
1 n = print("name is priyanka")
2 print(n)
```

Is there any error or is it correct?

```
1 L = [1,2,3]
2 n = L.append(4)
3 print(L)
4 print(n)
```



“Return” in Functions

```
1 def lists():
2     return [1,2,3]
3
4 a = lists()
5 b = lists()
6 print(a is b)
```

```
False
```

Return creates a new object everytime we call the functions, and return the reference.

```
1 def details(name,aff,year):
2     return name,aff,year
3
4 n = details("priyanka", "iitd", "first")
5 print(n)
6 n,a,y = details("svs", "iitd", "first")
7 print(n,a,y)
8 n, *a = details("subodh", "iitd", "first")
9 print(n,a)
```

Function can return reference to only one object, so multiple values are “packed” into a tuple automatically.

Now, in right hand side, we have a tuple object. All the rules of advance assignments apply here!

```
('priyanka', 'iitd', 'first')
svs iitd first
subodh ['iitd', 'first']
```

Scope!

Scope is the region of the program where a variable is accessible.

```
1 def add(*num):
2     print(f"inside function x: {x}")
3     sum1 = 0
4     for i in num:
5         sum1 += i
6     print(f"inside function sum:{sum1}")
7     return sum1
8
9 x = 10
10 s = add(1,2,3,4,5,6)
11 s1 = add(5,6,6,6,6)
12 print("total sum is", s+s1)
13 print(sum1)
```

X is defined outside of function definition, it can be accessed here, but no modification is allowed.

Sum1 is local to function. This will give us “name error”

```
inside function x: 10
inside function sum:21
inside function x: 10
inside function sum:29
total sum is 50
Traceback (most recent call last):
  File "run.py", line 1, in <module>
    import lec_main
  File "/home/p10783/lec_main.py", line 3, in <module>
    import lec19
  File "/home/p10783/lec19.py", line 13, in <module>
    print(sum1)
NameError: name 'sum1' is not defined
```

Scope!

Scope is the region of the program where a variable is accessible.

```
1  def add(*num):
2      y = x + 100
3      print(f"inside function execution x: {x}, and y: {y}")
4      sum1 = 0
5      for i in num:
6          sum1 += i
7      print(f"inside function sum:{sum1}")
8      return sum1
9
10 x = 10
11 y = 20
12 print(f"before function execution x: {x}, and y: {y}")
13 s = add(1,2,3,4,5,6)
14 s1 = add(5,6,6,6,6)
15 print(f"after function execution x: {x}, and y: {y}")
16 print("total sum is", s+s1)
```

This y is local to function add.
any chances in this y, doesn't reflect outside the execution of add

```
before function execution x: 10, and y: 20
inside function execution x: 10, and y: 110
inside function sum:21
inside function execution x: 10, and y: 110
inside function sum:29
after function execution x: 10, and y: 20
total sum is 50
```

Scope!

```python
def add(*num):
    x = x + 100
    print(f"inside function execution x: {x}, and y: {y}")
    sum1 = 0
    for i in num:
        sum1 += i
    print(f"inside function sum:{sum1}")
    return sum1

x = 10
y = 20
print(f"before function execution x: {x}, and y: {y}")
s = add(1,2,3,4,5,6)
s1 = add(5,6,6,6,6)
print(f"after function execution x: {x}, and y: {y}")
print("total sum is", s+s1)
```

Notice the only change from the previous slide

Because there is an assignment $x = x + 100$, Python assumes x is a **local variable** in this function.

But when evaluating $x + 100$, it tries to read the local x **before it has been created**, which leads to: UnboundLocalError.

```
before function execution x: 10, and y: 20
Traceback (most recent call last):
  File "run.py", line 1, in <module>
    import lec_main
  File "/home/p11141/lec_main.py", line 3, in <module>
    import lec19
  File "/home/p11141/lec19.py", line 13, in <module>
    s = add(1,2,3,4,5,6)
  File "/home/p11141/lec19.py", line 2, in add
    x = x + 100
UnboundLocalError: local variable 'x' referenced before assignment
```

Scope!

```
1  def add(*num):
2      global x
3      x = x + 100
4      print(f"inside function execution x: {x}, and y: {y}")
5      sum1 = 0
6      for i in num:
7          sum1 += i
8      print(f"inside function sum:{sum1}")
9      return sum1
10
11 x = 10
12 y = 20
13 print(f"before function execution x: {x}, and y: {y}")
14 s = add(1,2,3,4,5,6)
15 s1 = add(5,6,6,6,6)
16 print(f"after function execution x: {x}, and y: {y}")
17 print("total sum is", s+s1)
```

To tell Python, “I want to use the global variable, not create a local one”, you declare it with global

Modifies the global x

Before second execution of “add” value of x is 110

Notice the value of x

```
before function execution x: 10, and y: 20
inside function execution x: 110, and y: 20
inside function sum:21
inside function execution x: 210, and y: 20
inside function sum:29
after function execution x: 210, and y: 20
total sum is 50
```