

LECTURE 4.1

Functions

Write once, use everywhere, and how Python passes arguments

Today's key line of code

```
def add(a, b):  
    return a + b  
  
print(add(5, 6))
```

Where we are: Lecture 4, part 1

Part 1



Today you will

- 1 Name every part of a function definition and call
- 2 Send values in with parameters, get values out with return
- 3 Use keyword and default arguments
- 4 Know which variables a function can see
- 5 Predict what a function can and cannot change in the caller

The problem: the same lines, again and again

Without a function

Code

```
account = 20
balance = 100
balance = balance + 100
print("Account:", account)
print("Balance:", balance)
balance = balance - 50
print("Account:", account)
print("Balance:", balance)
balance = balance - 10
print("Account:", account)
print("Balance:", balance)
```

With a function

Code

```
def show(account, balance):
    print("Account:", account)
    print("Balance:", balance)

account = 20
balance = 100
balance = balance + 100
show(account, balance)
balance = balance - 50
show(account, balance)
balance = balance - 10
show(account, balance)
```

Change the report format once, inside **show**, and every call uses the new format.

You already use functions

1

Built-in

`print()`, `len()`, `type()`,
`input()`, `max()`: written by
Python's developers, ready to
use.

2

From modules

`math.sqrt()`,
`pd.read_csv()`: written by
others, loaded with `import`.

3

Your own

Written with `def`. Today's topic.

Anatomy of a function

Code

```
def hello():  
    print("Hello World!")
```

```
hello()
```

Output

```
Hello World!
```

Keyword **def** defines a function

Function name: you choose it

Brackets: inputs go here (none yet)

Body: indented, runs only when the function is called

Call: runs the body. Brackets are required

Parameters and arguments

Code

```
def hello(name):  
    print(f"Hello {name}!")  
  
hello("Joy")  
hello("Rini")
```

Parameter: a name for the input, used inside the function

The parameter is used like a normal variable

Argument: the actual value sent in

Another call, another argument, same code

Output

```
Hello Joy!  
Hello Rini!
```

What if we forget the argument?

Predict first

Code

```
def printme(text):  
    print(text)  
  
printme()
```



What will this print? Write your prediction as a comment before you run the cell.

Answer: What if we forget the argument?

Code

```
def printme(text):  
    print(text)  
  
printme()
```

Output

```
TypeError: printme() missing 1 required positional argument: 'text'
```

- `printme` has one required parameter
- The call sends nothing, so Python stops with a `TypeError`
- The message names the missing parameter: `text`
- Fix: `printme("Hello")`

return: hand a value back

Code

```
def add(a, b):  
    s = a + b  
    return s  
  
x = add(5, 6)  
print(x)
```

Two parameters, separated by a comma

return sends the value back and ends the function

The returned value is stored in x

5 goes to a, 6 goes to b, by position

Output

11

Watch it run: a function call

WATCH FOR

- Defining a function does not run it
- At the call, a new orange frame appears
- Parameters a and b get the argument values
- return sends 11 back, and the frame disappears

WATCH THE CODE RUN, STEP BY STEP

A function call, step by step

Watch the arguments go IN, a new frame appear, and the result come OUT.
Orange box = the function's own frame.

Click the video to play, click again to pause. Silent: talk the class through it.

print or return?

Predict first

Code

```
def double_print(x):  
    print(x * 2)  
  
def double_return(x):  
    return x * 2  
  
a = double_print(4)  
b = double_return(4)  
print(a, b)
```



What will this print? Write your prediction as a comment before you run the cell.

Answer: print or return?

Code

```
def double_print(x):  
    print(x * 2)  
  
def double_return(x):  
    return x * 2  
  
a = double_print(4)  
b = double_return(4)  
print(a, b)
```

Output

```
8  
None 8
```

- `double_print` shows 8, but gives nothing back
- A function without return gives back `None`
- `double_return` gives back 8, stored in b
- To use a result later, return it

Watch it run: print versus return

WATCH FOR

- Both functions produce 8
- Only `double_return` hands it back
- Watch `a` become `None`, `b` become 8
- Shown on screen is not the same as returned

WATCH THE CODE RUN, STEP BY STEP

print or return?

Both functions show or give 8. Only one of them gives it BACK.
Watch the values of `a` and `b` in the main frame.

Click the video to play, click again to pause. Silent: talk the class through it.

Returning several values

- `return a, b, c` packs the values into a tuple
- Unpack them into separate names (Lecture 3.2)
- The number of names must match
- Avoid names like `str` and `sum`: they hide built-ins

Code

```
def check():  
    return "ok", 10, 3.14  
  
result = check()  
print(result)  
print(type(result))  
status, num1, num2 = check()  
print(status, num1, num2)
```

Output

```
('ok', 10, 3.14)  
<class 'tuple'>  
ok 10 3.14
```

Keyword arguments: pass by name

- Name the parameter in the call: `age=50`
- Then order does not matter
- Makes calls with many inputs easier to read
- Lecture 6 uses this a lot:
`test_size=0.2`

Code

```
def printinfo(name, age):  
    print("Name:", name)  
    print("Age:", age)  
  
printinfo(age=50, name="Miki")
```

Output

```
Name: Miki  
Age: 50
```

Default arguments: a value if none is given

- `age=35` in the definition is a default
- Leave it out and 35 is used
- Pass a value and it replaces the default
- Parameters with defaults must come after those without

Code

```
def printinfo(name, age=35):  
    print("Name:", name, "| Age:", age)  
  
printinfo("Miki")  
printinfo("Miki", 50)  
printinfo(name="Joy", age=20)
```

Output

```
Name: Miki | Age: 35  
Name: Miki | Age: 50  
Name: Joy | Age: 20
```

The rules for arguments

Read the last line of an error first. It names the problem.

What you typed	What Python says	What it means
<pre>def f(age=35, name): pass</pre>	<code>SyntaxError: parameter without a default follows parameter with a default</code>	Parameters with defaults must come after required ones.
<pre>printinfo(name="Miki", 50)</pre>	<code>SyntaxError: positional argument follows keyword argument</code>	Positional arguments must come before keyword arguments.
<pre>printinfo("Miki", name="Joy")</pre>	<code>TypeError: printinfo() got multiple values for argument 'name'</code>	name got two values: one by position, one by keyword.

Variables inside a function stay inside

- `greeting` is a local variable: it exists only while the function runs
- Outside, Python has never heard of it
- Inputs come in through parameters
- Results go out through return

Code

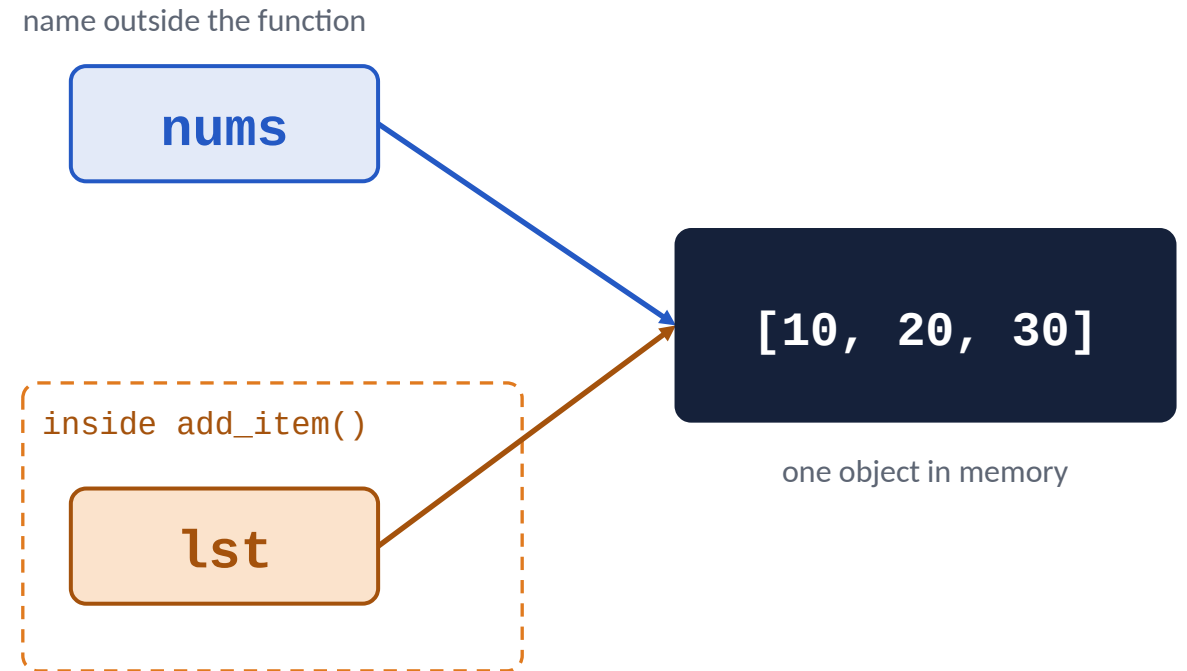
```
def make_greeting():  
    greeting = "Hi"  
    return greeting  
  
print(make_greeting())  
print(greeting)
```

Output

```
Hi  
NameError: name 'greeting' is not defined
```

How arguments are passed: names point to objects

- A variable is a name that points to an object
- Passing an argument does not copy the object
- The parameter becomes a second name for the same object
- Python's rule is called pass by object reference (or pass by assignment)
- It is neither classic call by value nor call by reference



Evidence: the same object inside and outside

- `id()` gives an object's identity number
- Same number means same object
- So the function received the object itself, not a copy
- Your numbers will differ; they only need to match

Code

```
def show_id(arg):  
    print("inside: ", id(arg))  
  
var = "Hello"  
print("outside:", id(var))  
show_id(var)
```

Output

```
outside: 140601564316704  
inside: 140601564316704
```

Changing a list inside a function changes it outside

- `lst.append(100)` changes the list object itself
- `nums` names that same object
- So the caller sees the change
- Lists and dictionaries are mutable: they can be changed

Code

```
def add_item(lst):  
    lst.append(100)  
  
nums = [10, 20, 30]  
add_item(nums)  
print(nums)
```

Output

```
[10, 20, 30, 100]
```

But reassigning inside a function does not

- `lst = [99]` points the local name at a new list
- The caller's name still points to the old list
- Under true call by reference, `nums` would now be `[99]`
- So Python is not call by reference

Code

```
def replace(lst):  
    lst = [99]  
    print("inside: ", lst)  
  
nums = [10, 20, 30]  
replace(nums)  
print("outside:", nums)
```

Output

```
inside:  [99]  
outside: [10, 20, 30]
```

Numbers and strings can only be replaced

- Integers and strings are immutable: no operation changes them
- `n = n + 1` makes a new object and moves the local name
- The caller's `x` is untouched
- To get a new value out, return it: `x = add_one(x)`

Code

```
def add_one(n):  
    n = n + 1  
    return n  
  
x = 10  
add_one(x)  
print(x)  
x = add_one(x)  
print(x)
```

Output

```
10  
11
```

Watch it run: change the object, or move the name

WATCH FOR

- Green note: lst and nums name the same list
- append changes that list, so nums changes too
- `lst = [99]` moves only the local name
- After replace, nums is unchanged

WATCH THE CODE RUN, STEP BY STEP

Lists inside functions: change vs replace

Watch the green note: 'same object as nums'.

Changing the object is visible outside. Moving a name is not.

Click the video to play, click again to pause. Silent: talk the class through it.

The one rule to remember

<code>lst.append(x)</code>	Changes the object: caller sees it
<code>lst[0] = x</code>	Changes the object: caller sees it
<code>d[key] = x</code>	Changes the object: caller sees it

<code>lst = [...]</code>	Moves the local name: caller does not
<code>n = n + 1</code>	New object: caller does not see it
<code>return value</code>	The reliable way to send results out

Built-in math functions

You type	Python gives	Meaning
<code>abs(-5)</code>	5	distance from zero
<code>max(5, 9)</code>	9	largest of the arguments
<code>max([4, 9, 2])</code>	9	or largest item of a list
<code>min(5, 9)</code>	5	smallest
<code>pow(2, 3)</code>	8	$2^{**}3$
<code>pow(2, 3, 5)</code>	3	$(2^{**}3) \% 5$
<code>round(5.8)</code>	6	nearest whole number
<code>round(3.14159, 2)</code>	3.14	round to 2 decimal places
<code>round(2.5)</code>	2	halves go to the even number
<code>sum([5, 6])</code>	11	sum needs a list, not <code>sum(5, 6)</code>

Mistakes to avoid

Read the last line of an error first. It names the problem.

What you typed	What Python says	What it means
<pre>sum(5, 6)</pre>	<pre>TypeError: 'int' object is not iterable</pre>	sum needs one list: <code>sum([5, 6])</code> .
<pre>sum = 10 print(sum([1, 2]))</pre>	<pre>TypeError: 'int' object is not callable</pre>	<code>sum = 10</code> hid the built-in <code>sum</code> . Never use built-in names for variables.
<pre>nums = [1, 2] nums = nums.append(3) print(nums)</pre>	<pre>None</pre>	No error, but <code>None</code> : <code>append</code> returns nothing, and <code>nums</code> lost its list.

Modules: borrowing more functions

- A module is a file of ready-made functions
- `import math` loads it; use `math.name`
- `from math import lcm` imports one name directly
- Use `math.gcd` to check your exercise answers

Code

```
import math
print(math.sqrt(16))
print(math.gcd(8, 12))
print(math.pi)

from math import lcm
print(lcm(8, 12))
```

Output

```
4.0
4
3.141592653589793
24
```

In-class exercises: write functions

Each task is a function that RETURNS its answer. Test it with the examples.

1 evens(n)

Return a list of even numbers from 1 to n. `evens(10)` gives [2, 4, 6, 8, 10].

2 gcd(a, b)

Greatest common divisor, with a loop. `gcd(8, 12)` gives 4.

3 lcm(a, b)

Use your gcd: $a * b // \text{gcd}(a, b)$. `lcm(8, 12)` gives 24.

4 mex(lst)

Smallest non-negative integer not in the list. `mex([2, 1, 4, 0, 5])` gives 3.

5 find_operator(number)

Return the operator name, or "INV" if not 11 characters or unknown prefix.

6 second_largest(lst)

Largest value smaller than the maximum. `second_largest([4, 9, 2, 9, 7])` gives 7.

Solutions

Show these after students have tried the exercises.

Solution: evens

Build the answer in a list, return it at the end. The caller decides whether to print it.

Code

```
def evens(n):  
    result = []  
    for i in range(1, n + 1):  
        if i % 2 == 0:  
            result.append(i)  
    return result  
  
print(evens(10))
```

Output

```
[2, 4, 6, 8, 10]
```

Solutions: gcd and lcm

lcm reuses gcd: functions calling functions is how bigger programs are built.

Code

```
def gcd(a, b):  
    result = 1  
    for i in range(1, min(a, b) + 1):  
        if a % i == 0 and b % i == 0:  
            result = i  
    return result  
  
def lcm(a, b):  
    return a * b // gcd(a, b)  
  
print(gcd(8, 12), gcd(86, 44), lcm(8, 12))
```

Output

4 2 24

Watch it run: a function that calls a function

WATCH FOR

- Frames stack up: main, sum_of_squares, square
- Each caller waits until its call returns
- square runs twice, each time with a fresh frame
- Like lcm calling gcd in the solution

WATCH THE CODE RUN, STEP BY STEP

A function that calls a function

Frames stack up: main waits for sum_of_squares, which waits for square.
Each call gets its own fresh frame.

Click the video to play, click again to pause. Silent: talk the class through it.

Solutions: mex and second_largest

`while m in lst` keeps counting up until it finds a number that is missing.

Code

```
def mex(lst):
    m = 0
    while m in lst:
        m = m + 1
    return m

def second_largest(lst):
    largest = max(lst)
    others = []
    for x in lst:
        if x != largest:
            others.append(x)
    return max(others)

print(mex([2, 1, 4, 0, 5]), mex([2, 1, 4, 0, 5, 3]))
print(second_largest([4, 9, 2, 9, 7]))
```

Output

```
3 6
7
```

Solution: find_operator

return ends the function at once, so an invalid number can never also print an operator name.

Code

```
def find_operator(number):  
    if len(number) != 11:  
        return "INV"  
    match number[:3]:  
        case "017" | "013": return "Grameenphone"  
        case "014" | "019": return "Banglalink"  
        case "015": return "Teletalk"  
        case "016": return "Airtel"  
        case "018": return "Robi"  
        case _: return "INV"  
  
print(find_operator("01411111111"))  
print(find_operator("0131234567890"))
```

Output

```
Banglalink  
INV
```

Interactive lab: Functions Lab

In the lab you can

1 Call visualiser

Step through calls: frames open, values go in, return comes back.

2 Argument matcher

See which argument fills which parameter, or why it fails.

3 Change or move?

Predict whether a function changes the caller's list.

4 Write and test

9 exercises with automatic tests checked against Python.

LECTURE 4.1 · INTERACTIVE LAB

Functions Lab

Watch call frames open and close, match arguments to parameters, and see what a function can change

Introduction to Data Science and Machine Learning
Md Mahbub E Noor · University of Barishal
[All Labs](#)

1 Call visualiser 2 Argument matcher 3 Change or move? 4 Write and test

Call visualiser: frames appear, values go in, values come back

Every example from the lecture. Watch Python skip the function body at def, open a new frame at each call, and remove it at return.

Example: Changing a list inside a function [Edit code](#)

```
1 def add_item(lst):  
2     lst.append(100)  
3  
4 nums = [10, 20, 30]  
5 add_item(nums)  
6 print(nums)
```

◀ Back ▶ Next step ▶ Play ▶ Run to end

↺ Start again

Play speed Step 5 of 8 **inside add_item(lst)**

Line 2: lst is now [10, 20, 30, 100].

FRAMES AND OBJECTS

☐ numbers and text inside the frame

Global frame

- add_item
- nums

function

- function add_item(lst)

list (changed)

- 10 20 30 100
- 0 1 2 3

add_item(lst)

- lst

Orange frame = the function running now. Orange arrow = a name that just got a new value.
Orange object = just created or changed.

OUTPUT

Open the lab ▶

Works offline in any browser
Labs_open_in_browser/
L4.1_Functions_Lab.html

Today's toolkit

<code>def name(p1, p2):</code>	Define a function
<code>return value</code>	Send a result back
<code>name(a1, a2)</code>	Call it with arguments
<code>name(p2=x, p1=y)</code>	Keyword arguments

<code>def f(p, q=35):</code>	Default value for q
<code>a, b = f()</code>	Unpack several return values
<code>import math</code>	Load a module
<code>math.sqrt(x)</code>	Use a module's function

Homework 4.1

Due before Lecture 4.2 • [date]

1 Predict, then run

Predict first: `def f(x): x.append(1);` then `a = [0]; f(a); print(a)`. And `def g(x): x = x + [1];` with the same test.

2 Temperature

Write `c_to_f(c)` that returns $c * 9 / 5 + 32$.
Print a table for 0, 25, 37, 100 using a loop.

3 Grade function

Turn the Lecture 3.1 grade code into `grade(score)` that returns A, B, C or F. Test all four.

4 Spam check

Write `is_spam(message, words)` that returns True or False. Default words: `["free", "win", "prize"]`.

5 Explain

In 3 sentences: why does `f` change `a` but `g` does not? Use the words `object` and `name`.

Submit on [LMS / Google Classroom]: your notebook file `HW4_1_<your ID>.ipynb`, or a Colab link set to “Anyone with the link can view”.

Recap: Lecture 4.1 in 90 seconds

RECAP

Lecture 4.1 in 90 seconds

A short recap of today's lecture. Pause any time.

Click to play or pause. Watch it again at home to revise.