

CSCI 145 · Introduction to Computing · Fall 2026

Variables & Functions

Lecture 2 — comparisons, logic, and writing our own functions

Outline

- More expressions and variables

- Why functions?

- Calling and writing functions

Warm-up: what's in memory?



```
y = 3  
x = 7  
y = 3 + 2 + 1
```

What does memory look like when this code runs?

y: 3 6

x: 7

Warm-up: tracing x

```
x = 5
print(x)      # a)  5
y = 10
x = x + y
print(x)      # b)  15
x + 3
print(x)      # c)  15
```

x:

5 15

y:

10

The bare expression x + 3 computes a value but doesn't store it anywhere.

What is the value of `x` after this code runs?

```
x = 5  
y = x  
y = 50
```

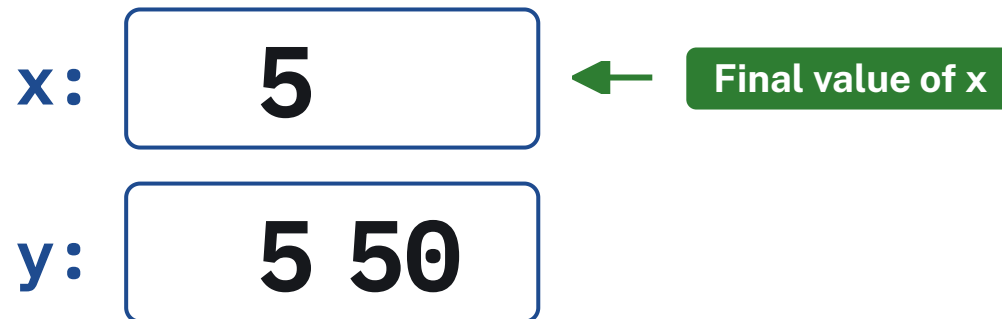
Go to **slido.com** and enter the code: `3674841`

Or [join directly here](#)



What is the value of **x** after this code runs?

```
x = 5  
y = x  
y = 50
```



What is the value of **b** after this code runs?

```
a = 10  
b = a  
a = a - 3  
b = b + a
```

Go to **slido.com** and enter the code: 3674841

Or [join directly here](#)



What is the value of **b** after this code runs?

```
a = 10  
b = a  
a = a - 3  
b = b + a
```

a: 10 7

b: 10 17 ← Final value of b

Part 1 of 3

More expressions & variables

Naming variables

Rules

Only letters, numbers, and underscores
Cannot start with a number
Cannot be a reserved Python keyword

Guidelines

Names should reflect their purpose
Start with a lowercase letter
Separate multi-word names with an underscore, e.g. `elapsed_time`

Good ✓

student_count
average_grade
is_valid
total_price

Poor

x
data
var
1

Illegal

2nd_list
break
my_list!
my-list

See [PEP 8](https://peps.python.org/pep-0008/#naming-conventions) for more: peps.python.org/pep-0008/#naming-conventions

Comparison operators

==

equal

!=

not equal

>

greater than

<

less than

>=

greater than or equal to

<=

less than or equal to

Comparisons take two values and produce a `bool` — `True` or `False`.

Try them out

```
>>> 5 > 8  
False
```

```
>>> 8 <= 8  
True
```

```
>>> 3 == "3"  
False
```

```
>>> "orange" < "Yellow"  
False
```

`3 == "3"` is False — an int and a str are never equal. Strings compare letter by letter using their character codes, and capital letters sort before lowercase ones, so `"Yellow"` comes before `"orange"`.

One thing to watch for

```
x = 5      # assigns 5 to x  
x == 5     # asks: does x equal 5?
```

`=` assigns a value. `==` compares two values. Both are valid Python that do very different things — mixing them up is a common bug, especially once we start writing conditionals.

Logical operators

and

true only if both sides are true

or

true if at least one side is true

not

flips True to False, or False to True

Logical operators combine or invert Booleans — the same way arithmetic operators combine numbers.

A	B	A and B	A or B
True	True	True	True
True	False	False	True
False	True	False	True
False	False	False	False

A	not A
True	False
False	True

Try them out

```
>>> True and False  
False
```

```
>>> True or False  
True
```

```
>>> not True  
False
```

```
>>> not False  
True
```

Putting it together

```
day = 14  
day > 10 and day < 20
```

`True` — day is between 10 and 20. Comparisons combine with `and` / `or` / `not` just like arithmetic values combine with `+` and `*`.

Order of operations, extended

()

grouping

**

exponent

-

negation (not subtraction)

*, /, %, //

multiplication, division, modulo, floor division

+, -

addition, subtraction

<, >, <=, >=, ==, !=

comparisons

not

logical negation

and

logical and

or

logical or

=

assignment, evaluated last

Activity

1. If you haven't already done so, choose a place on your computer (say `Documents`) and create a folder `cs145`, with subfolders `class`, `labs`, `homework`.
2. Download [day02.zip](#).
3. Drag `day02.zip` to your `class` folder. Extract it there: **Mac** — double-click it. **Windows** — right-click it and choose "Extract All...". Once you see the `day02` folder, delete `day02.zip`.
4. Launch Thonny. From the File menu, open `cs145/class/day02/day02_expressions.py`.
5. Read the code, predict the printed output, then uncomment the print statements to check your answers.

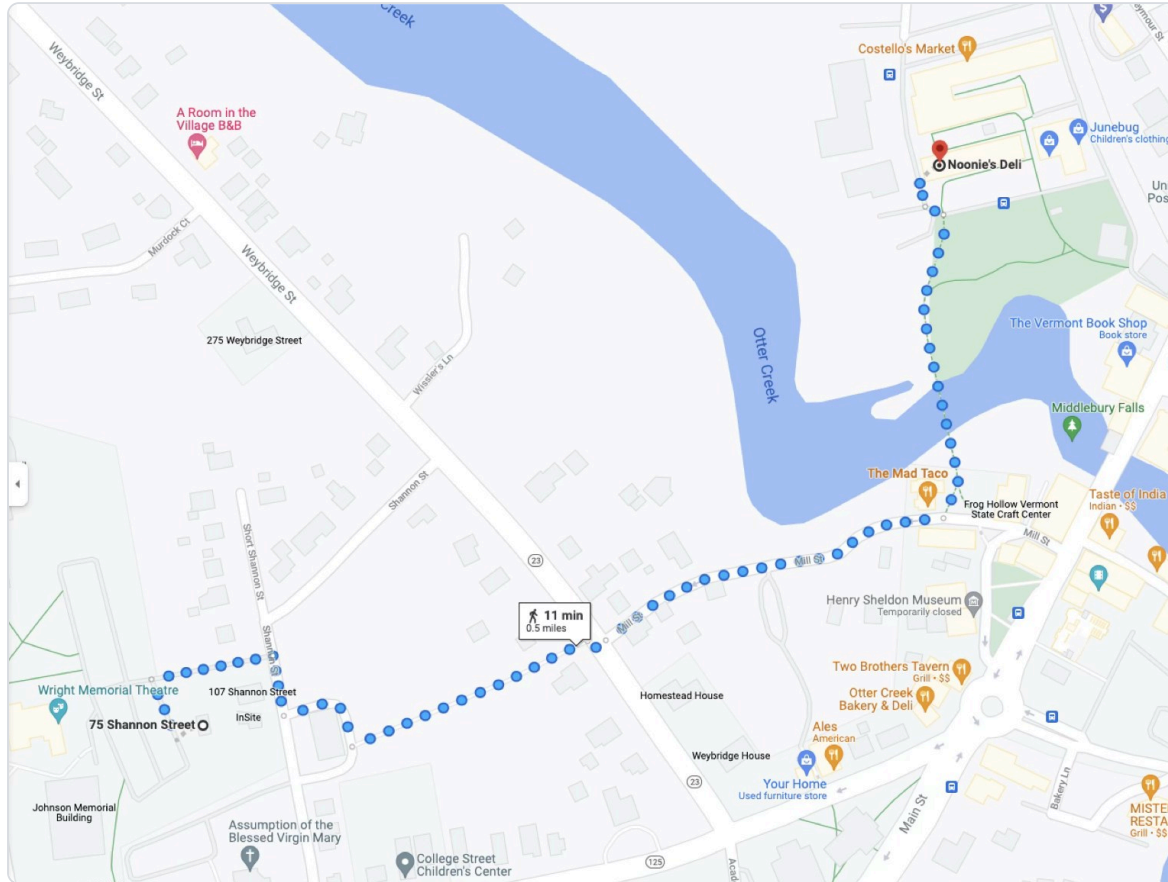
Part 2 of 3

Why functions?

What is a function?

Last time, we used `type()`, `int()`, `float()`, and `str()` without ever naming what they were. They're **functions** — a named sequence of instructions that performs a task or computation.

Abstraction: hiding the details



Every step, spelled out

- Attend 145 class
- **Go outside**
- **Head northwest on Shannon St**
- **Turn right, walk two blocks...**
- **Turn left, go inside, order lunch**
- **Turn right, walk back...**
- Visit office hours

Same plan, abstracted

- Attend 145 class
- **Eat lunch at Noonie's Deli**
- Visit office hours

The details aren't gone — we just don't need them right now. A function lets us hide a process behind a name.

Why abstraction helps

- makes a process clearer and easier to understand
- removes dependency on the details, so they can change without breaking the plan
- makes the process reusable

Functions we've already used

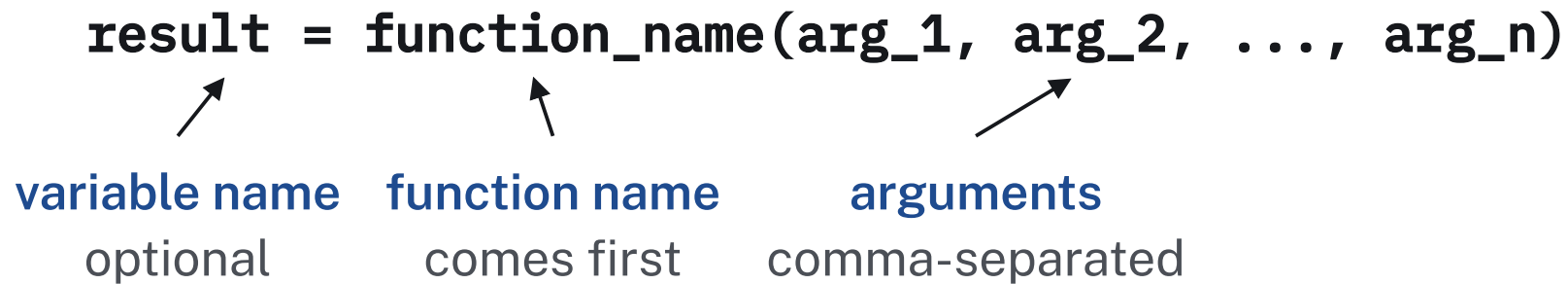
You've already been calling functions: `print()`, `str()`, `int()`, `float()`, `bool()`, `type()`, and `len()` are all **built-in functions** — functions Python provides for you.

Part 3 of 3

Calling & writing functions

Calling a function

`result = function_name(arg_1, arg_2, ..., arg_n)`

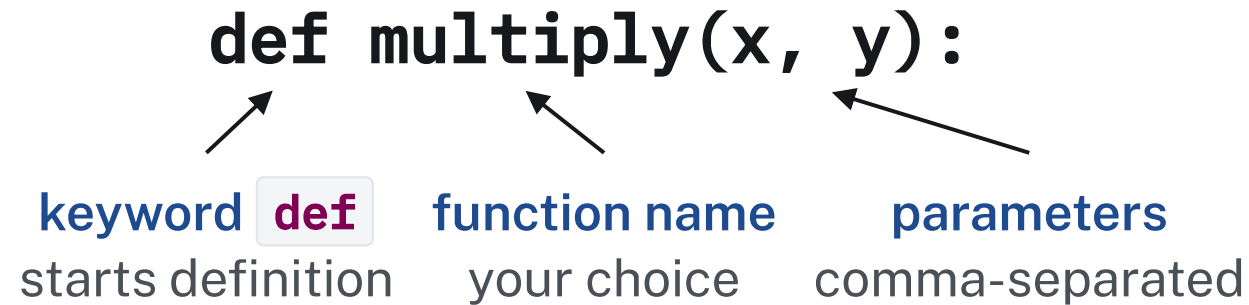


variable name **function name** **arguments**
optional comes first comma-separated

- the function's `name` comes first on the right-hand side
- arguments go in parentheses, separated by commas
- store the result in a variable if you want to use it later
- no arguments? still use the parentheses — `function_name()`

Writing a function

`def multiply(x, y):`



keyword `def` function name parameters
starts definition your choice comma-separated

```
def multiply(x, y):  
    """Multiply two numbers."""  
    return x * y
```

- a **docstring** (triple-quoted) describes what the function does
- the **body** is indented — it runs each time the function is called
- `return` sends a value back to the caller

A function with no parameters

```
def greet():  
    """Greet someone."""  
    print("Hello")  
    print("How are you?")
```

- a **docstring** (triple-quoted) describes what the function does
- the **body** is indented — it runs each time the function is called
- parameters and a **return** are both optional

Writing a good docstring

In this class, every function's docstring should include:

- 1–2 sentences describing what the function does
- a description of its parameters
- a description of what it returns, if anything

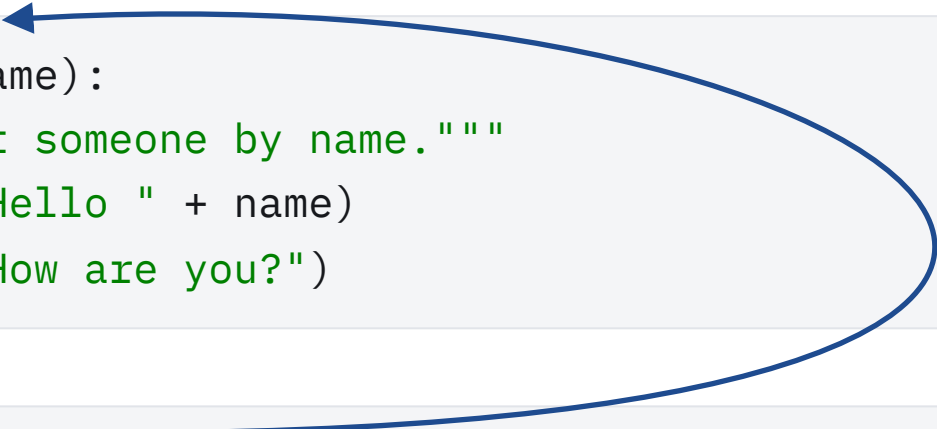
```
def function_name(param1, param2):  
    """  
    One or two sentences describing what the function does.  
  
    Params:  description and types of parameters  
    Returns: description and type of the returned value  
    """
```

Calling `greet()`

```
>>> greet()  
Hello  
How are you?
```

Every time we call `greet()`, Python runs the body from the top.

Functions can take parameters



```
def greet(name):  
    """Greet someone by name."""  
    print("Hello " + name)  
    print("How are you?")
```

```
>>> greet("Vicki")  
Hello Vicki  
How are you?
```

The **arguments** in the function call are evaluated, and those values are used to initialize the **parameters** in the function.

Here, the parameter `name` gets its value when the function is called with argument `"Vicki"`.

Functions can return a value

```
def double(x):  
    """Double the input."""  
    return x * 2
```

```
>>> double(5) + 1  
11
```

return sends a value back to wherever the function was called — the call itself becomes that value, so it can be used inside a larger expression.

Activity

1. From the Thonny File menu, open `cs145/class/day02/day02_functions.py`.
2. Click the green arrow to run the file. Nothing will print yet — this just loads the functions into Thonny. Then call them yourself in the shell at the bottom, e.g. type `greet1()` and press Enter.

To do

- Review this slide deck and these corresponding notes: [Notes 3](#), [Notes 4](#)
- Prepare for [Lab 1](#) this Friday
- See the [Practice materials](#) page to help you prepare for Quiz 1 next week

Next time

Lecture 3 — scope and the call stack