CSSE1001 — Week 3 Notes

Table of contents

Functions

See csse1001[1] for course logistics — this note covers Lecture 3A's technical content.

[1] courses/csse1001/csse1001.pdf

Today's outline

- What a function is and how to define one
- The `return` statement, and how it differs from `print`
- Improving functions with type hints and docstrings

Learning objectives

1. User-defined functions bundle code together and are the building blocks of more sophisticated programs.
2. `return` hands back a value and exits a function; it is not the same as `print`.
3. Functions can be documented with type hints and docstrings.

What is a function?

We have already used functions — `*` and `max`, for example — each of which takes input and returns output. User-defined functions let us name and reuse our own bundles of code, just like a mathematical function such as $f(x) = x^2 + x + 1$:

```
>>> def f(x):
...     return x**2 + x + 1
>>> f(3)
13
```

See python-functions[2] for the full syntax rules (indentation, multiple parameters, the `return` statement, `return` vs. `print`, type hints, and docstrings) — used throughout the rest of this note.

Building up Heron's formula

Exercise. The area of a triangle with side lengths a, b, c is $\sqrt{s(s-a)(s-b)(s-c)}$ where $s = \frac{1}{2}(a+b+c)$ (Heron's formula). What is the area of the triangle with sides 3, 4, and 5?

As a calculator, we'd have to type this out in full, and it would be tedious to repeat for other side lengths:

[2] courses/csse1001/python-functions.pdf

```
>>> (0.5*(3+4+5)
...  *(0.5*(3+4+5)-3)
...  *(0.5*(3+4+5)-4)
...  *(0.5*(3+4+5)-5))**0.5
6.0
```

Using names and sequencing (see 2025-08-04-python-memory-model[3]) avoids repeating the same sub-expression:

```
>>> a, b, c = 3, 4, 5
>>> s = (a + b + c)/2
>>> (s*(s-a)*(s-b)*(s-c))**0.5
6.0
```

Wrapping it in a function makes it reusable for *any* triangle:

```
>>> def heron(a, b, c):
...     s = (a + b + c)/2
...     return (s*(s-a)*(s-b)*(s-c))**0.5
>>> heron(3, 4, 5)
6.0
```

Improving the function

Adding type hints and a docstring (see python-functions[4]):

```
>>> def triangle_area(a: float, b: float, c: float) -> float:
...     """
...     Return the area of the triangle with sides length <a>,
...     <b>, and <c>.
...     Preconditions: <a>, <b>, <c> are all nonzero positive.
...     >>> triangle_area(3, 4, 5)
...     6.0
...     """
...     s = (a+b+c)/2
...     return (s*(s-a)*(s-b)*(s-c))**0.5
>>> triangle_area(2.2, 3.3, 4.4)
3.5147323866832307
```

[3] courses/csse1001/2025-08-04-python-memory-model.pdf
[4] courses/csse1001/python-functions.pdf

See python-pep8-style-guide[5] for the naming, spacing, and line-length conventions expected of functions like this.

`return` vs. `print`: quick check

```
>>> def example(x):
...     print(1*x)
...     return 3*x
...     print(2*x)   # never runs -- return already exited the function
>>> a = example(1)
1
>>> a
3
```

```
>>> def foo(x):
...     if x > 0:
...         print("Positive")
...     if x > 10**5:
...         print("Large positive")
>>> ans = foo(10**6)
Positive
Large positive
>>> type(ans)
<class 'NoneType'>
```

`foo` above never hits a `return`, so it defaults to returning `None` even though it printed something. Contrast with a version that returns instead:

```
>>> def bar(x):
...     if x > 0:
...         return "Positive"
...     if x > 10**5:
...         return "Large positive"
>>> ans = bar(10**6)
>>> ans
'Positive'
```

`bar` exits at the very first `return` it reaches, so `"Large positive"` is never returned even though `x > 10**5` is also true.

[5] courses/csse1001/python-pep8-style-guide.pdf

Exercise: the middle number

Write a function that takes three integers and returns the number that is not the largest or the smallest:

```
def middle_number(x: int, y: int, z: int) -> int:
    """
    Return the number that is not the largest or smallest
    among the three inputs.
    Precondition: the numbers are distinct.
    >>> middle_number(3, 1, 2)
    2
    >>> middle_number(2, 3, 1)
    2
    """
    return (x + y + z) - min(x, y, z) - max(x, y, z)
```

Summary

Blocks of code can be grouped into functions. Functions take zero or more inputs and hand back a single value designated by `return`.

Next lecture

2025-08-11-sequence-selection-and-iteration[6] — selection.

Sequence, Selection, and Iteration

See csse1001[7] for course logistics — this note covers Lecture 3B's technical content.

Today's outline

Despite the title, this lecture is entirely about **selection** (controlling program flow) — sequencing was already covered in 2025-08-04-python-memory-model[8], and iteration is the topic of the next lecture.

[6] courses/csse1001/2025-08-11-sequence-selection-and-iteration.pdf
[7] courses/csse1001/csse1001.pdf
[8] courses/csse1001/2025-08-04-python-memory-model.pdf

Learning objectives

1. Predicates and the logical operators `and`, `or`, `not` let us build up complex conditions.
2. The `if` statement (and its `elif`/`else` variants) lets us skip or select blocks of code based on a condition.
3. If-statements can often be refactored into simpler, equivalent forms.

Selection

See python-boolean-logic[9] for the full reference on booleans, predicates, `and`/`or`/`not`, precedence, short-circuiting, and truthiness, and python-if-statements[10] for the full reference on `if`/`elif`/`else`, the common elif-ordering bug, and simplifying if-statements.

Exercises

Bracket for `False`. Bracket the expression below so that it evaluates to `False`. How many different bracketings can you find?

```
False and False or True and False or False or True
```

Add a single `not`. Add a single `not` to the same expression so that it evaluates to `False`.

> Both exercises are posed but left unsolved in the source material — flagged here rather than guessed at.

Refactoring exercise. Refactor the following code:

```python
def foo(x, y):
    if x > 100 and y > 0:
        if y > 100 and x > 0:
            return "A"
        elif y > 100 or x > 0:
            return "A"
        else:
            return "B"
    elif y <= 0 or x <= 0:
        if x == y:
            return "A"
        if x <= y and x >= y:
            return "B"
```

[9] courses/csse1001/python-boolean-logic.pdf
[10] courses/csse1001/python-if-statements.pdf

```
        if y < x and x < y:
            return "C"
        else:
            return "A"
    else:
        if x <= 100 and y > 0:
            return "A"
        if x > 100 or y <= 0:
            return "B"
        else:
            return "D"
```

No worked solution is given in the source for this exercise either — left as an exercise rather than invented here.

Summary

Blocks of code can be skipped using `if` statements. This control flow depends on the evaluation of predicate (boolean-valued) statements.

Next lecture

2025-08-18-while-loops[11] — iteration.

Reference material

Python Boolean Logic

The basic operands of logic are `True`, `False`, `or`, `and`, and `not`.

Boolean domain and predicates

Let $\mathbb{B}$ denote the **boolean domain**, $\mathbb{B} = \{\texttt{True}, \texttt{False}\}$. Any function that maps into $\mathbb{B}$ (i.e. one that evaluates to `True` or `False`) is called a **predicate** — e.g. $>\colon \mathbb{Z} \times \mathbb{Z} \to \mathbb{B}$ ("greater than").

[11] courses/csse1001/2025-08-18-while-loops.pdf

```
>>> type(True), type(False)
(<class 'bool'>, <class 'bool'>)
>>> 7 > 3
True
>>> 7 >= 7 + 1        # addition happens first
False
>>> (7 >= 7) + 1
2
>>> int(True), int(False)
(1, 0)
```

and, or, not

and ($\mathbb{B} \times \mathbb{B} \to \mathbb{B}$) is true only when *both* inputs are true:

and	True	False
True	True	False
False	False	False

or ($\mathbb{B} \times \mathbb{B} \to \mathbb{B}$) is true when *at least one* input is true (false only when both are false):

or	True	False
True	True	True
False	True	False

```
>>> 3 > 7 or 7 > 3
True
>>> 3 > 7 and 7 > 3
False
>>> 0 < 3 and 3 < 8
True
>>> 0 < 3 < 8          # shorthand for the above
True
```

`True or False and False` is ambiguous without a precedence rule, since (`True or False`) `and False` = `False` but `True or (False and False)` = `True`. Python resolves this with `and` having **higher precedence than or** (so the expression above evaluates to `True`, as if bracketed `True or (False and False)`).

`not` is the negation of a logical statement (`not True == False`, `not False == True`) and is evaluated *first*, before `and`/`or`:

```
>>> a, b = 6, 7
>>> not (a == 6 and b != 5)
False
>>> not a == 6 and b != 5   # not happens before and
False
```

Avoid double negations (e.g. `not (a != 6 or not b != 5)`) — they're valid but hard to read.

Short circuits (lazy evaluation)

`or` stops as soon as it finds a truthy value; `and` stops as soon as it finds a falsy value — the remaining operand is never evaluated:

```
>>> True or 1/0
True
>>> False or 1/0
ZeroDivisionError: division by zero
>>> True or non_existent_variable
True   # Python never bothers looking up non_existent_variable
>>> True and 1/0
ZeroDivisionError: division by zero
>>> False and 1/0
False
```

Truthiness

Any object can be converted to a boolean with `bool`. **Truthy** values evaluate to `True` (non-zero numbers, non-empty strings/tuples/lists); **falsy** values evaluate to `False` (zero, the empty string, the empty tuple/list):

```
>>> bool(1), bool(-10), bool("Hello"), bool([1, 2, 3])
(True, True, True, True)
>>> bool(0), bool(""), bool([])
(False, False, False)
```

What `and`/`or` actually return

`and` and `or` don't always return `True`/`False` — `and` returns its first falsy input (or its last input, if none are falsy), and `or` returns its first truthy input (or its last input, if none are truthy):

```
>>> 0 or 2
2
>>> 0 and 2
0
>>> () or (1,) or (1, 2)
(1,)
>>> (1, 2) or (1,) or ()
(1, 2)
>>> () and (1,) and (1, 2)
()
>>> (1, 2) and (1,) and ()
()
```

Python Functions

User-defined functions bundle lines of code together so they can be reused, abstracting away complexity. Like mathematical functions, they take input and return output (we've already used built-in functions this way, e.g. `*` and `max`).

Defining a function

A mathematical function such as $f(x) = x^2 + x + 1$ is written in Python as:

```
>>> def f(x):
...     return x**2 + x + 1
>>> f(3)
13
```

Functions can take multiple parameters, and calls can be nested inside other expressions:

```
>>> def f(x, y):
...     return x*y
>>> f(-f(5, 2) + 12, f(2, 3))
12
```

Indentation

Four spaces of indentation are significant in Python — they associate a line of code with the control structure above it (here, the function body with its `def`). Inconsistent indentation raises `IndentationError: unexpected indent`.

The `return` statement

`return` is a reserved word (not a function) that hands a value back to the caller and immediately exits the function — any code after the first `return` a call actually reaches (including further `print`s or `return`s) never runs.

If a function has no `return` statement, Python presumes a `return None` as its last line.

`return` versus `print`

`print` displays something as a side effect but *returns* `None`. This looks similar to `return` but behaves very differently once the result is used in further computation:

```
>>> def f(x, y):
...     print(x+y)
>>> a = f(2, 3)
5
>>> b = f(3, 4)
7
>>> a + b
TypeError: unsupported operand type(s) for +: 'NoneType' and 'NoneType'
```

```
>>> def f(x, y):
...     return x + y
>>> a = f(2, 3)
>>> b = f(3, 4)
>>> a + b
12
```

`return` is a reserved word, not a function — we write `return x + y`, not `return(x + y)` (the latter happens to still work, since the brackets are just a redundant grouping, but it's misleading).

Type hints

Type hints annotate the expected type of each parameter and the return value, e.g. mapping to triangle_area : $\mathbb{R} \times \mathbb{R} \times \mathbb{R} \to \mathbb{R}$:

```
>>> def triangle_area(a: float, b: float, c: float) -> float:
...     s = (a+b+c)/2
...     return (s*(s-a)*(s-b)*(s-c))**0.5
```

Type hints are **not enforced** — they exist purely as documentation to make code more readable, and Python will not stop you calling a function with the "wrong" types.

Docstrings

A **docstring** (`"""..."""` immediately under the `def` line) documents what a function does and its preconditions, and gives example calls (written like REPL input/output) that double as tests:

```
>>> def triangle_area(a: float, b: float, c: float) -> float:
...     """
...     Return the area of the triangle with sides length <a>,
...     <b>, and <c>.
...     Preconditions: <a>, <b>, <c> are all nonzero positive.
...     >>> triangle_area(3, 4, 5)
...     6.0
...     """
...     s = (a+b+c)/2
...     return (s*(s-a)*(s-b)*(s-c))**0.5
```

General template

```
def function_name(arg0: type, arg1: type, ...) -> type:
    """
    Short description of the function for documentation.
    Preconditions (if any).
    >>> function_name(x, y, ...)
    expected output
    """
    ...
    function body
```

```
    ...
    return
```

See python-pep8-style-guide[12] for the naming, spacing, and line-length conventions used when writing functions like this.

Python If Statements

Given a condition C (a predicate — see python-boolean-logic[13]), an **if-statement** is a control structure that executes a block of code when C is `True` and skips it otherwise.

If-then

```
if <cond>:
    <code executed when cond == True>
```

Only the indented code runs when the condition holds; execution otherwise skips straight past it. Because of truthiness[14], the condition doesn't need to literally be `True`/`False`:

```
>>> x = 0
>>> if x:
...     x = x + 1
>>> x
0
>>> x = 1
>>> if x:
...     x = x + 1
>>> x
2
```

Warning. A variable only assigned inside an if-block does not exist if the condition was false:

```
>>> if False:
...     ans = 0
>>> ans
NameError: name 'ans' is not defined
```

[12] courses/csse1001/python-pep8-style-guide.pdf
[13] courses/csse1001/python-boolean-logic.pdf
[14] courses/csse1001/python-boolean-logic.pdf

If-then-else

```
if <cond>:
    <code>
else:
    <code>
```

`if-else` picks between exactly one of two instruction sets — unlike two separate `if`s, the condition is only checked once and the two branches can never both run.

Elif chains

```
if <cond0>:
    <code>
elif <cond1>:
    <code>
    ...
elif <condN>:
    <code>
```

Each `elif` condition is only checked if every condition above it was `False` — so later branches can safely assume the earlier conditions failed.

Common bug. Because later `elif`s implicitly assume the earlier ones were false, checking overlapping ranges in the wrong order silently picks the wrong branch:

```
>>> age = 60
>>> if age >= 18:
...     beverage = "cheap beer"
... elif age >= 30:
...     beverage = "standard beer"
... elif age >= 50:
...     beverage = "expensive beer"
>>> beverage
'cheap beer'   # not what was intended!
```

Two ways to fix it — bound each range explicitly:

```
>>> if 18 <= age < 30:
...     beverage = "cheap beer"
... elif 30 <= age < 50:
...     beverage = "standard beer"
... elif 50 <= age:
...     beverage = "expensive beer"
```

or check from highest to lowest, relying on the guarantee each `elif` gives about ranges already ruled out:

```
>>> if age >= 50:
...     beverage = "expensive beer"
... elif age >= 30:      # guaranteed age < 50
...     beverage = "standard beer"
... elif age >= 18:      # guaranteed age < 50 and age < 30
...     beverage = "cheap beer"
```

If-elif-else

```
if <cond0>:
    <code>
elif <cond1>:
    <code>
    ...
else:
    <code>
```

`else` is a catch-all — it runs whenever none of the preceding `if`/`elif` conditions were `True` (avoiding a `NameError` from a variable never getting assigned).

Factoring and refactoring

Factoring means breaking a complex problem into parts that are easier to conceive, understand, program, and maintain. **Refactoring** is the process of restructuring existing code — changing the factoring — without changing its behaviour.

Simplifying if-statements

Nested `if`s can often collapse into a single `and`ed condition:

```
if x > 1:                          if x > 1 and y > 2 and z > 3:
    if y > 2:             ==>          print("hello")
        if z > 3:
            print("hello")
```

An if/else that only assigns/returns `True`/`False` can be replaced by the condition itself:

```
def foo(x):                        def foo(x):
    if x > 0:             ==>          return x > 0
        return True
    else:
        return False

if x > 0:                          y = x > 0
    y = True          ==>
else:
    y = False
```

Comparing directly to `True`/`False` is redundant:

```
if x > y == True:    ==>    if x > y:
if x > y == False:   ==>    if not x > y:
```

Equivalence of `elif` vs. nested `if`. An `elif` condition can safely assume the earlier condition was false, so `elif x <= 0 and x % 2 == 0` is equivalent to just `elif x % 2 == 0` (given a preceding `if x > 0`). This is *not* true for a second, independent `if` — `if x <= 0 and x % 2 == 0` is *not* equivalent to a bare `if x % 2 == 0` placed after `if x > 0`, since the second `if` doesn't know the first one already ran (e.g. with `x = 2`, the `elif` form prints only `A`, but the two-independent-`if`s form prints both `A` and `B`).

Common errors

`or`/`and` do not distribute over a list of bare values — `x == 1 or 2 or 3` always evaluates truthy (`2` and `3` are truthy on their own, regardless of `x`). The comparison needs to be repeated instead: `x == 1 or x == 2 or x == 3` (later refactored to `x in [1, 2, 3]`).

`if x: pass else: <code>` is just a roundabout way of writing `if not x: <code>`.

Python PEP8 Style Guide

A **PEP** (Python Enhancement Proposal) is a design document describing conventions for how to style code. This course follows Google's PEP, a variant of **PEP8**.

Variable and function names

Names must start with a letter (not a digit) and can otherwise only contain letters, digits, and underscores (_). Names should be lowercase, with words separated by underscores for readability:

Yes	No
`descriptive_variable_name`	`DescriptiveVariableName`

Spacing

Put single spaces around binary operators; don't pad the inside of brackets:

Yes	No
`i = i + 1`	`i=i+1`
`hypot2 = x*x + y*y`	`hypot2 = x * x + y * y`
`c = (a+b) * (a-b)`	`c = (a + b) * (a - b)`

Breaking long lines

All lines should be strictly less than 80 characters wide. Wrap a long expression in a redundant enclosing bracket so it can be split across multiple lines (otherwise pressing enter would evaluate the expression early):

```
>>> income = (gross_wages
...           + taxable_interest
...           + (dividends - qualified_dividends)
...           - ira_deduction
...           - student_loan_interest)
```

Comments

Anything following a `#` is ignored by Python. Comments (and docstrings, see python-functions[15]) should explain *why* something not obvious was done, not simply restate what the code already shows:

```
x = x + 1   # Not helpful: Increment x
x = x + 1   # Helpful: Compensate for border
```

[15] courses/csse1001/python-functions.pdf