

ALGORITHMS

An algorithm is an **unambiguous** sequence of steps that, if followed exactly, solves the problem under consideration:

- has a **finite** number of steps
- each step is **well-defined**
- eventually **halts**
- solves a **general class** of problems, not just one instance

Pseudocode mixes natural language with familiar control structures (if/else, while) without committing to a real language's syntax. A **bug** is when code doesn't do what you intended; a **logical error** is a bug where the code runs but computes the wrong thing.

COMPILED VS. INTERPRETED

Compiled languages (C, C++, Java): source code is translated by a **compiler** into object code once, then run.

Interpreted languages (Python, JavaScript): an **interpreter** reads and runs source code directly, line by line — no separate compile step.

TYPES WE'VE SEEN SO FAR

- int** a whole number, positive or negative — e.g. -4, 0, 3
- float** a number with a decimal point — e.g. 3.14, 10.0
- bool** a Boolean value — True or False
- str** a sequence of characters (text) — e.g. 'hi', "hi"

`type(v)` tells you a value's type. Pick the type that matches what the value *means*: a count → int, a yes/no fact → bool, text → str, a measurement that can have a fraction → float. Convert with `int()`, `float()`, `str()`: `int(2.6)` is 2, `str(2)` is '2', `int("2")` is 2.

OPERATORS AND THEIR PRECEDENCE

Highest to lowest — Python evaluates higher rows first.

<code>()</code>	grouping
<code>**</code>	exponent
<code>-x</code>	negation (not subtraction)
<code>* / % //</code>	multiply, divide, modulo, floor-divide
<code>+ -</code>	addition, subtraction
<code>< > <= >= == !=</code>	comparisons
<code>not</code>	logical negation
<code>and</code>	logical and
<code>or</code>	logical or
<code>=</code>	assignment — evaluated last

NAMING VARIABLES

Rules: only letters, numbers, underscores; cannot start with a number; cannot be a reserved keyword (like `class`, `if`, `return`).

Guidelines: reflect the value's purpose, start lowercase, separate words with underscores (`elapsed_time`).

Good: `student_count`, `is_valid` Poor: `x`, `data`, `l` Illegal: `2nd_list`, `my-list`, `class`

COMPARISON & LOGICAL OPERATORS

`==` `!=` `>` `<` `>=` `<=` compare two values and produce a bool. An `int` and a `str` are **never** equal (`3 == "3"` is False). Strings compare letter-by-letter by character code — capital letters sort before lowercase ("`Yellow`" `<` "`orange`" is True).

Careful: `=` assigns a value; `==` asks if two values are equal. Mixing them up is a common bug.

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

`not True` → False `not False` → True

CODE EXECUTION & MEMORY MODEL

Python runs each line top to bottom. Assigning one variable to another copies its *current value* — it does not link the two names together.

```
x = 5
y = x
y = 50      # x is still 5; y is now 50
```

WRITING FUNCTIONS

```
def double(x):
    """
    Double the input.
    Params: x (int/float/str) -- value to double
    Returns: doubled value (int/float/str)
    """
    return x * 2

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

Every docstring in this course should state: (1) what the function does, (2) its parameters, (3) what it returns. The returned value becomes the value of the function call — a function with no return gives back `None`.

ABSTRACTION

Hiding the details behind a name (like a function) makes a process clearer, lets details change without breaking the plan, and makes the process reusable.

CSCI 145 · Exam 1 Reference Sheet (Lectures 1–4) — page 2 of 2

LOCAL VS. GLOBAL SCOPE

- The **scope** of a variable is the region of code where it can be accessed.
- Functions create **local scope** — variables created inside cannot be accessed from outside it.
- Variables created outside any function are in **global scope**, accessible anywhere.
- A parameter is local to its function, just like a variable assigned inside it.

TRACING THE CALL STACK

Every function call gets its own **frame**. A call inside another call pushes a new frame on top; when it returns, that frame pops and the value flows back to the line that called it. Calls can nest more than one level deep:

```
def f(x):
    return x + g(x)
def g(x):
    y = 10 * h(x) + h(x // 2)
    return y
def h(x):
    return 2 * x

z = 5
w = f(z)
```

main

z: 5 w: f(5) → 109

f(5)

x: 5 → 5 + g(5) → 5 + 104 → **109**

g(5)

x: 5 y → 10*h(5) + h(2) → 100 + 4 → **104**

h(5)

x: 5 → 2*5 → **10**

h(2)

x: 2 → 2*2 → **4**

COMMON GOTCHAS

- Floor division rounds **down**: `-7 // 2` is `-4`, not `-3`.
- A variable assigned inside a function does not exist once that call returns — using it afterward raises a `NameError`.

USING & IMPORTING A MODULE

<code>import math</code>	call as <code>math.sqrt(25)</code>
<code>import math as m</code>	call as <code>m.sqrt(25)</code>
<code>from math import sqrt, pow</code>	call as <code>sqrt(25)</code> — no prefix
<code>from random import *</code>	imports all names; call as <code>randint(1,6)</code>

`dir(math)` lists the names available in a module.

`help(math.sqrt)` shows its docstring/usage.

MATH & RANDOM MODULE FUNCTIONS

<code>math.sqrt(x)</code>	square root of x
<code>math.pow(x,y)</code>	x to the power of y
<code>math.sin(x)</code>	sine of x
<code>math.log(x,base)</code>	log of x in given base
<code>math.ceil(x)</code>	smallest integer $\geq x$
<code>random.random()</code>	float x, $0 \leq x < 1$
<code>random.randint(a,b)</code>	int x, $a \leq x \leq b$

None of these have side effects — all return a value.

STRINGS — INDEXING & SLICING

index	0	1	2	3	4	5	6	7
s =	C	O	M	P	U	T	E	R
neg. index	-8	-7	-6	-5	-4	-3	-2	-1

`s[i]` single character at index i `s[start:end]` slice, excluding end
`s[start:end:step]` every step-th char

```
s[2:5]    # 'MPU'            s[:3]    # 'COM'
s[::-1]   # 'RETUPMOC'
```

Strings are **immutable**: `+`, `*`, and methods build and return a **new** string, never modify in place.

STRING OPERATORS & METHODS

<code>str1 + str2</code>	concatenation
<code>str * n</code>	repeats the string n times
<code>x in string</code>	True if x appears in string
<code>len(string)</code>	number of characters
<code>s.lower()/s.upper()</code>	new string, lower/uppercase
<code>s.strip()</code>	outer whitespace removed
<code>s.find(sub)</code>	index of sub, or -1 if not found
<code>s.replace(old,new)</code>	every old → new
<code>s.split(sep)</code>	list of pieces split on sep

ESCAPE CHARACTERS & KEYWORD ARGUMENTS

`\n` newline `\t` tab `\"` literal quote `\\` literal backslash

```
s = "Tab:\tNew:\nQuote:\" Back:\""
>>> s                    # shows escapes literally
'Tab:\tNew:\nQuote:\" Back:\"'
>>> print(s)            # interprets escapes
Tab:    New:
Quote:" Back:\
```

```
print("a","b",sep="-")    # a-b
print("hi", end="")       # no trailing newline
```