CSCI 145 · Exam 2 Reference Sheet (Lectures 5–9) — page 1 of 2

COMPARISON & LOGICAL OPERATORS

`<` `>` `<=` `>=` `==` `!=` compare two values and produce a bool, `True` or `False`. **Careful:** = assigns a value; == asks whether two values are equal.

`a and b`	True only if both a and b are True
`a or b`	True if at least one of a, b is True
`not a`	True if a is False (flips it)

Evaluation order (first to last): `( )`, then arithmetic (`** * / // % + -`), then comparisons, then `not`, then `and`, then `or`.

CONDITIONALS: IF, IF/ELSE, IF/ELIF/ELSE

```
if condition:
    statements
```

Block runs only when the condition is True

```
if condition:
    statements
else:
    statements
```

Exactly one of the two blocks runs

```
if condition1:
    statements
elif condition2:
    statements
else:
    statements
```

- Conditions are checked **in order**; only the block for the **first** True one runs, and the rest are skipped
- You can chain as many `elif`s as you need; `else` is optional and must come last
- Separate `if` statements are each checked on their own, so more than one block can run
- The `:` ends each header line; indentation marks which lines are inside a block
- `return` ends a function immediately, even from inside a block or a loop

INPUT FROM THE USER

`input(prompt)`	Prints the prompt, waits for Enter, and returns what the user typed, **always as a string**
`int(s)`	Converts a string of digits to an int, e.g. `int("19")` is 19

THE RANDOM MODULE (import random)

`random.randint(a, b)`	A random int from a to b, **including** both a and b
`random.choice(seq)`	One item of a sequence, chosen at random (a character, for a string)

WHILE LOOPS

```
set up the loop variable
while condition:
    statements
    update the loop variable
```

- The condition is checked **before** each pass: if True, the body runs and Python checks again; if False, the loop ends
- If the condition is False the first time, the body runs **0 times**
- The body must eventually make the condition False. If it doesn't, you have an **infinite loop**

FOR LOOPS

```
for variable in sequence:
    statements
```

- The body runs once for each item in the sequence (a `range`, a string, or a list)
- Each pass, the loop variable holds the next item; the loop ends when the sequence runs out

```
for i in range(len(seq)):
    statements using seq[i]
```

Use the indices when you need the position too, or to change items of a list

THE RANGE FUNCTION

Produces a sequence of whole numbers. The **stop value is never included** (same rule as slicing).

Call	Values
`range(stop)`	0, 1, 2, … up to but not including stop
`range(start, stop)`	start, start + 1, … up to but not including stop
`range(start, stop, step)`	start, start + step, start + 2·step, … as long as they are less than stop

`range(2, 6)` → 2 3 4 5 ~~6~~

To include n itself, the stop has to be one past it: `range(1, n + 1)`

FOR OR WHILE?

- `for` when you know how many times to repeat, or have a sequence to go through
- `while` when you repeat until something happens

NESTED LOOPS

A loop body can contain another loop. The inner loop runs all the way through for each pass of the outer loop.

CSCI 145 · Exam 2 Reference Sheet (Lectures 5–9) — page 2 of 2

THE ACCUMULATOR PATTERN

```
result = starting value      # before the loop
for item in sequence:
    update result            # inside the loop
use result                   # after the loop
```

Typical starting value: `0` for a count or sum, `1` for a product, `""` for a string, `[]` for a list

- `x += y` is shorthand for `x = x + y`
- **Best so far:** start with one candidate, and replace it whenever you find a better one

LISTS

An ordered sequence of values of any type, in square brackets: `[4, "hi", 2.5]`. The empty list is `[]`.

Indexing, slicing, `len`, `+`, `*`, `in`, and `for` work the same way as for strings:

`seq[i]`	Item at index i; first is 0, last is `-1`
`seq[a:b]`	New sequence from index a up to, not including, b
`len(seq)`	Number of items
`a + b`	New sequence: items of a, then items of b
`seq * n`	New sequence: the items repeated n times
`x in seq`	`True` if x is in seq. Otherwise, `False`

MUTABLE VS. IMMUTABLE

- Lists are **mutable**: `items[i] = x` replaces the item at index i
- Strings are **immutable**: assigning to `s[i]` is an error
- String methods (`upper`, `replace`, ...) return a **new** string
- The list methods below change the list they are called on

LIST METHODS

Call	What it does	Returns
`items.append(x)`	Adds x to the end	None
`items.insert(i, x)`	Puts x at index i	None
`items.remove(x)`	Removes the first x	None
`items.pop()`	Removes the last item	the item
`items.pop(i)`	Removes item at index i	the item
`items.sort()`	Sorts, smallest first	None
`items.reverse()`	Reverses the order	None

BUILT-IN FUNCTIONS FOR LISTS

`sum(nums)`	The total of the items
`min(nums)`	The smallest item
`max(nums)`	The largest item
`sorted(seq)`	A **new** list of the items, smallest first; seq doesn't change

TWO NAMES, ONE LIST

- `b = a` does **not** copy the list: a and b are two names (**aliases**) for one list
- A change made to the list through either name shows up through both
- Assigning a new list, `b = [...]`, makes b name a new list; a still names the old one

Copy	Each makes a new list with the same items
`a[:]`	a slice of the whole list
`a.copy()`	the copy method
`list(a)`	a new list built from a

LISTS AND FUNCTIONS

- Passing a list to a function makes the parameter another name for the same list
- If the function changes that list, the caller sees the change
- Unless changing the list is the point, build and return a new list instead

MAP, FILTER, AND REDUCE

Pattern	What it does	Result
Map	Changes every item	a new list
Filter	Keeps some of the items	a new list
Reduce	Combines the items into one value	one value

LIST COMPREHENSIONS

A list comprehension builds a new list in one line.

```
[expression for variable in sequence]
[expression for variable in sequence if condition]
```

The second form builds the same list as this loop:

```
result = []
for variable in sequence:
    if condition:
        result.append(expression)
```

- `if` keeps only the items that pass (filter)
- The expression gives each new item (map)
- A comprehension can map and filter, but not reduce

NAMED CONSTANTS

```
MAX_TRIES = 5
VOWELS = "aeiou"
```

- A name in `ALL_CAPS` for a value that never changes, defined at the top of the file, outside any function
- Use one instead of a **magic number** (a value with no name)