Contents

Recal

There the five basic types we talked about day one:

- Numbers
- Characters
- **Booleans**
- Collections
- Tuple

Booleans

Booleans act as conditional values in CS. This is usually described in terms of truth tables and basic operations. The fundamental understanding is that a boolean is either *true* or *false*.

Gates (basic operations)

- Not (only unary)
- And
- NAnd
- Or
- XOr

The **Not** gate is defined as a unary operation that gives the inverse.

$$\text{Not}(a) = a^{\{-1\}} \tag{1}$$

$$\text{Not}(\text{true}) = \text{false} \tag{2}$$

We observe in Eq. 1 the mathematical inverse operation we denote as the word **Not**. It is less usual to see the word, rather we use a symbol like: $!, \sim, -, \ldots,$ etc. The truth table for the Not gate is as follows:

A	¬ **A**
True	False
False	True

The **And** gate is defined as a binary operation that combines two booleans (in this case).

$$\text{And}(a, b) = a * b \tag{3}$$

The truth table reads as follows:

A	B	$A \wedge B$
True	True	True
True	False	False
False	True	False
False	False	False

The **NAnd** gate is defined below by its truth table.

A	B	$A \neg \wedge B$
True	True	False
True	False	True
False	True	True
False	False	True

The **Or** gate is defined by its truth table below.

A	B	$A \vee B$
True	True	True
True	False	True
False	True	True
False	False	False

The **XOr** gate is defined by its truth table below. (most interesting gate)

A	B	$A \oplus B$
True	True	False
True	False	True
False	True	True
False	False	False

It is worth noting that booleans in literature can also appear as zero or one instead of false and true. This is to do with the context of binary, in particular the kind of “off and on” understanding of

binary. It is not important to memorize the truth tables but rather to understand the structure of Booleans and then the logic behind each operation.

Control Flow

We now explore the process of conditional logic. First we will explore this process of pseudocode rather than python. The concept of condition logic should be rather simple, of course *should* is presumptuous. We can think of the basic examples first then a more advanced type.

If-then-else

We can reference English heavily to understand if conditions. For example: **if** it is *raining* I will bring an *umbrella*, **otherwise** I will not. The condition is began with the beginning characters **if** next is the condition *raining*, then depending on the truth of *raining* we utilize the *umbrella*, the action if raining is false is described after the **otherwise** term. There are many ways to infer conditional logic in a pseudocode like form. The form Ill show is neither the most common nor the most readable, however it is the most clear version in my opinion.

```
if <boolean> then <action on true> else <action on false>
-- WARNING: the above is not runnable code
```

The above code explains the structure of how we create these if blocks.

$$\text{if } a \in \mathbb{Z} \text{ then } a+1 \text{ else } \sqrt{a} \tag{4}$$

We observe in Eq. 4 the pseudocode structure for a common if statement. This is very much like english, its the *mathy* way to write it. A much more common and equally valid approach (though) not as pretty is to use a more imparative process.

$$\begin{gathered}\text{if } a \in \mathbb{Z} \text{ then} \\ a+1 \\ \text{else} \\ \sqrt{a}\end{gathered} \tag{5}$$

This way of doing it is essentially the same, simply integrating end line characters for reading purposes[1]. The way this works that is the *condition* value evaluates to a boolean value. its much easier to see in the python code below.

```
cond = true
if cond:
    print("the true action")
else:
    print("the false action")
```

Match blocks

If your conditional logic is not linear or branch but rather large then match blocks can be useful. A match block is something that allows one to filter through *many* values all at once.

```
match color:
    case "blue":
        print("we have the blue color")
    case "red":
        print("we have the red color")
    case "green":
        print("we have the green color")
```

[1] it goes without saying that both styles will get full marks on an exam

```
    case _:
        print("unknown color")
```

The match block is comprised of two simple keywords in python: *match*, *case*. It is obvious how to build and use these blocks but there are a few infered rules that need attention. The main rule is that we always need a default case, if we look at the last case it lists the _ value, which of course is the **wildcard** matching any value. The other rule to keep in mind is the evaluated types of the match value and case values need to match.

Nomenclature

Σ l - is the *sumation* of the set l, this means that it takes each element of all and preforms the addition operation on them sequentially

$s.t$ or : - read “such that”

$\forall$ x $\in \mathbb{Z}$ - read “for all x in $\mathbb{Z}$”, denotes iteration over all indices of $\mathbb{Z}$

A $\longleftarrow$ B - denoting a binding where A will represent the value of B

A $\longrightarrow$ B - read A **implies** B, denoting the that the statement A being true implies that B is also true

[ ] - denoting a set

$\mathbb{Z}$ - set of all integers (real numbers or their negatives)

$\mathbb{R}$ - set of all real numbers (rational numbers/irrational numbers, including fractions and their negatives)

i $\in$ I - denotes that i is an element of the set I