Funktionen

→ Funktionen folgen immer diesem Muster:

```
return_type  function_name ( arguments ){
    // body
    // return statement
}
```

↳ mögl. return_types sind int / bool / float etc. dann sagen wir die Funktion gibt uns einen Wert mit diesem Typ zurück.
→ void als return type heisst wir geben nichts zurück

↳ Wenn eine Fkt aufgerufen wird, springt das Programm in diese Fkt und führt den Code aus bis ein return value; gefunden wird (oder bei einer void Fkt, die Fkt fertig ist.)

Bsp.

```
int blabla (){
    :
    if (... ){ return 5; }
    if (.. ) { return 6; }
    return 7;
}
```

Wichtig: Jeder mögliche Weg (heisst in allen if/else etc) muss ein return haben!

↳ Das sind die Fkt-Argumente z.B.

```
int add_two_numbers( int a, int b ){ ... }
```

Step-wise refinement

→ grosse Probleme angehen, indem ihr sie in kleinere unterteilt
⇒ im Code: Funktionen

→ e.g. Funktionen leer hinschreiben
↳ in Kommentaren schreiben was die Idee der Fkt ist
↳ schauen ob sie einzeln funktionieren
≙ unit testing

⇒ umgekehrt auch für debugging gültig
→ wenn es nicht funktioniert dann nicht einfach random suchen sondern quasi schrittweise annähern
→ bis hier stimmts
↳ bis hier ...
in dieser Fkt. etc.

PRE / POST

1. Pre-conditions are the things that must be true before a method is called. The method tells clients "this is what I expect from you".
2. Post-conditions are the things that must be true after the method is complete. The method tells clients "this is what I promise to do for you".
3. Invariants are the things that are always true and won't change. The method tells clients "if this was true before you called me, I promise it'll still be true when I'm done".

Pre → so weit wie mögl. ; ihr
↳ maximiert den Nutzen der Funktion nicht einschränken

Post → mögl. genau / eng
↳ soll ganz genau wissen was passiert

↳ einfach "lesbarer" sagen was die Fkt returned
e.g. ok wenn ihr sagt Fkt returned $\sum_{i=0}^{n} \frac{1}{i^2}$