Side Effects

- Ein Side Effect ist alles was eine Funktion macht, außer einfach einen Wert zurückzugeben
  - Globale Variable ändern
  - Internen Zustand eines Objekts ändern (OOP stuff)
  - Etwas in eine Datei schreiben oder Console stuff
  - Netzwerkverbindung aufbauen
- FP minimiert Side Effects oder trennt sie klar vom Rest des Codes → `Pure Functions`

TODO: EXPLIZIT darauf eingehen was die Nachteile davon sind mit Beispielen etc.

Side Effects sind alle Dinge die eine Funktion macht, die nicht Teil ihres Outputs sind. Das Ändern von globalen Variablen, das Modifizieren von Objekten, das Schreiben in Dateien oder das Aufbauen von Netzwerkverbindungen sind alles Beispiele für Side Effects. Der Nachteil von Side Effects ist, dass sie den Zustand des Programms verändern können, was zu unerwartetem Verhalten führen kann. Dies erschwert das Testen und Debuggen von Code, da die Funktion nicht mehr isoliert betrachtet werden kann. Außerdem leidet die Transparenz des Codes, da der Leser nicht sofort erkennen kann, welche Teile des Codes den Zustand verändern. Der Leser muss den gesamten Kontext verstehen, um die Auswirkungen einer Funktion vollständig zu begreifen. Ein Beispiel für Side Effects ist eine Funktion, die eine globale Variable ändert:

```
counter = 0
def increment():
    global counter
    counter += 1
increment()
print(counter)  # Ausgabe: 1
```

Funktionale Programmierung hingegen strebt danach, Side Effects zu minimieren oder klar vom Rest des Codes zu trennen, um die Vorteile reiner Funktionen zu nutzen. *Pure Functions* sind Funktionen, die für dieselben Eingaben immer dieselben Ausgaben liefern und keine Side Effects haben. Dies erleichtert das Verständnis, Testen und die Wiederverwendbarkeit von Code erheblich. Ein Beispiel für eine reine Funktion ist:

```
def increment(x):
    return x + 1

counter = increment(0)
print(counter)  # Ausgabe: 1
```

In diesem Beispiel hat die Funktion `increment` keine Side Effects, da sie nur den Wert ihrer Eingabe um 1 erhöht, ohne den Zustand des Programms zu verändern. Außerdem erkennt man sofort, dass `increment(0)` immer `1` zurückgibt, unabhängig vom Kontext.

Transparenz

TODO: man muss halt iwi interne Implementierung kennen weil SE nicht klar aldf

(wir müssen es schaffen nen convincing case vs. OOP zu machen)

- **FP:** Funktionen können isoliert betrachtet und verstanden werden
- **OOP:** Man muss die gesamte Klasse und ihren aktuellen Zustand verstehen, um eine einzige Methode zu verstehen
- *OOP* macht $f(x)$, während *FP* $f(5)$ macht, man sieht also direkt was *FP* macht (solange man keine kryptischen Namen benutzt)

- **WICHTIG:** komplett ohne *Side Effects* kommt man nicht aus, weil irgendwann muss das Programm irgendeinen Zustand ändern damit es sinnvoll ist **TODO: Drauf eingehen, dass man das an den Rand verlagern kann (Beispiel: Datenbank für auslagern von state)**

Concurrency

TODO: global mutable Variablen sind halt scheiße für multithreading nh

- *Shared Mutable State* ist Endgegner von Multithreading
- Beispiel: Bankautomat
  - Konto hat 100 Rubel
  - Thread 1 hebt 50 Euro ab
  - Thread 2 hebt 50 Drachmen ab
  - Beide lesen 100, ziehen 50 ab, beide schreiben 50 rein
  - Konto hat nach abziehen von $50 + 50 = 100$ immer noch 50
- In *OOP* muss man das mit *Locks* verhindern
- Thread 1 muss Transaktion erst abschließen und Thread 2 muss warten
- Führt oft zu *Deadlocks*

FP-Lösung:

Thread 1 und 2 möchten wieder von einem Account jeweils 50 abheben.
- `withdraw`-Funktion ist *rein* und der Account ist *immutable*
- Thread 1 bekommt ein neues Objekt vom Account
- Thread 2 bekommt ein neues Objekt vom Account
- Threads konkurrieren nicht mehr und beschreiben nicht denselben Speicher

Unter *Concurrency* versteht man die Fähigkeit eines Programms, mehrere Aufgaben gleichzeitig auszuführen. Ein Problem im *OOP* ist der Umgang mit *Shared Mutable State*, also mit gemeinsam genutztem, veränderbarem Zustand. Wenn mehrere Threads gleichzeitig auf denselben Zustand zugreifen und diesen ändern, kann es zu Inkonsistenzen kommen. Das führt zu unerwartetem Verhalten, da die Threads sich gegenseitig beeinflussen. In *OOP* wird dieses Problem mit *Locks* gelöst, die sicherstellen, dass nur ein Thread gleichzeitig auf den Zustand zugreifen kann. Dies kann jedoch zu *Deadlocks* führen, bei denen zwei oder mehr Threads sich gegenseitig blockieren und nicht weiterarbeiten können. In der *funktionalen Programmierung* hingegen sind Daten *immutable*. Ein Thread in der *funktionalen Programmierung* bekommt also keinen globalen Zustand, den er ändern kann, sondern eine Kopie des Zustands. Wenn ein Thread eine Änderung vornehmen möchte, dann erstellt er eine neue Version des Zustands, anstatt den ursprünglichen Zustand zu ändern. Damit können Threads nicht mehr konkurrieren und sich gegenseitig beeinflussen. In der *funktionalen Programmierung* wird das Problem der *Concurrency* also vermieden indem es keine gemeinsamen Zustände gibt.