Cheat Sheet

1. **SINGLE LEVEL OF ABSTRACTION:**
   a. High-Level <=> Low-Level-Code unterscheiden
   b. Lesbarkeit der Klasse und ggfs. Bei Bedarf Implementrierungsdetails nachlesen

Add_element()
Code in Hilfsfunktionen
Auffindbar

```
def add(self):
    message='Enter your Budget for every month in following format (Typ
    add='yes'
    while add.lower() in ['yes']:
        inputs=add_element('budget',message)
        bud=self.add_category(inputs[0])\
                .add_amount(inputs[1])\
                .add_message(inputs[2])\
                .add_month(inputs[3])\
                .add_year(inputs[4])
        add_element_in_db('budget',category=bud.category,amount=bud.amo
        add=questioner('Do you want to add more?', input_needed=True)
```

2. **QUELLTEXTKONENTIONEN.**
   a. Siehe Bild oben: Namensvergabe stehts eindeutig => Keine Kommentare notwendig
   b. Namensregeln wurden vergeben
3. **FEHLERVERFLOGUNG:**
   a. Probleme, Wünsche und offene Punkte werden strukturiert erfasst und aufgeschrieben (Geschehen in Tasks auf github)
   b. Aufgaben werden nicht vergessen und können nachverfolgt werden
4. **REVIEW:**
   a. Vier Augen sehen mehr als zwei
   b. Code erklären führt zu besseren Verständnis und zeigt Mängel auf
   c. Anwendung: Sehr früh im Entwicklungsprozess
5. **WIEDERHOLE DICH NICHT:**
   a. Doppelung von Code/Handgriffen begünstigt Fehler
   b. Erkennen von sich wiederholenden Code oder anderen Artefakten, die man selbst erzeugt hat
   c. Bereinigung durch Refaktorisierungen

Wird sowohl von Expenses und Budget benutzt => keine Doppelung im Code

```
def add_element(membership,message):
    inputs=questioner(*message,input_needed=True)
    add='y'
            add: str
    while add.lower() in ['yes','y']:
        try:
            inputs= inputs.split('/')
            if len(inputs)!=member[membership][0]:
                raise Exception
            return inputs

        except Exception:
            print('Your format was NOT correct. Try again (t) or quit (q)!')
            add=str(input())
```

6. **KISS - HALTE ES EINFACH:**
   a. Für die Evolvierbarkeit muss Code verständlich sein
   b. Nicht mehr tun als das Einfachste => Sonst Aufschiebung anderer wichtiger Tasks

Sehr einfache und verständliche Funktion zum Ausgeben der Einträge

```
def view_results(self,membership):
    results= data_db(membership)
    for entry in results:
        print(('\t|\t'.join([str(e) for e in entry])))
    return "200"
```

7. **IMPLEMENTIERUNG SPIEGELT ENTWURF:**
   a. In der Architektur definierte Komponenten auch im Code möglichst physisch trennen
   b. Verbesserung der Übersichtlichkeit und Testbarkeit
8. **OFFEN-GESCHLOSSEN-PRINZIP:**
   a. Module sollten offen für Erweiterungen und geschlossen für Modifikationen sein
   b. Veränderung existierenden Codes bei Erweiterung vermeiden (Fehlerquelle)
9. **SCHNITTSTELLENAUFTEILUNGSPRINZIP:(SIP)**
   a. Schnittstellen sollten eine hohe Kohäsion haben, also nur Dinge enthalten, die wirklich eng zusammengehören
   b. Schlanke Schnittstellen erfordern weniger Methodenimplementierungen

Einfache Schnittstellenimplementierung
=>
Add_element und delete_element müssen sowohl in expenses als auch budget implementiert werden

```
class Transaction(metaclass=ABCMeta):
    @classmethod
    def __subclasshook__(cls, subclass):
        return (hasattr(subclass, 'add_element') and
                callable(subclass.add_element) and
                hasattr(subclass, 'delete_element') and
                callable(subclass.delete_element) and
                hasattr(subclass,'loop') and
                callable(subclass.loop) or
                NotImplemented)

    @abstractmethod
    def add_element(self,category,amount, description, month,year):
        raise NotImplementedError

    @abstractmethod
    def delete_element(self,category,amount, description, month,year):
        raise NotImplementedError
```

10. **ABHÄNGIGKEIT-UMKEHRUNGS-PRINZIP(DIP).:**
    a. High-Level-Klassen sollten nicht von Low-Level-Klassen abhängig sein, sondern beide von Schnittstellen
    b. Schnittstellen sollten nicht von Details abhängig sein, sondern Details von Schnittstellen
11. **TEILNAHME AN FACHVERANSTALTUNGEN:**
    a. Am besten lernen wir von anderen und in der Gemeinschaft
    b. Austausch mit Entwicklern außerhalb des Teams, um andere Meinungen zu erfahren
12. **LESEN,LESEN,LESEN:**
    a. Softwaretechnik, Methoden und Werkzeuge entwickeln sich ständig weiter
    b. Regelmäßige Fachpublikationen lesen
    c. Auf den Laufenden bleiben
    d. Fähigkeit, informiert Entscheidungen zu treffen
13. **VORSICHT VOR OPTIMIERUNGEN:**
    a. Optimierungen kosten Aufwand und verringern die Code-Lesbarkeit
    b. Sie sind oft nicht notwendig oder nützlich
    c. Wenn, dann nur nach Analyse durch Profile (was ich in dem Projekt auch selber bin)
14. **URSACHENANALYSE:**
    a. Symptome behandeln bringt schnell Linderung, kostet langfristig aber mehr Aufwand
    b. Unter die Oberfläche von Problemen schauen ist letztendlich effizienter
15. **PRINZIP DER GERINGSTEN ÜBERRASCHUNG:**
    a. Wenn sich eine Komponente überraschender Weise anders verhält als erwartet, wird ihre Anwendung unnötig kompliziert und fehleranfällig
    b. Abfragemethoden wie GETVALUE() sollen den Zustand nicht verändern