Mögliche Lösungen zu den Aufgaben aus Block 2

Aufgabenblock 1: Die Studentenklasse

In der Videolecture haben wir die folgende Klasse `Student` entwickelt:

```
class Student:
    """
    This is a class called student.
    """
    profession = "student"

    def __init__(self, name, affiliation):
        self.name = name
        self.affiliation = affiliation
        self.beers = 0
        self.credits = 0
        self.friends = set()

    def __repr__(self):
        return self.name

    def drink(self, beers):
        self.beers += beers
        return beers*["Cheers!"]

    def study(self, cp_earned):
        self.credits += cp_earned

    def make_friend(self, new_friend):
        assert isinstance(new_friend, Student), "A student can only befriend other students!"
```

```
        self.friends.add(new_friend)

    def drink_with_friends(self, beers):
        for f in self.friends:
            f.drink(beers)
            self.drink(beers)
```

Aufgabe 1.1: Ergänze die `__repr__` Funktion des Studenten, sodass sie nicht nur seinen Namen, sondern auch die Anzahl der Credits und der Biere augegeben werden. Wenn die Instanz des Studierenden aufgerufen wird, soll die Ausgabe in etwa folgendermaßen ausseßen (vorausgesetzt, die Studenting heißt Claire, hat 5 Biere getrunken, und 10 credits erhalten):

```
print("Claire (5 Biere und 10 Credits)")
```

```
Claire (5 Biere und 10 Credits)
```

Hinweis: Denkt daran, dass ihr einzelne Strings mit dem Additionsoperator `+` miteinender verbinden könnt:

```
"Hallo" + " und " + "Tschuess!"
```

```
'Hallo und Tschuess!'
```

Hinweis 2: Das funktioniert nur mit strings! Zahlen müssen vorher in strings umgewandelt werden (siehe Funktion `str`).

```
class Student:
    """
    This is a class called student.
    """
    profession = "student"

    def __init__(self, name, affiliation):
        self.name = name
        self.affiliation = affiliation
        self.beers = 0
        self.credits = 0
        self.friends = set()

    def __repr__(self):
```

```
        return_val = self.name + " (" + str(self.beers) + " Biere und " + str(self.credits) +
        return return_val

    def drink(self, beers):
        self.beers += beers
        return beers*["Cheers!"]

    def study(self, cp_earned):
        self.credits += cp_earned

    def make_friend(self, new_friend):
        assert isinstance(new_friend, Student), "A student can only befriend other students!"
        self.friends.add(new_friend)

    def drink_with_friends(self, beers):
        for f in self.friends:
            f.drink(beers)
            self.drink(beers)
claire = Student("Claire", "Uni Erfurt")
claire
```

```
Claire (0 Biere und 0 Credits)
```

Aufgabe 1.2: Ergänzt die `__init__` Funktion, sodass jede Instanz des Studenten auch ein Instanzenattribut `title` hat. Dieses soll bei Instanziierung eines Studenten zunächst den Wert `'nichts'` haben.

```
class Student:
    """
    This is a class called student.
    """
    profession = "student"

    def __init__(self, name, affiliation):
        self.name = name
        self.affiliation = affiliation
        self.beers = 0
        self.credits = 0
        self.friends = set()
        self.title = "nichts"
```

```
    def __repr__(self):
        return_val = self.name + " (" + str(self.beers) + " Biere und " + str(self.credits) +
        return return_val

    def drink(self, beers):
        self.beers += beers
        return beers*["Cheers!"]

    def study(self, cp_earned):
        self.credits += cp_earned

    def make_friend(self, new_friend):
        assert isinstance(new_friend, Student), "A student can only befriend other students!"
        self.friends.add(new_friend)

    def drink_with_friends(self, beers):
        for f in self.friends:
            f.drink(beers)
            self.drink(beers)
claire = Student("Claire", "Uni Erfurt")
claire.title
```

```
'nichts'
```

Aufgabe 1.3: Ergänze die Klasse `Student` um eine neue Instanzenfunktion namens `graduate`. Diese Funktion soll überprüfen ob der Studierende mindestens 180 credits und 90 Biere hat.

Falls ja, soll das Attribut `title` auf den String `'BA'` geändert werden. Zudem soll dann die Funktion `drink_with_friends` mit 10 Bieren aufgerufen werden und eine Meldung (durch `print`) ausgegeben werden, dass der Student jetzt graduiert ist.

Falls nein, soll nur eine Meldung (durch `print` ausgegeben werden), dass der Student noch nicht graduieren kann.

Hinweis: Mehrere Abfragen in if/else statements können durch das `and` Keyword durchgeführt werden:

```
(2<3) and (4>2)
```

```
True
```

```
(2<3) and (2>2)
```

```
False
```

```
class Student:
    """
    This is a class called student.
    """
    profession = "student"

    def __init__(self, name, affiliation):
        self.name = name
        self.affiliation = affiliation
        self.beers = 0
        self.credits = 0
        self.friends = set()
        self.title = "nichts"

    def __repr__(self):
        return_val = self.name + " (" + str(self.beers) + " Biere und " + str(self.credits) +
        return return_val

    def drink(self, beers):
        self.beers += beers
        return beers*["Cheers!"]

    def study(self, cp_earned):
        self.credits += cp_earned

    def make_friend(self, new_friend):
        assert isinstance(new_friend, Student), "A student can only befriend other students!"
        self.friends.add(new_friend)

    def drink_with_friends(self, beers):
        for f in self.friends:
            f.drink(beers)
            self.drink(beers)

    def graduate(self):
        if (self.credits>=180) and (self.beers>=90):
            self.title = "BA"
            self.drink_with_friends(10)
```

```
            print("Hurra! Graduiert!")
        else:
            print("Oh weh, kann noch nicht graduieren...")

claire = Student("Claire", "Uni Erfurt")
claire.graduate()
```

```
Oh weh, kann noch nicht graduieren...
```

Aufgabe 1.4: Erstellt eine Instanz der Klasse `Student` und lasst sie so lange trinken und lernen, bis sie graduieren kann. Dann lasst sie graduieren.

```
claire = Student("Claire", "Uni Erfurt")
claire.study(180)
claire.drink(90)
claire.graduate()
```

```
Hurra! Graduiert!
```

Aufgabe 1.5: Schreibt eine Klasse `FlunkyBall`. Diese Klasse soll zwei Teams erstellen, die jeweils aus einer gleichen Anzahl von `Studenten` bestehen sollen. Die Anzahl der Studenten pro Team soll als Argument der `__init__` Funktion gegeben werden.

Dann soll die Klasse über eine Funktion `play` verfügen: diese Funktion nimmt ein Argument an. Dieses Argument spezifiziert, welches der Teams verliert. Alle Mitglieder des Verliererteams müssen nun ein Bier trinken.

Hinweis 1: Hier müssen wahrscheinlich if/else Befehle verwendet werden.

Hinweis 2: Ihr könnt euch hier ein wenig an der Klasse `StuRa` orientieren, die wir in den Video Lectures erstellt haben:

```
class StuRa:
    """
    A class for StuRas
    """
    function = "Organisation der Studis"

    def __init__(self, members):
        assert isinstance(members, set), "Members must given as set!"
        for m in members:
            assert isinstance(m, Student), "Members as such must be students!"
```

```
        self.members = members

    def befriend_members(self):
        """
        Befriends all members.
        """
        for m in self.members:
            for n in self.members:
                if m is not n:
                    m.make_friend(n)

    def hard_work(self, amount_work):
        for m in self.members:
            print(m, "works hard!")
            m.drink(amount_work)
```

Hinweis 3: Wir haben in den Video lectures gesehen wie man einfach eine größere Menge Studenten erstellt:

```
n_students = 10
list_of_students = {Student("student_" + str(x), "Uni Erfurt") for x in range(n_students)}
```

```
class FlunkyBall:
    def __init__(self, n_players):
        self.team_1 = {Student("student_" + str(x), "Uni Erfurt") for x in range(int(n_playe
        self.team_2 = {Student("student_" + str(x), "Uni Erfurt") for x in range(int(n_playe

    def play(self, winner):
        assert (winner==1) or (winner==2), "Must specify team 1 or 2!"
        if winner == 1:
            for m in self.team_2:
                m.drink(1)
        else:
            for m in self.team_1:
                m.dring(1)

funny_flunky = FlunkyBall(10)
funny_flunky.play(1)
```

Aufgabenblock 2: Das Solow Wachstumsmodell

In dieser Aufgabe sollt ihr eine Version von Solow's Wachstumsmodell erstellen. Klingt anspruchsvoll, aber mit ein paar Hinweisen könnt könnt ihr das schon!

Erster Schritt: Eine Klasse für die Ökonomie!: Als erster Schritt müsst ihr eine Klasse `Economy` erstellen.

Diese Klasse soll *drei Instanzeneigenschaften* haben: `capital`, `labor`, und `bip`. Diese Klasseneigenschaften sollen jeweils aus einer Liste bestehen. Deren erstes Element soll für `capital` und `labor` als Argument an die `__init__` Funktion übergeben werden. `bip` soll als leere Liste instantiiert werden.

Als Starthilfe hier schon einmal folgendes Grundgerüst, das ihr weiterverwenden könnt:

```
class Economy:
    """
    An economy.
    """
    def __init__(self, ):
        self.capital = capital
        self.labor
        self.
```

Die Lösung wäre:

```
class Economy:
    """
    An economy.
    """
    def __init__(self, ):
        self.capital = [capital]
        self.labor = [labor]
        self.bip = []
```

Zweiter Schritt: Eine Funktion für das BIP: Im nächsten Schritt müsst ihr eine Klassenfunktion schreiben, die für die aktuellen Werte von `capital` und `labor` den entsprechenden Wert für `BIP` berechnet.

Das Solow Wachstumsmodell verwendet hierfür ein *Cobb-Douglas* Produktionsfunktion:

$$Y_t = K_t^{\alpha} L_t^{1-\alpha}$$

Das bedeutet: der Wert für das BIP (Y in der Gleichung) zum Zeitpunkt t beträgt der aktuelle Kapitelstock exponiert mit dem Wert α multipliziert mit der Arbeitskraft der Ökonomie (L in der Gleichung), exponiert mit dem Wert $(1 - \alpha)$.

Außerhalb einer Klasse können wir diese Funktion folgendermaßen in Python implementieren:

```
def cobb_douglas(capital, labor, alpha):
    bip = capital**alpha * labor**(1-alpha)
    return bip
```

Implementiert diese Funktion als Klassenfunktion in der Klasse `Economy`. Vergesst nicht, dass `self` Argument hinzuzufügen!

```
class Economy:
    """
    An economy.
    """
    def __init__(self, capital, labor ):
        self.capital = [capital]
        self.labor = [labor]
        self.bip = []

    def cobb_douglas(self, capital, labor, alpha):
        bip = capital**alpha * labor**(1-alpha)
        return bip

expl_econ = Economy(5, 10)
```

Dritter Schritt: Anpassung der Basisklasse: Wir sind schon bald fertig! Ergänzt nun die `__init__` Funktion eurer Klasse `Economy` sodass sie ein weiteres Argument annimmt: `alpha`. Dieses soll als Instanzenattribut gespeichert werden, genauso wie `capital`, `labor` und `bip`, nur nicht als liste, sondern als `float`.

Auch könnt ihr nun die Instanzenfunktion `cobb_douglas` anpassen: sie soll nun nicht mehr `alpha` als Argument nehmen. Vielmehr kann sie `alpha` direkt als Instanzenattribut ansprechen.

Das ist ähnlich als ob bei der Funktion oben `alpha` vorher außerhalb der Funktion definiert worden wäre. Ihr müsst diese Logik nur an euren Klassenkontext angepasst werden (denkt an die Rolle von `self`):

```
alpha = 0.5
def cobb_douglas(capital, labor):
    bip = capital**alpha * labor**(1-alpha)
    return bip
cobb_douglas(10, 20)
```

```
14.142135623730953
```

Hinweis:

```
class Economy:
    """
    An economy.
    """
    def __init__(self, alpha, ):
        self.alpha =
        self.capital = capital
        self.labor
        self.
```

```
class Economy:
    """
    An economy.
    """
    def __init__(self, alpha, capital, labor):
        self.capital = [capital]
        self.labor = [labor]
        self.bip = []
        self.alpha = alpha

    def cobb_douglas(self, capital, labor):
        bip = capital**self.alpha * labor**(1-self.alpha)
        return bip

expl_econ = Economy(0.5, 10, 20)
```

Vierter Schritt: Ersten BIP Wert berechnen Ihr müsst nun ein letztes Mal die `__init__` Funktion anpassen: sie soll den ersten Wert für `self.bip` mit der Funktion `cobb_douglas` festlegen.

Hinweis:

```
class Economy:
    """
    An economy.
    """
    def __init__(self, alpha, ):
        self.alpha =
        self.capital = capital
        self.labor
        self.bip = []
        self.bip.append(self.cobb_douglas(self.capital[-1], self. ))
```

```
class Economy:
    """
    An economy.
    """
    def __init__(self, alpha, capital, labor):
        self.alpha = alpha
        self.capital = [capital]
        self.labor = [labor]
        self.bip = []
        self.bip.append(self.cobb_douglas(self.capital[-1], self.labor[-1]))


    def cobb_douglas(self, capital, labor):
        bip = capital**self.alpha * labor**(1-self.alpha)
        return bip

expl_econ = Economy(0.5, 10, 20)
expl_econ.bip
```

```
[14.142135623730953]
```

Fünfter Schritt: Die Zeitschritte: Ihr müsst nun eine weitere Instanzenfunktion hinzufügen: `timestep`. Diese Funktion nimmt als Input die aktuellen Werte von `capital`, `labor` und `bip` und berechnet den Wert für den nächsten Zeitschritt.

Die *Veränderungen* von `capital` und `labor` ergeben sich aus dem Investment in neues Kapital, der Kapitalabnutzung und dem Bevölkerungswachstum. Diese Veränderungen lassen sich durch folgende beiden Funktionen berechnen, die voraussetzen, dass 5% des BIP in Kapital investiert wird, 6% des aktuellen Kapitalstocks abgenutzt werden, und die Bevölkerung mit 3% pro Zeitschritt wächst.

```
class Economy:
    def change_capital(self):
        change_of_capital = self.bip[-1]*0.05 - self.capital[-1]*0.06
        return change_of_capital

    def change_labor(self):
        change_of_labor = self.labor[-1]*0.03
        return change_of_labor
```

Hinweis: Die `[-1]` Notation ruft das letzte Element einer Liste auf:

```
l_1 = [1,2,3,4]
l_1[-1]
```

```
4
```

Die Funktion `timestep` soll also alle drei Update Funktionen aufrufen, die neuen Werte berechnen, und an die Listen für `capital`, `labor` und `bip` anhängen (mit der `append` Methode).

Folgender Hinweis ist vielleicht hilfreich:

```
class Economy:
    def change_capital(self):
        change_of_capital = self.bip[-1]*0.05 - self.capital[-1]*0.06
        return change_of_capital

    def timestep(self):
        capital_change = self.change_capital()
        capital_new = self.capital[-1] + capital_change capital
        self.capital.append(capital_new)
        # similar operations for labor and bip
```

```
class Economy:
    """
    An economy.
    """
    def __init__(self, alpha, capital, labor):
        self.alpha = alpha
        self.capital = [capital]
        self.labor = [labor]
        self.bip = []
```

```
    def cobb_douglas(self, capital, labor):
        bip = capital**self.alpha * labor**(1-self.alpha)
        return bip

    def change_capital(self):
        change_of_capital = self.bip[-1]*0.05 - self.capital[-1]*0.06
        return change_of_capital

    def change_labor(self):
        change_of_labor = self.labor[-1]*0.03
        return change_of_labor

    def timestep(self):
        capital_change = self.change_capital()
        capital_new = self.capital[-1] + capital_change
        self.capital.append(capital_new)

        labor_change = self.change_labor()
        labor_new = self.labor[-1] + labor_change
        self.labor.append(labor_new)

        bip_new = self.cobb_douglas(self.capital[-1], self.labor[-1])
        self.bip.append(bip_new)

expl_econ = Economy(0.25, 5, 10)
expl_econ
```

```
<__main__.Economy at 0x118c6cfd0>
```

Sechster Schritt: Die run Funktion: Das ist der letzte Schritt. Definiert eine Funktion `run`, die als Argument die Anzahl der Zeitschritte nimmt, und dann `capital`, `labor` und `bip` für die Anzahl der Zeitschritte neu berechnet. In jedem Zeitschritt soll die Funktion `timestep` aufgerufen werden.

Hinweis:

```
class Economy:

    def run(self, ):
        for t in range( ):
            print(t)
            self.
```

```python
class Economy:
    """
    An economy.
    """
    def __init__(self, alpha, capital, labor):
        self.alpha = alpha
        self.capital = [capital]
        self.labor = [labor]
        self.bip = []
        self.bip.append(self.cobb_douglas(self.capital[-1], self.labor[-1]))

    def cobb_douglas(self, capital, labor):
        bip = capital**self.alpha * labor**(1-self.alpha)
        return bip

    def change_capital(self):
        change_of_capital = self.bip[-1]*0.05 - self.capital[-1]*0.06
        return change_of_capital

    def change_labor(self):
        change_of_labor = self.labor[-1]*0.03
        return change_of_labor

    def timestep(self):
        capital_change = self.change_capital()
        capital_new = self.capital[-1] + capital_change
        self.capital.append(capital_new)

        labor_change = self.change_labor()
        labor_new = self.labor[-1] + labor_change
        self.labor.append(labor_new)

        bip_new = self.cobb_douglas(self.capital[-1], self.labor[-1])
        self.bip.append(bip_new)

    def run(self, t_steps):
        for t in range(t_steps):
            print(t)
            self.timestep()
```

Siebter Schritt Erstellt eine Instanz eurer Ökonomie, lasst das Modell für 10 Zeitschritte laufen, und lasst euch die finalen Werte für `capital`, `labor` und `bip` ausgeben. Ihr könnt hier

folgendes Muster verwenden:

```
expl_econ = Economy(0.4, 5, )
expl_econ.run( )
```

```
expl_econ = Economy(0.4, 5, 8)
expl_econ.run(10)
```

```
0
1
2
3
4
5
6
7
8
9
```

Hier sind die relevanten Statusvariablen:

```
expl_econ.labor
```

```
[8,
 8.24,
 8.4872,
 8.741816,
 9.00407048,
 9.274192594399999,
 9.552418372232,
 9.83899092339896,
 10.13416065110093,
 10.438185470633957,
 10.751331034752976]
```

```
expl_econ.capital
```

```
[5,
 5.031445401733999,
 5.06778191105444,
```

```
 5.108982601976881,
 5.155029407627849,
 5.205912785920756,
 5.261631412835216,
 5.322191901872839,
 5.3876085483457805,
 5.457903097233226,
 5.533104533415775]
```

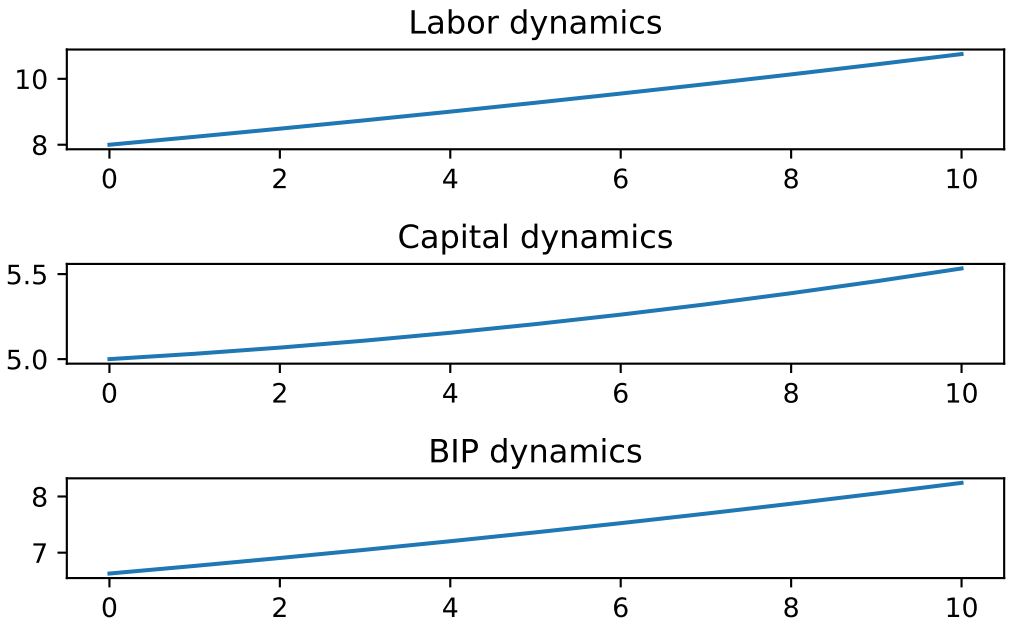
```
expl_econ.bip
```

```
[6.628908034679974,
 6.764464668489624,
 6.9053521117141425,
 7.0517152353916135,
 7.203702855011553,
 7.361467881394116,
 7.525167476154716,
 7.694963211706246,
 7.871021235763835,
 8.053512440330865,
 8.242612635159615]
```

Wir können das auch visualisieren (mehr dazu aber in einer späteren Session):

```
import matplotlib.pyplot as plt
import numpy as np
fig, axes = plt.subplots(3,1)

axes[0].plot(np.linspace(0, 10, 11), expl_econ.labor)
axes[0].title.set_text("Labor dynamics")
axes[1].plot(np.linspace(0, 10, 11), expl_econ.capital)
axes[1].title.set_text("Capital dynamics")
axes[2].plot(np.linspace(0, 10, 11), expl_econ.bip)
axes[2].title.set_text("BIP dynamics")
fig.tight_layout()
```


Labor dynamics
10
8
0
2
4
6
8
10
Capital dynamics
5.5
5.0
0
2
4
6
8
10
BIP dynamics
8
7
0
2
4
6
8
10