Informatik I: Einführung in die Programmierung

8. Objekte und Datenklassen

Albert-Ludwigs-Universität Freiburg

UNI FREIBURG

Prof. Dr. Peter Thiemann

02. Dezember 2020

UNI FREIBURG

Objekte und Datenklassen

UNI FREIBURG

Objekte

Objekte und Attribute

- Alle *Werte* in Python sind in Wirklichkeit *Objekte*.

Objekte und Attribute

UNI FREIBURG

- Alle *Werte* in Python sind in Wirklichkeit *Objekte*.
- Damit ist gemeint, dass sie assoziierte *Attribute* und *Methoden* haben, auf die mit der Punktnotation

```
ausdruck.attribut
```

zugegriffen werden kann:

Objekte und Attribute

UNI FREIBURG

- Alle *Werte* in Python sind in Wirklichkeit *Objekte*.
- Damit ist gemeint, dass sie assoziierte *Attribute* und *Methoden* haben, auf die mit der Punktnotation

  `ausdruck.attribut`

  zugegriffen werden kann:

Python-Interpreter

```
>>> x = complex(10, 3)
>>> x.real, x.imag
10.0 3.0
>>> "spam".index("a")
2
>>> (10 + 10).__neg__()
-20
```

Identität: `is` und `is not`

- Jedes Objekt besitzt eine eigene Identität.
- Die Operatoren `is` und `is not` testen die Identität:
- `x is y` ist `True`, wenn `x` und `y` dasselbe Objekt bezeichnen, und ansonsten `False` (**`is not`** umgekehrt):

Python-Interpreter

```
>>> x, y = ["ham", "spam", "jam"], ["ham", "spam", "jam"]
>>> z = y
>>> x is y, x is z, y is z
(False, False, True)
>>> x is not y, x is not z, y is not z
(True, True, False)
>>> del y[1]
>>> x, y, z
(['ham', 'spam', 'jam'], ['ham', 'jam'], ['ham', 'jam'])
```

UNI FREIBURG

Identität und Gleichheit

Identität vs. Gleichheit

UNI FREIBURG

- Außer Zahlen und Strings können auch Listen und Tupel auf Gleichheit getestet werden. Der Unterschied zum Identitätstest ist wichtig:

Python-Interpreter

```
>>> x = ["ham", "spam", "jam"]
>>> y = ["ham", "spam", "jam"]
>>> x == y, x is y
(True, False)
```

- Test auf *Gleichheit*: haben `x` und `y` den gleichen Typ, sind sie gleich lang und sind korrespondierende Elemente gleich?
  (die Definition ist rekursiv)
- Test auf *Identität*: bezeichnen `x` und `y` dasselbe Objekt?

Faustregel

Verwende in der Regel den Gleichheitstest.

Spezialfall `x is None`

UNI FREIBURG

Anmerkung zu None:

- Der Typ `NoneType` hat nur einen einzigen Wert: `None`. Daher ist es egal, ob ein Vergleich mit `None` per Gleichheit oder per Identität erfolgt.
- Es hat sich eingebürgert, Vergleiche mit `None` immer als
  `x is None` bzw. `x is not None`
  und nicht als
  `x == None` bzw. `x != None`
  zu schreiben.

Veränderlich oder unveränderlich?

Jetzt können wir auch genauer sagen, was es mit veränderlichen (*mutable*) und unveränderlichen (*immutable*) Datentypen auf sich hat:

- Instanzen von veränderlichen Datentypen können modifiziert werden.
  Vorsicht bei Zuweisungen wie `x = y`:
  Nachfolgende Operationen auf `x` beeinflussen auch `y` (und umgekehrt).
  - Beispiel: Listen (**`list`**)
- Instanzen von unveränderlichen Datentypen können nicht modifiziert werden.
  Daher sind Zuweisungen wie `x = y` völlig unkritisch:
  Da das durch `x` bezeichnete Objekt nicht verändert werden kann, besteht keine Gefahr für `y`.
  - Beispiele: Zahlen (**`int`**, **`float`**, **`complex`**), Strings (**`str`**), Tupel (**`tuple`**)

UNI FREIBURG

Datenklassen für Records

Klassen beschreiben Objekte

UNI FREIBURG

- Bisher haben wir vorgefertigte Objekte verwendet,

Klassen beschreiben Objekte

- Bisher haben wir vorgefertigte Objekte verwendet,
- Jetzt beginnen wir selbst welche zu bauen!

Klassen beschreiben Objekte

- Bisher haben wir vorgefertigte Objekte verwendet,
- Jetzt beginnen wir selbst welche zu bauen!
- Dafür benötigen wir einen Bauplan.

Klassen beschreiben Objekte

- Bisher haben wir vorgefertigte Objekte verwendet,
- Jetzt beginnen wir selbst welche zu bauen!
- Dafür benötigen wir einen Bauplan.

Definition

Ein **Record** ist ein Objekt, das mehrere untergeordnete Objekte, die **Attribute**, enthält.

- alternativ: **Struct**; deutsch: Reihung, Struktur
- Objekte heißen auch **Instanzen**.
- Attribute heißen auch **Felder**.

Beispiel für ein Record: Ware

UNI FREIBURG

Beschreibung für Ware

Ein Händler beschreibt eine Ware durch den Namen und den Angebotspreis.

Beispiel für ein Record: Ware

UNI FREIBURG

Beschreibung für Ware

Ein Händler beschreibt eine Ware durch den Namen und den Angebotspreis.

Schritt 1: Bezeichner und Datentypen

Ein Händler beschreibt eine Ware (`Article`) durch die Attribute

- `name : str`, den Namen und
- `price : int`, den Angebotspreis (in cent), immer ≥ 0.

Klassendefinition

Definieren von Records und Klassen

UNI FREIBURG

Python-Interpreter

```
>>> class Article:
...     pass # nur notwendig für leere Klasse!
...
>>> Article
<class '__main__.Article'>
>>> int
<class 'int'>
```

- Neue Records und Klassen werden mit der `class`-Anweisung eingeführt (Konvention: CamelCase-Namen).

Definieren von Records und Klassen

UNI FREIBURG

Python-Interpreter

```
>>> class Article:
...     pass # nur notwendig für leere Klasse!
...
>>> Article
<class '__main__.Article'>
>>> int
<class 'int'>
```

- Neue Records und Klassen werden mit der `class`-Anweisung eingeführt (Konvention: CamelCase-Namen).
- Die `class`-Anweisung muss ausgeführt werden. Sie sollte nicht in einer bedingten Anweisung verborgen werden!

Definieren von Records und Klassen

UNI FREIBURG

Python-Interpreter

```
>>> class Article:
...     pass # nur notwendig für leere Klasse!
...
>>> Article
<class '__main__.Article'>
>>> int
<class 'int'>
```

- Neue Records und Klassen werden mit der `class`-Anweisung eingeführt (Konvention: CamelCase-Namen).
- Die `class`-Anweisung muss ausgeführt werden. Sie sollte nicht in einer bedingten Anweisung verborgen werden!
- Der Klassenname `Article` wird als neuer Type definiert.

UNI FREIBURG

Instanzenerzeugung

Erzeugung von Instanzen

UNI FREIBURG

- Jeder Aufruf der Klasse als Funktion erzeugt ein neue Instanz der Klasse.

Erzeugung von Instanzen

UNI FREIBURG

- Jeder Aufruf der Klasse als Funktion erzeugt ein neue Instanz der Klasse.

Python-Interpreter

```
>>> class Article:
...     pass
...
>>> instance1 = Article()
>>> instance2 = Article()
>>> instance1 is instance2, instance1 == instance2
(False, False)
>>> isinstance(instance1, Article) , isinstance(0, Article)
(True, False)
```

Erzeugung von Instanzen

UNI FREIBURG

- Jeder Aufruf der Klasse als Funktion erzeugt ein neue Instanz der Klasse.

Python-Interpreter

```
>>> class Article:
...     pass
...
>>> instance1 = Article()
>>> instance2 = Article()
>>> instance1 is instance2, instance1 == instance2
(False, False)
>>> isinstance(instance1, Article) , isinstance(0, Article)
(True, False)
```

- Alle erzeugten Instanzen sind untereinander nicht-identisch und ungleich!

Erzeugung von Instanzen

UNI FREIBURG

- Jeder Aufruf der Klasse als Funktion erzeugt ein neue Instanz der Klasse.

Python-Interpreter

```
>>> class Article:
...     pass
...
>>> instance1 = Article()
>>> instance2 = Article()
>>> instance1 is instance2, instance1 == instance2
(False, False)
>>> isinstance(instance1, Article) , isinstance(0, Article)
(True, False)
```

- Alle erzeugten Instanzen sind untereinander nicht-identisch und ungleich!
- `isinstance` prüft ob ein Objekt Instanz einer bestimmten Klasse ist.

Instanzen sind dynamische Strukturen/Records

UNI FREIBURG

Python-Interpreter

```
>>> class Article:
...     pass
...
>>> phone = Article()
>>> phone.name = "Smartphone"
>>> phone.price = 49500
>>> phone.price * 0.19 / 1.19
7903.361344537815
```

- Instanzen können *dynamisch* neue Attribute erhalten.

Instanzen sind dynamische Strukturen/Records

UNI FREIBURG

Python-Interpreter

```
>>> class Article:
...     pass
...
>>> phone = Article()
>>> phone.name = "Smartphone"
>>> phone.price = 49500
>>> phone.price * 0.19 / 1.19
7903.361344537815
```

- Instanzen können *dynamisch* neue Attribute erhalten.
- Jede Instanz hat einen eigenen Namensraum, auf den die Punktnotation zugreift.

Instanzen sind dynamische Strukturen/Records

UNI FREIBURG

Python-Interpreter

```
>>> class Article:
...     pass
...
>>> phone = Article()
>>> phone.name = "Smartphone"
>>> phone.price = 49500
>>> phone.price * 0.19 / 1.19
7903.361344537815
```

- Instanzen können *dynamisch* neue Attribute erhalten.
- Jede Instanz hat einen eigenen Namensraum, auf den die Punktnotation zugreift.
- Besser: gleiche Attribute für alle Instanzen einer Klasse!

Einfache Klasse: Article

UNI FREIBURG

Schritt 2: Klassengerüst

```
from dataclasses import dataclass
@dataclass
class Article:
    name : str
    price : int
```

- Die Klasse `Article` kann nun als Funktion mit zwei Parametern (name, price) aufgerufen werden.
- Alle Instanzen haben die Attribute `name` und `price`.
- Instanzen von Datenklassen sind gleich (==), falls alle Attribute gleich sind.

Beispiel

UNI FREIBURG

Python-Interpreter

```
>>> @dataclass
>>> class Article:
...       name : str
...       price : int
...
>>> phone = Article("Smartphone", 49500)
>>> phone
Article(name='Smartphone', price=49500)
>>> phone.price * 0.19 / 1.19
7903.361344537815
>>> myphone = Article("Smartphone", 49500)
>>> myphone == phone
True
```

Funktionen auf Records

Beispiel

UNI FREIBURG

Angebotspreis

Der Händler will seine Preise am Black Friday um 25% herabsetzen. Der Angebotspreis soll dynamisch nur an der Kasse berechnet werden.

Beispiel

UNI FREIBURG

Angebotspreis

Der Händler will seine Preise am Black Friday um 25% herabsetzen. Der Angebotspreis soll dynamisch nur an der Kasse berechnet werden.

Schritt 1: Bezeichner und Datentypen

Der Händler braucht für die Kasse eine Funktion `sale_price`, die als Parameter

- `article : Article`, die Ware, und
- `discount : int`, den Rabattsatz (in Prozent zwischen 0 und 100)

erwartet und den Verkaufspreis `:int` (in Cent) berechnet.

Beispiel Angebotspreis

UNI FREIBURG

Schritt 2: Funktionsgerüst

```
def sale_price (
        article : Article,
        discount : int) -> int:
    # fill in
    return 0
```

- Neu: im Rumpf können wir die Attribute von `article` über die Punktnotation verwenden.

Beispiel Angebotspreis

UNI FREIBURG

Schritt 3: Beispiele

```
a1 = Article ("Mausefalle", 2000)
a2 = Article ("Promo␣Lutscher", 0)
a3 = Article ("Nougat", 2000)
assert sale_price (a1, 25) == 1500
assert sale_price (a1, 10) == 1800
assert sale_price (a3, 10) == 1800
assert sale_price (a2, 25) == 0
```

Beispiel Angebotspreis

UNI FREIBURG

Schritt 4: Funktionsdefinition

```
def sale_price (
        article : Article,
        discount : int) -> int:
    return article.price * (100 - discount) // 100
```

Beispiel Angebotspreis

UNI FREIBURG

Schritt 4: Funktionsdefinition

```
def sale_price (
        article : Article,
        discount : int) -> int:
    return article.price * (100 - discount) // 100
```

Bemerkung

Die Funktion funktioniert für jedes Objekt mit einem `price` Attribut.

UNI FREIBURG

Geschachtelte Records

Beispiel

UNI FREIBURG

Terminplanung

Ein (Besprechungs-) Termin hat einen Titel, Teilnehmer, eine Anfangszeit und eine Endzeit. Eine (Uhr-) Zeit wird durch Stunde und Minute repräsentiert.

1. Wie lange dauert ein Termin?
2. Stehen zwei Termine in Konflikt?

Beispiel

UNI FREIBURG

Terminplanung

Ein (Besprechungs-) Termin hat einen Titel, Teilnehmer, eine Anfangszeit und eine Endzeit. Eine (Uhr-) Zeit wird durch Stunde und Minute repräsentiert.

1. Wie lange dauert ein Termin?
2. Stehen zwei Termine in Konflikt?

Bemerkungen

- Zwei Datenklassen beteiligt: für Termin und für Zeit
- Frage 2 muss noch präzisiert werden

Terminplanung

UNI FREIBURG

Schritt 1: Bezeichner und Datentypen

Eine Zeit `Time` besteht aus

- einer Stundenzahl `hour : int` zwischen 0 und 23 inklusive.
- einer Minutenzahl `minute : int` zwischen 0 und 59 inklusive.

Ein Termin `Appointment` hat

- einen Titel `title : str`
- (mehrere) Teilnehmer `participants : list[str]`
- eine Anfangszeit `start : Time`
- eine Endzeit `end : Time` nicht vor `start`

Terminplanung

UNI FREIBURG

Schritt 1: Bezeichner und Datentypen

Eine Zeit `Time` besteht aus

- einer Stundenzahl `hour : int` zwischen 0 und 23 inklusive.
- einer Minutenzahl `minute : int` zwischen 0 und 59 inklusive.

Ein Termin `Appointment` hat

- einen Titel `title : str`
- (mehrere) Teilnehmer `participants : list[str]`
- eine Anfangszeit `start : Time`
- eine Endzeit `end : Time` nicht vor `start`

Bemerkung

- Ein `Appointment`-Objekt enthält zwei `Time`-Objekte

Terminplanung

UNI FREIBURG

Schritt 2: Klassengerüst

```
@dataclass
class Time:
    hour   : int # 0 <= hour < 24
    minute : int # 0 <= minute < 60

@dataclass
class Appointment:
    title:str
    participants:list[str]
    start:Time
    end:Time     # less than start
```

Funktionen der Terminplanung

UNI FREIBURG

Schritt 1: Bezeichner und Datentypen

Wie lange dauert ein Termin?
Die Funktion `duration` nimmt einen Termin `app : Appointment` und bestimmt seine Dauer in Minuten (`int`).

Funktionen der Terminplanung

UNI FREIBURG

Schritt 1: Bezeichner und Datentypen

Wie lange dauert ein Termin?
Die Funktion `duration` nimmt einen Termin `app : Appointment` und bestimmt seine Dauer in Minuten (`int`).

Schritt 2: Funktionsgerüst

```
def duration (app : Appointment) -> int:
    # fill in
    return 0
```

Funktionen der Terminplanung

UNI FREIBURG

Schritt 3: Beispiele

```
t1 = Time (12, 50)
t2 = Time (13, 10)
t3 = Time (10, 05)
t4 = Time (12, 45)
m1 = Appointment ("lunch", [], t1, t2)
m2 = Appointment ("lecture", [], t3, t4)
m3 = Appointment ("alarm", [], t4, t4)
assert duration(m1) == 20
assert duration(m2) == 160
assert duration(m3) == 0
```

Funktionen der Terminplanung

UNI FREIBURG

Schritt 4: Funktionsdefinition

```
def duration (app : Appointment) -> int:
    return difference (app.end, app.start)
```

Funktionen der Terminplanung

UNI FREIBURG

Schritt 4: Funktionsdefinition

```
def duration (app : Appointment) -> int:
    return difference (app.end, app.start)
```

Prinzip Wunschdenken

- Zur Erledigung der Aufgabe in `Appointment` benötigen wir eine Operation, die nur mit `Time` zu tun hat.
- Daher lagern wir sie in eine Hilfsfunktion aus!
- **Wunschdenken** heißt, wir geben der gewünschten Funktion einen Namen und erstellen einen Vertrag für sie.
- Dann verwenden wir sie, bevor sie entworfen und implementiert ist.

Differenz von Zeiten

UNI FREIBURG

Schritt 1: Bezeichner und Datentypen

Bestimme die Differenz zweier Zeitangaben.
Die Funktion `difference` nimmt zwei Zeitangaben `t1, t2 : Time` und bestimmt die Differenz `t1 - t2` in Minuten (`int`). Dabei nehmen wir an, dass `t1 >= t2` ist.

Differenz von Zeiten

UNI FREIBURG

Schritt 1: Bezeichner und Datentypen

Bestimme die Differenz zweier Zeitangaben.
Die Funktion `difference` nimmt zwei Zeitangaben `t1, t2 : Time` und bestimmt die Differenz `t1 - t2` in Minuten (`int`). Dabei nehmen wir an, dass `t1 >= t2` ist.

Schritt 2: Funktionsgerüst

```
def difference (t1 : Time, t2 : Time) -> int:
    # fill in
    return 0
```

Differenz von Zeiten

UNI FREIBURG

Schritt 3: Beispiele

```
t1 = Time (12, 50)
t2 = Time (13, 10)
t3 = Time (10, 05)
t4 = Time (12, 45)
assert difference(t2, t1) == 20
assert difference(t4, t3) == 160
assert difference(t1, t1) == 0
```

Differenz von Zeiten

UNI FREIBURG

Schritt 4: Funktionsdefinition

```
def difference (t1 : Time, t2 : Time) -> int:
    return ((t1.hour - t2.hour) * 60
            + t1.minute - t2.minute)
```

Differenz von Zeiten

UNI FREIBURG

Schritt 4: Funktionsdefinition

```
def difference (t1 : Time, t2 : Time) -> int:
    return ((t1.hour - t2.hour) * 60
            + t1.minute - t2.minute)
```

In der Regel

- In Funktionen die Punktnotation nur zum Zugriff auf direkte Attribute verwenden.
- Also nicht tiefer als eine Ebene zugreifen.

Terminkonflikt

UNI FREIBURG

Präzisierung der Fragestellung

Stehen zwei Termine in Konflikt?

Terminkonflikt

UNI FREIBURG

Präzisierung der Fragestellung

Stehen zwei Termine in Konflikt?

- Überschneiden sich zwei Termine zeitlich?

Terminkonflikt

UNI FREIBURG

Präzisierung der Fragestellung

Stehen zwei Termine in Konflikt?

- Überschneiden sich zwei Termine zeitlich?
- Haben zwei Termine gemeinsame Teilnehmer?

Terminkonflikt

UNI FREIBURG

Präzisierung der Fragestellung

Stehen zwei Termine in Konflikt?

- Überschneiden sich zwei Termine zeitlich?
- Haben zwei Termine gemeinsame Teilnehmer?
- Konflikt nur, falls beides zutrifft!

Terminkonflikt

UNI FREIBURG

Präzisierung der Fragestellung

Stehen zwei Termine in Konflikt?

- Überschneiden sich zwei Termine zeitlich?
- Haben zwei Termine gemeinsame Teilnehmer?
- Konflikt nur, falls beides zutrifft!

Schritt 1: Bezeichner und Datentypen

Stehen zwei Termine in Konflikt?
Die Funktion `conflict` nimmt zwei Termine `a1, a2 : Appointment` und stellt fest, ob sie in Konflikt stehen (`bool`).

Terminkonflikt

UNI FREIBURG

Schritt 2: Funktionsgerüst

```
def conflict (a1 : Appointment,
              a2 : Appointment) -> bool:
    # fill in
    return False
```

Terminkonflikt

UNI FREIBURG

Schritt 3: Beispiele

```
t1 = Time (12, 00)
t2 = Time (12, 30)
t3 = Time (10, 05)
t4 = Time (12, 45)
a1 = Appointment ("lunch", ["jim", "jack"], t1, t2)
a2 = Appointment ("lecture", ["jeff", "jim"], t3, t4)
a3 = Appointment ("coffee", ["jack", "jill"], t2, t4)
#
assert conflict(a1, a2) and conflict (a2, a1)
assert not conflict(a1, a3)
assert not conflict(a2, a3)
```

Terminkonflikt

UNI FREIBURG

Schritt 4: Funktionsdefinition

```
def conflict (a1 : Appointment,
              a2 : Appointment) -> bool:
    time_ok = (before(a1.end, a2.start)
               or before(a2.end, a1.start))
    participants_ok = not (
        intersection (a1.participants, a2.participants))
    return not (time_ok and participants_ok)
```

Terminkonflikt

UNI FREIBURG

Wunschdenken

```
def before (t1 : Time, t2 : Time) -> bool:
    ''' check whether t1 is no later than t2 '''
    return False

def intersection (lst1 : list, lst2 : list) -> list:
    ''' return the list of elements both in lst1 and lst2 '''
    return []
```

Terminkonflikt

UNI FREIBURG

Wunschdenken

```
def before (t1 : Time, t2 : Time) -> bool:
    ''' check whether t1 is no later than t2 '''
    return False

def intersection (lst1 : list, lst2 : list) -> list:
    ''' return the list of elements both in lst1 and lst2 '''
    return []
```

Weitere Ausführung selbst

- `before`: Bedingung auf den Attributen von `Time`-Objekten
- `intersection`: `for`-Schleife auf einer der Listen, Akkumulator für das

UNI FREIBURG

Entwurf mit Alternativen

Entwurf mit Alternativen

UNI FREIBURG

Spielkarten

Eine Spielkarte ist entweder

- ein Joker oder
- eine natürliche Karte mit einer Farbe und einem Wert.

Entwurf mit Alternativen

UNI FREIBURG

Spielkarten

Eine Spielkarte ist entweder

- ein Joker oder
- eine natürliche Karte mit einer Farbe und einem Wert.

Schritt 1: Bezeichner und Datentypen

Eine Spielkarte hat eine von zwei Ausprägungen.

- Joker werden durch Objekte der Klasse `Joker` repräsentiert.
- Natürliche Karten durch Objekte der Klasse `Card` mit Attributen `suit` (Farbe) und `rank` (Wert).

Farbe ist ***C**lubs*, ***S**pades*, ***H**earts*, ***D**iamonds*
Wert ist 2, 3, 4, 5, 6, 7, 8, 9, 10, **J**ack, **Q**ueen, **K**ing, **A**ce

Entwurf mit Alternativen

UNI FREIBURG

Schritt 2: Klassengerüst

```
from typing import Union

@dataclass
class Joker:
    pass # no attributes

@dataclass
class Card:
    suit:str       # 'C'lubs, 'S'pades, 'H'earts, 'D'iamonds
    rank:Union[int,str]

AllCards = Union[Joker,Card]
```

Beispiel: Rommé

UNI FREIBURG

Rommé Figuren erkennen

Ein Figur im Rommé ist entweder

- ein Satz (*set*): drei oder vier Karten gleichen Werts in verschiedenen Farben,
- eine Reihe (*run*): drei oder mehr Karten der gleichen Farbe mit aufsteigenden Werten

Eine Karte in einer Figur kann durch einen Joker ersetzt werden. Joker (dürfen nicht nebeneinander liegen und) dürfen nicht in der Überzahl sein.
Erste Aufgabe: Erkenne einen Satz

Beispiel: Rommé

UNI FREIBURG

Rommé Figuren erkennen

Ein Figur im Rommé ist entweder

- ein Satz (*set*): drei oder vier Karten gleichen Werts in verschiedenen Farben,
- eine Reihe (*run*): drei oder mehr Karten der gleichen Farbe mit aufsteigenden Werten

Eine Karte in einer Figur kann durch einen Joker ersetzt werden. Joker (dürfen nicht nebeneinander liegen und) dürfen nicht in der Überzahl sein.
Erste Aufgabe: Erkenne einen Satz

Schritt 1: Bezeichner und Datentypen

Die Funktion `is_set` nimmt als Eingabe eine Liste `cards` von Spielkarten und liefert `True` gdw `cards` ein Satz ist.

Satz erkennen

UNI FREIBURG

Schritt 2: Funktionsgerüst

```
def is_set (cards : list[Union[Card,Joker]]) -> bool:
    # initialization of acc
    for card in cards:
        # action on single card
    # finalization
    return True
```

Satz erkennen

UNI FREIBURG

Schritt 2: Funktionsgerüst

```
def is_set (cards : list[Union[Card,Joker]]) -> bool:
    # initialization of acc
    for card in cards:
        # action on single card
    # finalization
    return True
```

- Liste `cards` verarbeiten: `for` Schleife mit Akkumulator

Satz erkennen

UNI FREIBURG

Schritt 2: Funktionsgerüst

```
def is_set (cards : list[Union[Card,Joker]]) -> bool:
    # initialization of acc
    for card in cards:
        # action on single card
    # finalization
    return True
```

- Liste `cards` verarbeiten: `for` Schleife mit Akkumulator
- Länge der Liste prüfen (drei oder vier)

Satz erkennen

UNI FREIBURG

Schritt 2: Funktionsgerüst

```
def is_set (cards : list[Union[Card,Joker]]) -> bool:
    # initialization of acc
    for card in cards:
        # action on single card
    # finalization
    return True
```

- Liste `cards` verarbeiten: `for` Schleife mit Akkumulator
- Länge der Liste prüfen (drei oder vier)
- Anzahl der Joker prüfen (nicht in der Überzahl)

Satz erkennen

UNI FREIBURG

Schritt 2: Funktionsgerüst

```
def is_set (cards : list[Union[Card,Joker]]) -> bool:
    # initialization of acc
    for card in cards:
        # action on single card
    # finalization
    return True
```

- Liste `cards` verarbeiten: `for` Schleife mit Akkumulator
- Länge der Liste prüfen (drei oder vier)
- Anzahl der Joker prüfen (nicht in der Überzahl)
- auf gleichen Wert prüfen

Satz erkennen

UNI FREIBURG

Schritt 2: Funktionsgerüst

```
def is_set (cards : list[Union[Card,Joker]]) -> bool:
    # initialization of acc
    for card in cards:
        # action on single card
    # finalization
    return True
```

- Liste `cards` verarbeiten: `for` Schleife mit Akkumulator
- Länge der Liste prüfen (drei oder vier)
- Anzahl der Joker prüfen (nicht in der Überzahl)
- auf gleichen Wert prüfen

Satz erkennen

Schritt 3: Beispiele

```
c1 = Card ('C', 'Q')
c2 = Card ('H', 'Q')
c3 = Card ('S', 'Q')
c4 = Card ('D', 'Q')
c5 = Card ('D', 'K')
j1 = Joker ()

assert not is_set ([c1,c2])
assert is_set ([c1, c2, c3])
assert is_set ([c1, c2, j1])
assert is_set ([j1, c2, c3])
assert not is_set ([j1, c5, c4])
assert is_set ([c2, c3, c1, c4])
```

Satz erkennen

Schritt 4: Funktionsdefinition

```
def is_set (cards) -> bool:
    if len (cards) < 3 or len (cards) > 4:
        return False
    rank = None    # common rank
    suits = []     # suits already seen
    nr_jokers = 0
    for card in cards:
        if is_joker (card):
            nr_jokers = nr_jokers + 1
        else:      # a natural card
            if rank and rank != card.rank:
                return False
            else:
                rank = card.rank
            if card.suit in suits:
                return False # repeated suit
            else:
                suits = suits + [card.suit]
    return 2 * nr_jokers <= len (cards)
```

Satz erkennen / Klassentest

UNI FREIBURG

Schritt 4: Funktionsdefinition (Wunschdenken)

```
def is_joker (card : Union[Card,Joker]) -> bool:
    return type(card) is Joker
```

Satz erkennen / Klassentest

UNI FREIBURG

Schritt 4: Funktionsdefinition (Wunschdenken)

```
def is_joker (card : Union[Card,Joker]) -> bool:
    return type(card) is Joker
```

- Klassentest

Satz erkennen / Klassentest

UNI FREIBURG

Schritt 4: Funktionsdefinition (Wunschdenken)

```
def is_joker (card : Union[Card,Joker]) -> bool:
    return type(card) is Joker
```

- Klassentest
- `type(x)` liefert immer das Klassenobjekt zum Wert in `x`

Satz erkennen / Klassentest

UNI FREIBURG

Schritt 4: Funktionsdefinition (Wunschdenken)

```
def is_joker (card : Union[Card,Joker]) -> bool:
    return type(card) is Joker
```

- Klassentest
- `type(x)` liefert immer das Klassenobjekt zum Wert in `x`
- Das Klassenobjekt ist eindeutig, daher kann es mit `is` verglichen werden.

Satz erkennen / Klassentest

UNI FREIBURG

Schritt 4: Funktionsdefinition (Wunschdenken)

```
def is_joker (card : Union[Card,Joker]) -> bool:
    return type(card) is Joker
```

- Klassentest
- `type(x)` liefert immer das Klassenobjekt zum Wert in `x`
- Das Klassenobjekt ist eindeutig, daher kann es mit `is` verglichen werden.
- Verwendung im Gerüst, immer wenn ein Argument zu verschiedenen Klassen gehören kann.

Satz erkennen / Klassentest

UNI FREIBURG

Schritt 4: Funktionsdefinition (Wunschdenken)

```
def is_joker (card : Union[Card,Joker]) -> bool:
    return type(card) is Joker
```

- Klassentest
- `type(x)` liefert immer das Klassenobjekt zum Wert in `x`
- Das Klassenobjekt ist eindeutig, daher kann es mit `is` verglichen werden.
- Verwendung im Gerüst, immer wenn ein Argument zu verschiedenen Klassen gehören kann.
- Alternative: `isinstance(card, Joker)`.

Reihe erkennen

Schritt 1: Bezeichner und Datentypen

Die Funktion `is_run` nimmt als Eingabe eine Liste `cards : list[Union[Card,Joker]]` von Spielkarten und liefert `True` gdw `cards` eine Reihe ist.

Reihe erkennen

UNI FREIBURG

Schritt 2: Funktionsgerüst

```
def is_run (cards : list[Union[Card,Joker]]) -> bool:
    # initialization of acc
    for card in cards:
        # action on single card
    # finalization
    return True
```

Reihe erkennen

UNI FREIBURG

Schritt 2: Funktionsgerüst

```
def is_run (cards : list[Union[Card,Joker]]) -> bool:
    # initialization of acc
    for card in cards:
        # action on single card
    # finalization
    return True
```

- Liste verarbeiten: `for` Schleife mit Akkumulator

Reihe erkennen

UNI FREIBURG

Schritt 2: Funktionsgerüst

```
def is_run (cards : list[Union[Card,Joker]]) -> bool:
    # initialization of acc
    for card in cards:
        # action on single card
    # finalization
    return True
```

- Liste verarbeiten: `for` Schleife mit Akkumulator
- Länge der Liste prüfen

Reihe erkennen

UNI FREIBURG

Schritt 2: Funktionsgerüst

```
def is_run (cards : list[Union[Card,Joker]]) -> bool:
    # initialization of acc
    for card in cards:
        # action on single card
    # finalization
    return True
```

- Liste verarbeiten: `for` Schleife mit Akkumulator
- Länge der Liste prüfen
- Anzahl der Joker prüfen

Reihe erkennen

UNI FREIBURG

Schritt 2: Funktionsgerüst

```
def is_run (cards : list[Union[Card,Joker]]) -> bool:
    # initialization of acc
    for card in cards:
        # action on single card
    # finalization
    return True
```

- Liste verarbeiten: `for` Schleife mit Akkumulator
- Länge der Liste prüfen
- Anzahl der Joker prüfen
- auf gleiche Farbe prüfen

Reihe erkennen

UNI FREIBURG

Schritt 2: Funktionsgerüst

```
def is_run (cards : list[Union[Card,Joker]]) -> bool:
    # initialization of acc
    for card in cards:
        # action on single card
    # finalization
    return True
```

- Liste verarbeiten: `for` Schleife mit Akkumulator
- Länge der Liste prüfen
- Anzahl der Joker prüfen
- auf gleiche Farbe prüfen

Reihe erkennen

Schritt 3: Beispiele

```
cq = Card ('C', 'Q')
ck = Card ('C', 'K')
sa = Card ('S', 'A')
dq = Card ('D', 'Q')
d10 = Card ('D', '10')
jj = Joker ()

assert not is_run ([cq, ck])
assert is_run ([cq, ck, sa])
assert is_run ([dq,ck,sa])
assert is_run ([d10,jj,dq])
assert is_run ([d10,jj,dq,ck])
assert not is_run ([s10, dq, ck])
assert not is_run ([d10, jj, jj])
```

Reihe erkennen

Schritt 3: Funktionsdefinition

UNI FREIBURG


```
def is_run (cards : list[Union[Card,Joker]]) -> bool:
    if len (cards) < 3:      # check length of list
        return False
    else:
        # initialization of acc
        nr_jokers = 0        # count jokers
        current_rank = None # keep track of rank
        suit = None
    for card in cards:
        if current_rank:
            current_rank = next_rank (current_rank)
        # action on single card
        if is_joker (card):
            nr_jokers = nr_jokers + 1
        else:
            if not current_rank:
                current_rank = card.rank
            elif current_rank != card.rank:
                return False
            if not suit:
                suit = card.suit
            elif suit != card.suit:
                return False
    # finalization
    return 2 * nr_jokers <= len (cards)
```

Reihe erkennen

UNI FREIBURG

Was noch fehlt . . .

- Wunschdenken: `next_rank`
- Joker nebeneinander?
- Joker außerhalb der Reihe. . .

Zusammenfassung & Ausblick

Zusammenfassung & Ausblick

- Alle Werte in Python sind Objekte.

Zusammenfassung & Ausblick

UNI FREIBURG

- Alle Werte in Python sind Objekte.
- Veränderliche Objekte besitzen eine Identität.

Zusammenfassung & Ausblick

UNI FREIBURG

- Alle Werte in Python sind Objekte.
- Veränderliche Objekte besitzen eine Identität.
- Eine Klasse beschreibt Objekte/Instanzen.

Zusammenfassung & Ausblick

UNI FREIBURG

- Alle Werte in Python sind Objekte.
- Veränderliche Objekte besitzen eine Identität.
- Eine Klasse beschreibt Objekte/Instanzen.
- Eine Instanz enthält Attribute, d.h. untergeordnete Objekte.

Zusammenfassung & Ausblick

UNI FREIBURG

- Alle Werte in Python sind Objekte.
- Veränderliche Objekte besitzen eine Identität.
- Eine Klasse beschreibt Objekte/Instanzen.
- Eine Instanz enthält Attribute, d.h. untergeordnete Objekte.
- Eine Datenklasse (Record) enthält nichts anderes als Attribute.

Zusammenfassung & Ausblick

UNI FREIBURG

- Alle Werte in Python sind Objekte.
- Veränderliche Objekte besitzen eine Identität.
- Eine Klasse beschreibt Objekte/Instanzen.
- Eine Instanz enthält Attribute, d.h. untergeordnete Objekte.
- Eine Datenklasse (Record) enthält nichts anderes als Attribute.
- Funktionsentwurf mit einfachen Records.

Zusammenfassung & Ausblick

UNI FREIBURG

- Alle Werte in Python sind Objekte.
- Veränderliche Objekte besitzen eine Identität.
- Eine Klasse beschreibt Objekte/Instanzen.
- Eine Instanz enthält Attribute, d.h. untergeordnete Objekte.
- Eine Datenklasse (Record) enthält nichts anderes als Attribute.
- Funktionsentwurf mit einfachen Records.
- Funktionsentwurf mit geschachtelten Records.

Zusammenfassung & Ausblick

UNI FREIBURG

- Alle Werte in Python sind Objekte.
- Veränderliche Objekte besitzen eine Identität.
- Eine Klasse beschreibt Objekte/Instanzen.
- Eine Instanz enthält Attribute, d.h. untergeordnete Objekte.
- Eine Datenklasse (Record) enthält nichts anderes als Attribute.
- Funktionsentwurf mit einfachen Records.
- Funktionsentwurf mit geschachtelten Records.
- Entwurf mit Alternativen.

Zusammenfassung & Ausblick

UNI FREIBURG

- Alle Werte in Python sind Objekte.
- Veränderliche Objekte besitzen eine Identität.
- Eine Klasse beschreibt Objekte/Instanzen.
- Eine Instanz enthält Attribute, d.h. untergeordnete Objekte.
- Eine Datenklasse (Record) enthält nichts anderes als Attribute.
- Funktionsentwurf mit einfachen Records.
- Funktionsentwurf mit geschachtelten Records.
- Entwurf mit Alternativen.
- Der Typtest geschieht durch Identitätstest gegen die Klasse.