

Informatik I: Einführung in die Programmierung

9. Alternativen und Pattern Matching

Albert-Ludwigs-Universität Freiburg



Prof. Dr. Peter Thiemann

21. November 2023

1 Alternativen und Pattern Matching



- Entwurf mit Alternativen
- Pattern Matching
- Aufzählungstypen

Alternativen
und Pattern
Matching

Entwurf mit
Alternativen
Pattern Matching
Aufzählungstypen

Zusammen-
fassung

1 Alternativen und Pattern Matching



- Entwurf mit Alternativen
- Pattern Matching
- Aufzählungstypen

Alternativen
und Pattern
Matching

Entwurf mit
Alternativen

Pattern Matching

Aufzählungstypen

Zusammen-
fassung

Spielkarten

Eine Spielkarte ist (alternativ) 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 werden durch Objekte der Klasse `Card` mit Attributen `suit` (Farbe) und `rank` (Wert) repräsentiert.

Farbe ist eine von **'Clubs', 'Spades', 'Hearts', 'Diamonds'**.

Wert ist einer von 2, 3, 4, 5, 6, 7, 8, 9, 10, **'Jack', 'Queen', 'King', 'Ace'**

Schritt 2: Klassengerüst

```
@dataclass
class Joker:
    pass # no attributes

@dataclass
class Card:
    suit: str          # 'Clubs' | 'Spades' | 'Hearts' | 'Diamonds'
    rank: int | str

type AllCards = Card | Joker
```

- Eine Karte AllCards kann alternativ Card oder Joker sein.
- Das lässt sich ausdrücken durch einen **Union-Typ**: Card | Joker.
- Das Schlüsselwort **type** leitet ein *Typalias* ein.

Figuren in Rommé erkennen

Ein Figur im Rommé ist entweder

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

Eine Karte in einer Figur darf 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_rummy_set` nimmt als Argument eine Liste `cards` von Spielkarten und liefert `True` gdw. `cards` ein Satz ist.

Satz erkennen

Schritt 2: Funktionsgerüst

```
def is_rummy_set (cards : list[AllCards]) -> bool:  
    # initialization of acc  
    for card in cards:  
        pass # action on single card  
    # finalization  
    return ...
```

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

Alternativen
und Pattern
Matching

Entwurf mit
Alternativen
Pattern Matching
Aufzählungstypen

Zusammen-
fassung

Schritt 3: Beispiele

```
c1 = Card ('Clubs', 'Queen')
c2 = Card ('Hearts', 'Queen')
c3 = Card ('Spades', 'Queen')
c4 = Card ('Diamonds', 'Queen')
c5 = Card ('Diamonds', 'King')
j1 = Joker ()
```

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

Alternativen
und Pattern
Matching

Entwurf mit
Alternativen
Pattern Matching
Aufzählungstypen

Zusammen-
fassung

Schritt 4: Funktionsdefinition

```
def is_rummy_set (cards : list[AllCards]) -> bool:
  if len (cards) < 3 or len (cards) > 4:
    return False
  common_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 not common_rank:
        common_rank = card.rank
      elif common_rank != card.rank:
        return False
      if card.suit in suits:
        return False # repeated suit
      else:
        suits = suits + [card.suit]
  return 2 * nr_jokers <= len (cards)
```

Alternativen
und Pattern
Matching

Entwurf mit
Alternativen
Pattern Matching
Aufzählungstypen

Zusammen-
fassung

Schritt 4: Funktionsdefinition (Wunschdenken)

```
def is_joker (card : AllCards) -> bool:  
    return type(card) is Joker
```

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

Alternativen
und Pattern
Matching

Entwurf mit
Alternativen
Pattern Matching
Aufzählungstypen

Zusammen-
fassung

Schritt 1: Bezeichner und Datentypen

Die Funktion `is_rummy_run` nimmt als Argument eine Liste

`cards` : `list`[`AllCards`] von Spielkarten und liefert `True` gdw. `cards` eine Reihe ist.

Alternativen
und Pattern
Matching

Entwurf mit
Alternativen

Pattern Matching

Aufzählungstypen

Zusammen-
fassung

Schritt 2: Funktionsgerüst

```
def is_rummy_run (cards : list[AllCards]) -> bool:  
    # initialization of acc  
    for card in cards:  
        pass # action on single card  
    # finalization  
    return ...
```

- Länge der Liste prüfen
- Liste verarbeiten: for Schleife mit Akkumulator
- Anzahl der Joker prüfen
- Natürliche Karten auf gleiche Farbe prüfen
- Natürliche Karten auf aufsteigende Werte prüfen

Alternativen
und Pattern
Matching

Entwurf mit
Alternativen
Pattern Matching
Aufzählungstypen

Zusammen-
fassung

Schritt 3: Beispiele

```
c2, cq = Card ('Clubs', 2), Card ('Clubs', 'Queen')
ck, ca = Card ('Clubs', 'King'), Card ('Clubs', 'Ace')
dq, d10 = Card ('Diamonds', 'Queen'), Card ('Diamonds', 10)
jj      = Joker ()
```

```
assert not is_rummy_run ([cq, ck])
assert is_rummy_run ([cq, ck, ca])
assert not is_rummy_run ([dq, ck, ca])
assert is_rummy_run ([d10,jj,dq])
assert not is_rummy_run ([d10,jj,dq,ck])
assert not is_rummy_run ([ck, ca, c2])
assert not is_rummy_run ([d10, jj, jj])
```

Alternativen
und Pattern
Matching

Entwurf mit
Alternativen
Pattern Matching
Aufzählungstypen

Zusammen-
fassung

Reihe erkennen

Schritt 3: Funktionsdefinition



```
def is_rummy_run (cards : list[AllCards]) -> bool:
  if len (cards) < 3:      # check length of list
    return False
  # initialization of accumulators
  nr_jokers = 0             # count jokers
  current_rank = None     # keep track of rank
  common_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 common_suit:
        common_suit = card.suit
      elif common_suit != card.suit:
        return False
  # finalization
  return 2 * nr_jokers <= len (cards)
```

Alternativen
und Pattern
Matching

Entwurf mit
Alternativen
Pattern Matching
Aufzählungstypen

Zusammen-
fassung

Was noch fehlt ...

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

Alternativen
und Pattern
Matching

Entwurf mit
Alternativen

Pattern Matching

Aufzählungstypen

Zusammen-
fassung

1 Alternativen und Pattern Matching



- Entwurf mit Alternativen
- Pattern Matching
- Aufzählungstypen

Alternativen
und Pattern
Matching

Entwurf mit
Alternativen

Pattern Matching

Aufzählungstypen

Zusammen-
fassung

- Beim Entwurf mit Alternativen
 - ein Argument kann aus einer von mehreren Klassen stammen
 - erkennbar am Union-Typ
- Funktionsgerüst auf Alternative $T_1 \mid T_2 \mid \dots$
 - erst Typtest auf T_i
 - dann Zugriff auf die Attribute von T_i
 - mögliche Fehlerquellen
 - Verwechslung der Attribute!
 - Fälle übersehen
- Abhilfe: neue Anweisung **Pattern Matching**
 - Gleichzeitiger Test und Zugriff auf die Attribute
 - Funktionen wie `is_joker` werden überflüssig!

Syntax

```
match expr:  
  case pattern1:  
    block1  
  case pattern2:  
    block2  
  . . .
```

- `match` und `case` sind Schlüsselworte.
- `expr` ist ein beliebiger Ausdruck.
- Jedes `pattern` beschreibt einen Test gegen den Wert von `expr`.

Alternativen
und Pattern
Matching

Entwurf mit
Alternativen

Pattern Matching

Aufzählungstypen

Zusammen-
fassung

Beispiel: Kartenwerte in Rommé

Für die Punktabrechnung besitzt jede Spielkarte in Rommé einen Wert.

Rang	Wert in Punkten
Zwei bis Neun	Entsprechend dem Rang (2, 3, 4, 5, 6, 7, 8, 9)
Zehn, Bube, Dame, König	10
Ass	11
Joker	20

Alternativen
und Pattern
Matching

Entwurf mit
Alternativen
Pattern Matching
Aufzählungstypen

Zusammen-
fassung

Wert einer Karte in Rommé

Die Funktion `card_value` nimmt als Argument eine `card`: `AllCards` und liefert als Ergebnis ein `int` entsprechend dem Wert von `card`.

```
assert card_value (Joker()) == 20
assert card_value (Card ('Hearts', 'Ace')) == 11
assert card_value (Card ('Spades', 'Queen')) == 10
assert card_value (Card ('Diamonds', 6)) == 6
```

```
def card_value (card: AllCards) -> int:  
  match card:  
    case Joker():  
      return ...  
    case Card(suit, rank):  
      return ...
```

- Das Pattern `Joker()` passt nur, wenn `card` eine Instanz von `Joker` ist.
- Das Pattern `Card(suit, rank)` passt, wenn `card` eine Instanz von `Card` ist.
 - Im zugehörigen Block sind `suit` und `rank` an die entsprechenden Attribute von `card` gebunden.
 - Die Reihenfolge der Attribute entspricht der Reihenfolge in der Deklaration der Klasse `Card` als `@dataclass`.

Alternativen
und Pattern
Matching

Entwurf mit
Alternativen

Pattern Matching

Aufzählungstypen

Zusammen-
fassung

```
def card_value (card: AllCards) -> int:  
  match card:  
    case Joker():  
      return 20  
    case Card(_, rank):  
      return ...
```

- Das erste Pattern gibt das Ergebnis für Joker vor.
- Die Farbe spielt für die Bestimmung der Kartenwerts keine Rolle. Dort wird das **Wildcard-Pattern** `_` verwendet, das auf jeden beliebigen Wert passt und keine Variablenbindung vornimmt.
- Zur weiteren Analyse des `rank`-Attributes können Pattern **geschachtelt** werden.

Alternativen
und Pattern
Matching

Entwurf mit
Alternativen

Pattern Matching

Aufzählungstypen

Zusammen-
fassung

```
match card:  
  case Card(_, 'Ace'):  
    return 11  
  case Card(_, 'Jack' | 'Queen' | 'King'):  
    return 10  
  case Card(_, int(i)):  
    return i
```

- Das Literal-Pattern `'Ace'` passt nur auf den String `'Ace'`.
- Das Oder-Pattern `'Jack' | 'Queen' | 'King'` passt auf einen von `'Jack'` oder `'Queen'` oder `'King'`.
- Das Pattern `int(i)` passt, falls `card` eine Instanz von `int` ist und bindet die Zahl an `i`.

Schritt 4: Funktionsdefinition

```
def is_rummy_set(cards: list[AllCards]) -> bool:
    if len(cards) < 3 or len(cards) > 4:
        return False
    common_rank = None # common rank
    suits = [] # suits already seen
    nr_jokers = 0
    for card in cards:
        match card:
            case Joker():
                nr_jokers = nr_jokers + 1
            case Card(suit, rank):
                if not common_rank:
                    common_rank = rank
                elif common_rank != rank:
                    return False
                if suit in suits:
                    return False
                suits = suits + [suit]
    return 2 * nr_jokers <= len(cards)
```

Alternativen
und Pattern
Matching

Entwurf mit
Alternativen

Pattern Matching

Aufzählungstypen

Zusammen-
fassung

- Variable:
Das Pattern passt auf jeden Wert und weist ihn der Variable zu.
- Klassenname ($pattern_1, \dots, pattern_n$):
Das Pattern passt, wenn der Wert eine Instanz von Klassenname ist **und** alle Teilpattern $pattern_i$ auf das entsprechende Attribut der Instanz passen. Es dürfen nicht mehr Pattern angegeben werden, als Attribute vorhanden sind. Ansonsten werden die ersten n Attribute geprüft.
- Konstante:
Das Pattern passt, wenn der Wert gleich der Konstante ist.
- weitere Möglichkeiten:
Listen von Patterns, Tupel von Patterns, ... Vgl. Dokumentation.

Semantik

```
match expr :  
  case pattern1 :  
    block1  
  case pattern2 :  
    block2  
  :
```

- Werte zuerst *expr* zu *v* aus.
- Dann prüfe ob *pattern*₁ zu *v* passt; falls ja, führe *block*₁ aus; Variablen im Pattern werden entsprechend *v* zugewiesen; danach nächste Anweisung nach dem `match`.
- Sonst prüfe *pattern*₂ usw bis das erste passende Pattern gefunden wird.
- Nächste Anweisung, falls kein Pattern passt.

Alternativen
und Pattern
Matching

Entwurf mit
Alternativen

Pattern Matching

Aufzählungstypen

Zusammen-
fassung

1 Alternativen und Pattern Matching



- Entwurf mit Alternativen
- Pattern Matching
- Aufzählungstypen

Alternativen
und Pattern
Matching

Entwurf mit
Alternativen

Pattern Matching

Aufzählungstypen

Zusammen-
fassung

- Das Rommé-Beispiel verwendet verschiedene Strings zur Modellierung der Farben und einiger Ränge.
- Das ist fehleranfällig, weil so leicht illegale Karten erzeugt werden können:
`illegal_card = Card ('Ace', 'Ace') ## ??`
- Bei 'Clubs' | 'Spades' | 'Hearts' | 'Diamonds' handelt es sich um eine degenerierte Alternative, wo die einzelnen Alternativen keine weiteren Objekte mit sich führen.
- Dies kann mit einem **Aufzählungstypen** (Enumeration) modelliert werden!

Definition von Farben als Enumeration



```
from enum import Enum
class Suit(Enum):
    CLUBS = 'Clubs'
    SPADES = 'Spades'
    HEARTS = 'Hearts'
    DIAMONDS = 'Diamonds'
```

- Definiert vier Objekte `Suit.CLUBS`, `Suit.SPADES`, `Suit.HEARTS` und `Suit.DIAMONDS`.
- Sie alle sind Instanzen der *Enumeration* `Suit`.
- Es gilt `Suit.CLUBS == Suit('Clubs') == Suit['CLUBS']`.
- Aber `Suit('Ace')` liefert einen Fehler!
- Alle Instanzen einer Enumeration besitzen Attribute `name` und `value` mit `Suit.CLUBS.name == 'CLUBS'` und `Suit.CLUBS.value == 'Clubs'`.

Alternativen
und Pattern
Matching

Entwurf mit
Alternativen
Pattern Matching
Aufzählungstypen

Zusammen-
fassung

```
@dataclass
class Card:
    suit: Suit
    rank: int | str
    ...
    match card:
        # is this the ace of spades?
        case Card (Suit.SPADES, 'Ace'):
            ...
```

- Das Pattern `Suit.SPADES` passt auf den entsprechenden Wert.
- (Erweiterung: Enumeration `Figure` für Ass, König, Dame, Bube ...)

Alternativen
und Pattern
Matching

Entwurf mit
Alternativen
Pattern Matching
Aufzählungstypen

Zusammen-
fassung

Rommé Satz erzeugen

Gegeben einen Rang, erzeuge den Satz zu diesem Rang aus natürlichen Karten.

Gerüst

```
def create_rummy_set (rank: int | str) -> list[Card]:  
    # fill in  
    return ...
```

Implementierung

```
def create_rummy_set (rank: int | str) -> list[Card]:  
    result = []  
    for s in Suit:          # iterate over all elements of Suit  
        result = result + [Card (s, rank)]  
    return result
```

2 Zusammenfassung



Alternativen
und Pattern
Matching

Zusammen-
fassung

- Entwurf mit **Alternativen**.
- Ein **Typtest** kann durch Identitätstest gegen die Klasse, `isinstance` oder Pattern Matching geschehen.
- Pattern Matching vereinigt alle Typtests, die Projektion der Attribute und die Fallunterscheidungen.
- Patterns können geschachtelt werden.
- Aufzählungstypen enthalten eine fest vordefinierte Anzahl von “frischen” Werten.
- Iteration über die Elemente eines Aufzählungstyps.