Informatik I: Einführung in die Programmierung

9. Alternativen und Pattern Matching

Albert-Ludwigs-Universität Freiburg

Prof. Dr. Peter Thiemann

20. November 2024

1 Alternativen und Pattern Matching

- Entwurf mit Alternativen
- Pattern Matching
- Aufzählungstypen

1 Alternativen und Pattern Matching

- Entwurf mit Alternativen
- Pattern Matching
- Aufzählungstypen

Entwurf mit Alternativen

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 `FaceCard` 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'`

Typen für Farben und Werte

Farben

```
from typing import Literal
type Suit = Literal['Clubs','Spades','Hearts','Diamonds']
```

Der Typ `Suit` enthält genau die aufgelisteten Strings.

Werte

```
type Rank = ( Literal[2,3,4,5,6,7,8,9,10]
            | Literal['Ace','King','Queen','Jack'])
```

Der Typ `Rank` enthält genau die aufgelisteten Werte, eine Mischung aus **`int`** und **`str`**.

Entwurf mit Alternativen

Schritt 2: Klassengerüst

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

@dataclass
class FaceCard:
    suit: Suit
    rank: Rank

type Card = FaceCard | Joker
```

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

Beispiel: Rommé

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[Card]) -> 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

Satz erkennen

Schritt 3: Beispiele

```
c1 = FaceCard ('Clubs', 'Queen')
c2 = FaceCard ('Hearts', 'Queen')
c3 = FaceCard ('Spades', 'Queen')
c4 = FaceCard ('Diamonds', 'Queen')
c5 = FaceCard ('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])
```

Satz erkennen

Schritt 4: Funktionsdefinition

```
def is_rummy_set (cards : list[Card]) -> 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)
```

Satz erkennen / is_joker

Schritt 4: Funktionsdefinition (Wunschdenken)

```
def is_joker (card : Card) -> bool:
    match card:
        case Joker():
            return True
        case FaceCard():
            return False
```

- Verwendet Pattern Matching!
- Die Funktion `is_joker` ist eigentlich überflüssig, weil Pattern Matching direkt anstelle der `if`-Anweisung verwendet werden kann.

1 Alternativen und Pattern Matching

- Entwurf mit Alternativen
- Pattern Matching
- Aufzählungstypen

Pattern Matching

- Beim Entwurf mit Alternativen
  - ein Argument kann aus einer von mehreren Klassen stammen
  - erkennbar am Union-Typ
- Funktionsgerüst für Alternative auf `x : T1 | T2 | ...`
  - Pattern Matching gegen `x`: `match x:`
  - Ein Fall für jedes `T_i`: `case T_():`
  - Im Rumpf des `case` Zugriff auf die Attribute von `T_i`
  - mögliche Fehlerquelle: Verwechselung der Attribute!
- Noch besser:
  - Gleichzeitiger Test und Zugriff auf die Attribute
  - Schreibe die Fälle als `case T_i(a1, ..., a_n):`
  - die `a_i` sind Variable für die Attribute sind.

Pattern Matching

Syntax

```
match expr:
    case pattern_1:
        block_1
    case pattern_2:
        block_2
    ...
```

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

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

Wert einer Karte in Rommé

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

Beispiele


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

Gerüst

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

- Das Pattern `Joker()` passt nur, wenn `card` eine Instanz von `Joker` ist.
- Das Pattern `FaceCard(suit, rank)` passt, wenn `card` eine Instanz von `FaceCard` 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 `FaceCard` als `@dataclass`.

Gerüst + erstes Beispiel


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

Gerüst + restliche Beispiele

```
match card:
    case FaceCard(_, 'Ace'):
        return 11
    case FaceCard(_, 'Jack' | 'Queen' | 'King'):
        return 10
    case FaceCard(_, 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`.

Beispiel: Rommé Satz erkennen

Schritt 4: Funktionsdefinition

```
def is_rummy_set(cards: list[Card]) -> 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 FaceCard(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)
```

Patterns passen auf Werte

- Variable:
  Das Pattern passt auf jeden Wert und weist ihn der Variable zu.
- `Klassenname` (*pattern*$_1$`,` `...`, *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.

Pattern Matching

Semantik

```
match expr:
    case pattern_1:
        block_1
    case pattern_2:
        block_2
    :
```

- 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.

Reihe erkennen

Schritt 1: Bezeichner und Datentypen

Die Funktion `is_rummy_run` nimmt als Argument eine Liste `cards : list[Card]` von Spielkarten und liefert `True` gdw. `cards` eine Reihe ist.

Reihe erkennen

Schritt 2: Funktionsgerüst

```
def is_rummy_run (cards : list[Card]) -> 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

Reihe erkennen

Schritt 3: Beispiele

```
c2, cq  = FaceCard ('Clubs', 2), FaceCard ('Clubs', 'Queen')
ck, ca  = FaceCard ('Clubs', 'King'), FaceCard ('Clubs', 'Ace')
dq, d10 = FaceCard ('Diamonds', 'Queen'), FaceCard ('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])
```

Reihe erkennen

Schritt 3: Funktionsdefinition

```python
def is_rummy_run (cards : list[Card]) -> 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
        match card:
            case Joker():
                nr_jokers = nr_jokers + 1
            case FaceCard(suit, rank):
                if not current_rank:
                    current_rank = rank
                elif current_rank != rank:
                    return False
                if not common_suit:
                    common_suit = suit
                elif common_suit != suit:
                    return False
    # finalization
    return 2 * nr_jokers <= len (cards)
```

Reihe erkennen

Was noch fehlt . . .

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

1 Alternativen und Pattern Matching

- Entwurf mit Alternativen
- Pattern Matching
- Aufzählungstypen

Aufzählungstypen (Enumerations)

- 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 = FaceCard ('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'`.

Enumerations in Patterns


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

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

Iteration über Enumeration

Rommé Satz erzeugen

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

Gerüst

```
def create_rummy_set (rank: Rank) -> list[FaceCard]:
    # fill in
    return ...
```

Implementierung

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

2 Zusammenfassung

Zusammenfassung

- 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.