Informatik 1

Einführung in die Programmierung

SS 2022

Prof. Dr. Peter Thiemann
Institut für Informatik
Albert-Ludwigs-Universität Freiburg

- Für die Bearbeitung der Aufgaben haben Sie **120 Minuten** Zeit.
- Es sind **keine Hilfsmittel** wie Skripte, Bücher, Notizen oder Taschenrechner erlaubt. Desweiteren sind alle elektronischen Geräte (wie z.B. Handys) auszuschalten. **Ausnahme: Fremdsprachige Wörterbücher** sind erlaubt.
- Falls Sie **mehrere Lösungsansätze** einer Aufgabe erarbeiten, markieren Sie deutlich, welcher gewertet werden soll. Die "Zielfunktion" darf nur einmal in der Abgabe definiert werden, alles andere muss auskommentiert oder gelöscht werden.
- Verwenden Sie **Typannotationen** um die Rückgabe- und Parameter-Typen Ihrer Funktion anzugeben. Fehlende Typannotationen führen zu Punktabzug.
- Bearbeiten Sie die einzelnen Aufgaben in den vorgegebenen **Templates**, z.B. `ex1_sequences.py`. **Falsch benannte Funktionen** werden nicht bewertet. **Neu erstellte Dateien** werden nicht bewertet.
- Bei Bedarf können Sie **weitere Module** aus der Standardbibliothek importieren. Zum Lösen der Aufgaben ist dies aber nicht notwendig.

	Erreichbare Punkte	**Erzielte Punkte**	**Nicht bearbeitet**
Aufgabe 1	30		
Aufgabe 2	10		
Aufgabe 3	30		
Aufgabe 4	5		
Aufgabe 5	15		
Aufgabe 6	15		
Aufgabe 7	15		
Gesamt	120		

Aufgabe 1 (Sequenzen; Punkte: 30).

(a) (6 Punkte) Schreiben Sie eine Funktion `initials`, die einen Text als Argument nimmt und die Initialen der enthaltenen Worte zurückgibt. Worte werde durch ein oder mehrere Leerzeichen voneinander getrennt.

Beispiel:

```
>>> initials('Hello to the World!')
'HttW'
```

(b) (7 Punkte)

Schreiben Sie eine Funktion `balanced_stars`, die einen Text als Argument nimmt und zurückgibt, ob der Text eine gerade Anzahl an `'*'`-Zeichen enthält.

Beispiel:

```
>>> balanced_stars('Hello to the *World*!')
True
>>> balanced_stars('Hello *to the *World*!')
False
```

(c) (7 Punkte) In der englischen Sprache können viele Adjektive zu ihrem zugehörigen Substantiv umgewandelt werden, indem man das Suffix `'ness'` anhängt. Zum Beispiel wird so aus dem Adjektiv `'good'` das Substantiv `'goodness'`.

Ein Sonderfall besteht jedoch, wenn das Adjektiv mit einem `'y'` endet. Hier muss zusätzlich das `'y'` durch ein `'i'` ersetzt werden. Zum Beispiel wird so aus dem Adjektiv `'happy'` das Substantiv `'happiness'`.

Schreiben Sie eine Funktion `remove_ness`, die diesen Prozess umkehrt und den Sonderfall mit `'y'` berücksichtigt. Sie können annehmen, dass es keine weiteren Sonderfälle gibt und die Funktion ausschließlich auf einem einzelnen Wort aufgerufen wird. Endet das Wort nicht mit `'ness'`, so soll das Wort einfach unverändert zurückgegeben werden.

Beispiele:

```
>>> remove_ness('goodness')
'good'
>>> remove_ness('happiness')
'happy'
>>> remove_ness('cow')
'cow'
```

Hinweis: Es empfiehlt sich in dieser Aufgabe ausgiebig *Slicing* zu verwenden.

(d) (10 Punkte) Ein *Klammerpaar* besteht aus einem Symbol für eine öffnende Klammer, z.B. `'('`, und einem Symbol für eine zugehörige schließende Klammer, z.B. `')'`.

Ein Text ist *klammer-korrekt* bezüglich einer Liste von Klammerpaaren, wenn jede öffnende Klammer durch ihre zugehörige schließende Klammer geschlossen wird.

Schreiben Sie eine Funktion `is_bracket_correct`, die einen Text als Argument nimmt und zurückgibt, ob der Text klammer-korrekt bezüglich den Klammerpaaren `('(', ')')` und `('{', '}')` ist.

Beispiele:

```
>>> is_bracket_correct('abc(x)(y)')
True
>>> is_bracket_correct('({}())')
True
>>> is_bracket_correct('({}()')
False  # Das erste '(' wurde nicht geschlossen.
>>> is_bracket_correct('({)}')
False  # Das erste '{' wurde durch ein ')' geschlossen
```

Hinweis: Es empfiehlt sich während der Iteration eine Liste zu verwenden, in der man sich die noch zu schließenden Klammer-Symbole merkt.

Aufgabe 2 (Dictionaries und Sets; Punkte: 10).

Schreiben Sie eine Funktion `words_by_length`, die einen Text als Argument nimmt und ein Dictionary zurückgibt, welches Zahlen `n` auf die Menge der Wörter mit Länge `n` abbildet.
Das Dictionary darf nur Einträge für Wortlängen enthalten, für die es mindestens ein zugehöriges Wort im Text gibt.
Beispiel:

```
>>> words_by_length("das ja genau das will ich")
{ 2: {"ja"}, 3: {"das", "ich"}, 4: {"will"}, 5: {"genau"} }
```

Aufgabe 3 (Dataclasses; Punkte: 30).

Ein Punkt (`Point`) besteht aus zwei ganzzahligen Koordinaten `x` und `y`. Wir betrachten nun zwei Arten von 2-dimensionalen geometrischen Formen (`Shape`):

1. Ein Rechteck (`Rectangle`) besteht aus zwei Punkten (`p_min`, `p_max`). Es gilt die Invariante, dass die Koordinaten von `p_min` jeweils kleiner oder gleich wie die entsprechenden Koordinaten von `p_max` sein müssen.
2. Ein Dreieck (`Triangle`) besteht aus den Eckpunkten des Dreiecks (`p1`, `p2`, `p3`).

Die *Bounding Box* einer geometrischen Form, ist das kleinst-möglichste Rechteck, das die geometrische Form vollständig enthält.
Jede geometrische Form hat eine Bounding Box, weshalb wir `Shape` wie folgt definieren:

```
@dataclass
class Shape:
    # Calculate the bounding box of the shape.
    def bbox(self) -> 'Rectangle':
        raise Exception("bbox needs to be overriden by subclasses of Shape!")
```

Ihre Aufgaben sind:

(a) (7 Punkte) Implementieren Sie die Datenklasse `Point`. Die Klasse hat zwei Methoden `min` und `max`, die das komponentenweise Minimum bzw. Maximum zwischen zwei Punkten berechnen. Beispiel:

```
>>> p1 = Point(1, 100)
>>> p2 = Point(2, 20)
>>> p1.min(p2)
Point(1, 20)
>>> p1.max(p2)
Point(2, 100)
```

(b) (8 Punkte) Implementieren Sie die Datenklasse `Rectangle` als Subklasse von `Shape`. Stellen Sie sicher, dass die Invariante beim Erstellen eines Rechtecks eingehalten wird (`assert`). Die Bounding Box eines Rechtecks ist das Rechteck selbst. Die Klasse hat zusätzlich eine Methode `union`, die für zwei Rechtecke das kleinst-möglichste Rechteck zurückgibt, welches beide Rechtecke enthält.

```
>>> r1 = Rectangle(Point(1, 1), Point(3, 3))
>>> r2 = Rectangle(Point(10, 10), Point(30, 30))
>>> r1.union(r2)
Rectangle(Point(1, 1), Point(30, 30))
```

Hinweis: es bietet sich an bei `union` die `min`- und `max`-Methoden von `Point` zu verwenden.

(c) (7 Punkte) Implementieren Sie die Datenklasse `Triangle` als Subklasse von `Shape`. Die Bounding Box eines Dreiecks ergibt sich aus dem Minimum und Maximum der Eckpunkte.

Beispiel:

```
>>> t = Triangle(Point(0,0), Point(1,0), Point(0,1))
>>> t.bbox()
Rectangle(Point(0, 0), Point(1, 1))
```

(d) (8 Punkte) Schreiben Sie eine Funktion `bboxes`, die eine Liste von geometrischen Formen als Argument nimmt und eine Bounding Box zurückgibt, die alle geometrischen Formen aus der Liste enthält.

Verwenden Sie hierzu die `bbox`-Methode.

Ist die Liste leer, so soll `None` zurückgegeben werden.

Beispiel:

```
>>> r = Rectangle(Point(2,2), Point(4,8))
>>> t = Triangle(Point(0,0), Point(1,0), Point(0,1))
>>> bboxes([t, r])
Rectangle(Point(0, 0), Point(4, 8))
```

Aufgabe 4 (Tests; Punkte: 5).

Schreiben Sie `pytest`-kompatible Unittests für die folgende Funktion. Dabei soll jede `return`-Anweisung durch genau eine Testfunktion abgedeckt werden.

```
def leapyear(year: int) -> bool:
    if year % 4 == 0:
        if year % 100 != 0:
            return True
        elif year % 100 == 0 and year % 400 == 0:
            return True
        else:
            return False
    else:
        return False
```

Aufgabe 5 (Generatoren; Punkte: 15).

Der Import von zusätzlichen Modulen ist in dieser Aufgabe **verboten**.
Vermeiden Sie unnötigen Speicherverbrauch: Funktionen, die iterierbare Objekte (Iterables) als Argument nehmen, dürfen diese nicht unnötigerweise in eine Liste umwandeln.

(a) (5 Punkte) Schreiben Sie eine Generator-Funktion `without`, die ein iterierbares Objekt `xs` und eine Liste `ys` als Argumente nimmt und diejenigen Elemente aus `xs` generiert, die nicht in `ys` enthalten sind.

Beispiele:

```
assert list(without(range(0, 6), []))     == [0, 1, 2, 3, 4, 5]
assert list(without(range(0, 6), [1, 3])) == [0,    2,    4, 5]
```

(b) (10 Punkte) Schreiben Sie eine Generator-Funktion `my_split`, die einen Text als Argument nimmt und die Wörter des Texts generiert. Wörter sind hierbei durch ein oder mehrere Leerzeichen voneinander getrennt. Die `str.split()`-Methode darf hierbei *nicht* verwendet werden.

Beispiele:

```
assert list(my_split('')) == []

s1 = 'this is a sentence'
assert list(my_split(s1)) == ['this', 'is', 'a', 'sentence']

s2 = '    this   is  a     sentence   '
assert list(my_split(s2)) == ['this', 'is', 'a', 'sentence']
```

Aufgabe 6 (Funktionale Programmierung; Punkte: 15).

In den folgenden Teilaufgaben müssen Sie *keine* Typannotationen angeben.
Implementieren Sie die Funktionen aus folgenden Teilaufgaben im funktionalen Stil - also entweder mit einem Rumpf, der aus genau einer `return`-Anweisung besteht, oder durch ein Lambda, das einer Variable zugewiesen wird.

(a) (5 Punkte) Schreiben Sie eine Funktion `same`, die zwei einstellige Funktionen `f` und `g` als Argument nimmt und eine Funktion zurückgibt, die ein Argument `x` nimmt und überprüft ob `f` und `g` für `x` den gleichen Wert zurückgeben.

Beispiel:

```
>>> f = same(lambda x: x*2+1, lambda x: x*3)
>>> f(0)
False
>>> f(1)
True
>>> f(2)
False
```

(b) (5 Punkte) Schreiben sie eine Funktion `check_fun`, die eine einstellige Funktion `f` und eine Liste von Paaren `ps` als Argument nimmt und eine Liste zurückgibt.

Die Funktion `check_fun` soll dabei für jedes Paar `(x, y)` aus `ps` überprüfen ob `f(x) == y` gilt und wenn dies nicht der Fall ist, das Tripel `(x, f(x), y)` in die zurückzugebende Liste aufnehmen.

Verwenden Sie hierfür genau eine List-Comprehension.

Beispiel:

```
>>> f = lambda x: x * 2
>>> ps = [(1, 2), (2, 40), (3, 60), (4, 8)]
>>> check_fun(f, ps)
[(2, 4, 40), (3, 6, 60)]
```

(c) (5 Punkte) Die folgende Funktion nimmt zwei Strings `xs` und `ys` als Argumente und gibt eine Liste der möglichen Kombinationen der Zeichen von `xs` und `ys` zurück:

```
def combinations(xs: str, ys: str) -> list[str]:
    return [ [ x + y for y in ys ] for x in xs ]
```

Beispiel:

```
>>> nested_combinations('abc', '12')
[['a1', 'a2'], ['b1', 'b2'], ['c1', 'c2']]
```

Schreiben Sie eine Funktion `combinations2`, die sich wie die Funktion `combinations` verhält, aber verwenden Sie bei der Implementierung statt List-Comprehensions mehrere Aufrufe der `map`- und `list`-Funktionen und einen `lambda`-Ausdruck.

Aufgabe 7 (Rekursion; Punkte: 15).

In dieser Aufgabe befassen wir uns mit einer (vereinfachten) Baumstruktur für die Markup-Sprache *HTML* und wie man diese in Text umwandelt.
Ein HTML-Knoten ist entweder:

- ein Text-Knoten, der aus einem String besteht; oder
- ein Element-Knoten, der aus einem Namen und einer Liste von HTML-Knoten besteht.

Die zugehörige Python-Definition ist wie folgt:

```
@dataclass
class TextNode:
    text: str

@dataclass
class ElemNode:
    name: str
    children: list['Node']

Node = TextNode | ElemNode
```

Ihre Aufgaben sind:

(a) (10 Punkte) Schreiben Sie eine Funktion `node_to_str`, die einen `Node` als Argument nimmt und den zugehörigen HTML-Code als Text zurückgibt.

Beispiele:

```
>>> node_to_str(TextNode('important'))
'important'

>>> node_to_str(ElemNode('b', []))
'<b></b>'

>>> node_to_str(ElemNode('b', [TextNode('important')]))
'<b>important</b>'

>>> node_to_str(
        ElemNode('p', [
            TextNode('It is very '),
            ElemNode('b', [
                TextNode('important'),
            ]),
            TextNode(' to pay attention.'),
        ])
    )
>>> node_to_str(node)
'<p>It is very <b>important</b> to pay attention</p>'
```

(b) (5 Punkte) Wie man in den Beispielen aus dem vorherigen Aufgabenteil sieht, ist es sehr unleserlich auf diese Weise Bäume zu beschreiben.

Schreiben Sie deshalb zwei Hilfsfunktionen `p` und `b`, die es erlauben

```
p('It is very ', b('important'), ' to pay attention.')
```

zu schreiben und dabei den selben Baum zu erhalten wie

```
ElemNode('p', [
    TextNode('It is very '),
    ElemNode('b', [
        TextNode('important'),
    ]),
    TextNode(' to pay attention.'),
])
```

Hierbei muss beachtet werden, dass `p` und `b` eine beliebige Anzahl an Argumenten nehmen können, da in einem `ElemNode` auch eine beliebige Anzahl an Kind-Knoten erlaubt sind. Beispiele:

```
assert p()             == ElemNode('p', [])
assert p('foo')        == ElemNode('p', [TextNode('foo')])
assert p('foo', 'bar') == ElemNode('p', [TextNode('foo'), TextNode('bar')])
```