Aufgabe 1 (Sequence; Punkte: 10).

In dieser Aufgabe soll ermittelt werden wie oft ein Blatt Papier gefaltet werden kann. Wir betrachten hier ein idealisiertes Blatt, dessen Breite und Höhe nur ganzzahlige Werte größer null annehmen. Das Falten wird mit einer Ganzzahldivision durch zwei simuliert. Außerdem soll immer nur die längere von beiden Seiten gefaltet werden. Schreiben Sie eine Funktion `count_folds`, welche die Breite `width` und Höhe `height` von einem Blatt Papier als Argumente nimmt, und berechnet wie oft man das Papier falten kann bis eine der Seiten die Länge null erreicht hat.


```
>>> count_folds(2, 1)
2
>>> count_folds(15, 7)
6
```

Aufgabe 2 (Dict; Punkte: 10).

Das folgende Dictionary weist Mahlzeiten Ihre Zutatenliste zu:


```
>>> recipes = { 'Pizza Margherita': ['Mehl', 'Hefe', 'Tomaten',
                'Pasta Napoli':     ['Penne', 'Tomaten'],
                'Nudelauflauf':     ['Penne', 'Käse', 'Eier', 'Sahne'] }
```

Schreiben Sie eine Funktion `cluster_by_ingredience`, die solch ein Dictionary `recipes` als Argument nimmt und ein Dictionary zurückgibt, welches jeder Zutat diejenigen Mahlzeiten zuordnet, die die Zutat enthalten.
Beispiel:

```
>>> cluster_by_ingredience(recipes)
{ 'Mehl':    ['Pizza Margherita'],
  'Hefe':    ['Pizza Margherita'],
  'Tomaten': ['Pizza Margherita', 'Pasta Napoli'],
  'Käse':    ['Pizza Margherita', 'Nudelauflauf'],
  'Penne':   ['Pasta Napoli', 'Nudelauflauf'],
  'Eier':    ['Nudelauflauf'],
  'Sahne':   ['Nudelauflauf'] }
```

Aufgabe 3 (Strings; Punkte: 20).

Sehr große und sehr kleine Zahlen werden häufig in *wissenschaftlicher Notation* geschrieben. Zahlen in dieser Notation haben die Form `xey`, wobei x eine Fließkommazahl zwischen 0.0 und 10.0 und y eine beliebige ganze Zahl sein kann. Dabei kodiert `xey` die Zahl $x \cdot 10^y$. Beispiele:

```
>>> 3.0e5
300000.0
>>> 2.1e-3
0.0021
>>> -7.0e13
-70000000000000.0
```

Schreiben Sie eine Funktion `is_scientific`, welche einen String `s` als Argument nimmt und `True` zurückgibt, wenn `s` eine Zahl in wissenschaftlicher Notation repräsentiert. Andernfalls soll `False` zurück gegeben werden. Beachten Sie bei ihrer Implementierung, dass in den Teilstrings `x` und `y` Vorzeichen (`+`, `-`) erlaubt sind.
Beispiel:

```
>>> is_scientific("3.0e5")
True
>>> is_scientific("3.0")
False
```

Hinweis: Die Verwendung von `isdigit` wird empfohlen.

Aufgabe 4 (Dataclasses; Punkte: 20).

(a) (5 Punkte) In einem Computerspiel sollen Roboter im Kampf gegeneinander antreten. Modellieren Sie dafür die Datenklasse `Robot` mit folgenden Attributen:

- Lebenspunkte: `hp: int`
- Rüstung: `armor: int`
- Angriffspunkte: `attack: int`
- Typ: `robot_typ: int`

(b) (5 Punkte) Es soll drei Typen von Robotern geben: `1`, `2` und `3`. Es ist daher ungültig wenn das Attribut `robot_typ` andere Werte annimmt. Garantieren Sie die Einhaltung dieser Invariante beim Erzeugen einer neuen `Robot`-Instanz durch ein `assert`-Statement.

(c) (5 Punkte) Nun kommt es zum Kampf zwischen 2 Robotern. Schreiben Sie eine Methode `hit_damage`, welche den Schaden eines Roboters `self` auf einen anderen Roboter `other` berechnet. Die Berechnung läuft folgendermaßen:

- Der Schaden berechnet sich Grundlegend aus `self.attack - other.armor`
- Roboter vom Typ 1 machen einen Extraschaden von `+1`, wenn sie gegen einen Roboter vom Typ 2 kämpfen. Kämpfen sie gegen einen Roboter vom Typ 3 sind es sogar `+2` Extraschaden.
- Roboter vom Typ 2 haben eine um `+2` erhöhte Rüstung wenn sie gegen Roboter vom Typ 1 kämpfen
- Jeder Schlag muss, unabhängig von den vorherigen Bedingungen, mindestens einen Schaden von 1 machen.

(d) (5 Punkte) Implementieren Sie die Vergleichsmethode `>=` (`__ge__`) um zwei Roboter zu vergleichen. Es soll `r1 >= r2` gelten, wenn `r1` und `r2` beides Roboter sind und `r1` mindestens so stark ist wie `r2`. Dies ist der Fall, wenn `r1` in einem Kampf `r2` besiegen würde - auch wenn `r1` dabei selbst vernichtet werden würde.

Ein Kampf läuft wie folgt ab: Beide Roboter stehen sich gegenüber und schlagen sich gleichzeitig. Die Lebenspunkte der beiden verringern sich dabei jeweils um den Wert der mit `hit_damage` berechnet wird. Dies wird solange wiederholt bis mindestens einer der beiden Roboter vernichtet wurde.

```
>>> r1 = Robot(10, 1, 5, 1)
>>> r2 = Robot(16, 2, 3, 2)
>>> r1 >= r2
False
>>> r1 >= Robot(10, 1, 5, 1)
True
```

Aufgabe 5 (Tests; Punkte: 10).

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

```
def letter_occurence(input: str) -> str:
    chars = dict()
    for c in input:
        if not c.isupper() and not c.islower():
            return "error"
        if c in chars:
            chars[c] += 1
        else:
            chars[c] = 1

    maximum_doubling = max(chars.values())
    if maximum_doubling <= 1:
        return "einfach"
    elif maximum_doubling <= 2:
        return "doppelt"
    else:
        return "mehrfach"
```

Aufgabe 6 (Rekursion; Punkte: 15).

Im Folgenden betrachten wir binäre Bäume, die wie in der Vorlesung über eine Datenklasse `Node` implementiert sind.

```
@dataclass
class Node:
    mark: str
    left: Optional['Node']
    right: Optional['Node']
```

Schreiben Sie eine Funktion `layer`, die einen natürliche Zahl `n` und einen Baum `node` als Argument nimmt und eine Liste aller Markierungen zurückgibt, deren Knoten die Tiefe `n` haben.
Beispiel:

```
>>> tree = Node("0", Node("1a", None, Node("2", None, None)),
...                  Node("1b", None, None))
>>> layer(0, tree)
['0']
>>> layer(1, tree)
['1a', '1b']
>>> layer(2, tree)
['2']
```

In der zurückgegebenen Markierungsliste sollen erst die Markierungen aus den linken Teilbäumen und dann die aus den rechten Teilbäumen enthalten sein - wie im Beispiel.

Aufgabe 7 (Generatoren; Punkte: 20).

Verwenden Sie in den folgenden Teilaufgaben *keine* von Python bereitgestellten Generator-Funktionen *außer* `range`. Die Funktionen `map`, `filter` und `enumerate` sind also z.B. verboten.
Vermeiden Sie unnötigen Speicherverbrauch: Funktionen die Generatoren als Argument nehmen dürfen diese nicht unnötigerweise in eine Liste umwandeln.
In den folgenden Teilaufgaben müssen Sie *keine* Typannotationen angeben.

(a) (5 Punkte) Schreiben Sie eine Generator-Funktion `accumulate`, die eine Generator `xs` von ganzen Zahlen als Argument nimmt und die Summen über alle Anfangsstücke von `xs` generiert.

Beispiel:

```
>>> list(accumulate(iter([1, 2, 3, 4, 5, 6])))
[1, 3, 6, 10, 15, 21]
```

(b) (5 Punkte) Schreiben Sie eine Generator-Funktion `my_map`, die eine Funktion `f` und einen Generator `xs` als Argumente nimmt und sich wie `xs` verhält, aber auf die generierten Elemente zusätzlich `f` anwendet.

Beispiel:

```
def double(n: int) -> int:
    return n * 2

>>> list(my_map(double, iter([1, 2, 3, 4])))
[2, 4, 6, 8]
```

(c) (10 Punkte) Schreiben Sie eine Generator-Funktion `init`, die einen Generator `xs` als Argument nimmt und sich wie `xs` verhält, aber das letzte Element weglässt. Erzeugt `xs` überhaupt keine Elemente, so soll auch `init(xs)` keine Elemente erzeugen.

```
>>> list(init(iter([1, 2, 3, 4])))
[1, 2, 3]
>>> list(init(iter([])))
[]
```

Aufgabe 8 (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 `paired`, die zwei einstellige Funktionen `f` und `g` als Argument nimmt und eine zweistellige Funktion zurückgibt, die `f` und `g` auf jeweils einem der Argumente anwendet und ein Paar der Ergebnisse zurückgibt.

Beispiel:

```
>>> paired(lambda x: x*2, lambda x: x*3)(5, 10)
(10, 30)
```

(b) (10 Punkte) Schreiben Sie eine Funktion `is_prime`, die eine positive ganze Zahl `n` als Argument nimmt und zurückgibt ob `n` prim ist.

Verwenden Sie hierfür eine Generator-Comprehension und die `any`-Funktion.

Beispiel:

```
>>> is_prime(0), is_prime(4), is_prime(5), is_prime(13)
    False,       False,       True,        True
```

Hinweis: Die `any`-Funktion nimmt ein iterierbares Objekt von `bool`schen Werten als Argument und gibt zurück ob mindestens einer dieser Werte `True` ist.

Hinweis: Eine ganze Zahl `n` heißt prim wenn sie größer 1 ist und außer 1 und `n` keine Teiler hat.