## Aufgabe 1: count_iterations (10)

Gegeben ist die Funktion f auf ganzen Zahlen mit
$f(x) = x + 4$; für x teilbar durch 3
$f(x) = \frac{x}{2}$; für x nicht teilbar durch 3, aber teilbar durch 4
$f(x) = x - 1$; für x nicht teilbar durch 3 und nicht teilbar durch 4

Schreiben Sie eine Funktion `count_iterations`, die eine ganze Zahl `n` als Argument nimmt und zurückgibt, wie häufig die Funktion f auf diese Zahl angewendet werden muss, bis das Ergebnis 0 ist.

Bsp.: `assert count_iterations(5) == 4`
Begründung: $f(5) = 4; f(4) = 2; f(2) = 1; f(1) = 0$

## Aufgabe 2: Dictionary (10/15?)

Gegeben ist folgendes Dictionary `points`, in dem Studenten und ihre jeweils erreichten Punkte gespeichert sind:

```
points = {"Paul": 15, "Frank": 44, "Tim": 20, "Anna": 29}
```

Formulieren Sie eine Funktion `cluster_by_grade`, die ein solches Dictionary nimmt und ein neues zurückgibt, in dem die Studenten-Namen den jeweiligen Punkten zugeordnet werden. Dabei sollen die Punktzahlen immer auf die nächste 10er-Zahl abgerundet werden. Im zurückgegebenen Dictionary sollen dabei nur die Punktebereiche auftauchen, zu denen es auch tatsächlich Studenten gibt. Es sollen also keine leeren Listen enthalten sein.

Bsp.: `assert cluster_by_points(points) == {10: ["Paul"], 20: ["Tim", "Anna"], 40: ["Frank"]}`

## Aufgabe 3: Strings (20)

Schreiben Sie eine Funktion `is_strong(pw: str)`, die ein Passwort als String nimmt und `True` oder `False` zurückgibt, je nachdem ob das Passwort als stark eingestuft wird oder nicht.
Ein starkes Passwort muss dabei

(1) Mindestens 8 Zeichen lang sein
(2) Mindestens eine Ziffer enthalten
(3) Wenn es maximal drei Ziffern enthält, dann muss das Passwort mind. 1 Sonderzeichen enthalten
(4) Wenn es weniger als drei Großbuchstaben enthält, dann muss das Passwort mind. 1 Sonderzeichen enthalten
(5) Ist sowohl wegen Punkt (3) also auch wegen Punkt (4) ein Sonderzeichen nötig, so muss ein starkes Passwort mind. 2 Sonderzeichen enthalten

Der Einfachheit halber gehen wir davon aus, dass ein Passwort nur aus `a-z`, `A-Z`, `0-9` und den Sonderzeichen `"!"`, `"?"`, `"+"` und `"*"` besteht.
Hinweis: Es bietet sich an, die Methoden `isupper` und `isdigit` zu verwenden.

## Aufgabe 4: Dataclasses (6 + 6 + 8 = 20)

a) Implementieren Sie eine Datenklasse Ranking, mit folgenden Attributen:
- `club: str`
- `wins: int`
- `draws: int`
- `losses: int`
- `goals_achieved: int`
- `goals_conceded: int`

b) Implementieren Sie eine Methode, die eine String-Repräsentation eines Rankings zurückgibt. Dieser String soll den Namen, die Anzahl der Spiele, die Siege, die Unentschieden, die Niederlagen, das Torverhältnis, die Tordifferenz und die Punkte enthalten. Das Torverhältnis soll dargestellt werden als die Anzahl der erzielten Tore und die Anzahl der gefangenen Tore, getrennt durch einen Doppelpunkt. Die Punkte berechnen sich als 3 Punkte pro Sieg und 1 Punkt für jedes Unentschieden. Niederlagen geben weder Punkte noch Punkabzug.

Bsp.: `r1 = Ranking("FC H", 6, 2, 2, 23, 14)`
`r1.show() == "FC H 10 6 2 2 23:14 9 20"`

c) Implementieren Sie die Methode < (`__lt__()`), die die `Rankings` zweier Vereine vergleicht. Stellen Sie dabei sicher, dass tatsächlich zwei `Rankings` verglichen werden. Zunächst sollen die Punkte vergliche werden. Bei Punktgleichstand dann die Tordifferenz und wenn auch diese gleich ist, dann bewertet sich der Vergleich nach der Anzahl an Siegen.

Bsp.: `r1 = Ranking("FC H", 6, 2, 2, 23, 14)`
`r2 = Ranking("SC B", 5, 5, 0, 21, 12)`
`assert r2 < r1`
`assert not (r1 < r1)`

## Aufgabe 5: Testen

Gegeben ist die folgende Funktion `is_prime`:

```
import math
def is_prime(n: int) -> bool:
    n == 2:
        return True
    elif n < 2 or n % 2 == 0:
        return False
    x = math.floor(math.sqrt(n)) + 1
    for i in range(3, x, 2):
        if n % i == 0:
            return False
    return True
```

Schreiben Sie für jedes `return` Statement exakt einen mit `pytest` ausführbaren `unittest`.

## Aufgabe 6: Trees

Gegeben ist die folgende aus der Vorlesung bekannte Implementation eines Binärbaumes:

```
from typing import Optional
from dataclasses import dataclass
@dataclass
class Node:
    mark: str
    left: Optional['Node']
    right: Optional['Node']
```

Implementieren Sie eine Funktion `find_substr`, die einen String als Argument nimmt und eine Liste alle `marks` zurückgibt, in denen der String enthalten ist. Verwenden Sie dabei *Pre-Order*.

## Aufgabe 7: Generators

In dieser Aufgabe sind *keine* Typannotationen nötig. Verwenden Sie bei den folgenden Teilaufgaben keine von `python` bereitgestellten Generator-Funktionen außer `range`. Also insbesondere nicht `map`, `filter` oder `enumerate`. Dabei soll unnötiger Speicherverbrauch vermieden werden, wandeln Sie also insbesondere nicht zu Beginn den Generator in eine Liste um.

a) Implementieren Sie die Funktion `my_enumerate`, die sich genauso verhält, wie die von `python` bereitgestellte `enumerate`-Funktion.

   Bsp.: `assert list(my_enumerate("abcd")) == [(0, "a"), (1, "b"), (2, "c"), (3, "d")]`

b) Implementieren Sie eine Funktion `prefixes`, die eine Liste von ganzen Zahlen als Argument nimmt und nach und nach alle Anfangs-Teillisten zurückgibt.

   Bsp.: `assert list(prefixes([1, 2, 3, 4]) == [[], [1], [1, 2], [1, 2, 3], [1, 2, 3, 4]]`

c) Implementieren Sie eine Funktion `alternate`, die zwei Generatoren als Argumente nimmt und immer abwechselnd die Werte der beiden zurückgibt. Ist der eine erschöpft, aber vom anderen noch Elemente übrig, so sollen dann die verbleibenden Elemente direkt hintereinander zurückgegeben werden.

   Bsp.: `assert list(alternate("abcdefg", [1, 2, 3, 4])) == ["a", 1, "b", 2, "c", 3, "d", 4, "e", "f", "g"]`

## Aufgabe 8: Functional Programming

In dieser Aufgabe sind *keine* Typannotationen nötig. Verwenden Sie bei den folgenden Teilaufgaben funktionale Programmierung, also entweder Funktionen, deren Funktionsrumpf ausschließlich ein `return` Statement enthält, oder `lambda`s, die einer Variable zugewiesen werden.

a) Implementieren Sie eine Funktion `twice`, die eine Funktion f als Argument nimmt und eine Funktion zurückgibt, die die Funktion f zweimal auf die jeweilige Eingabe anwendet.

   Bsp.: `assert twice(lambda x: 2 * x)(5) == 20`

b) Implementieren Sie eine Funktion `pythagorean_triples`, die eine ganze Zahl `n > 0` nimmt und eine Liste aller Tripel aus ganzen Zahlen < n `(a, b, c)` zurückgibt, für die gilt: $a^2 + b^2 = c^2$. Verwenden Sie dazu genau eine List-Comprehension. Es sollen dabei Duplikate vermieden werden. Ein Duplikat in diesem Sinn ist auch `(b, a, c)`, wenn bereits `(a, b, c)` in der Liste enthalten ist.

   Bsp.: `assert pythagorean_triples(15) == [(3, 4, 5), (6, 8, 10), (5, 12, 13)]`
   Hinweis: die Reihenfolge der Tripel ist irrelevant.