# Aufgabenblatt 2

## 2.1 Operationen und Präzedenzen (8p)

Gegeben sei eine Reihe von Anweisungen:

```
a = 1 + 2 * 3 - 4
b = a ** 2 ** a
c = a == 23
d = [] or None and [1,2,3]
e = [0] or None
f = not True or d == 0
g = 3 and -4 + 1
```

### 2.1.1 Werte (3p)

Erstellen Sie ein kleines Skript namens *precedences.py* mit einem Header, der Auskunft gibt darüber, welche Python-Variante Sie verwenden (`#!...` als erste Zeile), und wer das Skript erstellt hat bzw. was im Skript geübt wird (in einem kurzen DocString). Kopieren Sie anschließend die angegebenen Anweisungen ins Skript (als einfache Anweisungsfolge, ohne Einrückung).

- Gestalten Sie für jede angegebene Anweisung zwei Print-Anweisungen.
  - Die erste Ausgabe soll über den **Wert der erzeugten Variable** berichten, **sowie Ihre Hypothese** über den erwarteten Wert hartkodiert angeben.
  - Die zweite, anschließende Anweisung soll den **entsprechenden boolschen Wert** der Variable ausgeben, wieder sollten Sie **Ihre Annahme** hartkodiert angeben.

  Beispiel[1]:

```
a = 1 + 2 * 3 - 4
print("a = " + str(a) + " -- erwarteter Wert: 3")
print("bool(a) = " + str(bool(a)) + " -- erwarteter boolscher
    Wert: True")
print()
```

- Führen Sie das Skript erst jetzt aus (im Python-Interpreter oder von der Kommandozeile aus).
- Korrigieren Sie ggf. die erwarteten Angaben in den Print-Anweisungen falls vom tatsächlichen abweichend.

[1] Eine leere Print-Anweisung erzeugt eine leere Zeile, und kann als mögliche Trennung der Ausgaben eingesetzt werden.

### 2.1.2 Präzedenzen (5p)

- Identifizieren Sie Operatoren und Operanden in den Anweisungen und **markieren Sie die Berechnungsreihenfolge** der Operationen entsprechend der Präzedenz (Bindestärke) der Operatoren durch Klammerung ( `( )` ). Die Operation mit dem am schwachsten bindenden Operanden soll nicht geklammert werden.
  Beispiel: `a = ((1 + (2 * 3)) - 4)`
- Führen Sie das Skript erneut aus, und überprüfen Sie, ob der Wert der einzelnen Variablen wie in den Print-Anweisungen erwartet aussieht. Korrigieren Sie die Klammerung bei Bedarf.
- Kommentieren Sie einige der Anweisungen:
  - Kommentar zu `d`: Welche/r Operand/en muss/müssen in der Anweisung **nicht ausgewertet** werden um `d` zu erzeugen, und warum nicht?
  - Identifizieren Sie eine weitere Anweisung, in der ein Operand **nicht ausgewertet** wird, und kommentieren Sie diese mit der Feststellung und dem Grund.

## 2.2 (Un)veränderbare Datentypen (12p)

### 2.2.1 Unveränderbare Typen (5p)

Öffnen Sie das beigefügte Skript *immutables.py* in einem Editor. Führen Sie die Vorübung (im Kommentar ab Zeile 9) durch. Beantworten Sie nach diesen und ggf. weiteren Übungen im Python-Interpreter die Fragen, welche als Kommentar in der Funktion `check_immutables()` verfasst wurden. Beantworten Sie die Fragen durch Ergänzung des Kommentars. Geben Sie im DocString zum Skript Ihren Namen als Autor an.

### 2.2.2 Veränderbare Typen – Listen (7p)

Öffnen Sie das beigefügte Skript *mutables.py* in einem Editor. Führen Sie die Vorübung (im Kommentar ab Zeile 8) durch. Beantworten Sie nach diesen und ggf. weiteren Übungen im Python-Interpreter die Fragen, welche als Kommentar in der Funktion `check_immutables()` verfasst wurden. Beantworten Sie die Fragen durch Ergänzung des Kommentars. Geben Sie im DocString zum Skript Ihren Namen als Autor an.

## 2.3 Miniprogramm (10p)

Schreiben sie ein kurzes Programm im Skript namens *add_ numbers.py*, das alle ganzen Zahlen von einer gegebenen Zahl bis einer zweiten (z. B. von 3 bis 5) aufaddiert und das Ergebnis (z. B. 3+4+5 = 12) ausgibt.

- Das Skript soll mit einem **Header** ähnlich wie für Aufgabe 2.1 beginnen.
- Das Skript soll zwei Funktionen sowie einen Main-Teil enthalten.
  - **Funktion** `user_input(str promt) -> int`[2] soll mit der im Argument angegebenen Prompt-Anfrage nach einer ganzen Zahl fragen und diese als Integer zurückgeben.[3]
  - **Funktion** `sum_from_to(int from_int, int to_int) -> int` soll zwei Integers als Argumente nehmen, und einen Integer zurückgeben, wobei der Rückgabewert das aufsummierte Ergebnis der Zahlfolge zwischen den beiden Eingabezahlen sein soll (beide Zahlen inklusive).
  - Beide Funktionen sollten mit je einem **DocString** versehen werden, der über die Funktionalität und Eingabe bzw. Rückgabe der Funktion berichtet.
  - Der **Main-Teil** – eingerückger Block nach einer Zeile
    `if __name__ == "__main__":`
    – soll mithilfe der beiden geschriebenen Funktionen sowie weiteren Anweisungen und Kontrollstrukturen so gestaltet werden, dass zwei Ganzzahlen eingelesen werden, welche für die Berechnung der Aufsummierung mit `sum_from_to()` dienen. Das Ergebnis soll ausgegeben werden.
  - Wird als zweite Zahl eine kleinere Zahl als die erste vom Benutzer (Sie) eingegeben, soll keine Berechnung erfolgen, sondern der Benutzer soll von den erwarteten Eingaben informiert werden.
  - Testen Sie Ihr Programm mit unterschiedlichen Zahlen.
- Beispielaufrufe (von der Kommandozeile aus):

```
$ python add_numbers.py
Anfangs-Zahl: 3
End-Zahl: 5
Aufsummiertes Ergebnis: 12
$ python add_numbers.py
Anfangs-Zahl: 3
End-Zahl: 3
Aufsummiertes Ergebnis: 3
$ python add_numbers.py
```

[2] Die Angabe soll als Signatur der Funktion interpretiert werden.
[3] Man nimmt an, dass der Benutzer tatsächlich Zahlen eingibt, sodass falsche Benutzereingaben wie Nicht-Zahlen hier nicht behandelt werden müssen.

```
Anfangs-Zahl: 3
End-Zahl: 2
Die zweite eingegebene Zahl soll mindestens so groß sein, wie
   die erste.
```

---

Abgabe:

- Die erstellten und ergänzten Dateien (*precedences.py*, *immutables.py*, *mutables.py* und *add_numbers.py*) müssen in einem komprimierten Ordner als E-Mail-Anhang abgegeben werden.
- Der Betreff der E-Mail soll lauten: `[Prog1_WS17]` ***`NAME, VORNAME,`*** `HA02` – Name und Vorname bitte entsprechend eintragen.