Stefan Kurtz **Universität Hamburg**
Jeanine Liebold **Zentrum für Bioinformatik**

Programmierung für Naturwissenschaften 1
Wintersemester 2022/2023
Übungen zur Vorlesung: Ausgabe am 11.01.2023

Selbsttest-Aufgaben zum Thema Klassen und Funktionen in Python

Um ihre eigenen Fähigkeiten in der Python-Programmierung zu testen, lösen Sie bitte selbständig die folgenden Aufgaben. Nutzen Sie zunächst möglichst nicht den Python-Interpreter, sondern nur Papier und Stift. Sie können Ihre Lösung natürlich später mit dem Python-Interpreter verifizieren. Ihre Lösungen geben Sie nicht ab. Nutzen Sie die ersten 15 Minuten der Übung für diese Aufgaben. Es wird nicht erwartet, dass Sie alle Aufgabe in dieser Zeit lösen können.

1.
   - Geben Sie bitte an, was das folgende Programmstück ausgibt.
   - Warum ist es bei der Anwendung von `zip` in der Methode `__pow__` nicht notwendig, sich direkt auf die Instanz-Variable `_values` zu beziehen?
   - Vereinfachen Sie den Ausdruck `v0.__pow__(v1)` in der `print`-Anweisung, ohne das die Ausgabe sich ändert.

```
class Vector:
  def __init__(self,values):
    self._values = values
  def __str__(self):
    return '({})'.format(', '.join(map(str,self)))
  def __len__(self):
    return len(self._values)
  def __iter__(self):
    return iter(self._values)
  def __pow__(self,o_vector):
    assert len(self) == len(o_vector)
    return Vector([b ** e for b,e in zip(self,o_vector)])
v0 = Vector([2,3,4])
v1 = Vector([3,4,5])
print('{} ** {} = {}'.format(v0,v1,v0.__pow__(v1)))
```

2.
   - Geben Sie bitte an, was das folgende Programmstück ausgibt.
   - Nutzen Sie `pysearch.py`, um zu klären, was `isinstance(param,str)` bedeutet. Dokumentieren Sie diese Bedeutung und überlegen Sie sich, wofür sie nützlich sein könnte.
   - Welche Art von Werten wird in der Instanz-Variablen `self._bits` gespeichert?
   - Wie sieht die Ausgabe aus, wenn man in der Wertzuweisung an `bs0` ein Vorkommen von `0` durch einen Unterstrich ersetzt?
   - Schreiben Sie den Ausdruck `bs0 | bs1` aus der `print`-Anweisung ohne Verwendung des Infix-Operators `|` auf und zwar so, dass die Ausgabe sich nicht verändert.

```
class BitString:
  def __init__(self,param):
    if isinstance(param,str):
      self._bits = [a == '1' for a in param]
    else:
      self._bits = param
  def __str__(self):
    return ''.join(['1' if b else '0' for b in self._bits])
  def __or__(self,o_bitvector):
    assert len(self._bits) == len(o_bitvector._bits)
    return BitString([x or y for x,y in zip(self._bits,o_bitvector
       ._bits)])
bs0 = BitString('10110')
bs1 = BitString('11000')
print('{} | {} = {}'.format(bs0,bs1,bs0 | bs1))
```

3. Geben Sie bitte an, was das folgende Programmstück ausgibt. Charakterisieren Sie Strings `s`, so dass `bracket_checker(s)` den Wert `True` liefert.

```
def bracket_checker(s):
  br_open = '([{'
  br_close = ')]}'
  comrade = { bo : br for bo, br in zip(br_open,br_close) }
  stack = list()
  for cc in s:
    if cc in br_open:
      stack.append(comrade[cc])
    elif cc in br_close:
      if len(stack) == 0 or stack[-1] != cc:
        return False
      stack.pop()
  return len(stack) == 0
bracket_expressions = ['{}','{[}','{','(([))]','{{([[]]){}}}','
   {{([])}}']
print([bracket_checker(s) for s in bracket_expressions])
```

4. Beschreiben Sie, was die folgende Funktion berechnet, wenn `n` eine positive ganze Zahl ist und `factors` eine Liste von positiven ganzen Zahlen. Was wird durch die `for`-Schleife nach der Funktionsdefinition berechnet und ausgegeben? Dabei ist `primes_up_to_100` eine Liste der Primzahlen ≤ 100.

```
def decompose(n,factors):
  if n == 1:
    return list()
  for i in factors:
    if n % i == 0:
      pfd = decompose(n // i, factors)
      if pfd is not None:
        pfd.append(i)
        return pfd
  return None
```

```
for n in range(100,130+1):
  pfd = decompose(n,primes_up_to_100)
  if pfd is not None:
    print('{}\t{}'.format(n,pfd))
```

Aufgabe 10.1 (2 Punkte)

Aus Zeitgründen können wir in der Vorlesung das Thema „Extraktion von Sequenzen aus Dateien im multiplen Fasta-Format“ nicht besprechen.. Dieser Abschnitt wird daher im Peer-Teaching Format behandelt.

Für diese Übungsaufgabe muss sich aus jeder Kleingruppe ein Student bzw. eine Studentin anhand der Vorlesungsfolien auf dieses Thema vorbereiten. Sie sollten untereinander frühzeitig klären, wer diese Vorbereitung in dieser Woche übernimmt.

Das in der Vorbereitung erworbene Wissen soll an die anderen Mitglieder der jeweiligen Kleingruppe dadurch weitergegeben werden, dass man zusammen den Inhalt der Folien zum Thema *Parsing sequences from a multiple FASTA file*) (siehe `python-slides.pdf`, Section 21, frame 1-12) bespricht. Das kann entweder in den ersten 20-25 Minuten der Übung erfolgen oder zu einem anderen Zeitpunkt außerhalb des Übungstermins.

Der oder die Studierende, die sich vorbereitet hat, kann ggf. bei Unklarheiten Fragen, zumindest zu den Grundbegriffen, beantworten oder Erläuterungen mündlich ergänzen. Falls Sie den in den Folien dargestellten Programmcode ausprobieren möchten, können Sie auf die Materialien zurückgreifen.

Nach Bearbeitung dieser Aufgabe dokumentiert jede Kleingruppe in der Textdatei `bearbeitung.txt` in höchstens 15 Zeilen mit maximal 80 Zeichen pro Zeile ihr Vorgehen bei der Erarbeitung des Themas und ggf. noch bestehende Verständnisfragen oder Hinweise zu Unklarheiten in den Folien. Willkommen sind natürlich auch Bemerkungen zur Lehrform selbst. Es soll nicht der Inhalt der Folien wiedergegeben werden. Selbstverständlich müssen Sie in der Datei auch die üblichen Metadaten angeben.

Aufgabe 10.2 (4 Punkte)

In dieser Aufgabe soll auf Basis der Funktionen aus Aufgabe 8.3 ein lauffähiges Programm `pwgen.py` zum Generieren von Passwörtern implementiert werden. In den Materialien zu dieser Aufgabe finden Sie die Datei `pwgen_template.py`, die Sie in in `pwgen.py` umbenennen. In der Datei `pwgen_functions.py` finden Sie die Musterlösung zu Aufgabe 8.3, die Sie gerne nutzen dürfen.

Zunächst implementieren Sie die Funktion `password_generate(wd,struct_str)`, die entsprechend des Dictionaries `wd` (berechnet durch `word_dict_get()`) und des Strukturstrings `struct_str` ein Paar `(pw,choice_list)` zurückliefert. Dabei ist `pw` ein zufällig generiertes Passwort und `choice_list` ist die Liste der Anzahlen der möglichen Passwörter für die einzelnen Strukturelemente in `struct_str`.

Zur Berechnung der Liste der Strukturelemente verwenden Sie die Funktion `structure_elements_list` aus Aufgabe 8.3. Für die Generierung der Teile des Passwortes entsprechend eines `d`- bzw. `p`-Strukturelements der Länge `n` verwenden Sie den Ausdruck `randstring(string.digits,n)` bzw. `randstring(string.punctuation,n)`. Dabei ist `randstring` die in Aufgabe 8.3 entwickelte Funktion. Hierfür müssen Sie das Modul `string` importieren. Für die Generierung der Teile des Passwortes mit einem `w`-Strukturelement der Länge `m` wählen Sie ein Wort aus der Liste

`wd[m]` zufällig aus. Falls `m` kein Schlüssel in `wd` ist (also kein Wort der Länge `m` in `wordlist.txt` existiert), muss eine Fehlermeldung generiert und das Programm mit `exit(1)` abgebrochen werden.

Die einzelnen Werte in `choice_list` ergeben sich wie folgt:

- bei einem `d`- oder `p`-Strukturelement berechnet man den Wert aus der Anzahl der Zeichen im entsprechenden Alphabet und der Länge `m`.
- bei einem `w`-Strukturelement ist der Wert die Anzahl der Worte in `wd[m]`

Beispiel: Für den Strukturstring `w4p2d2w5` mit 4 Strukturelementen ist `choice_list` die Liste `[2143,1024,100,3132]`.

Im zweiten Schritt implementieren Sie eine Funktion `parse_command_line(argv)`, die unter Nutzung des Moduls `argparse` einen Argumentparser `p` mit den folgenden Optionen spezifiziert und `p.parse_args(argv)` zurückliefert. Siehe auch `python-slides.pdf`, Abschnitt *Functions*, frame 63-71.

- eine Option `-s/--structure` mit genau einem String-Argument, nämlich einem Strukturstring, wie oben beschrieben. Der default-Wert ist `'w4p2d2w5p1d1w8'` und dieser soll bei der Spezifikation dieser Option in `parse_command_line` angegeben werden.
- eine Option `-n/--number` mit genau einem ganzzahligen Argument, das die Anzahl der auszugebenden Passwörter spezifiziert. Der default-Wert ist 1 und dieser muss bei der Spezifikation dieser Option in `parse_command_line` angegeben werden.

Die Funktion `main()`, die das Hauptprogramm implementiert und ihr Aufruf sind bereits in der vorgegebenen Datei vorhanden. Durch `make test` verifizieren Sie, dass Ihr Programm für einige Testdaten korrekt funktioniert.

Punkteverteilung:

- 2 Punkte für die Implementierung der Funktion `password_generate`.
- 1 Punkt für die Implementierung von `parse_command_line`.
- 1 Punkt für die bestandenen Tests.

Aufgabe 10.3 (3 Punkte)

Sie haben eine neue Messmethode entwickelt, die für ein chemisches, physikalisches oder biologisches System Koordinaten im zweidimensionalen Raum liefert. Diese Messmethode kann z.B. durch die Wahl verschiedener Einstellungen variiert werden. Sie haben Ihre Methode mit verschiedenen Einstellungen ausprobiert und entsprechende Koordinaten ermittelt. Diese bilden damit Vorhersagen der Realität. Sie sollen nun die Qualität der einzelnen Messungen jeweils durch einen Vergleich Ihrer vorhergesagten Werte mit einem Goldstandard ermitteln, der durch eine sehr aufwändige, aber anerkannte Messmethode ermittelt wurde.

Die Qualität Ihrer Methode soll durch die Bestimmung der Sensitivität und der Spezifität relativ zum Goldstandard bestimmt werden. Die Sensitivität macht eine Aussage über die Fähigkeit der Methode, Koordinaten entsprechend des Goldstandards richtig vorherzusagen. Die Spezifität macht eine Aussage über die Fähigkeit der Methode, keine bzgl. des Goldstandards falschen Werte vorherzusagen. Die formale Definition dieser Begriffe basiert auf drei Mengen, P, G und TP. Dabei ist P die Menge der Messwerte Ihrer Messmethode, G die Menge der Werte des Goldstandards und $TP = P \cap G$, also die Schnittmenge von P und G. TP heißt auch die Menge der *True Positives*, also der korrekten Vorhersagen. Die Sensitivität $se(P, G)$ und die Spezifität $sp(P, G)$ sind definiert

durch

$$se(P, G) = 100 \cdot \frac{|TP|}{|G|} \qquad sp(P, G) = 100 \cdot \frac{|TP|}{|P|}$$

Dabei bezeichnet $|S|$ die Größe einer Menge S. Da $TP \subseteq G$ und $TP \subseteq P$, liegen beide Werte zwischen 0 und 100. Eine ideale Methode erreicht jeweils Werte von 100%, d.h. sie liefert die gleichen Messwerte wie der Goldstandard. In vielen Anwendungen erreicht man optimale Sensitivität nur auf Kosten geringer Spezifität und umgekehrt. Daher kombiniert man oft beide Werte, indem man den harmonischen Durchschnitt der Sensitivität und Spezifität berechnet. Für $0 \leq a, b \leq 100$ ist $\frac{2}{\frac{1}{a}+\frac{1}{b}}$ der harmonische Durchschnitt von a und b.

Implementieren Sie ein Programm `predictionqual.py`, das die Qualität der vorhergesagten Messdaten berechnet und formatiert ausgibt. Das Programm soll eine Option `-g/--gold_standard` mit genau einem String-Argument haben. Dieses String-Argument ist der Name der Datei mit dem Goldstandard (z.B. `goldstandard.tsv` im Material). Alle weiteren Argumente sind die Namen der Dateien mit den Koordinaten der Messungen (`prediction*.tsv` im Material).

Alle genannten Dateien enthalten jeweils ein Paar von x, y-Koordinaten pro Zeile, separiert durch das Zeichen \t. Zur Vereinfachung sind die Koordinaten durch ganze Zahlen repräsentiert. Ein Koordinatenpaar (a, b) ist identisch mit einem Koordinatenpaar (a', b'), wenn $a = a'$ und $b = b'$ ist. True Positives sind die identischen Koordinatenpaare. Die Ausgabe soll zeilenweise für jede Datei die Qualitätswerte der Messungen ausgeben, und zwar durch das Zeichen \t separiert und in folgendem Format:

- Spalte 1: Name der Datei `(filename)`
- Spalte 2: Anzahl der True Positives `(tp)`
- Spalte 3: Sensitivität `(sens)`
- Spalte 4: Spezifität `(spec)`
- Spalte 5: Harmonischer Durchschnitt der Sensitivität und Spezifität `(hmean)`

Die Werte müssen zeilenweise und tabulatorsepariert ausgegeben werden. Die Fließkommawerte müssen mit zwei Nachkommastellen ausgegeben werden. In der Datei `quality_out.tsv` finden Sie das erwartete Ergebnis.

Hinweise:

- Die Dateien enthalten jedes Koordinatenpaar jeweils genau einmal.
- Verwenden Sie die Klasse `set` zur Speicherung der Koordinatenpaare. Diese Klasse bietet u.a. die folgenden Operationen:
  - Eine leere Menge `s` wird durch `s = set()` erzeugt.
  - Durch `s.add(x)` fügen Sie ein neues Element `x` zur Menge `s` hinzu.
  - Für Mengen `s` und `t` liefert der Ausdruck `s & t` die Schnittmenge `s∩t` von `s` und `t`.
  - Für eine Menge `s` liefert `len(s)` die Größe dieser Menge.
- Gliedern Sie Ihr Programm durch die Implementierung von fünf Funktionen:
  - eine Funktion `parse_command_line(argv)`, die unter Nutzung des Moduls `argparse` einen Argumentparser `p` spezifiziert und `p.parse_args(argv)` zurückliefert. Siehe auch `python-slides.pdf`, Abschnitt *Functions*, frame 63-71. Beachten Sie die

Verwendung des Parameters `argv`, wodurch diese Funktion mit dem Argument `sys.argv[1:]` aufgerufen werden muss.

- eine Funktion `inputfile2set(inputfile)` erhält genau eine Eingabedatei im beschriebenen Format und liefert mit einer `return`-Anweisung die Menge der Koordinatenpaare.
- eine Funktion `evaluate(gs,prediction)` erhält zwei Mengen `gs` und `prediction` mit den Koordinaten eines Goldstandards und einer Vorhersage und liefert mit einer `return`-Anweisung die Anzahl der True Positives, die Sensitivität und die Spezifität bzgl. dieser beiden Mengen.
- eine Funktion `harmonic_mean(a,b)` liefert den harmonischen Durchschnitt von `a` und `b`.
- eine Funktion `main()`, die nacheinander die obigen Funktionen aufruft und mit `print` die Ausgabe erzeugt. Diese Funktion wird aufgerufen, wenn die Bedingung `__name__ == '__main__'` gilt.

In den Materialien finden Sie neben den erwähnten Dateien mit dem Goldstandard und den Messwerten ein Makefile. Durch `make test` verifizieren Sie, dass Ihr Programm für diese Testdaten korrekt funktioniert.

Punkteverteilung:

- Implementierung der fünft oben genannten Funktionen und des Hauptprogramms: je 0.5 Punkte
- bestandene Tests: 0.5 Punkte

Aufgabe 10.4 (6 Punkte)

Eine Permutation einer Menge S ist eine Liste der Elemente aus S, in der jedes Element genau einmal vorkommt. Jede Permutation ist also eine Liste der Länge n, wenn n die Anzahl der Elemente in S ist. Verschiedene Permutationen von S unterscheiden sich durch die Reihenfolge der Elemente aus S. Die Menge der Permutationen von S wird durch $Perms(S)$ bezeichnet.

Beispiel: Für $S = \{0, 1, 2\}$ ist $Perms(S) = \{[2, 1, 0], [1, 2, 0], [2, 0, 1], [0, 2, 1], [1, 0, 2], [0, 1, 2]\}$.

Für Permutationen gibt es vielfältige Anwendungen in der Mathematik und Informatik.

Ein rekursiver Algorithmus zur Berechnung von $Perms(S)$ funktioniert nach den folgenden Regeln:

- Falls $S = \emptyset$, dann ist $Perms(S) = \{[\,]\}$, d.h. die leere Liste ist die einzige Permutation der leeren Menge S.
- Falls $S = \{a\}$ (d.h. S besteht aus genau einem Element a), dann ist $Perms(S) = \{[a]\}$.
- Falls S mindestens zwei Elemente enthält, dann berechnet man für alle $a \in S$ die Menge $R(S, a) = Perms(S \setminus \{a\})$.[1] $Perms(S)$ besteht in diesem Fall aus den Listen $p.append(a)$ für alle $a \in S$ und $p \in R(S, a)$.

Beispiel: Sei $S = \{0, 1, 2\}$. Nach dem obigen Verfahren berechnet man zunächst die Mengen $R(S, 0)$, $R(S, 1)$ und $R(S, 2)$. Es ist $R(S, 0) = Perms(S \setminus \{0\}) = Perms(\{1, 2\})$. Daher berechnet

[1] Der Operator $\setminus$ steht für die Mengendifferenz, d.h. für zwei Mengen A und B ist $A \setminus B = \{a \mid a \in A, a \notin B\}$.

man zunächst die Mengen $R(\{1,2\},1)$ und $R(\{1,2\},2)$. Es gilt:

$$R(\{1,2\},1) = Perms(\{2\}) = \{[2]\}$$
$$R(\{1,2\},2) = Perms(\{1\}) = \{[1]\}$$

Damit ergibt sich $R(S,0) = \{[2,1],[1,2]\}$. Analog erhält man $R(S,1) = \{[2,0],[0,2]\}$ und $R(S,2) = \{[1,0],[0,1]\}$. Aus $R(S,0)$ berechnet man durch Anhängen von 0 die beiden Permutationen $[2,1,0],[1,2,0]$. Aus $R(S,1)$ berechnet man durch Anhängen von 1 die beiden Permutationen $[2,0,1],[0,2,1]$. Aus $R(S,2)$ berechnet man durch Anhängen von 2 die zwei Permutationen $[1,0,2],[0,1,2]$. Damit wurden alle Permutationen von S berechnet.

Alle Permutationen berechnen

Benennen Sie die Datei `allperms_template.py` in `allperms.py` um. Implementieren Sie darin eine rekursive Python-Funktion `all_perms_rec(all_perms_list,elems)`, die nach dem obigen Algorithmus alle Permutationen der Elemente aus `elems` berechnet und jeweils an die Liste `all_perms_list` anhängt. `all_perms_rec` hat keinen Rückgabewert. Die Funktion wird in der bereits implementierten Funktion `all_permutations(elems)` aufgerufen.

Die Menge der Elemente aus S bzw. ihre Teilmengen können Sie jeweils als Liste darstellen, d.h. `elems` ist eine Liste. In der Implementierung müssen Sie Elemente aus Listen löschen. Durch `l.pop(idx)` können Sie das Element an Index `idx` in einer Liste `l` löschen.

Beachten Sie, dass eine Wertzuweisung `p = q` für zwei Listen `p` und `q` eine Referenz `p` auf die Liste `q` erzeugt. Da `all_perms_rec` mit verschiedenen Listen (als zweitem Argument) aufgerufen wird, müssen Listen kopiert werden. Die Funktion `q.copy()` liefert eine Kopie der Liste `q`.

Vollständigkeit und Korrektheit der Berechnung verifizieren

Implementieren Sie eine Funktion `all_permutations_verify(all_perms_list,elems)`, die verifiziert, dass die Liste `all_perms_list` genau die Liste aller Permutationen von `elems` ist. Dabei sind die Elemente in `elems` aufsteigend sortiert. In der Funktion `all_permutations_verify` müssen Sie mit Hilfe von `assert` die folgenden Bedingungen verifizieren:

1. Die Länge von `all_perms_list` ist $n!$, wobei n die Anzahl der Elemente in `elems` ist.
2. Es gibt keine Permutation in `all_perms_list`, die mehr als einmal vorkommt.
3. Alle Elemente aus `all_perms_list` sind Permutation von `elems`.

Durch die Verwendung von `assert` bricht das Programm ab, wenn eine der genannten Bedingungen nicht zutrifft.

Hinweis: Die Verifikation von Duplikaten (siehe 2) darf nicht dadurch erfolgen, dass man alle möglichen Elemente aus `all_perms` paarweise miteinander vergleicht. Nutzen Sie stattdessen Mengen, repräsentiert durch Objekte der Klasse `set` (siehe auch Aufgabe 10.2). Beachten Sie dabei, dass man mit dieser Klasse nur hashbare Objekte verarbeiten kann. Listen gehören nicht dazu. Sie müssen also jeweils eine Permutation in ein hashbares Objekt verwandeln, um es in die Menge einzufügen.

Im Hauptprogramm werden die beiden genannten Funktionen für Listen der Länge n aufgerufen,

wobei n das einzige Argument des Hauptprogramms ist. Durch `make test` verifizieren Sie die Korrektheit Ihrer Implementierung für alle n, $2 \leq n \leq 7$.

Begründen Sie in der Datei `allperms_test_correct.txt`, warum alle drei Bedingungen 1.-3. zusammen ausreichen, um zu verifizieren, dass eine Liste `all_perms_list` genau die Liste aller Permutationen von `elems` ist. Genau bedeutet hier, dass keine Permutationen fehlen (Vollständigkeit) und dass keine zusätzlichen Elemente in der Liste sind (Korrektheit).

Punkteverteilung:

- Implementierung von `all_perms_rec`: 3 Punkte
- Implementierung von `all_permutations_verify`: 2 Punkte
- Begründung zu den Bedingungen 1.-3.: 1 Punkt

Bitte die Lösungen zu diesen Aufgaben bis zum 16.01.2023 um 18:00 Uhr an pfn1@zbh.uni-hamburg.de schicken.