Einführung in die Programmierung

Prof. Dr. Peter Thiemann
Marius Weidner, Hannes Saffrich
Simon Dorer, Sebastian Klähn

Universität Freiburg
Institut für Informatik
Wintersemester 2024

Übungsblatt 9

Abgabe: Montag, 16.12.2023, 9:00 Uhr

Korrektur zur Vorlesung: Private Attribute

In der Vorlesung haben Sie private Attribute von Datenklassen kennengelernt, auf die von außerhalb nicht ohne weiteres zugegriffen werden kann. Beim Zugriff auf ein privates Attribut liefert Python eine Exception. Diese Attribute werden in Python mit zwei Unterstrichen (`__`) zu Beginn gekennzeichnet. Attribute, die mit nur einem Unterstrich (`_`) beginnen sind dagegen lediglich per Konvention privat und können wie gewohnt von außen gelesen und geschrieben werden. **Im Rahmen der Vorlesung und Übungen sollen Sie private Attribute mit zwei Unterstrichen verwenden.**

Ein Beispiel:

```
@dataclass
class Foo:
    bar: int  # Normales Attribut
    _baz: int  # Nur per Konvention privat (NICHT VERWENDEN)
    __bax: int  # Privates Attribut

    def add_bax(self, value: int) -> int:
        return self.__bax + value
```

Definiert man nun eine Instanz der Klasse `Foo` und versucht auf die Attribute zuzugreifen, erhält man folgendes Resultat:

```
>>> foo = Foo(1, 2, 3)
>>> foo.bar  # Kein Fehler
1
>>> foo._baz  # Kein Fehler
2
>>> foo.__bax  # Fehler
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'Foo' object has no attribute '__bax'. Did you mean: '_baz'?
```

Innerhalb der Klasse, kann jedoch ohne Probleme auf das private Attribut `__bax` zugegriffen werden:

```
>>> foo.add_bax(5)
8
```

Zweite Korrektur zur Vorlesung: Dunder-Methoden

Entsprechend der vorherigen Blätter müssen Sie für jede Funktion vollständige Typannotation angeben, dazu zählen auch die Argumente von Dunder-Methoden, zum Beispiel `other`. In den annotierten Funktionen dürfen Sie dann davon ausgehen, dass die Argumente die richtigen Typen haben und müssen dies nicht explizit mit match überprüfen. In dem Spezialfall, dass der Typ einer Variable eine Union ist, kann es jedoch trotzdem sinnvoll sein zu matchen. Siehe Beispiel:

```
@dataclass
class Foo:
    bar: str

    def __add__(self, other: str | int) -> bool:
        match other:
            case str():
                return self.bar + other
            case int():
                return self.bar + str(other)
```

Aufgabe 9.1 (Dateiverwaltung; 4 Punkte; Datei: `files.py`)

In dieser Aufgabe sollen Sie Datenklassen schreiben, die mehrere Invarianten einhalten. In Ihrem git-Repo befindet sich eine Testdatei `test_files.py`, mit der Sie ihre Abgabe testen können. Achten Sie darauf, dass Namen von internen Attributen mit zwei Unterstrichen (`__`) beginnen, auch wenn diese in der Aufgabenstellung nicht so genannt werden. Denken Sie auch an die 3 Regeln für Invarianten aus der Vorlesung!

(a) **File; 2.5 Punkte**
Erstellen Sie eine Datenklasse `File` mit den Attributen `name` (Dateiname) und `content` (Dateiinhalt). Das Attribut `content` soll standardmäßig auf den leereren String (`""`) gesetzt werden. Das Attribut `name` soll ein Property-Attribut mit Getter und Setter sein und soll die folgenden Invarianten einhalten:

- Der Dateiname muss genau einen Punkt (`"."`) enthalten.
- Der Teil des Dateinamens vor dem Punkt (Hauptname) und der Teil nach dem Punkt (Dateiendung) dürfen nicht leer sein.
- Die Dateiendung darf ausschließlich aus Buchstaben bestehen.
- Der Hauptname darf sowohl aus Buchstaben als auch aus Zahlen bestehen.

Sie können Sie zur Implementierung die String-Methoden `isalpha`[1] und `isalnum`[2] verwenden.

(b) **Directory; 1.5 Punkte**
Erstellen Sie eine Datenklasse `Directory`. Der Konstruktor soll keine Argumente erwarten, aber ein internes Attribut `files` anlegen, das eine Liste von

[1] siehe: https://docs.python.org/3.12/library/stdtypes.html#str.isalpha
[2] siehe: https://docs.python.org/3.12/library/stdtypes.html#str.isalnum

`File`-Objekten (initial die leere Liste `[]`) repräsentiert, die sich im Verzeichnis befinden. Das Attribut `files` soll nur lesbar, aber nicht direkt veränderbar sein (d.h., es ist eine Property mit einem Getter, aber ohne Setter).

Definieren Sie eine Methode `add_file`, die eine `File`-Instanz als Argument erwartet und versucht, diese zur Liste `files` hinzuzufügen. Falls eine Datei mit dem gleichen Namen bereits existiert, soll `False` zurückgegeben werden. Ansonsten wird die Datei am Ende der Liste hinzugefügt und `True` zurückgegeben.

Aufgabe 9.2 (Unendlichkeit; 4 Punkte; Datei: `infinity.py`)

In dieser Aufgabe erweitern wir die Menge der ganzen Zahlen $\mathbb{Z}$ um folgende zusätzliche Elemente:

- ∞ (positive Unendlichkeit),
- $-\infty$ (negative Unendlichkeit),
- `NaN` (undefinierte Ganzzahl).

Die erweiterte Menge ist definiert als $\mathcal{Z} = \mathbb{Z} \cup \{\infty, -\infty, \texttt{NaN}\}$. In Python repräsentieren wir $\mathcal{Z}$ durch den Typen `ExtendedInt` und die neu hinzugefügten Elemente durch die drei Datenklassen `PosInf`, `NegInf` und `UndefinedInt`. Die Menge der ganzen Zahlen $\mathbb{Z}$ wird wie gewohnt durch den Typen `int` repräsentiert:

```
type ExtendedInt = int | "PosInf" | "NegInf" | "UndefinedInt"

@dataclass
class PosInf:
    ...

@dataclass
class NegInf:
    ...

@dataclass
class UndefinedInt:
    ...
```

Ihre Aufgabe ist es, die arithmetischen Operationen Addition, Negation und Subtraktion für die neuen Elemente zu implementieren. Kopieren Sie dazu die gegebene Code-Struktur in Ihre Datei `infinity.py` und überschreiben Sie `...` mit den jeweiligen Methoden. Sie können Ihre Implementierung mit der Datei `test_infinity.py` testen. Gehen Sie wie folgt vor:

(a) **Negation; 1 Punkt**
Implementieren Sie die Negationsmethode `__neg__` für die Klassen `PosInf`, `NegInf` und `UndefinedInt`. Diese bekommen außer `self` keine weiteren Argumente und geben einen `ExtendedInt` zurück, der die Negation von `self`

repräsentiert. Dabei ist eine negierte undefinierte Ganzzahl wieder eine undefinierte Ganzzahl (d.h. $-u = u$). Die Negation von ∞ ist $-\infty$ und die Negation von $-\infty$ ist ∞.

(b) **Addition; 2 Punkte**
Definieren Sie die Methoden `__add__` (Addition von *links*) und `__radd__` (Addition von *rechts*) für die Klassen `PosInf`, `NegInf` und `UndefinedInt`. Die Methoden sollen neben `self` einen `ExtendedInt` als Argument bekommen und das Ergebnis der Addition vom Typ `ExtendedInt` zurückgeben. Die Regeln sind:

- Ganzzahl $+\infty = \infty$ und Ganzzahl $+(-\infty) = -\infty$
- $\infty + \infty = \infty$ und $-\infty + (-\infty) = -\infty$
- $\infty + (-\infty) =$ `NaN` und $-\infty + \infty =$ `NaN`
- `NaN` $+ x =$ `NaN` (für beliebiges $x \in \mathcal{Z}$).

Verwenden Sie In Ihrer Implementierung Pattern-Matching und keine if-Abfagen.

Definieren Sie nun die Addition von *rechts* (`__radd__`) für die Klassen `PosInf`, `NegInf` und `UndefinedInt`. Verwenden Sie hierzu die bereits definierte Methode `__add__` und nutzen sie die Kommutativität ($a + b = b + a$) der Addition.

(c) **Subtraktion; 1 Punkte**
Implementieren Sie die Subtraktion von links (`__sub__`) und von rechts (`__rsub__`) für die Klassen `PosInf`, `NegInf` und `UndefinedInt`. Verwenden Sie hierzu ausschließlich die in Aufgabenteil a) und b) definierten Methoden zur Negation und Addition der erweiterten Ganzzahlen. Sie können dabei ausnutzen dass die Subtraktion $a - b$ als Addition $a + (-b)$ dargestellt werden kann.

Aufgabe 9.3 (Datenkapselung und dunder-Methoden; 8 Punkte; Datei: `times.py`)

In dieser Aufgabe sollen Sie eine Datenklasse schreiben, die das Prinzip der *Datenkapselung* verwendet und zusätzlich einige *dunder-Methoden* besitzt. In Ihrem git-Repo befindet sich eine Testdatei `test_times.py`, mit der Sie ihre Abgabe testen können.

(a) **Datenklasse und Properties; 3 Punkte**
Schreiben Sie eine Datenklasse `Time`, die eine Uhrzeit bzw. relative Zeitangabe im 24-Stunden-Format beschreibt. Die Klasse soll als äußeres Interface die beiden ganzzahligen property-Attribute `hours` (Stunden) und `minutes` (Minuten) mit Gettern und Settern besitzen. Die interne Repräsentation soll jedoch nur aus einer einzigen Ganzzahl `__time` bestehen, die die gesamte Zeit in Minuten darstellt. Achten Sie bei der Umwandlung darauf, dass die interne Repräsentation jeweils zyklisch auf die Stunden $\{0, \ldots, 23\}$ bzw. Minuten $\{0, \ldots, 59\}$ abgebildet wird.[3]

[3] An die Klasse übergebene Argumente können beliebig groß oder klein sein. Orientieren Sie sich ggf. an den Tests, um zu verstehen, was bei der Unter-/Überschreitung der Intervalle passiert.

```
>>> t1 = Time(12, 23)
>>> assert t1.hours == 12 and t1.minutes == 23
>>> t2 = Time(-13, 61)
>>> assert t2.hours == 12 and t2.minutes == 1
```

(b) **Vergleichsoperatoren; 2.5 Punkte**
Überladen Sie die Vergleichsoperatoren `==`, `>`, `<`, `>=` und `<=` für jeweils zwei Objekte der Klasse `Time`. Vergleichen Sie dabei zuerst die Stundenzahl. Ist die Stundenzahl der Operanden gleich, muss die Minutenzahl verglichen werden.

(c) **Arithmetik; 1.5 Punkte**
Überladen Sie die arithmetischen Operatoren `+` und `-`, die jeweils zwei Zeitangaben addieren bzw. voneinander abziehen.

```
>>> assert Time(12, 34) + Time(12, 32) == Time(1, 6)
>>> assert Time(10, 34) - Time(12, 44) == Time(21, 50)
```

(d) **String-Repräsentation; 1 Punkte**
Fügen Sie der Klasse die dunder-Methode `__str__`[4] hinzu, die keine Parameter außer self hat und eine String-Repräsentation des `Time`-Objekts der Form `"hh:mm"` zurückgibt, wobei `h` für eine Stunden-Ziffer und `m` für eine Minuten-Ziffer steht.

Aufgabe 9.4 (Dictionaries invertieren; 3 Punkte; Datei: `dict_invert.py`)

In dieser Aufgabe geht es um ein klassisches Dictionary Problem.

(a) **Invertieren; 2 Punkte**
Schreiben Sie eine Funktion `invert_dict(d)`, die ein Dictionary `d`, welches Objekte (Strings) einer Kategorie (String) zuweist, als Argument nimmt. Die Funktion soll ein neues Dictionary zurückgeben, bei dem jeder Kategorie eine Liste aller Objekte zugeordnet ist, die in `d` zu dieser Kategorie gehörten.

```
>>> d = {"Apfel": "Obst", "Karotte": "Gemüse", "Banane": "Obst"}
>>> invert_dict(d)
{'Obst': ['Apfel', 'Banane'], 'Gemüse': ['Karotte']}
```

(b) **Mehrere Schlüssel; 1 Punkte**
Schreiben Sie eine Funktion `multiple_keys(d)`, die `True` zurückgibt, wenn es in `d` mindestens einen Wert gibt, dem mehrere Schlüssel zugeordnet sind. Ansonsten soll `False` zurückgegeben werden. Für obiges Beispiel wäre das Ergebnis `True`, da der Wert `Obst` zwei Schlüssel hat.

```
>>> d = {"Apfel": "Obst", "Karotte": "Gemüse", "Banane": "Obst"}
>>> multiple_keys(d)
True
```

Aufgabe 9.5 (Erfahrungen; Datei: `NOTES.md`)

[4] Diese Methode ermöglicht die Umwandlung mit der Funktion `str()` und die Ausgabe mit `print`

Notieren Sie Ihre Erfahrungen mit diesem Übungsblatt (benötigter Zeitaufwand, Probleme, Bezug zur Vorlesung, Interessantes, etc.).

Editieren Sie hierzu die Datei `NOTES.md` im Abgabeordner dieses Übungsblattes auf unserer Webplattform. Halten Sie sich an das dort vorgegebene Format, da wir den Zeitbedarf mit einem Python-Skript automatisch statistisch auswerten. Die Zeitangabe `7.5 h` steht für 7 Stunden 30 Minuten.

Der Build-Server überprüft ebenfalls, ob Sie das Format korrekt angegeben haben. Prüfen Sie, ob der Build-Server mit Ihrer Abgabe zufrieden ist, so wie es im Video zur Lehrplattform gezeigt wurde.