Vorbemerkung

Ab dem Jahr 2026 wird im hessischen Landesabitur eine Pflichtaufgabe zum Halbjahresthema 'Algorithmik und objektorientierte Programmierung' in den Programmiersprachen Python und Java angeboten. Die Auswahl der Programmiersprache erfolgt im Rahmen der durch Erlass vorgegebenen Möglichkeiten durch die Fachkonferenz. Die Programmiersprache dient als Werkzeug und steht nicht im Mittelpunkt des Unterrichts.

Die Umsetzung der Inhalte des KCGOs zu den Halbjahresthemen 'Einführung in die Informatik' und 'Algorithmik und objektorientierte Programmierung' wird im Folgenden erläutert. Dieses Dokument und die Beispielaufgaben werden auch auf dem hessischen Bildungsserver ggf. aktualisiert und erweitert zu finden sein.

Bezeichner

In Python verwenden wir die üblichen Namenskonventionen[1].

- Klassenbezeichner beginnen mit einem Großbuchstaben und verwenden das CamelCase-Format.
- Module, Methoden und Variablen beginnen mit einem Kleinbuchstaben, wobei einzelne Wörter durch Unterstriche getrennt werden.

```
from knoten import Knoten

class BinBaum:

    def ip_von_domain(self):
        pass
```

Hinweis: `from knoten import Knoten` importiert aus dem Modul *knoten* die Klasse *Knoten*. Auf den Import ganzer Module soll verzichtet werden.
Bei Quellcode in Aufgaben des Landesabiturs werden wir in den meisten Fällen import Statements nicht explizit angeben.

Datenkapselung, Sichtbarkeit und get-/set-Methoden

Datenkapselung ist ein zentrales Konzept der objektorientierten Programmierung, auch in Python. Die Sichtbarkeit von Attributen bestimmt, ob auf sie von außerhalb der Klasse zugegriffen werden kann. Wir verwenden die folgenden Konventionen, um das Geheimnisprinzip zu wahren:

- Private Attribute und Methoden: Präfix '__'
- Geschützte Attribute und Methoden: Präfix '_'

Der Zugriff erfolgt über get- und set-Methoden, während die Nutzung von Properties im Landesabitur nicht vorgesehen ist.

1 PEP 8 – Style Guide for Python Code

Typinformationen

Python verwendet dynamische Typisierung, daher finden keine expliziten Variablendeklarationen statt. Der Datentyp ergibt sich aus der Initialisierung durch eine Wertzuweisung. Für UML-Diagramme und Quellcodedarstellungen nutzen wir Typinformationen (type hints) [2]. Diese Typinformationen sind in Python lediglich Hinweise und beeinflussen nicht das Verhalten der Laufzeitumgebung.

In Python können Typannotationen zu einem Fehler führen, wenn sie sich auf Typen beziehen, die zum Zeitpunkt ihrer Definition noch nicht bekannt sind. Dies ist besonders problematisch bei selbstreferenzierenden Klassen oder Klassen, die sich gegenseitig referenzieren. In der folgenden Beispielaufgabe befindet sich die selbstreferenzierende Klasse *Knoten*, die zwei Attribute vom Typ *Knoten* enthält. Die Verwendung von **from __future__ import annotations** löst dieses Problem, indem alle Typannotationen als Zeichenketten behandelt werden, die erst zur Laufzeit aufgelöst werden.
Auch dieses import Statement werden wir in Aufgaben des Landesabiturs nicht explizit angeben. Auch in Lösungen der Schülerinnen und Schüler wird dies nicht verlangt werden.

Beispielaufgabe

Überführen Sie die beiden Klassen in ein UML-Klassendiagramm.

Klasse BinBaum

```
from knoten import Knoten

class BinBaum:

    def ____init____(self):
        self._wurzel: Knoten = None

    def ip_von_domain(self, domain_name: str) -> str:
        # in Aufgabe 3 zu implementieren

    def ip_erfragen(self, domain_name: str) -> str:
        # hier muss ein übergeordneter DNS-Server angefragt werden
        # die Methode soll als implementiert angenommen werden
```

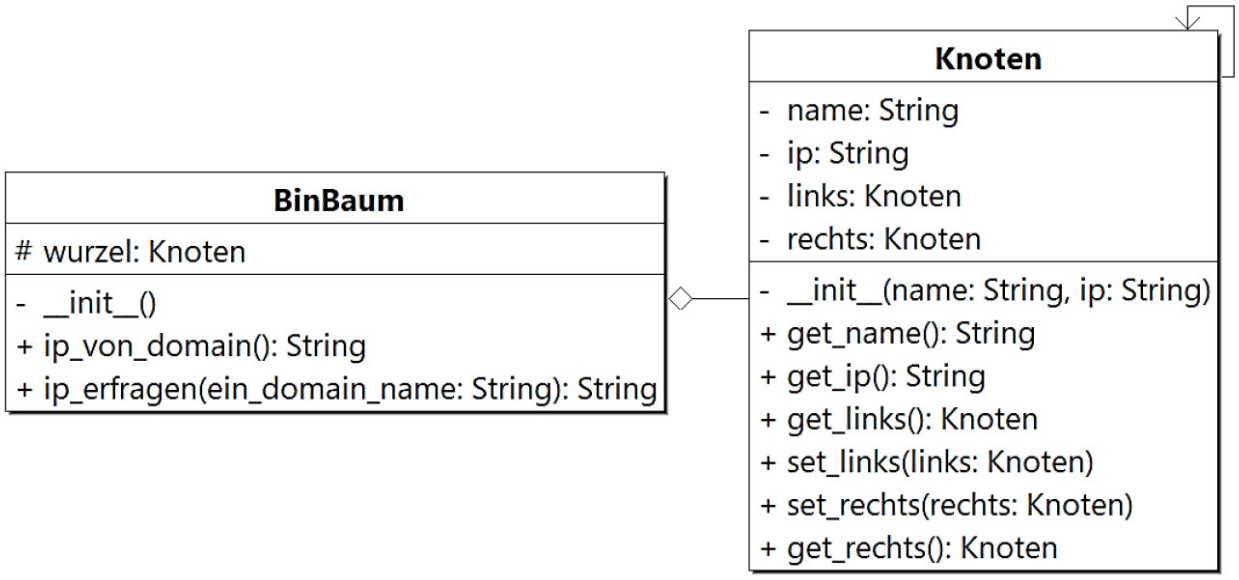
Klasse Knoten

```
from __future__ import annotations

class Knoten:

    def __init__(self, name: str, ip: str):
        self.__name: str = name
        self.__ip: str = ip
        self.__links: Knoten = None
        self.__rechts: Knoten = None

    def get_name(self) -> str:
        return self.__name

    def get_ip(self) -> str:
```

[2] Seit Version 3.6 sind die Typhinweise in Python möglich. Selbst Pythons Gründervater Guido van Rossum unterstützt das Vorgehen mit dem Type-Checker-Projekt mypy.

```
        return self.__ip

    def get_links(self) -> Knoten:
        return self.__links

    def set_links(self, links: Knoten):
        self.__links = links

    def get_rechts(self) -> Knoten:
        return self.__rechts

    def set_rechts(self, rechts: Knoten):
        self.__rechts = rechts
```

Lösung:



Typinformationen erleichtern auch die Analyse von Quellcode, da hierdurch die Bedeutung der Variablen verdeutlicht wird. Wir werden daher im Landesabitur in den Aufgabenstellungen und Lösungen stets Typinformationen angeben.

Beispielaufgabe:

Analysieren Sie die Methode *ausgeben(...)* aus dem Material.

Die Methode *ausgeben(…)* der Klasse *Arzneiverwaltung*

```
01 def ausgeben(self, arzneiname: str) -> bool:
02     for arznei in self.__arzneien:
03         if arznei.get_name() == arzneiname:
04             if arznei.get_anzahl() > 0:
05                 arznei.ändere_anzahl_um(-1)
06                 return True
07             else:
08                 return False
09     return False
```

Sortieralgorithmen

Sortieralgorithmen sind in vielen Anwendungen erforderlich. Die Lernenden sollen einfache Sortieralgorithmen entwickeln und diese beschreiben, darstellen und implementieren. Die vordefinierte Funktion *sort()* auf Listen soll dabei nicht verwendet werden. Der Dreieckstausch kann durch die in Python übliche direkte Vertauschung der Listeninhalte ersetzt werden.

Beispielaufgabe:

Implementieren Sie die Methode *sortiere_arzneien()* der Klasse *Arzneiverwaltung*, die die Arzneien alphabetisch nach Arzneinamen sortiert.
Hinweis: Die Methoden *sort()* und *sorted()* für Listen sollen in dieser Aufgabe nicht verwendet werden.

Lösung:

```
def sortiere_arzneien(self):
    for i in range(len(self.__arzneien) - 1):
        min = i
        for j in range(i + 1, len(self.__arzneien)):
            if (self.__arzneien[j].get_name()
                    < self.__arzneien[min].get_name()):
                min = j
        (self.__arzneien[i], self.__arzneien[min]) \
            = (self.__arzneien[min], self.__arzneien[i])
```

List Comprehensions

In Anbetracht der Lesbarkeit und Verständlichkeit des Codes präferieren wir traditionelle Schleifenkonstrukte gegenüber List Comprehensions. Letztere, obwohl sie für ihre Kompaktheit und Eleganz bekannt sind, können bei komplexer oder spezifischer Logik, die Codeverständlichkeit beeinträchtigen. Daher wird empfohlen, Schleifenkonstrukte direkt zu verwenden, um die Klarheit und Nachvollziehbarkeit des Codes zu gewährleisten.

Beispiel:

```
class Person:
    def __init__(self, name: str, alter: str):
        self._name = name
        self._alter = alter

personen = [Person("Ada", 25), Person("Bernd", 30), Person("Charlie", 20)]
```

Mit List Comprehensions

```
namen_unter_30 = [person._name for person in personen if person._alter < 30]
```

Ohne List Comprehensions

```
namen_unter_30 = []
for person in personen:
    if person._alter < 30:
        namen_unter_30.append(person._name)
```

weiteres Beispiel:

```
def counting_sort(self):
    anzahl_abstände: int = 7

    #so:
    c: list[int] = [] #c hat so viele Einträge,
    #wie es der maximale Abstand im Gitter ermöglicht +1 (6 + 1).
    for i in range(anzahl_abstände):
        c.append(0)
    #und nicht so:
    c = [0 for i in range(anzahl_abstände)]

    #so:
    b: list[Auto] = [] #Diese Liste wird sortiert aufgebaut
    for i in range(len(self.__verfügbare_autos)):
        b.append(None)
    #und nicht so:
    b = [None for i in range(len(self.__verfügbare_autos))]

    for i in range (len(self.__verfügbare_autos)):
        abstand: int =  self.__verfügbare_autos[i].bestimme_abstand()
        c[abstand] = c[abstand] + 1

    for i in range (anzahl_abstände - 1):
        c[i] = c[i] + c[i-1]

    for i in range (len(self.__verfügbare_autos)):
        abstand: int = self.__verfügbare_autos[i].bestimme_abstand()
        b[c[abstand]-1] = self.__verfügbare_autos[i]
        c[abstand] = c[abstand]-1

    self.__verfügbare_autos = b
```

Slicing

Python bietet slicing als Operation auf Listen an. Slicing ermöglicht es häufig, Algorithmen sehr effizient zu implementieren, kann dabei aber auch die Lesbarkeit des Quellcodes verringern. Aus diesem Grund verzichten wir vollständig auf Slicing und gehen wie oben unter List Comprehensions beschrieben vor. Schülerinnen und Schüler dürfen bei Implementierungen in Abituraufgaben Slicing verwenden.

Beispiel:

```
class Senso:

    def __init__(self):
        self.__farbfolge: list[Licht] = []
        self.__farbfolge.append(Licht("blau"))
        self.__farbfolge.append(Licht("gelb"))
        self.__farbfolge.append(Licht("rot"))
        self.__farbfolge.append(Licht("gelb"))
```

Mit Slicing

```
    def spielen(self):
        for runde in range(len(self.__farbfolge)):
            print("Runde ", runde + 1)
            for licht in self.__farbfolge[:runde+1]:
                licht.aufblinken()
```

Ohne Slicing

```
    def spielen(self):
        for runde in range(len(self.__farbfolge)):
            print("Runde ", runde + 1)
            for i in range(runde + 1):
                self.__farbfolge[i].aufblinken()
```

Lesbarkeit von Quellcode

Insgesamt werden wir eine bessere Lesbarkeit des Quellcodes gegenüber dessen Kürze oder Eleganz vorziehen. Dies gilt insbesondere, wenn ein Quellcode analysiert werden soll.

Es steht Lehrerinnen und Lehrern jedoch hierbei frei, in ihren Kursen den Schwerpunkt anders zu setzen. Schülerinnen und Schüler sind bei Implementierungen in Abituraufgaben nicht verpflichtet diesen Grundsatz zu befolgen. Die Priorisierung der Lesbarkeit soll lediglich als Empfehlung dienen, um das Verständnis und die Nachvollziehbarkeit von Quellcode zu fördern.

Beispiel:

Kurz

```
  def erstes_element(self) -> Element:
      return self.__feld[0]
```

Besser lesbar

```
  def erstes_element(self) -> Element:
      if self.__feld:
          return self.__feld[0]
      else:
          return None
```

Listen in Python

In Python existiert nicht direkt die Datenstruktur Feld (Array), wie sie in anderen Programmiersprachen wie Java oder Delphi vorkommt. Stattdessen wird ein List-Objekt verwendet, das Methoden für grundlegende Operationen bereitstellt. Während in Python ein indexbasierter Zugriff auf List-Objekte möglich ist, fordert die Vergrößerung eines Feldes nicht das Neuerzeugen und Umkopieren der bereits vorhandenen Inhalte, wie es in Java nötig ist. Stattdessen werden in Python integrierte Methoden wie *append*, *pop*, … verwendet, um die Inhalte eines Feldes zu manipulieren.

Da die Datenstruktur Feld eine fundamentale in der Informatik ist, soll diese weiterhin auch im Unterricht mit Python thematisiert werden. Für die Aufgaben im Landesabitur hat dies folgende Konsequenzen:

Soll die Datenstruktur im Mittelpunkt stehen, wie beispielsweise. im Themenfeld der Sortieralgorithmen, wird in der Regel auf einem gegebenen Feld indexbasiert operiert. Wenn die Datenstruktur Feld hingegen nur zum Speichern von Inhalten verwendet werden, wird dies in Python mithilfe der zur Verfügung stehenden Methoden implementiert.

Beispiel:

Feld steht als Datenstruktur nicht im Vordergrund

Die Methoden *einfügen()* und *erstes_löschen()* werden mit Hilfe der Methoden *append()* und *pop()* implementiert.

```
class Element:

    def __init__(self, nachname: str, vorname: str, ist_notfall: bool):
        self.__nachname: str = nachname
        self.__vorname: str = vorname
        self.__ist_notfall: bool = ist_notfall

    def get_nachname(self) -> str:
        return self.__nachname

    def get_ist_notfall(self) -> bool:
        return self.__ist_notfall

class Warteschlange:

    def __init__(self):
        self.__feld: list[Element] = []

    def ist_leer(self) -> bool:
        return self.__feld == []

    def einfügen(self, nachname: str, vorname: str, ist_notfall: bool):
        self.__feld.append(Element(nachname, vorname, ist_notfall))

    def erstes_element(self) -> Element:
        if self.__feld:
            return self.__feld[0]
        return None

    def erstes_löschen(self):
        if self.__feld:
            self.__feld.pop(0)
```

Feld ist zentraler Inhalt der Aufgabe

Die Methoden *einfügen_mit_notfall(…)* und *sortiere_nach_nachnamen()* werden ohne Zuhilfenahme von integrierten Python-Methoden implementiert.

```
    def einfügen_mit_notfall(self, nachname: str, vorname: str,
            ist_notfall: bool):
        if not ist_notfall:
            self.einfügen(nachname, vorname, ist_notfall)
        else:
            i: int = len(self.__feld)
            while i > 0 and not self.__feld[i-1].get_ist_notfall():
                i = i - 1
            self.__feld.insert(i, Element(nachname, vorname, ist_notfall))

    def sortiere_nach_nachnamen(self):
        for i in range(len(self.__feld), 0, -1):
            for j in range(i-1):
                if self.__feld[j].get_nachname() >
                        self.__feld[j+1].get_nachname():
                    self.__feld[j], self.__feld[j+1] = self.__feld[j+1],
                        self.__feld[j]
```