Straßenkarten als Graphen

Allgemeines

- Da dieser Versuch aus mehreren Quelldateien und XML-Dateien besteht. Speichern Sie diese in ein Verzeichnis mit dem Namen:
  `p04NameVorname`

  Name ist durch Ihren Nachnamen und Vorname durch Ihren Vornamen zu ersetzen. Der Verzeichnisname sollte weiterhin keine Leerzeichen enthalten.

  Geben Sie den Versuch dann als gepackte ZIP-Datei `p04NameVorname.zip` ab, in dem der Inhalt des o. g. Verzeichnisses enthalten ist.

- Verwenden Sie zur Programmentwicklung als Modulnamen (Dateinamen) für dieses Praktikum `node1.py` (Aufgabe 1.1), `node2.py` (Aufgabe 1.2), `node3.py` (Aufgabe 1.3-2.1). Wobei `node2.py` eine Verbesserung von `node1.py` darstellt und `node3.py` eine Verbesserung von `node2.py`.
  In Aufgabe 2 wird die Klasse Edge im Modul `edge.py` entwickelt.
  Ab Aufgabe 2.2 wird aus `node3.py` dann `node.py`. Damit ist dann `node.py` fertiggestellt.
  In Aufgabe 3 kommt dann noch die Klasse `Graph` hinzu, die in `graph.py` entwickelt wird.

- Bevor Sie Ihre Lösung den Betreuern vorstellen, können Sie mit dem in moodle bereitgestellten Unit-Test `p04unit.py` überprüfen, ob Ihre Lösung die richtigen Ergebnisse liefert.
  - Im Anaconda-Prompt können Sie den Unit-Test in Ihrem Projektverzeichnis über folgende Anweisung ausführen:
    `python p04unit.py`
  - Achten Sie **haargenau** auf die in der Aufgabenstellung dargestellte Ausgabe:
    Ein Leerzeichen zu viel, Groß-/Kleinschreibung nicht beachtet oder falsche Zeilenvorschübe gibt der Unittest als Fehler aus!
- Schreiben Sie Ihre Funktionstests so, dass diese nicht bei Aufruf des Unit-Tests aufgerufen werden.
  Die geforderten Funktionstests finden Sie in den Teilaufgaben angegeben.

- Sollten in der Aufgabenstellung Fragen zu beantworten sein, dann beantworten Sie diese bitte in einem Dokumentationskommentar am Anfang Ihre Python-Datei, z. B.:

```
"""
      Antworten zu den in der Aufgabenstellung gestellten
      Fragen:
      Frage 1: Ihre Antwort
      Frage 2: Ihre Antwort
      ...........
"""
```

Straßenkarten als Graphen

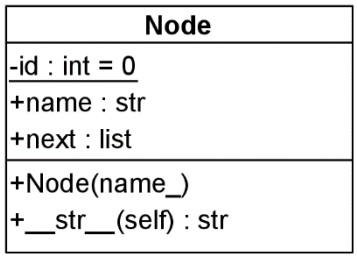
Eine Straßenkarte soll im Rechner repräsentiert werden, als grundlegende Datenstruktur soll ein Graph zum Einsatz kommen. Die Knoten des Graphen stehen dabei für Straßenkreuzungen (und Einmündungen), die Kanten für die Straßen zwischen den Kreuzungen. Der Einfachheit halber sind die Kanten gerichtet (können also als Einbahnstraßen interpretiert werden).

Sowohl die Knoten als auch die Kanten besitzen einen unveränderlichen Namen, die Kanten zusätzlich Gewichte (die der Weglänge entsprechen). Auch weitere Informationen (wie. z. B. die Art der Straße, Geschwindigkeitsbeschränkungen etc.) können sinnvoll sein.

Für die Arbeit mit Karten sind neben den genannten Daten auch eine Reihe von Operationen wichtig. Um die Übersicht zu behalten und den Nutzern eine einfache Programmierschnittstelle zu bieten, empfiehlt sich die Organisation in Klassen.

1. Aufbau der Klasse Node

1.1. Implementieren Sie die Klasse `Node`, deren Struktur in folgendem Klassendiagramm abgebildet ist. `next` ist dabei eine Liste der Nachfolgeknoten. `id` eine ganze Zahl (Sie können die **statische** Klassenvariable zunächst einfach auf 0 setzen).
Konstruieren Sie in der Methode `__str__` zuerst einen String, der die nötigen Informationen für den Knoten beinhaltet. Speichern Sie die Klasse im Modul `node1.py`.



Testen Sie die Klasse nach der Klassendefinition
(ohne dass der Test im Unittest mit-aufgerufen wird!)

```
n1 = Node('A')
n2 = Node('B')
n3 = Node('C')

n1.next.append(n2)
n1.next.append(n3)

print(n3.name)
print(n1)
print(n2)
```

Folgende Ausgabe soll sich dabei ergeben:

```
C
A ---> B
  ---> C
B <end>
```

1.2. Kopieren Sie nun die Datei `node1.py` in die Datei `node2.py`.
Modifizieren Sie die Klasse Node der letzten Teilaufgabe so, dass `name` eine nur lesbare Eigenschaft (des entsprechenden privaten Attributs) darstellt.
Test innerhalb von `node2.py`:

```
n1 = Node('A')
print(n1.name)
n1.name = 'B'        # die Zeile soll zu einem Fehler führen!
```

1.3. Kopieren Sie nun die Datei `node2.py` in die Datei `node3.py`.
Modifizieren Sie die Klasse Node der letzten Teilaufgabe, so dass es möglich ist, den Konstruktor auch ohne Angabe eines Knotennamens aufzurufen. In diesem Fall soll der (eindeutige) Standardname "`Knoten XXX`" verwendet werden, wobei XXX für die laufende Nummer des Knotens steht.

1.4 Modifizieren Sie die Klasse Node des Moduls `node3.py` so, dass das Hinzufügen eines Nachfolgeknotens nur noch über die Methode `connect` möglich ist.

Testen Sie die Klasse z. B. mittels[1]

```
n1 = Node()
n2 = Node('B')
n3 = Node()
n1.connect(n2)
n1.connect(n3)
print(n1)
print(n2)
```

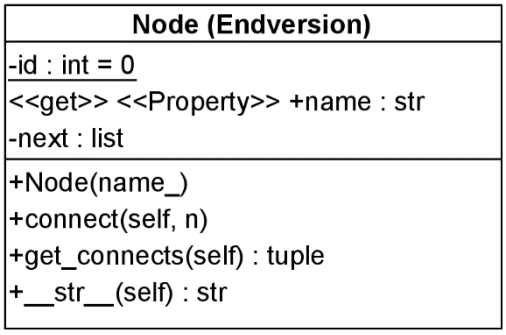
Folgende Ausgabe soll sich dabei ergeben:

```
Knoten 1 ---> B
         ---> Knoten 3
B <end>
```

Beachten Sie, dass die Pfeildarstellungen abhängig vom Knotennamen in derselben Spalte beginnen!

1.5. Ergänzen Sie die Klasse Node des Moduls `node3.py` um die Methode `get_connects`, die die Listenelemente der Nachfolgeknoten als `tuple` zurückgibt.

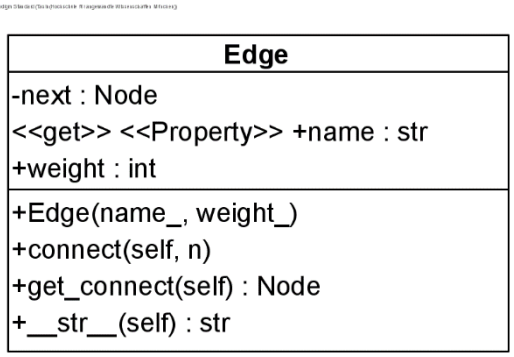
Das endgültige Klassendiagramm der Klasse Node ist damit



[1] wie bereits erwähnt: Ohne dass der Test im Unittest mitaufgerufen wird.

2. Die Klasse Edge

2.1. Implementieren Sie die Klasse `Edge`, deren Struktur in folgendem Klassendiagramm abgebildet ist. `connect` setzt dabei den (maximal einen) Zielknoten der Kante (der Ausgangsknoten wird hier zunächst nicht betrachtet).
`get_connect` gibt den Zielknoten zurück. `name` stellt wie in der Klasse `Node` eine nur lesbare Eigenschaft dar, Speichern Sie die Klasse im Modul `edge.py`.



Testen Sie die Klasse mittels

```
import node3
n1 = node3.Node('A')
e1 = Edge('E', 5)
e1.connect(n1)
print('vorher:', e1)
e1.weight=3
print('nachher:', e1)
```

Folgende Ausgabe soll sich dabei ergeben:

```
vorher: E/5
nachher: E/3
```

2.2. Kopieren Sie nun die Datei `node3.py` in die Datei `node.py`. Modifizieren sie dann die Klasse `Node`, so dass die Methode `connect` nun jeweils eine Kante mit dem Knoten verbindet (und dadurch eine andere Ausgabe liefert).

Testen Sie die Klasse mittels

```
n1 = node.Node('A')
n2 = node.Node('B')
n3 = node.Node('C')
e1 = edge.Edge('E', 5)
e2 = edge.Edge('F', 2)
n1.connect(e1)
n1.connect(e2)
e1.connect(n2)
e2.connect(n3)
print(n1)
print(n2)
```

Folgende Ausgabe soll sich dabei ergeben:

```
A --E/5--> B
  --F/2--> C
B <end>
```

3. Die Klasse Graph

3.1. Implementieren Sie eine Klasse `Graph`, die eine Liste von Knoten und eine Liste von Kanten enthält, in der Datei `graph.py`.

Das folgende Testprogramm soll mit der Klasse Graph durchführbar sein:

```
g = Graph()
n1 = g.new_node('A')
n2 = g.new_node()
e1 = g.new_edge('E', 5)
n1.connect(e1)
e1.connect(n2)
print(g)
```

Folgende Ausgabe soll sich dabei ergeben:

```
Knoten:
-------
A --E/5--> Knoten 2
Knoten 2 <end>

Kanten:
-------
E/5
```

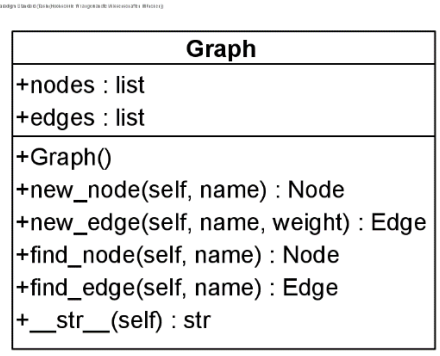
3.2. Erweitern Sie die Klasse `Graph` um die Methoden `find_node(self, name)` und `find_edge(self, name)`, die den Knoten mit dem angegebenen Namen zurückgibt. Testen Sie die Methode z. B. mit `print(g.find_node('Knoten 2'))`.

Tipp: Eine mögliche Implementierung von `find_edge` ist:

```
def find_edge(self, name):
    r = None
    for e in self.edges:
        if e.name == name:
            r = e
            break
    return r
```

Für Profis: Versuchen Sie die Implementierung mit der internen Funktion `next`. Recherchieren Sie die Funktionsweise von `next` im Internet.

Insgesamt sieht das Klassendiagramm der Klasse `Graph` dann wie folgt aus:

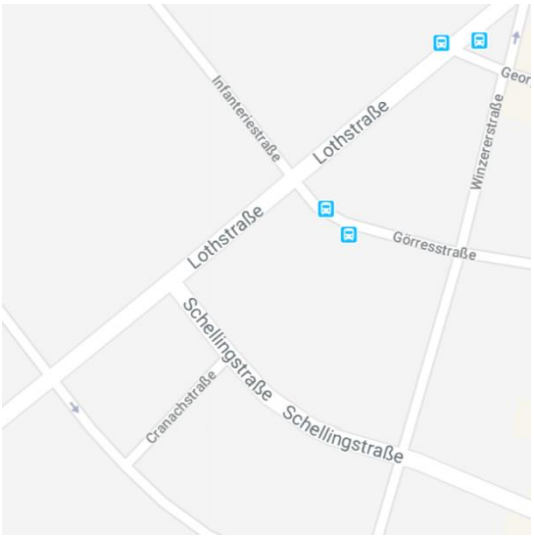


3.3 Unter einem *Pfad* versteht man eine Folge im Graph benachbarter Knoten. Erweitern Sie die Klasse `Graph` um die Methode `path_length(self, path_node_names)`, die die Knotennamen des Pfades als Liste akzeptiert und die Gesamtlänge des Pfades ermittelt.
Existiert so ein Pfad nicht, soll `-1` zurückgegeben werden. Testen Sie die Methode an einem selbst definierten Graphen.
Tipp: Es kann sinnvoll sein, eine statische Methode `find_edge_between(n1, n2)` einzuführen, die im Graphen eine Kante (Edge-Objekt) sucht die die beiden Node-Objekte n1 und n2 verbindet. Falls eine derartige Kante nicht existiert gibt sie None zurück.

4. Die bereitgestellte Klasse `Reader` und XML

Die Ihnen zur Verfügung gestellte Klasse `Reader` im Modul `reader.py` implementiert eine Lesemethode für XML-Dateien, die einen Graphen in textueller Form darstellen.
Reader besitzt dabei die Methode `read`, die als Argument den Pfad zur XML-Datei erwartet und den Graphen als Instanz der Klasse `Graph` zurückgibt. `Reader` importiert dazu die Klasse `Graph` und nutzt deren Methoden und Attribute.

4.1. Betrachten Sie die beigefügten beispielhaften XML-Dateien und versuchen Sie, sie zu verstehen. Sie können sich diese in einem Text-Editor ansehen z. B. auch über Spyder.
Erstellen Sie eine eigene XML-Datei, die den folgenden Ausschnitt einer Straßenkarte repräsentiert. (Die Kreuzungen können beliebig benannt werden. Es reicht, Einbahnstraßen zu modellieren; achten Sie aber auf die Möglichkeit, von der Kreuzung Winzererstr./Görresstr. zu beliebigen Stellen der Karte zu gelangen.) Speichern Sie die Karte unter dem Namen `Karte.xml`. Die Gewichte sind dabei die Längen der Teilstrecken in Meter, die frei (aber vernünftig) gewählt werden können.
Tipp: Sie können die Längen auch über Google-Maps nachmessen.



4.2. Schreiben Sie ein Programm, das `Karte.xml` einliest und eine Liste der Knoten und Kanten (wie in Teilaufgabe 3.1. definiert) ausgibt.

4.3. *Für Profis (freiwillig):* Implementieren Sie eine Methode `find_path(self, start, target)` der Klasse Graph, die einen Weg von einem Knoten `start` zu einem Knoten `target` findet. Der Weg muss nicht optimal sein. Testen Sie Ihre Methode, indem Sie einen Weg von der Kreuzung Winzererstr./Görresstr. zur Kreuzung Schellingstr./Cranachstr. suchen.

Tipp: Versuchen Sie, das Problem, von A nach B zu kommen, (rekursiv) durch einfachere Probleme auszudrücken. Ein zusätzliches Attribut `is_visited` in der Klasse `Node` (oder ein anderes geeignetes Attribut in der Klasse `Graph`) kann helfen.
Zum Testen benötigen Sie die `Node`-Objekte `start` und `target`, die Sie z. B. mit der Methode `Graph`/`find_node` aus dem Kontennamen ermitteln können.

Bemerkung:
Um in einem Graphen, dessen Kanten keine negativen Gewichte besitzen, den (bzw. einen) kürzesten Pfad von einem Startknoten zu einem Endknoten zu finden, kann man den Algorithmus von **Dijkstra** verwenden. Dabei handelt es sich um einen Greedy-Algorithmus der die spezielle Datenstruktur Priority Queue verwendet. Er wird im Detail im FWP2 Wahlfach ***Algorithmendesign und höhere Datenstrukturen*** des Kollegen Prof. Dr. Ressel besprochen.

4.4. *Für Profis (freiwillig):* Das bisherige Klassendesign berücksichtigt nicht vollständig die *Kapselung* von Daten innerhalb einer Klasse: Auf Klassenattribute sollte gar nicht direkt von außen zugegriffen werden (dafür gibt es Properties bzw. Getter und Setter), bei Methoden sollte zwischen privaten (internen) und öffentlichen unterschieden werden. Überprüfen Sie Ihr Design in Hinblick auf Kapselung und modifizieren Sie es entsprechend. Passen Sie auch die Klassendiagramme an.