Coding in GIS

Nils Ratnaweera

Sep 07, 2020

CODING IN GIS I

Dieser kurze Kurs ist Bestandteil des übergreifenden Moduls "Angewandte Geoinformatik" der Zürcher Hochschule für Angewandte Wissenschaften (ZHAW). Er soll einen Einstieg in die Programmierwelt von Python bieten und spezifisch zeigen wie man räumliche Fragestellungen mit frei verfügbaren Programmen lösen kann. Die Voraussetzung für dieser Kurs ist eine Offenheit neue Tools und Ansätze kennen zu lernen sowie die Bereitschaft für Lösungorientiertes arbeiten und etwas Hartnäckigkeit.

Das github-repo ist hier ratnanil/codingingis.

Online und pdf

Dieses Buch ist als online version hier gehostet: https://ratnanil.github.io/codingingis. Es kann aber auch als pdf hier heruntergelanden werden.

CHAPTER

ONE

TYPOGRAPHIE

Noch ein kurzer Hinweis zur Typografie dieses Dokumentes:

- Wenn sich im Fliesstext (Python- oder R-) Code befindet, wird er in `dieser Festschriftart` dargestellt
- Alleinstehende Codezeilen werden folgendermassen dargestellt:

```
print("Coding in GIS!")
```

- Englische Fachbegriffe, deren Übersetzung eher verwirrend als nützlich wäreb, werden *in dieser Weise* hervorgehoben
- Da viele von euch bereits Erfahrung in R haben, stelle ich immer wieder den Bezug zu dieser Programmiersprache her. Diese Verweise sind folgendermassen gekennzeichnet:

Für R Nutzer

- hier wird ein Bezug zu R gemacht

1.1 Einleitung zu diesem Block

!EIN PAAR WORTE ZUR HEUTIGEN ÜBUNG!

Übungsziele

- JupyterLabs aufstarten, kennenlernen und bei Bedarf personalisieren
- Python kennen lernen, erste Interaktionen
- Die wichtigsten Datentypen in Python kennen lernen (bool, text, integer, float, list, dictionary, dataframe)
- Pandas DataFrames kennen lernen und einfache Manipulationen durchführen
- Geodaten (Vektor) in Python kennenlernen
- Geodaten in Python einfach visualisieren können

1.2 Primitive Datentypen

Bei primitiven Datentypen handelt es sich um die einfachste Datenstruktur. Sie werden deshalb auch "atomare Datentypen" genannt: Alle komplexeren Datentypen (Tabellarische Daten, Bilder, Geodaten) basieren auf diesen einfachen Strukturen.

1.2.1 Boolean

Hierbei handelt es sich um den einfachsten Datentyp. Er beinhaltet nur zwei Zustände: Wahr oder Falsch. In Python werden diese mit `True` oder `False` definiert. Beispielsweise sind das Antworten auf geschlossene Fragen.

```
regen = True
```

```
sonne = False

type(sonne)
```

```
bool
```

1.2.2 String

In sogenannten *Strings* werden Textinformationen gespeichert. Beispielsweise können das die Namen von Ortschaften sein.

```
stadt = "Bern"
land = "Schweiz"
```

```
type(stadt)
```

```
str
```

Strings können mit + miteinander verbunden werden

```
stadt +" ist die Hauptstadt der " + land
```

```
'Bern ist die Hauptstadt der Schweiz'
```

1.2.3 Integer

In Integerwerten werden ganzzahlige Werte gespeichert, beispielsweise die Anzahl Einwohner einer Stadt.

```
bern_einwohner = 133115
```

```
type(bern_einwohner)
```

```
int
```

1.2.4 Float

Als `Float` werden Zahlen mit Nachkommastellen gespeichert, wie zum Beispiel die Temeratur in Grad Celsius.

```
bern_temp = 22.5
```

```
type(bern_temp)
```

```
float
```

1.2.5 Weitere Datentypen

Es gibt noch weitere Datentypen, auf die wir an dieser Stelle jedoch nicht eingehen.

1.3 Zusammengesetzte Datentypen

Basierend auf den "atomaren" oder primitiven Datentypen exisiteren komplexere Datenstrukturen, die aber an sich immer noch sehr einfach sind.

1.3.1 List

Eine `List`:

- Enthält mehrere Werte
- Die Reihenfolge wird gespeichert
- auch unterschiedliche Datentypen möglich

```
hexerei = [3,1,2]
```

Der erste Wert wird in Python mit `0` aufgerufen:

```
hexerei[0]
```

```
3
```

```
type(hexerei)
```

```
list
```

In einer List können verschiedene Datentypen enthalten sein, auch weitere, verschachtelte Lists

```
chaos = [23, "ja", [1,2,3]]
```

```
# Der Inhalt vom ersten Wert ist vom Typ "Int"
type(chaos[0])
```

```
int
```

```
# Der Inhalt vom dritten Wert ist vom Typ "List"

type(chaos[2])
```

```
list
```

1.3.2 Dict

Eine `Dict`:

- Enthält mehrere Werte
- Reihenfolge wird nicht gespeichert
- jeder Wert («Value») hat einen Schlüssel («Key»)
- unterschiedliche Datentypen möglich

```
menue = {"Vorspeise": "Suppe",
         "Hauptspeise": "Gratin",
         "Dessert": "Eis"}
```

```
menue["Vorspeise"]
```

```
'Suppe'
```

Auch hier sind verschiedene Datentypen möglich:

```
menue = {"Vorspeise": "Suppe",
         "Hauptspeise": ["Gratin","Spinat", "Salat"],
         "Dessert": "Himbeerglacé",
         "Preis": 50}
```

```
type(menue)
```

```
dict
```

1.3.3 DataFrame

Bei den bisherigen Datentypen handelte es sich um Strukturen, die in der Standartinstallation von Python enhalten sind. Tabellarische Strukturen sind in Python, im Gegensatz zu R, nicht standardmässig vorhanden. Dazu brauchen wir eine Erweiterung zu Python: Was es sich damit auf sich hat und wie diese installiert wird erfahren wir später. An dieser Stelle möchte ich nur die Struktur `DataFrame` vorstellen. `DataFrames` sind:

- Tabellarische Daten
- Ein Spezialfall einer Dict:
  - `values` sind `Lists` von gleicher Länge
  - jede `List` besteht aus einem einzigen Datentyp
- jeder `key` ist ein Spaltenname

```
import pandas as pd
```

```
menue2 = {"Typ": ["Vorspeise","Hauptspeise","Dessert"],
          "Beschreibung": ["Suppe","Gratin mit Spinat", "Himbeerglacé"],
          "Preis": [7.50, 32.0, 10.50]}

menue2
```

```
{'Typ': ['Vorspeise', 'Hauptspeise', 'Dessert'],
 'Beschreibung': ['Suppe', 'Gratin mit Spinat', 'Himbeerglacé'],
 'Preis': [7.5, 32.0, 10.5]}
```

```
type(menue2)
```

```
dict
```

```
menue_df = pd.DataFrame(menue2)

menue_df
```

```
           Typ       Beschreibung  Preis
0    Vorspeise              Suppe    7.5
1  Hauptspeise  Gratin mit Spinat   32.0
2      Dessert       Himbeerglacé   10.5
```

```
type(menue_df)
```

```
pandas.core.frame.DataFrame
```

1.4 Python Basics

1.4.1 Übung 1: Variablen erstellen

Wir beginnen damit, einfache Datentypen wie Zahlen, Text, boolsche Variabeln (siehe Vorlesungsfolien) sogenannten Variablen zu zuweisen. Führe die folgenden Schritte aus und schau dir nach jedem Schritt die erstellten Variablen jeweils im im Variable explorer von Spyder an

1. Erstelle eine Variabel `vorname` mit deinem Vornamen
2. Erstelle eine zweite Variabel `nachname` mit deinem Nachnamen
3. Was sind `vorname` und `nachname` für Datentypen?
4. Klebe die beiden Variabeln zusammen, ohne den Output zu speichern
5. Erstelle eine Variabel `groesse` mit deiner Körpergrösse in Zentimeter. Was ist das für ein Datentyp?
6. Ermittle deine Grösse in Meter auf der Basis von `groesse`. Was ist das für ein Datentyp?
7. Erstelle eine boolsche Variable blond und setzte sie auf `True` wenn diese Eigenschaft auf dich zutrifft und `False` falls nicht.

```
vorname = "Guido"
nachname = "van Rossum"

type(vorname) # es handelt sich um den Datentyp "str", also String (Text)
vorname+nachname

groesse = 184
type(groesse) # es handelt sich hierbei um den Datentyp "integer"

groesse_meter = groesse/100
type(groesse_meter) # es handelt sich um den Datentyp "float"

blond = False
```

Für R Nutzer

- In R gibt es die gleichen Primitiven Datentypen wie in Python: Boolean, String, Integer, Float. Die Bezeichnung ist jedoch leicht anders (Logical, Character, Integer, Numeric, Double)
- In R können Variablen genau gleich zugewiesen werden wie in Python (mit dem = Symbol). In R gibt es zudem noch die Möglichkeit, Variablen mit <− zu zuweisen, in Python jedoch nicht

1.4.2 Übung 2: Lists

1. Erstelle eine Variable `vornamen` bestehend aus einer *List* mit 3 Vornamen
2. Erstelle eine zweite Variable `nachnamen` bestehend aus einer *List* mit 3 Nachnamen
3. Erstelle eine Variable `groessen` bestehend aus einer *List* mit 3 Grössenangaben in Zentimeter.

Für R Nutzer

Eine Python List entspricht eines R Vectors

```
vornamen = ["Christopher", "Henning", "Severin"]
nachnamen = ["Annen","May", "Kantereit"]

groessen = [174, 182, 162]
```

1.4.3 Übung 3: Elemente aus Liste ansprechen

Wie erhältst du den ersten Eintrag in der Variable `vornamen`? Wie erhältst du den letzten Eintrag? Tipp: nutze dazu `[` und `]` sowie eine Zahl.

Für R Nutzer

In R werden Elemente aus Vectors auf die gleiche Weise extrahiert. Nur ist in R das erste Element die Nummer 1, Python beginnt bei 0

1.4.4 Übung 4: Liste ergänzen

Listen können durch der Method `append` ergänzt werden.

```
vornamen.append("Malte")
```

Ergänze in die Listen `vornamen`, `nachnamen` und `groessen` durch je einen Eintrag.

```
nachnamen.append("Huck")

groessen.append(177)
```

1.4.5 Input: Dictionary (Teil I)

Ähnlich wie eine List, ist eine Dictionary ein Behälter wo mehrere Elemente abgespeichert werden können. Wie bei einem Wörterbuch bekommt jedes Element ein "Schlüsselwort", mit dem man den Eintrag finden kann. Unter dem Eintrag "trump" findet man im Langenscheidt Wörterbuch (1977) die Erklärung "erdichten, schwindeln, sich aus den Fingern saugen".


III. v/i. 5. trumpf ausspielen, trumpfen, stechen.
trump² [trʌmp] v/t. ~ up erdichten, zs.-schwindeln, sich aus den Fingern saugen; 'trumped-up [trʌmpt] adj. erfunden, erlogen, falsch: ~ charges.
trump·er·y ['trʌmpəri] I. s. 1. Plunder m, Ramsch m, Schund m; 2. fig. Gewäsch n, Quatsch m; II. adj. 3. Schund..., Kitsch..., kitschig, geschmacklos; 4. fig. billig, nichtssagend: ~ arguments.
trum·pet ['trʌmpit] I. s. 1. ♪ Trom'pete f: to blow one's own ~ fig. sein eigenes Lob singen; the last ~ die Posaune des Jüngsten Gerichts; 2.

In Python würde man diese *Dictionary* folgendermassen erstellen:

```
langenscheidt = {"trump":"erdichten- schwindeln- sich aus den Fingern saugen"}
```

Schlüssel (von nun an mit *Key* bezeichnet) des Eintrages lautet "trump" und der dazugehörige Wert (*Value*) "erdichten- schwindeln- aus den Fingern saugen". Beachte die geschweiften Klammern (`{` und `}`) bei der Erstellung einer Dictionary.

Eine *Dictionary* besteht aber meistens nicht aus einem, sondern aus mehreren Einträgen: Diese werden Kommagetrennt aufgeführt.

```
langenscheidt = {"trump":"erdichten- schwindeln- sich aus den Fingern saugen",
↪"trumpery":"Plunder- Ramsch- Schund"}
```

Der Clou der *Dictionary* ist, dass man nun einen Eintrag mittels dem *Key* aufrufen kann. Wenn wir also nun wissen wollen was "trump" heisst, ermitteln wir dies mit der nachstehenden Codezeile:

```
langenscheidt["trump"]
```

```
'erdichten- schwindeln- sich aus den Fingern saugen'
```

Für R Nutzer

Eine Python *Dictionary* entspricht einer R *List*. Diese werden in R nicht gleich erstellt, dafür aber auf die gleiche weise abgefragt wie *Dictionaries* in Python

1.4.6 Übung 5: Dictionary

Erstelle eine *Dictionary* mit folgenden Einträgen: Vorname und Nachname von (d)einer Person. Weise diese Dictionary der Variable `me` zu.

```
me = {"vorname": "Guido", "nachname": "van Rossum"}
```

1.4.7 Übung 6: Elemente aus Dictionary ansprechen

Rufe verschiedene Elemente aus der Dictionary via dem *Key* ab.

```
me["vorname"]

me["nachname"]
```

```
'van Rossum'
```

1.4.8 Übung 7: Key ergänzen

Um einer *Dictionary* mit einem weiteren Eintrag zu ergänzen, geht man sehr ähnlich vor wie beim Abrufen von Einträgen.

```
langenscheidt["trumpet"] = "trompete"
```

Ergänze gemäss nachstehendem Beispiel die Variable `me` durch den Eintrag `groesse`.

Für R Nutzer

Python *Dictionaries* werden nicht nur gleich abgerufen wie R *Lists*, sonder auch gleich ergänzt.

```
me["groesse"] = 181
```

1.4.9 Input: Dictionary (Teil II)

Ein *Key* kann auch mehrere Einträge enthalten. An unserem Langenscheidts Beispiel: Das Wort "trump" ist zwar eindeutig, doch "trumpery" hat vier verschiedene Bedeutungen. In so einem Fall können wir einem Eintrag auch eine *List* von Werten zuweisen. Beachte die Eckigen Klammern und die Kommas, welche die Listeneinträge voneinander trennt.

```
langenscheidt["trumpery"] = ["Plunder- Ramsch- Schund", "Gewäsch- Quatsch", "Schund-␣
↪Kitsch", "billig- nichtssagend"]
langenscheidt["trumpery"]
```

```
['Plunder- Ramsch- Schund',
 'Gewäsch- Quatsch',
 'Schund- Kitsch',
 'billig- nichtssagend']
```

1.4.10 Übung 8: Dictionary mit List

Erstelle eine neue Dictionary mit den gleichen Keys wie `me`, aber diesmal mit mehreren Einträgen pro *Key* (also mehreren Vornamen, Nachnamen usw.). Beachte, dass nun jeder Eintrag eine *List* sein muss. Weise diese Dictionary der Variabel `people` zu.

```
people = {"vornamen": ["Christopher", "Henning", "Severin"], "nachnamen": ["Annen",
↪"May", "Kantereit"], "groessen": [174, 182, 162]}
```

1.4.11 Übung 9: Functions

Bisher haben wir Objekte erstellt. Richtig interessant wird programmieren aber erst, wenn wir mit Objekte durch Funktionen (*Functions*) verändern. *Functions* führen bestimmte Tasks aus. Wende die Functions `len()` und `sum()` an den Listen `groessen` und `vornamen` an.

```
len(groessen)
```

```
4
```

```
sum(groessen)
```

```
695
```

```
len(vornamen)
```

```
4
```

```
# sum(vornamen)
```

1.5 Python Modules

Ähnlich wie R basiert Python auf Erweiterungen: Diese Erweiterungen heissen in R *Libraries* / *Packages*, in Python werden sie *Modules* genannt. Sie sind dazu da, gewisse Teilbereiche unseres Arbeitsprozesses zu vereinfachen. Eine Analogie dazu: Um ein Haus zu bauen sind wir auf verschiedene Spezialisten / Spezialistinnen angewiesen: Wir brauchen zum Beispiel eine Malerin oder einen Maler. Im Telfonbuch sind seitenweise Maler*innen aufgelistet, und jede*r arbeitet etwas anders. Um eine spezifische Malerin anzuheuern müssen wir zuerst den Kontakt herstellen und die Vertragsmodalitäten vereinbaren. Erst dann können wir sie in unseren Arbeitsprozess (Haus bauen) einbinden.

Um diese Analogie auf unser Projekt zu übertragen: Das "Haus bauen" ist unser Forschungsprojekt (z.B. eine Bachelorarbeit). Ein "Telefonbuch", wo die Spezialisten erfasst sind nennt ein *Reposoritory*. Den Erstkontakt mit der Malerin zu erstellen und die Vertragsmodalitäten zu vereinbaren bedeutet, die Erweiterung zu installieren. In R wird eine Erweiterung folgendermassen installiert:

```
in R:
------------------------------------------------------------
install.packages("malerinMina")
```

In diesem Beispiel heisst die Erweiterung `malerinMina`. Das *Reposoritory* geben wir in R oft nicht an, weil in RStudio typischerweise schon eine Adresse hinterlegt ist. Wie wir in Python ein Modul installieren lernen wir später. Nehmen wir an dieser Stelle an, wir haben die Erweiterung `malerinMina` in Python installiert. Treiben wir an dieser Stelle die Analogie etwas weiter: Der Erstkontakt mit der Malerin ist also erstellt und alle Vertragsmodalitäten sind vereinbart. Nun wollen wir an einem bestimmten Tag mit ihr arbeiten. Dafür müssen wir sie zuerst auf die Baustelle bestellen. Übersetzt auf programmieren bedeutet dies, wir müssen die Erweiterung in unsere Session laden. In R sieht der Befehl so aus:

```
in R:                       |       in Python:
----------------------------+-----------------------------------
library(malerinMina)        |       import malerinMina
```

Erst jetzt können wir mit der Erweiterung arbeiten und die Fachexpertise unserer Malerin nutzen. Eine Expertise unserer Malerin ist es, Wände zu bemalen. Dafür gibt es eine *Function* `wand_bemalen()`. In R kann ich diese *Function* "einfach so" aufrufen. In Python hingegen muss ich die Erweiterung, in der die *Function* enthalten ist, der *Function* mit einem Punkt voranstellen. Das sieht also folgendermassen aus:

```
in R:                       |       in Python:
----------------------------+-----------------------------------
wand_bemalen()              |       malerinMina.wand_bemalen()
```

Das ist zwar umständlicher, aber dafür weniger Fehleranfällig. Angenommen, unser Maurer kann ebenfalls Wände bemalen und hat die entsprechende *Function* `wand_bemalen()` ebenfalls. Dann ist in R nicht klar, welche Erweiterung gemeint ist und das kann zu Missverständnissen führen (vielleicht bemalt der Mauerer die Wände etwas anders als die Malerin). In Python ist im obigen Beispiel unmissverständlich, dass ich `wand_bemalen()` aus dem Modul `malerinMina` meine.

Das wichtigste in Kürze

- Erweiterungen heissen in Python *Modules*
- Vor der erstmaligen Nutzung muss ein *Module* installiert werden
- Vor der Verwendung in einer Session muss ein *Module* "geladen" werden. Dies muss für jede Session wiederholt werden!
- Ein *Module* wird in Python mit `import modulename` geladen
- Eine *Function* aus einem *Module* wird folgendermassen aufgerufen: `modulname.function()`

1.5.1 Modul mit Alias importieren

Wenn es uns zu Umständlich ist jedesmal `malerinMina.wand_bemalen()` voll auszuschreiben können wir beim Importieren dem Modul auch einen "Alias" vergeben. Dies kann beispielsweise folgendermassen aussehen:

```
import malerinMina as mm
mm.wand_bemalen()
```

Dies ist deshalb wichtig, weil sich für viele Module haben sich bestimmte Aliasse eingebürgert haben. Ihr macht sich das Leben leichter, wenn ihr euch an diese Konventionen (welche ihr noch kennenlernen werdet) hält.

1.5.2 Einzelne *Function* importieren

Was man auch noch machen kann, ist eine explizite *Function* aus einem Modul zu laden. Wenn man dies macht, kann man die Funktion ohne vorangestelltes Modul nutzen (genau wie in R). Dies sieht folgendermassen aus:

```
from malerinMina import wand_bemalen()
wand_bemalen()
```

1.6 Tabellarische Daten

Schauen wir uns nochmals die *Dictionary* `people` an. Diese ist ein Spezialfall einer Dictionary: Jeder Eintrag besteht aus einer Liste von gleich vielen Werten. Diese Dictionary schreit förmlich danach, tabellarisch dargestellt zu werden. Dies wollen wir gewähren und brauchen dafür das Modul `pandas`. Importiere das Modul mit dem Alias `pd`.

!! ACHTUNG: HIER FEHLT NOCH DIE EINFÜHRUNG INSTALLATION VON MODULEN !!

1.6.1 Übung 1: von einer *Dictionary* zu einer *DataFrame*

Wandle die Dictionary `people` in eine DataFrame um: Dazu musst du `people` als Argument der Funktion `DataFrame` übergeben: `pd.DataFrame(people)`. Weise den Output der Variable `people_df` zu.

Für R Nutzer: Auch in R gibt es *Dataframes*, diese sind aber in Base R integriert (brauchen deswegen keine Erweiterung wie `pandas`)

1.6.2 Übung 2: *DataFrame* in csv umwandeln

In der Praxis kommen Tabellarische Daten meist als «csv» Dateien daher. Wir können aus unserer eben erstellten DataFrame sehr einfach eine csv Datei erstellen. Führe das mit folgendem Code aus:

```
people_df.to_csv("people.csv")
```

1.6.3 Übung 3: CSV als *DataFrame* importieren

Genau so einfach ist es eine csv zu importieren. Lade dazu die Datei "zeckenstiche.csv" von Moodle herunter und speichere es im aktuellen Arbeitsverzeichnis ab. Importiere mit folgendem Code die Datei "zeckenstiche.csv". Schau dir `zeckenstiche` nach dem importieren an.

```
zeckenstiche = pd.read_csv("zeckenstiche.csv")
```

Hinweis: Dieser Code funktioniert nur, wenn "zeckenstiche.csv" im aktuellen Arbeitsverzeichnis (*Current Working Directory*) abgespeichert wird. Wenn du nicht sicher bist, wo dein aktuelles Arbeitsverzeichnis liegt, kannst du das Modul `os` laden und dies mit der Funktion `os.getcwd()` (get **c**urrent**w**orking**d**irectory) herausfinden.

```
import os

os.getcwd()
```

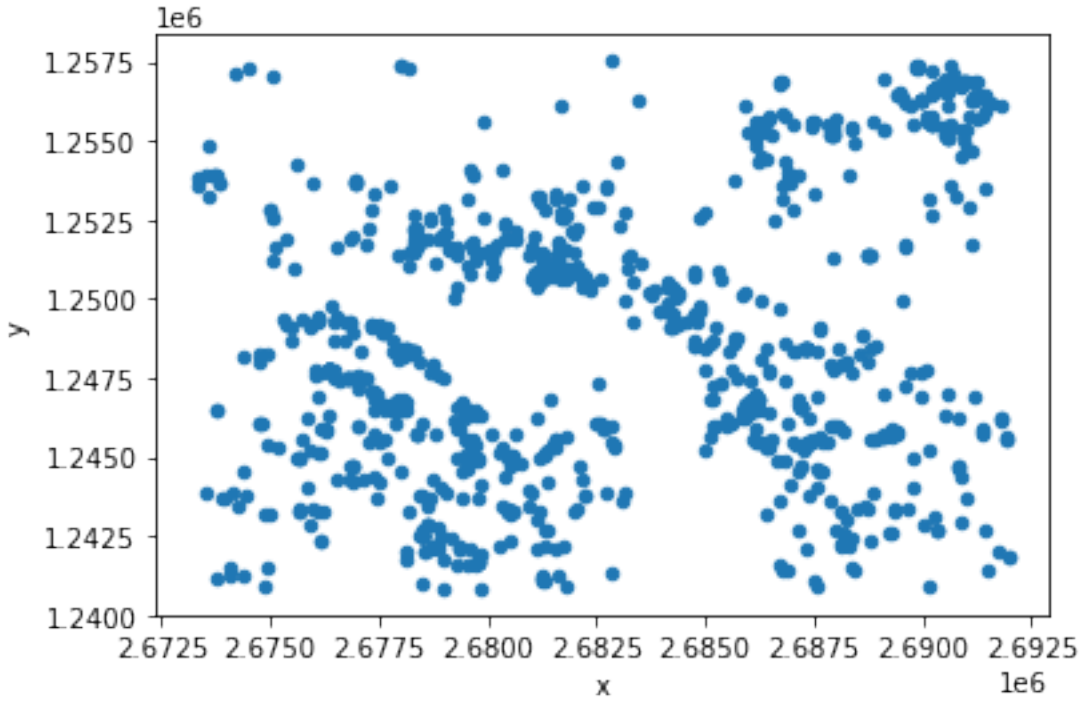
```
'/home/runner/work/codingingis/codingingis'
```

1.6.4 Übung 4: Koordinaten räumlich darstellen

Die *DataFrame* `zeckenstiche` beinhaltet x und y Koordinaten für jeden Unfall in den gleichnamigen Spalten. Wir können die Stiche mit einem Scatterplot räumlich visualisieren. Führe dazu folgenden Code aus:

```
zeckenstiche.plot.scatter("x","y")
```

```
<AxesSubplot:xlabel='x', ylabel='y'>
```



1.6.5 Übung 5: Einzelne Spalte selektieren

Um eine einzelne Spalte zu selektieren (z.B. die Spalte "ID") kann man gleich vorgehen wie bei der Selektion eines Eintrage in einer *Dictionary*. Probiere es aus.

```
zeckenstiche["ID"]
```

```
0          0
1          1
2          2
3          3
4          4
        ...
1071    1071
1072    1072
1073    1073
1074    1074
1075    1075
Name: ID, Length: 1076, dtype: int64
```

1.6.6 Übung 6: Neue Spalte erstellen

Auch das Erstellen einer neuen Spalte ist identisch mit der Erstellung eines neuen *Dictionary* Eintrags. Folgende Zeile erstellt eine neue Spalte "Stichtyp" mit dem gleichen Wert in allen Zeilen ("Zecke"). Erstelle ebenfalls eine solche Spalte.

```
zeckenstiche["Stichtyp"] = "Zecke"

zeckenstiche
```

```
        ID    accuracy          x          y Stichtyp
0        0   65.617154  2678971.0  1240824.0    Zecke
1        1  257.492052  2679837.0  1240858.0    Zecke
2        2  163.533834  2687539.0  1240881.0    Zecke
3        3  185.000000  2674848.0  1240913.0    Zecke
4        4  228.215231  2681771.0  1240922.0    Zecke
...    ...         ...        ...        ...      ...
1071  1071  109.531946  2678005.0  1257344.0    Zecke
1072  1072  100.489274  2678005.0  1257347.0    Zecke
1073  1073  301.748529  2689893.0  1257351.0    Zecke
1074  1074  301.748542  2690668.0  1257369.0    Zecke
1075  1075  226.000000  2682852.0  1257531.0    Zecke

[1076 rows x 5 columns]
```

1.7 GIS in Python

Die bisherigen Aufgaben hatten noch nicht viel mit GIS zu tun. Die Koordinaten der Zeckenstiche sind uns zwar bekannt, es handelt es dabei aber lediglich um Numerische xy-Werte, nicht um Geometrische Punkte mit einem Raumbezug. So bleiben uns bis jetzt alle räumlichen Operationen vorenthalten (zum Beispiel ob sich ein Stich im Wald befindet oder nicht), und wir können auch keine schönen Karten generieren.

Um mit Geodaten in Python arbeiten zu können, müssen wir ein neues Modul importieren. Im Grunde genommen sind Vektordaten nicht mehr als Tabellen mit einer zusätzlichen "Geometrie"-Spalte, dementsprechend baut die "Geo"-Erweiterung auf `pandas` auf und heisst: `geopandas`.

Du solltest `geopandas` bereits installiert haben und das Modul in dein Script importieren. Importiere `geopandas` mit dem Alias `gpd`.

1.7.1 Übung 1: *DataFrame* zu *GeoDataFrame*

Wie erwähnt sind die Zeckenstichdaten bisher lediglich als tabellarische Daten vorhanden. In ArcGIS Terminologie müssen wir die Operation "XY Table to Point" durchführen. In Python heisst das: Wir wandeln eine *DataFrame* in eine *GeoDataFrame* um. Zuerst erstellen wir eine Geometrie-Spalte aus den xy-Koordinaten mit der Funktion `points_from_xy` aus dem Modul `geopandas`.

```
zeckenstiche["geometry"] = gpd.points_from_xy(zeckenstiche.x, zeckenstiche.y)
```

Der Datensatz `zeckenstiche` hat jetzt aber noch nicht begriffen, dass es jetzt eine `GeoDataFrame` ist. Dies müssen wir dem Objekt erst noch mitteilen:

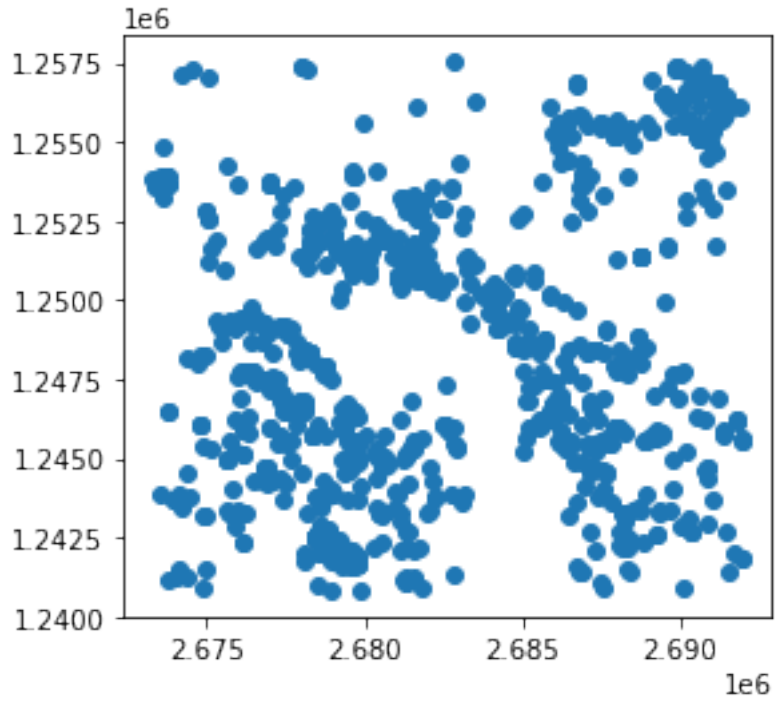
```
zeckenstiche = gpd.GeoDataFrame(zeckenstiche)

zeckenstiche
```

```
        ID    accuracy          x          y                          geometry
0        0   65.617154  2678971.0  1240824.0  POINT (2678971.000 1240824.000)
1        1  257.492052  2679837.0  1240858.0  POINT (2679837.000 1240858.000)
2        2  163.533834  2687539.0  1240881.0  POINT (2687539.000 1240881.000)
3        3  185.000000  2674848.0  1240913.0  POINT (2674848.000 1240913.000)
4        4  228.215231  2681771.0  1240922.0  POINT (2681771.000 1240922.000)
...    ...         ...        ...        ...                            ...
1071  1071  109.531946  2678005.0  1257344.0  POINT (2678005.000 1257344.000)
1072  1072  100.489274  2678005.0  1257347.0  POINT (2678005.000 1257347.000)
1073  1073  301.748529  2689893.0  1257351.0  POINT (2689893.000 1257351.000)
1074  1074  301.748542  2690668.0  1257369.0  POINT (2690668.000 1257369.000)
1075  1075  226.000000  2682852.0  1257531.0  POINT (2682852.000 1257531.000)

[1076 rows x 5 columns]
```

Jetzt, wo zeckenstiche eine GeoDataFrame ist, gibt es einen einfachen weg die Punkte räumlich zu visualisieren:

```
zeckenstiche.plot()
```

```
<AxesSubplot:>
```



1.7.2 Übung 2: Koordinatensystem festlegen

Wir wissen zwar, das unsere *GeoDataFrame* in Schweizer Landeskoordinaten (CH1903 LV95) zu verstehen ist, aber dies haben wir noch nirgends festgehalten. Das Koordinatensystem (Coordinate Reference System, CRS) können wir über das Attribut `crs` der *GeoDataFrame* festhalten. Das Koordinatensystem CH1903 LV95 hat den EPSG Code 2056, demnach muss das CRS folgendermassen festgelegt werden:

```
zeckenstiche = zeckenstiche.set_crs(epsg = 2056)
```

Nun ist das Koordinatensystem (*CRS*) als Attribut der *GeoDataFrame* gespeichert:

```
zeckenstiche.crs
```

```
<Projected CRS: EPSG:2056>
Name: CH1903+ / LV95
Axis Info [cartesian]:
- E[east]: Easting (metre)
- N[north]: Northing (metre)
Area of Use:
- name: Europe - Liechtenstein and Switzerland
- bounds: (5.96, 45.82, 10.49, 47.81)
Coordinate Operation:
- name: Swiss Oblique Mercator 1995
- method: Hotine Oblique Mercator (variant B)
Datum: CH1903+
- Ellipsoid: Bessel 1841
- Prime Meridian: Greenwich
```

1.7.3 Übung 3: Zeckenstiche als Shapefile exportieren

Zum Schluss exportieren wir unser Datensatz in ein Shapefile, damit wir das nächste Mal direkt mit einer *GeoDataFrame* arbeiten können. Genau wie wir in einer vorherigen Übung eine pandas DataFrame mit `.to_csv` in eine csv exportiert haben, gibt es für GeoDataFrames die Methode `.to_file`. Exportiere zeckenstiche mit dieser Methode in eine Shapefile.

```
zeckenstiche.to_file("zeckenstiche.shp")
```

1.8 Einleitung zu diesem Block

Letzte Woche habt ihr einen weiten Bogen gespannt: Von einfachen (primitiven) Datentypen bis hin zu Geodaten und deren Visualisierung. In dieser und der nächsten Übung (Coding in GIS II und III) wollen wir uns weiter mit den Zeckenstichdaten befassen. Wir werden im Wesentlichen ein Teil der Übung aus «Datenqualität und Unsicherheit» in Python rekonstruieren. In der Übung geht es um folgendes: Wir wissen das die Lagegenauigkeit der Zeckenstichmeldungen mit einer gewissen Unsicherheit behaftet sind. Um die Frage "Welcher Anteil der der Zeckenstiche befinden sich im Wald?" unter Berücksichtigung dieser Unsicherheit beantworten zu können, führen wir eine (Monte Carlo) Simulation durch. In dieser Simulation verändern wir die Position der Zeckenstichmeldungen zufällig und berechnen den Anteil der Zeckenstiche im Wald. Das zufällige Verschieben und berechnen wiederholen wir beliebig lange und bekommen für jede Wiederholung einen leicht unterschiedlichen Prozentwert. Die Verteilung dieser Prozentwerte ist die Antwort auf die ursprüngliche Frage ("Welcher Anteil…") unter Berücksichtigung der Unsicherheit.

Um eine solche, etwas komplexere Aufgabe lösen zu können müssen wir sie in einfachere Einzelschritte aufteilen. Diese bearbeiten wir in dieser und der kommenden Woche:

- Schritt 1: Einen Einzelpunkt zufällig verschieben
- Schritt 2: Alle Punkte einer GeoDataFrame zufällig verschieben (1 «Run»)
- Schritt 3: Alle Punkte einer GeoDataFrame mehrfach zufällig verschieben (z.B. 50 «Runs»)
- Schritt 4: Anteil der Punkte im Wald pro «Run» ermitteln
- Schritt 5: Verteilung dieser Mittelwerte visualisieren

Übungsziele

- Functions kennenlernen und beherrschen
- Einfache Geometrien manipulieren lernen
- Eigene Python-Scripts importieren können
- Function auf eine ganze Spalte einer (Geo-) DataFrame anwenden können.

1.9 Input zu *Functions*

Bevor wir uns in die Simulation stürzen müssen wir noch lernen eigene *Functions* zu schreiben: Dies ist auch nicht weiter schwierig: Eine *Function* wird mit `def` eingeleitet, braucht einen Namen, einen Input und einen Output.

Wenn wir zum Beispiel eine Function erstellen wollen die uns grüsst, so geht dies folgendermassen:

```
def sag_hallo():
    return("Hallo!")
```

- Mit `def` sagen wir: "Jetzt definiere ich eine Function".
- Danach kommt der Name der *Function*, in unserem Fall `sag_hallo` (mit diesem Namen können wir die *Function* später wieder abrufen).
- Als drittes kommen die runden Klammern, wo wir bei Bedarf Inputvariablen (sogenannte Parameter) festlegen können. In diesem ersten Beispiel habe ich keine Parameter festgelegt
- Nach der Klammer kommt ein Doppelpunkt was bedeutet: "jetzt wird gleich definiert, was die Funktion tun soll"
- Auf einer neuen Zeile wird eingerückt festgelegt, was die Function eben tun soll. Meist sind hier ein paar Zeilen Code vorhanden
- Die letzte eingerückte Zeile (in unserem Fall ist das die einzige Zeile) gibt mit return an, was die *Function* zurück geben soll (der Output). In unserem Fall soll sie "Hallo!" zurück geben.
- Voila, das war's schon! Jetzt können wir diese Function schon nutzen:

```
sag_hallo()
```

```
'Hallo!'
```

Diese *Function* ohne Input ist wenig nützlich. Meist wollen wir der *Function* etwas - einen Input - übergeben können. Um eine *Function* zu erstellen die ein Argument annimmt geht man folgendermassen vor:

```
def sag_hallo(vorname):
    return("Hallo "+vorname+"!")
```

Nun können wir der Function einen Paramter übergeben, damit wir persönlich gegrüsst werden. In folgendem Beispiel ist `vorname` ein Parameter, "Guido" ist sein Argument.

```
sag_hallo("Guido")
```

```
'Hallo Guido!'
```

Den Output können wir wie gewohnt einer neuen Variabel zuweisen:

```
persoenlicher_gruss = sag_hallo("Guido")
persoenlicher_gruss
```

```
'Hallo Guido!'
```

1.10 Übungen zu *Functions*

1.10.1 Übung 1: Erste *Function* erstellen

Erstelle eine Function, die `gruezi` heisst, einen Nachnamen als Input annimmt und per Sie grüsst. Das Resultat soll in etwa folgendermassen aussehen:

```
def gruezi(nachname):
    return("Guten Tag, "+nachname)
```

```
gruezi("Guido")
```

```
'Guten Tag, Guido'
```

1.10.2 Übung 2: *Function* erweitern

Erweitere die *Function* `gruezi` indem eine du einen weiteren Parameter namens `anrede` implementierst. Das Resultat soll in etwa folgendermassen aussehen:

```
def gruezi(nachname, anrede):
    return("Guten Tag, "+anrede+" "+nachname)
```

```
gruezi("van Rossum","Herr")
```

```
'Guten Tag, Herr van Rossum'
```

1.10.3 Übung 3: Default-Werte festlegen

Man kann für die Parameter folgendermassen einen Standardwert festlegen: Beim Definieren der Function wird dem Parameter schon innerhalb der Klammer ein Argument zugewiesen (z.B. `anrede = "Herr oder Frau"`). Wenn anrede bei der Verwendung von `gruezi` nicht definiert wird, entspricht die Anrede nun «Herr oder Frau». Setzte einen Standardwert in der Anrede und teste die *Function*. Das Resultat soll in etwa folgendermassen aussehen:

```
def gruezi(nachname, anrede = "Herr oder Frau"):
    return("Guten Tag, "+anrede+" "+nachname)
```

```
gruezi("van Rossum")
```

```
'Guten Tag, Herr oder Frau van Rossum'
```

1.10.4 Übung 4: Python-File in Module konvertieren

In der Vorlesung habt ihr gesehen, dass Modules nichts weiter sind als Python-Skripte in einem bestimmten Verzeichnis. Um das zu verdeutlichen kreieren wir nun unser eignes Module. Speichert dazu eure *Function* `gruezi` in einem neuen Skript mit dem Namen "myownmodule.py" in eurer Working Directory ab. Nun könnt ihr `myownmodule` wie jedes andere Module in euer Skript importieren.

```
import myownmodule

myownmodule.gruezi("van Rossum","Herr")
```

```
'Guten Tag, Herr van Rossum'
```

1.10.5 Weitere Hinweise zu Functions

Hier noch zwei Hinweise zum Definieren von Functions:

- Wenn Parameter Default-Argumente zugewiesen werden, dann werden sie für den Nutzer automatisch zu optionalen Parametern.
- In der Definition einer Function werden optionale Parameter immer nach erforderlichen Parameter definiert.

Jetzt noch zwei Hinweise zum Abrufen von Functions:

- Wenn die richtige Reihenfolge eingehalten wird, müssen die Parameter (z.B: `anrede =`, `nachname =`) nicht spezifiziert werden. Zum Beispiel:

```
gruezi("van Rossum", "Herr")
```

```
'Guten Tag, Herr van Rossum'
```

- Wenn die Parameter der Argumente spezifiziert werden, ist die Reihenfolge wiederum egal:

```
gruezi(anrede = "Herr", nachname = "von Rossum")
```

```
'Guten Tag, Herr von Rossum'
```

1.11 Einen Einzelpunkt zufällig verschieben

Um die Monte Carlo Simulation in Angriff zu nehmen, müssen wir als erstes einen Weg finden, wie wir Punkte zufällig verschieben können, zum Beispiel um 100m. Bei *GeoDataFrames* handelt es sich bei den Punkten um `Point()`-Objekte des Modules `shapely.geometry`. Wir importieren also genau diese *Function* und bei der Gelegenheit auch `geopandas`:

```
import geopandas as gpd
from shapely.geometry import Point
```

```
/opt/hostedtoolcache/Python/3.7.9/x64/lib/python3.7/site-packages/geopandas/_compat.
↪py:88: UserWarning: The Shapely GEOS version (3.8.0-CAPI-1.13.1 ) is incompatible␣
↪with the GEOS version PyGEOS was compiled with (3.8.1-CAPI-1.13.3). Conversions␣
↪between both will be slow.
  shapely_geos_version, geos_capi_version_string
```

Als erster Schritt kreieren wir einen "Testpunkt" und entwickeln eine Methode, diesen zufällig zu verschieben (was wir mit einem Punkt schaffen, schaffen wir es auch mit Tausend Punkten). Beispielsweise können wir den Testpunkt mit den Koordinaten der alten Sternwarte Bern (bzw. deren Gedenktafel, s.u.) erstellen.

```
sternwarte = Point(2600000,1200000)
print(sternwarte)
```

```
POINT (2600000 1200000)
```

Dieser konstruierte Punkt liegt im neuen Schweizer Koordinatensystem (CH1903+ LV95) auf der alten Sternwarte in Bern, bzw. auf deren Gedenktafel.

Fig. 1.1: Links: Koordinatensystem der Schweiz, mit dem (alten) Referenzwert Sternwarte Bern. Quelle: lv95.bve.be.ch rechts: Gedenktafel an die Alte Sternwarte Bern. Quelle: aiub.unibe.ch

1.11.1 Übung 1

Wenn wir die Position eines Punktes verschieben wollen, dann müssen wir die Koordinaten wissen. Wir suchen also nach einem Weg, wie wir die x,y-Werte unseres Punktes `sternwarte` zurückerhalten. Erstelle den Punkt `sternwarte`. Versuche herauszufinden, wie das geht. Ich empfehle dazu folgende Schritte:

1. Finde heraus, um was für einen Datentyp es sich beim Punkt handelt, nutze dafür `type()`
2. Recherchier diesen Datentyp im Internet
3. Finde heraus, was für *Attributes* und *Methods* mit dem Datentyp assoziiert sind

Wenn du die `x` und `y` Koordinaten hast, weise sie den Variabeln `x_alt`, resp. `y_alt` zu. Hinweis: Die Antwort ist verblüffend einfach. Verbringe nicht allzu viel Zeit mit dieser Übung. Frage mich oder deine Nachbarn, wenn du hier nicht weiterkommst.

```
# Schritt 1
type(sternwarte)
```

```
shapely.geometry.point.Point
```

```
# Schritt 2
# Es handelt sich offenbar um ein "Shapely" objekt https://shapely.readthedocs.io/en/
↪latest/manual.html
```

```
# Schritt 3
# Es sind diverse Methods mit diesem Datentyp assoziiert: https://shapely.readthedocs.
↪io/en/latest/manual.html#general-attributes-and-methods
sternwarte.geom_type

# Punkt objekte haben x, y und evt. z Koordinaten: https://shapely.readthedocs.io/en/
↪latest/manual.html#points
sternwarte.x
sternwarte.y
```

```
1200000.0
```

1.11.2 Übung 2: Zufallswert addieren

Jetzt, wo wir die Koordinaten aus extrahieren können weisen wir sie zwei neuen Variabeln zu: `x_alt`, `y_alt`. Danach müssen wir eine Zufallswert zwischen -100 und 100 generieren, den wir zu `x_alt` resp. `y_alt` addieren können (das Koordinatensystem ist ja in Metern). Wenn ich nach «Python random number» im Internet suche dann ist mein erstes Resultat bereits die Funktion `randrange` aus dem Modul `random`.

Importiere dieses Modul und nutze die genannte Funktion um einen Versatz je für `x` und `y` zu generieren. Addiere diese beiden Zahlen zu `x_alt` resp. `y_alt` und speichere die Outputs als `x_neu` und `y_neu`. Nun können wir wieder `Point()` verwenden um aus `x_neu`, `y_neu` eine neue Punkt-Geometrie zu erstellen. Speichere diese neue Geometrie unter der Variabel `sternwarte_neu`.

```
x_alt = sternwarte.x
y_alt = sternwarte.y

import random

x_neu = x_alt + random.randrange(-100,100)
y_neu = y_alt + random.randrange(-100,100)

sternwarte_neu = Point(x_neu, y_neu)
print(sternwarte_neu)
```

```
POINT (2599988 1200053)
```

1.11.3 Übung 3: Arbeitsschritte in eine *Function* verwandeln

Jetzt sind die Einzelschritte zur Verschiebung eines Punktes klar. Da wir dies für viele Punkte machen müssen, ist es sinnvoll, aus den Arbeitsschritten eine *Function* zu erstellen. Erstelle eine Function `point_offset` welche als Input ein `Point()`-Objekt nimmt und ein leicht verschobenes `Point()`-Objekt zurück gibt. Wenn du möchtest kannst du die Distanz der Verschiebung als optionalen Parameter (wichtig!) definieren.

```
def point_offset(point, distance = 100):
    x_alt = point.x
    y_alt = point.y

```

(continues on next page)

(continued from previous page)

```
    distance = int(distance)

    x_neu = x_alt + random.randrange(-distance,distance)
    y_neu = y_alt + random.randrange(-distance,distance)

    point_off = Point(x_neu, y_neu)

    return(point_off)

point_offset(sternwarte, 100)
```

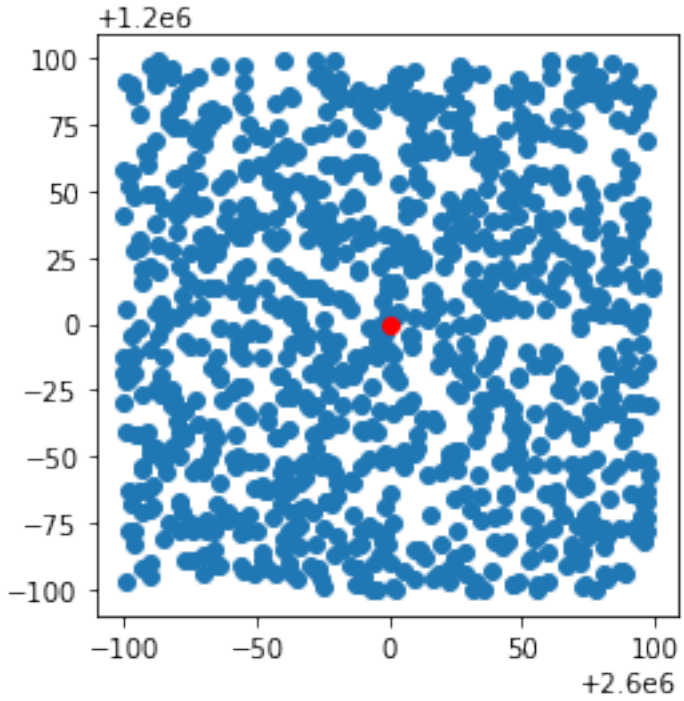
```
<shapely.geometry.point.Point at 0x7fd242a4a110>
```

1.11.4 Output visualisieren

Nun ist es wichtig, dass wir unsere Function visuell überprüfen. Wir wenden die Function 1'000 mal auf den Punkt `sternwarte` an und schauen mal wie die Punkte verteilt werden. Für diesen Schritt gebe ich euch den fertigen Code, da ihr die dafür benötigten Techniken noch nicht gelernt habt. Füge diesen Code in dein Script ein und führe ihn aus. Allenfalls musst du den Code leicht an deine Situation anpassen.

```
ax = gpd.GeoSeries([point_offset(sternwarte) for x in range(1,1000)]).plot()
gpd.GeoSeries(sternwarte).plot(ax = ax, color = "red")
```

```
<AxesSubplot:>
```

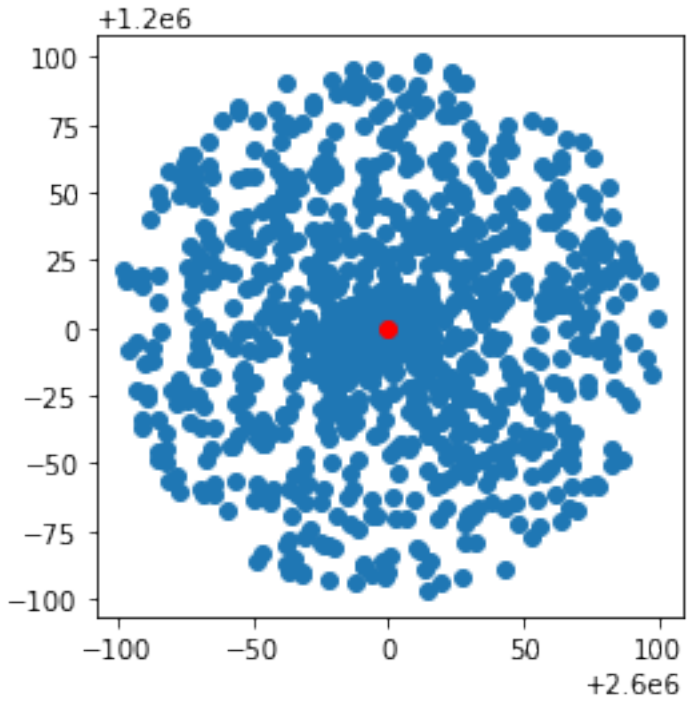


1.11.5 Optionale Übung für *Hoch*motivierte (sehr anspruchsvoll)

Wie in der obigen Grafik ersichtlich ist die Verteilung der Punkte bei unserer Herangehensweise Quadratisch. Im Extremfall liegt ein Punkt 100m östlich und 100m nördlich vom Ursprungspunkt entfernt, die euklidische Distanz zum Ursprungspunkt beträgt dann 141 Meter ($\sqrt{100^2 + 100^2}$). Wen das auch stört, der kann eine alternative Methode entwickeln den Punkt um maximal 100m zu verschieben. Tipps dazu könnt ihr gerne bei uns abholen.

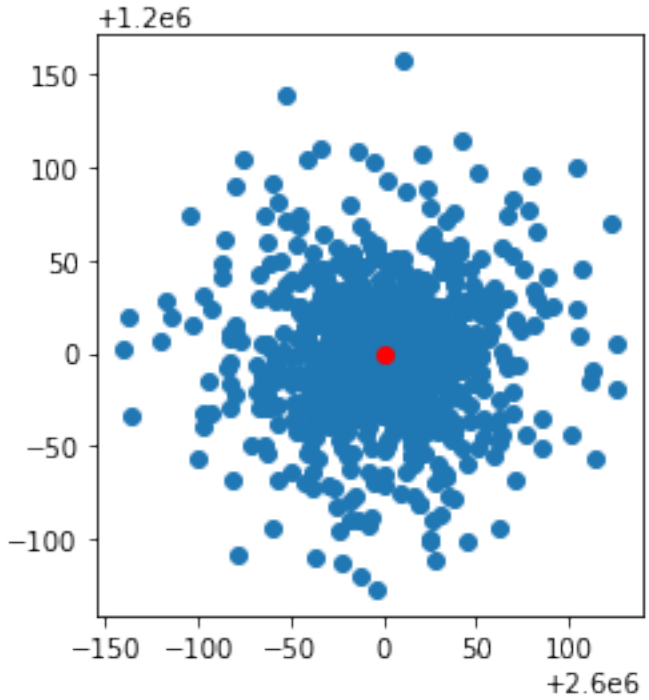
```
def point_offset2(point,distance, distribution = "uniform"):
    import random
    import math
    import shapely
    x = point.x
    y = point.y
    # Ein Winkel von 360 wird in Radians konvertiert
    rad = math.radians(random.uniform(0,360))
    # zwei Varianten: "uniforme Verteilung oder Normalverteilung"
    if distribution == "uniform":
        offdist = random.randrange(-distance,distance)
    if distribution == "normal":
        offdist = random.normalvariate(mu = 0, sigma = distance/2)
    # Mit der Cosinusfunktion, dem Winkel (in Radians) und der Distanz wird der y_
↪offset bestimmt
    y_offset = math.cos(rad)*offdist
    # Mit der Sinusfunktion wird der x_offset bestimmt
    x_offset = math.sin(rad)*offdist
    x_neu = x+x_offset
    y_neu = y+y_offset
    point_off = Point(x_neu, y_neu)
    return(point_off)

ax = gpd.GeoSeries([point_offset2(sternwarte,100, "uniform") for x in range(1000)]).
↪plot()
gpd.GeoSeries(sternwarte).plot(ax = ax, color = "red")
```

```
<AxesSubplot:>
```




```
ax = gpd.GeoSeries([point_offset2(sternwarte,100, "normal") for x in range(1000)]).
↪plot()
gpd.GeoSeries(sternwarte).plot(ax = ax, color = "red")
```

```
<AxesSubplot:>
```



1.12 Punkte einer GeoDataFrame zufällig verschieben

Nun wollen wir den ganzen Zauber auf unsere richtigen Daten, die Zeckenstiche, anwenden. Dazu müssen wir zuerst den Zeckenstichdatensatz einlesen. Nutzt dazu die Function `gpd.read_file()` um das Shapefile, welches wir letzte Woche abgespeichert hatten, zu importieren. Wer dieses Shapefile nicht hat, findet eine Version auf Moodle. Speichere die Daten in als Variabel `zeckenstiche`.

```
import pandas as pd
import geopandas as gpd
import random
from shapely.geometry import Point

zeckenstiche = gpd.read_file("zeckenstiche.shp")


def point_offset(point, distance = 100):
    x_alt = point.x
    y_alt = point.y

    distance = int(distance)

    x_neu = x_alt + random.randrange(-distance,distance)
    y_neu = y_alt + random.randrange(-distance,distance)

    point_off = Point(x_neu, y_neu)

    return(point_off)
```

```
/opt/hostedtoolcache/Python/3.7.9/x64/lib/python3.7/site-packages/geopandas/_compat.
↪py:88: UserWarning: The Shapely GEOS version (3.8.0-CAPI-1.13.1 ) is incompatible␣
↪with the GEOS version PyGEOS was compiled with (3.8.1-CAPI-1.13.3). Conversions␣
↪between both will be slow.
  shapely_geos_version, geos_capi_version_string
```

1.12.1 Übung 1: Alle Zeckenstiche zufällig verschieben

Um in *DataFrames* / *GeoDataFrames* ganze Spalten oder Zeilen zu verändern, stehen einem die *Methods* `map`, `apply` und `applymap` zur Verfügung. Es lohnt sich sehr, die verschiedenen Anwendungsbereiche dieser *Methods* kennen zu lernen, doch das würde den Umfang dieses Kurses überschreiten. Heute brauchen wir lediglich `apply`, welches wir auf die `geometry`-Spalte unserer *GeoDataFrame* anwenden. Um unsere eigene Function point_offset auf die Spalte geometry anzuwenden gehst du wie folgt vor: Selektiere die entsprechende Spalte (mit `["spaltenname"]`) und verwende die Method apply mit der selbst erstellt Function point_offset als einziges Argument, und zwar ohne Klammer.

```
zeckenstiche["geometry"].apply(point_offset).head() # .head kann man weglassen
```

```
0    POINT (2678906.000 1240891.000)
1    POINT (2679816.000 1240891.000)
2    POINT (2687490.000 1240898.000)
3    POINT (2674771.000 1240828.000)
4    POINT (2681736.000 1240918.000)
Name: geometry, dtype: geometry
```

1.12.2 Übung 2: Neue GeoDataFrame mit simulierten Punkten erstellen

Was bei `apply()` rauskommt, sind nur die neuen, verschobenen Punkte. Um diese abzuspeichern erstellen wir eine leere GeoDataFrame und weisen den Output aus `apply()` einer neuen Spalte namens `geometry` zu.

```
zeckenstiche_sim = gpd.GeoDataFrame()
zeckenstiche_sim["geometry"] = zeckenstiche["geometry"].apply(point_offset)
```

1.12.3 Übung 3: ID übernehmen

Der generierte Datensatz `zeckenstiche_sim` verfügt nun nur über eine Geometriespalte, die anderen Spalten von `zeckenstiche` haben wir nicht übernommen. Bei den meisten Spalten ist uns das auch egal, aber die Zeckenstich ID wäre praktisch, um den simulierten Punkt zum Original Zeckenstich zurück führen zu können.

Übertrage die ID von `zeckenstiche` auf `zeckenstiche_sim`. Verwende dazu die Selektion mit eckigen Klammern.

```
zeckenstiche_sim["ID"] = zeckenstiche["ID"]
zeckenstiche_sim.head()
```

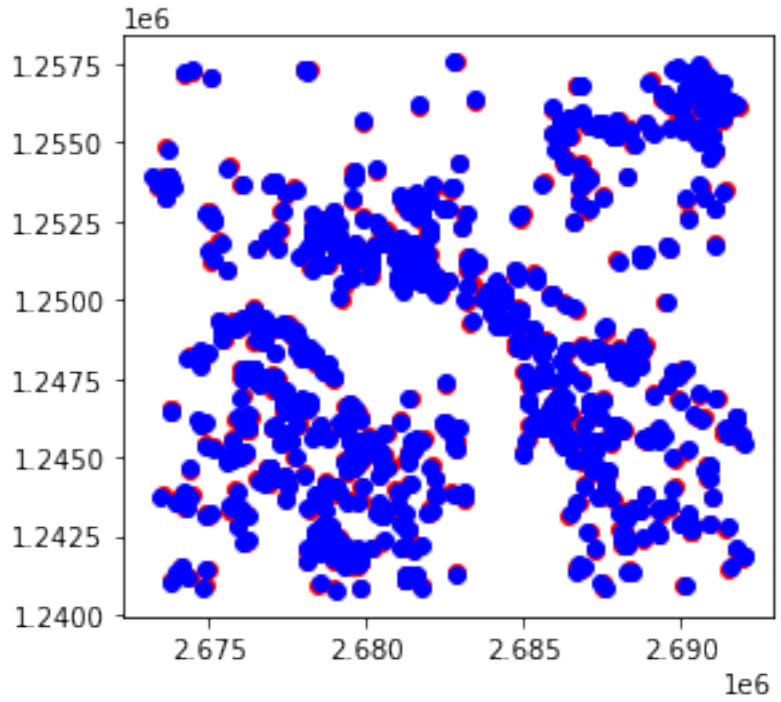
```
                         geometry  ID
0  POINT (2679037.000 1240774.000)   0
1  POINT (2679838.000 1240865.000)   1
2  POINT (2687592.000 1240828.000)   2
3  POINT (2674799.000 1240861.000)   3
4  POINT (2681727.000 1240833.000)   4
```

1.12.4 Übung 4: Mehrere *GeoDataFrames* visualisieren

Um zwei *GeoDataFrames* im gleichen Plot darzustellen, wird folgendermassen vorgegangen. Der erste Datensatz wird mit `.plot()` visualisiert, wobei der Output einer Variabel (z.B. `basemap`) zugewiesen wird. Danach wird der zweite Datensatz ebenfalls mit `.plot()` visualisiert, wobei auf den ersten Plot via dem Argument `ax` verwiesen wird.

```
basemap = zeckenstiche.plot(color = "red")
zeckenstiche_sim.plot(ax = basemap, color = "blue")
```

```
<AxesSubplot:>
```



1.13 Einleitung zu diesem Block

!! HIER NOCH EINE EINLEITUNG SCHREIBEN !!

Übungsziele

- Functions kennenlernen und beherrschen
- Einfache Geometrien manipulieren lernen
- Eigene Python-Scripts importieren können
- Function auf eine ganze Spalte einer (Geo-) DataFrame anwenden können.

1.14 For Loops

Nirgends ist der Aspekt der Automatisierung so sichtbar wie in For Loops. Loops sind «Schleifen» wo eine Aufgabe so lange wiederholt wird, bis ein Ende erreicht worden ist. Auch For-Loops sind im Grunde genommen sehr einfach:

```
for platzhalter in [0,1,2]:
    print("Iteration",platzhalter)
```

```
Iteration 0
Iteration 1
Iteration 2
```

- for legt fest, dass eine For-Loop beginnt

- Nach «for» kommt eine Platzhalter-Variabel, die beliebig heissen kann. Diese Variabel verändert sich innerhalb des-Loops
- Nach dem Platzhalter kommt der Begriff «in»
- Nach in wird der «Iterator» festgelegt, also worüber der For-Loop iterieren soll (hier: über eine Liste mit den Zahlen 0,1,2).
- Danach kommt, wie schon bei der Funktion ein Doppelpunkt «:» der Zeigt: «Nun legen wir gleich fest was im For-Loop passieren soll»
- Auf einer neuen Zeile wird eingerückt festgelegt, was in der For-Loop passieren soll. In unserem Fall wird wieder etwas Nonsens in die Konsole ausgespukt. Achtung: return() gibt's in For-Loops nicht.
- Der Output des obigen For-Loops sieht folgendermassen aus:

```
Iteration 0
Iteration 1
Iteration 2
```

- Die Variabel «platzhalter» hat nach Beendigung des-Loops den gespeichert, der als letztes verwendet wurde:

```
platzhalter
```

```
2
```

1.14.1 Übung 1: Erste For-Loop erstellen

Erstelle eine For-Loop, die über eine Liste von 3 Namen iteriert, und jede Person in der Liste grüsst (Output in die Konsole mittels `print`).

```
for name in ["Il Buono", "Il Brutto", "Il Cattivo"]:
    print("Ciao ",name)
```

```
Ciao  Il Buono
Ciao  Il Brutto
Ciao  Il Cattivo
```

1.14.2 Übung 2: For-Loop mit `range()`

Im Dummy-Beispiel der Einführung iterieren wir über eine Liste mit den Werten `0`, `1` und `2`. Wenn wir aber über viele Werte iterieren wollen, ist es zu mühsam händisch eine Liste mit allen Werten zu erstellen. Mit range(n) erstellt Python ein Iterator mit den Zahlen `0` bis `n`. Repliziere den For-Loop aus der Einführung und ersetzte `[0,1,2]` mit `range(3)`.

```
for platzhalter in range(3):
    print("Iteration",platzhalter)
```

```
Iteration 0
Iteration 1
Iteration 2
```

1.14.3 Input: Output aus For-Loop

Bis jetzt haben wir lediglich Sachen in die Konsole herausgeben lassen, doch wie schon bei Functions ist der Zweck einer For-Loop meist, dass nach Durchführung etwas davon zurückbleibt. return() gibt es aber bei For-Loops nicht. Nehmen wir folgendes Beispiel (Brooks, M., 1997):

```
for schlagwort in ["bitch","lover","child","mother","sinner","saint"]:
    liedzeile = "I'm a "+ schlagwort
    print(liedzeile)
```

```
I'm a bitch
I'm a lover
I'm a child
I'm a mother
I'm a sinner
I'm a saint
```

Der Output von dieser For-Loop sind zwar sechs Liederzeilen, wenn wir die Variabel `liedzeile` anschauen ist dort nur das Resultat aus der letzten Durchführung gespeichert. Das gleiche gilt auch für die variabel `schlagwort`.

```
liedzeile
```

```
"I'm a saint"
```

```
schlagwort
```

```
'saint'
```

Das verrät uns etwas über die Funktionsweise des *For-Loops*: Bei jedem Durchgang werden die Variablen immer wieder überschrieben. Wenn wir also den Output des ganzen For-Loops abspeichern wollen, müssen wir dies etwas vorbereiten. Dafür erstellen wir unmittelbar for dem For-Loop einen leeren Behälter, zum Beispiel eine leere Liste (`strophe = []`). Nun können wir innerhalb des *Loops* `append()` nutzen, um den Output von einem Durchgang dieser Liste hinzu zu fügen.

```
strophe = []

for schlagwort in ["bitch","lover","child","mother","sinner","saint"]:
    liedzeile = "I'm a "+ schlagwort
    strophe.append(liedzeile)

strophe
```

```
["I'm a bitch",
 "I'm a lover",
 "I'm a child",
 "I'm a mother",
 "I'm a sinner",
 "I'm a saint"]
```

1.14.4 Übung 3: Output aus For-Loop speichern

Erweitere deinen *For-Loop* aus Übung 1 so, dass der Output aller Durchgänge in einer Liste gespeichert werden.

```
mylist = []

for name in ["Il Buono", "Il Brutto", "Il Cattivo"]:
    mylist.append("Ciao "+name)

mylist
```

```
['Ciao Il Buono', 'Ciao Il Brutto', 'Ciao Il Cattivo']
```

1.15 Zeckenstich Simulation mit Loop

Letzte Woche hattet ihr alle Punkte Zeckenstich-GeoDataFrame einmal zufällig verschoben. Nun wo ihr die Funktionsweise von Loops kennt, könnt ihr diesen Schritt eine beliebige Anzahl mal wiederholen. Um auf den Stand der letzten Woche zu kommen müsst ihr folgende Schritte ausführen:

1. Importiert die notwenigen Module (pandas, geopandas, random)
2. Importiert die notwendigen Functions (Point aus shapely.geometry sowie die selbst erstellte Function point_offset())
3. Importiert den Zeckenstichdatensatz
4. Rekonstruiert Übung 10 und 11 aus letzter Woche um die Zeckenstiche 1x zufällig zu verschieben

Der Code für diese Schritte 1 – 4 lauten folgendermassen:

```
# Schritt 1
import pandas as pd
import geopandas as gpd
import random

# Schritt 2
from shapely.geometry import Point

def point_offset(point, distance = 100):
    x_alt = point.x
    y_alt = point.y

    distance = int(distance)

    x_neu = x_alt + random.randrange(-distance,distance)
    y_neu = y_alt + random.randrange(-distance,distance)

    point_off = Point(x_neu, y_neu)

    return(point_off)

# Schritt 3
zeckenstiche = gpd.read_file("zeckenstiche.shp")

# Schritt 4
zeckenstiche_sim = gpd.GeoDataFrame()
```

(continues on next page)

(continued from previous page)

```
zeckenstiche_sim["geometry"] = zeckenstiche["geometry"].apply(point_offset)
zeckenstiche_sim["ID"] = zeckenstiche["ID"]
```

```
/opt/hostedtoolcache/Python/3.7.9/x64/lib/python3.7/site-packages/geopandas/_compat.
↪py:88: UserWarning: The Shapely GEOS version (3.8.0-CAPI-1.13.1 ) is incompatible␣
↪with the GEOS version PyGEOS was compiled with (3.8.1-CAPI-1.13.3). Conversions␣
↪between both will be slow.
  shapely_geos_version, geos_capi_version_string
```

1.15.1 Übung 1: Mit For-Loop Monte Carlo Simulation durchführen

Kombiniere den Code aus Schritt 4 (siehe vorherige Seite) mit deinem Wissen über Loops, um diese einmalige Verschiebung der Punkte 50-mal (mit `range(50)`) zu wiederholen. Denk daran: Du brauchst vor dem Loop eine leere Liste (z.B. `monte_carlo = []`) damit du den Output aus jedem Loop mit `append()` abspeichern kannst. Erstelle auch eine neue Spalte `Run_Nr` mit der Nummer der Durchführung (die du vom Platzhalter erhältst).

```
monte_carlo = []
for i in range(50):
    zeckenstiche_sim = gpd.GeoDataFrame()
    zeckenstiche_sim["geometry"] = zeckenstiche["geometry"].apply(point_offset)
    zeckenstiche_sim["ID"] = zeckenstiche["ID"]
    zeckenstiche_sim["Run_Nr"] = i
    monte_carlo.append(zeckenstiche_sim)
```

1.15.2 Übung 2: GeoDataFrames aus Simulation zusammenführen

Schau dir die Outputs an.

- Mit `type()`:
  - Was für ein Datentyp ist `zeckenstiche_sim`?
  - Was für ein Datentyp ist `monte_carlo`?
- Mit `len()`:
  - Wie vielen Elemente hat `zeckenstiche_sim`?
  - Wie viele Elemente hat `monte_carlo`?

Worauf ich hinaus will: `zeckenstiche_sim` ist eine *GeoDataFrame* und `monte_carlo` ist eine Liste von *GeoDataFrames*. Glücklicherweise kann man eine Liste von ähnlichen GeoDataFrames (ähnlich im Sinne von: gleiche Spaltennamen und -typen) mit der Funktion `concat()` aus `pandas` zu einer einzigen *GeoDataFrame* zusammenführen. Führe die Funktion aus und speichere den Output als `monte_carlo_df`.

```
type(zeckenstiche_sim)
```

```
geopandas.geodataframe.GeoDataFrame
```

```
type(monte_carlo)
```

```
list
```

```
len(zeckenstiche_sim)
```

```
1076
```

```
len(monte_carlo)
```

```
50
```

```
monte_carlo_df = pd.concat(monte_carlo)
```

1.15.3 Übung 3: Simulierte Daten visualisieren

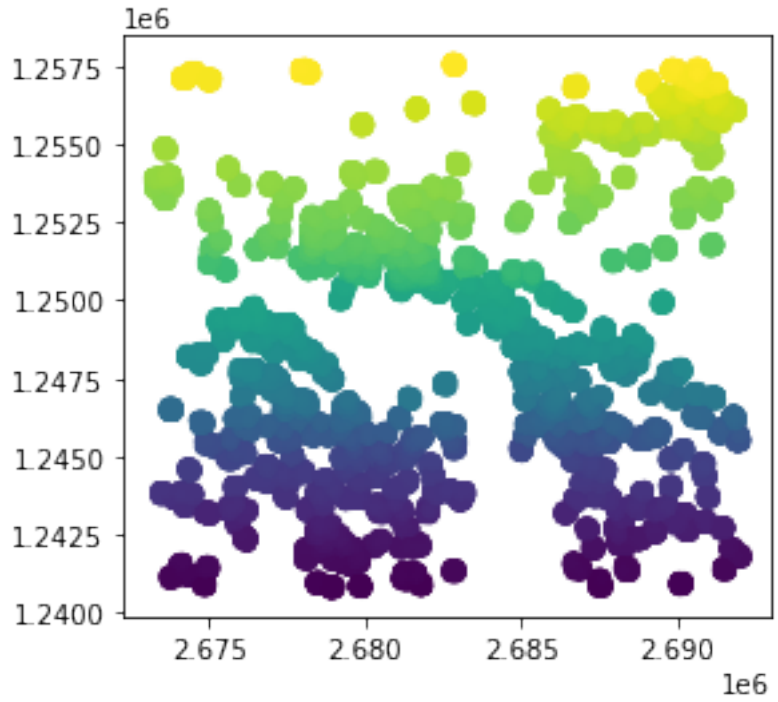
Exploriere nun `monte_carlo_df`? Was ist es für ein Datentyp? Was hat es für Spalten? Visualisiere den Datensatz räumlich mit `monte_carlo_df.plot()`. Optional: Wenn du Punkte mit der gleichen ID einer Farbe zuweisen möchtest, dann erreichst du dies mit der Option `column = "ID"` (das geht natürlich nur, wenn du im Loop auch eine solche Spalte erstellt hast)

```
monte_carlo_df.head()
```

```
                          geometry  ID  Run_Nr
0  POINT (2679065.000 1240806.000)   0       0
1  POINT (2679864.000 1240956.000)   1       0
2  POINT (2687449.000 1240804.000)   2       0
3  POINT (2674903.000 1241004.000)   3       0
4  POINT (2681762.000 1240910.000)   4       0
```

```
monte_carlo_df.plot(column = "ID")
```

```
<AxesSubplot:>
```




```
monte_carlo_df.to_file("monte_carlo_df.shp")
```

1.16 Waldanteil berechnen

Nun sind wir so weit, dass wir 50 Simulation der Zeckenstiche mit zufällig verschobenen Punkten vorbereitet haben. Wir haben also die gleiche Ausgangslage, mit der ihr den Themenblock «Datenqualität und Unsicherheiten» gestartet habt. Nun geht es darum:

1. Für jeden Simulierten Punkt zu bestimmen ob er innerhalb oder ausserhalb des Waldes liegt
2. Den Anteil der Punkte im Wald pro Simulation zu bestimmen
3. Die Verteilung dieser prozentualen Anteile über alle Simulationen zusammen zu fassen (mit einem Boxplot)

Lade dafür den Datensatz “Wald.zip” von Moodle herunter und entpacke das Shapefile in deine *Working Directory*. Importiere den Datensatz mit `pd.read_file()` und speichere es als Variable `wald`.

```
import pandas as pd
import geopandas as gpd

monte_carlo_df = gpd.read_file("monte_carlo_df.shp")

monte_carlo_df

wald = gpd.read_file("wald.shp")
zeckenstiche = gpd.read_file("zeckenstiche.shp")
```

```
/opt/hostedtoolcache/Python/3.7.9/x64/lib/python3.7/site-packages/geopandas/_compat.
↪py:88: UserWarning: The Shapely GEOS version (3.8.0-CAPI-1.13.1 ) is incompatible␣
↪with the GEOS version PyGEOS was compiled with (3.8.1-CAPI-1.13.3). Conversions␣
↪between both will be slow.
  shapely_geos_version, geos_capi_version_string
```

1.16.1 Übung 1: Wald oder nicht Wald?

Als erstes stellt sich die Frage, welche Punkte sich innerhalb eines Wald-Polygons befinden. In GIS Terminologie handelt es sich hier um einen *Spatial Join*.

Spatial Join ist als Funktion im Modul `geopandas` mit dem namen sjoin vorhanden. Wie auf der Hilfeseite beschrieben, müssen wir dieser *Function* zwei *GeoDataFrames* übergeben, die ge-joined werden sollen. Es können weitere, optionale Parameter angepasst werden, doch bei uns passen die Default werte.

Führe `gpd.sjoin()` auf die beiden Datensätze `monte_carlo_df` und `wald` aus. Beachte, das die Reihenfolge, mit welchen du die beiden GeoDataFrames der Funktion übergibst eine Rolle spielt. Versuche beide Varianten und wähle die korrekte aus. Stelle dir dazu die Frage: Was für ein Geometrietyp (Punkt / Linie / Polygon) soll der Output haben? Speichere den Output als `mote_carlo_sjoin`. Hinweis: Allenfalls müssen das Koordinatensystem der beiden GeoDataFrames nochmals explizit gesetzt werden (z.B. mit `wald.set_crs(epsg = 2056,allow_override = True)`)

```
monte_carlo_df = monte_carlo_df.set_crs(epsg = 2056)
wald = wald.set_crs(epsg = 2056,allow_override = True)

monte_carlo_sjoin = gpd.sjoin(wald,monte_carlo_df)

monte_carlo_sjoin.head()
```

```
   Wald     Shape_Leng    Shape_Area  \
0     1  921225.341854  7.963237e+07
0     1  921225.341854  7.963237e+07
0     1  921225.341854  7.963237e+07
0     1  921225.341854  7.963237e+07
0     1  921225.341854  7.963237e+07

                                            geometry  index_right   ID  Run_Nr
0  POLYGON Z ((2689962.355 1245335.250 644.591, 2...        34650  218      32
0  POLYGON Z ((2689962.355 1245335.250 644.591, 2...        42182  218      39
0  POLYGON Z ((2689962.355 1245335.250 644.591, 2...        43258  218      40
0  POLYGON Z ((2689962.355 1245335.250 644.591, 2...        24966  218      23
0  POLYGON Z ((2689962.355 1245335.250 644.591, 2...        16358  218      15
```

1.16.2 Übung 2: Anteil der Punkte im Wald

Wenn wir mit `len()` die Anzahl Zeilen zwischen `monte_carlo_df` und `mote_carlo_sjoin` vergleichen, sehen wir, dass sich die Anzahl von 53'800 auf 24'021 reduziert hat. Im Mittel, über alle Simulationen, befinden sich also 44.5% ($\frac{24021}{53800}$) der Zeckenstiche im Wald. Das ist ja schon mal spannend, aber uns interessiert ja vor allem die Verteilung dieser Mittelwerte zwischen den Simulationen.

Wir müssen also die von sjoin übrig gebliebenen Punkte pro Durchgang zählen und durch ursprüngliche Anzahl Zeckenstiche dividieren. Ursprünglich waren es 1'076 Punkte, wie uns len(zeckenstiche) zeigt.

In der Logik: DataFrame => Nach «Durchgang» Gruppieren => zählen => dividieren. Genau in dieser Schreibweise können wir unsere Befehlskette aufbauen:

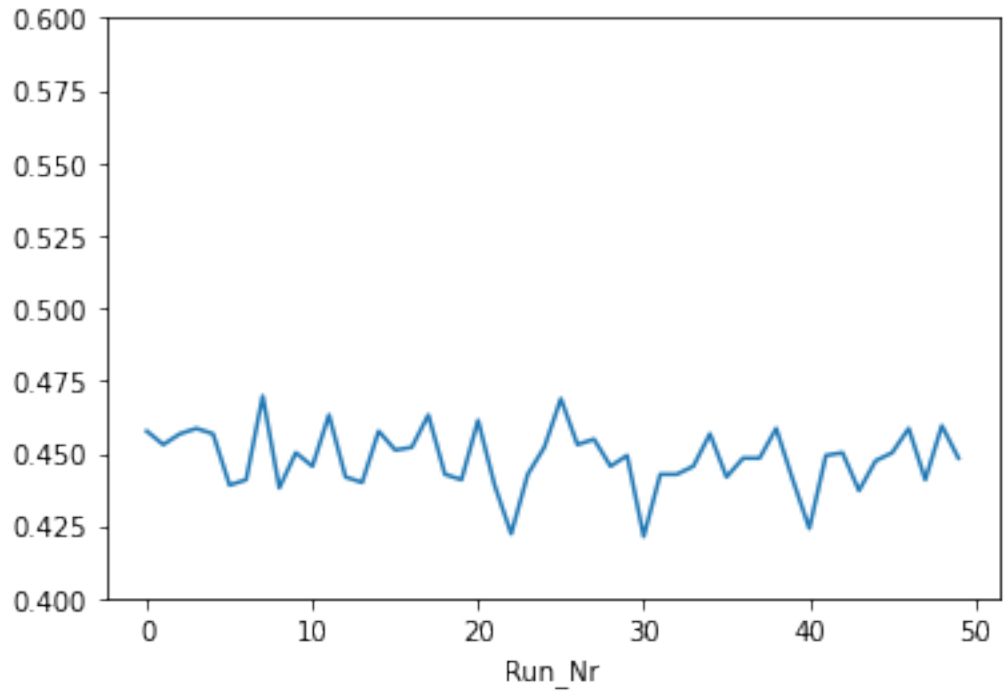
```
mittelwerte = monte_carlo_sjoin.groupby("Run_Nr").size()/1075
```

1.16.3 Übung 3: Mittelwerte Visualisieren

Gratuliere! Wenn du an diesem Punkt angekommen bist hast du eine ganze Monte Carlo Simulation von A bis Z mit Python durchgeführt. Von hier an steht dir der Weg frei für noch komplexere Analysen. Zum Abschluss kannst du die Mittelwerte wir nun auf einfache Weise visualisieren. Versuche dabei die Methods `plot()` und `plot.box()` sowie `plot.hist()`.

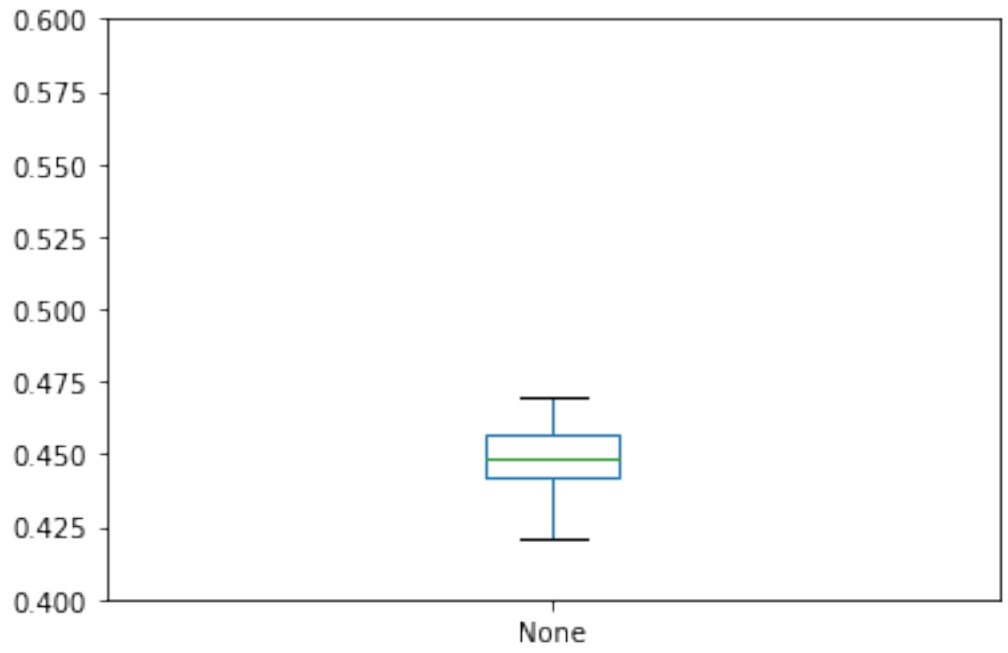
```
mittelwerte.plot().set_ylim(0.4,0.6)
```

```
(0.4, 0.6)
```




```
mittelwerte.plot.box().set_ylim(0.4,0.6)
```

```
(0.4, 0.6)
```




```
mittelwerte.plot.hist()
```

```
<AxesSubplot:ylabel='Frequency'>
```