Coding in GIS

Nils Ratnaweera

02.10.2020

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 verfügbarer Software lösen kann.

Die Voraussetzung für dieser Kurs ist eine Offenheit, neue Tools und Ansätze kennen zu lernen, die Bereitschaft für lösungorientiertes Arbeiten sowie etwas Hartnäckigkeit.

Dieses Buch auch als pdf version verfügbar

Wir empfehlen, dass ihr im Unterricht die Online Version dieser Übungsunterlagen nutzt. Diese spiegeln immer den neusten Stand, sind responsive (passen sich an Endgeräte wie Tablets usw. an) und können die Musterlösungen interaktiv darstellen (sobald diese freigeschaltet sind).

Als Doku für euch ist aber auch eine PDF Version der Unterlagen verfügbar. Speichert euch die erst am ende vom Kurs ab, damit ihr die neuste Version inkl. allen Musterlösungen habt.

- online (**empfohlen**): https://ratnanil.github.io/codingingis
- pdf (nur für Doku / Notizen): https://github.com/ratnanil/codingingis/raw/master/codingingis.pdf

---

Noch ein paar Hinweise zur Handhabung dieses Dokumentes:

- Die Musterlösungen zu allen Aufgaben stehen bereit. Wir werden diese bald einblenden
- Wenn sich im Fliesstext (Python- oder R-) Code befindet, wird er in `dieser Festschriftart` dargestellt
- Englische Begriffe, deren Übersetzung eher verwirrend als nützlich wären, werden *in dieser Weise* hervorgehoben
- Da viele von euch bereits Erfahrung in R haben, stelle ich immer wieder den Bezug zu dieser Programmiersprache her.
- Alleinstehende Codezeilen werden folgendermassen dargestellt:
- ```
  print("Coding in GIS!")
  ```
- Der gesamte Quellcode um dieses Buch zu erstellen ist in dem folgenden github-repo verfügbar: ratnanil/codingingis.
  ```

KAPITEL 1

Einleitung zu diesem Block

In diesem Block bekommt ihr euren ersten Kontakt mit Python und lernt dabei auch gerade JupyterLabs kennen, um mit Python zu interagieren. Um euch den Einstieg zu erleichtern müsst ihr noch nichts lokal auf euren Rechnern installieren, sondern könnt auf einem ZHAW-Server arbeiten. Ihr könnt euch mit folgendem Link und eurem ZHAW Kürzel (ohne „@students.zhaw.ch“) und Passwort einloggen:

jupyterhub01.zhaw.ch

Um die Übungen zu lösen, könnt ihr nach dem Einloggen wie folgt Vorgehen (siehe dazu auch die Vorlesungsfolien)

1. Erstellt einen neuen Ordner (z.B. „CodinginGIS“)
2. Erstellt darin ein neues Jupyter-Notebook-File (File > New > Notebook)
3. Bennent das File um (z.B. in „CodinginGIS_1.ipynb“)
4. Startet den Variable Inspector

Nun könnt ihr mit den Übungen beginnen. Ich empfehle, jede Übung mit einer „Markdown“-Zelle zu starten, um eure Lösung zu gliedern.

Übungsziele

- JupyterLabs aufstarten, kennenlernen und bei Bedarf personalisieren
- Python kennen lernen, erste Interaktionen
- Die wichtigsten Datentypen in Python kennen lernen (`bool`, `str`, `int`, `float`, `list`, `dict`)
- Pandas DataFrames kennen lernen und einfache Manipulationen durchführen

KAPITEL 2

Aufgabe 1: Primitive Datentypen

2.1 Theorie

Bei primitiven Datentypen handelt es sich um die kleinste Einheit der Programmiersprache, sie werden deshalb auch „atomare Datentypen“ genannt. Alle komplexeren Datentypen (Tabellarische Daten, Bilder, Geodaten) basieren auf diesen einfachen Strukturen. Die für uns wichtigsten Datentypen lauten: *Boolean*, *String*, *Integer* und *Float*. Das sind ähnliche Datentypen wie ihr bereits aus R kennt:

Python	R	Beschreibung	Beispiel	In Python
Boolean	Logical	Logische Werte ja / nein	Antwort auf geschlossene Fragen	`regen = True`
String	Character	Textinformation	Bern, Luzern	`stadt = "Bern"`
Integer	Integer	Zahl ohne Nachkommastelle	Anzahl Einwohner in einer Stadt	`bern = 133115`
Float	Double	Zahl mit Nachkommastelle	Temperatur	`temp = 22.5`

2.1.1 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 (diese Schreibweise muss genau beachtet werden). Beispielsweise sind das Antworten auf geschlossene Fragen.

```
regen = True # "es regnet"

sonne = False # "die Sonne scheint nicht"

type(sonne)
```

```
bool
```

Um zu prüfen, ob ein bestimmter Wert `True` oder `False` ist verwendet man `is True`. Will man also fragen ob es regnet, wir dies folgendermassen fomuliert:

```
# regnet es?
regen is True
```

```
True
```

Ob die Sonne scheint, lautet folgendermassen (natürlich müssen dazu die Variabel `sonne` bereits existieren):

```
# scheint die Sonne?
sonne is True
```

```
False
```

2.1.2 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'
```

2.1.3 Integer

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

```
bern_einwohner = 133115

type(bern_einwohner)
```

```
int
```

2.1.4 Float

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

```
bern_flaeche = 51.62

type(bern_flaeche)
```

```
float
```

2.2 Übungen

2.2.1 Übung 1.1: Variablen erstellen

Erstelle eine Variabel `vorname` mit deinem Vornamen und eine zweite Variabel `nachname` mit deinem Nachnamen. Was sind `vorname` und `nachname` für Datentypen?

```
# Musterlösung

vorname = "Guido"
nachname = "van Rossum"

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

```
str
```

2.2.2 Übung 1.2: String verbinden

„Klebe“ die beiden Variabeln mit einem Leerschlag dazwischen zusammen.

```
# Musterlösung

vorname+" "+nachname
```

```
'Guido van Rossum'
```

2.2.3 Übung 1.3: Zahl ohne Nachkommastelle

Erstelle eine Variabel `groesse_cm` mit deiner Körpergrösse in Zentimeter. Was ist das für ein Datentyp?

```
# Musterlösung

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

```
int
```

2.2.4 Übung 1.4: Zahl mit Nachkommastelle

Ermittle deine Grösse in Fuss auf der Basis von `groesse_cm` (1 Fuss entspricht 30.48 cm). Was ist das für ein Datentyp?

```
# Musterlösung

groesse_fuss = groesse_cm/30.48
type(groesse_fuss) # es handelt sich um den Datentyp "float"
```

```
float
```

2.2.5 Übung 1.5: Boolsche Variabeln

Erstelle eine boolsche Variable blond und setzte sie auf `True` wenn diese Eigenschaft auf dich zutrifft und `False` falls nicht.

```
# Musterlösung

blond = False
```

2.2.6 Übung 1.6: Einwohnerdichte

Erstelle eine Variabel `einwohner` mit der Einwohnerzahl der Schweiz (8'603'900, per31. Dezember 2019). Erstelle eine zweite Variabel `flaeche` (ohne Umlaute!) mit der Flächengrösse der Schweiz (41'285 km^2). Berechne nun die Einwohnerdichte.

```
# Musterlösung

einwohner = 8603900
flaeche = 41285

dichte = einwohner/flaeche

dichte
```

```
208.40256751846917
```

2.2.7 Übung 1.7: BMI

Erstelle eine Variabel `gewicht_kg` (kg) und `groesse_cm` (m) und berechne aufgrund von `gewicht_kg` und `groesse_m` ein BodyMassIndex ($BMI = \frac{m}{l^2}$, m: Körpermasse in Kilogramm, l: Körpergrösse in Meter).

```
# Musterlösung

gewicht_kg = 85
groesse_m = groesse_cm/100

gewicht_kg/(groesse_m*groesse_m)
```

```
25.10633270321361
```

KAPITEL 3

Komplexe Datentypen

Im letzten Kapitel haben wir primitive Datentypen angeschaut. Diese stellen eine gute Basis dar, in der Praxis haben wir aber meistens nicht *einen* Temperaturwert, sondern eine Liste von Temperaturwerten. Wir haben nicht *einen* Vornamen sondern eine Tabelle mit Vor- und Nachnamen. Dafür gibt es in Python komplexere Datenstrukturen die als Gefässe für primitive Datentypen betrachtet werden können. Auch hier finden wir viele Ähnlichkeiten mit R:

Python	R	Beschreibung	Beispiel
List	(Vector)	werden über die Position abgerufen	`hexerei = [3,2,1]`
Dict	List	werden über ein Schlüsselwort abgerufen	`langenscheidt = { ↪"trump": ↪"erdichten"}`
DataFrame	Dataframe	Tabellarische Daten	`pd. ↪DataFrame(langenscheidt)`

In Python gibt es noch weitere komplexe Datentypen wie *Tuples* und *Sets*. Diese spielen in unserem Kurs aber eine untergeordnete Rolle. Ich erwähne an dieser Stelle zwei häufig genannte Typen, damit ihr sie schon mal gehört habt:

- *Tuples*:
  - sind ähnlich wie *Lists*, nur können sie nachträglich nicht verändert werden. Das heisst, es ist nach der Erstellung keine Ergänzung von neuen Werten oder Löschung von bestehenden Werten möglich.
  - sie werden mit runden Klammern erstellt: `mytuple = (2,2,1)`
- *Sets*
  - sind ähnlich wie *Dicts*, verfügen aber nicht über `keys` und `values`
  - jeder Wert wird nur 1x gespeichert (Duplikate werden automatisch entfernt)
  - sie werden mit geschweiften Klammern erstellt: `myset = {3,2,2}`

KAPITEL 4

Aufgabe 2: Listen

4.1 Theorie

Wohl das einfachste Gefäss, um mehrere Werte zu speichern sind Python-Listen, sogenannte *Lists*. Diese *Lists* werden mit eckigen Klammern erstellt. Die Reihenfolge, in denen die Werte angegeben werden, wird gespeichert. Das erlaubt es uns, bestimmte Werte aufgrund ihrer Position abzurufen.

Eine *List* wird folgendermassen erstellt:

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

Der erste Wert wird in Python mit `0` (!!!) aufgerufen:

```
hexerei[0]
```

```
3
```

```
type(hexerei)
```

```
list
```

Im Prinzip sind *Lists* ähnlich wie *Vectors* in R, mit dem Unterschied das in Python-Lists unterschiedliche Datentypen abgespeichert werden können. Zum Beispiel 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
```

4.2 Übungen

4.2.1 Übung 2.1: 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.

```
# Musterlösung

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

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

4.2.2 Übung 2.2: Elemente aus Liste ansprechen

Wie erhältst du den ersten Eintrag in der Variable `vornamen`?

```
# Musterlösung

vornamen[0]
```

```
'Christopher'
```

4.2.3 Übung 2.3: Liste ergänzen

Listen können durch die Methode `append` ergänzt werden (s.u.). Ergänze die Listen `vornamen`, `nachnamen` und `groessen` durch je einen Eintrag.

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

```
# Musterlösung

nachnamen.append("Huck")

groessen.append(177)
```

4.2.4 Übung 2.4: Summen berechnen

Ermittle die Summe aller Werte in `groesse`. Tip: Nutze dazu `sum()`

```
# Musterlösung

sum(groessen)
```

```
695
```

4.2.5 Übung 2.5: Anzahl Werte ermitteln

Ermittle die Anzahl Werte in `groesse` . Tip: Nutze dazu `len()`

```
# Musterlösung

len(groessen)
```

```
4
```

4.2.6 Übung 2.6: Mittelwert berechnen

Berechne die durchschnittliche Grösse aller Personen in `groesse`. Tip: Nutze dazu `len()` und `sum()`.

```
# Musterlösung

sum(groessen)/len(groessen)
```

```
173.75
```

4.2.7 Übung 2.7: Minimum-/Maximumwerte

Ermittle nun noch die Minimum- und Maximumwerte aus `grossen` (finde die dazugehörige Funktion selber heraus).

```
# Musterlösung

min(groessen)
max(groessen)
```

```
182
```

KAPITEL 5

Aufgabe 3: Dictionaries

5.1 Theorie

In den letzten Übungen haben wir einen Fokus auf Listen gelegt. Nun wollen wir ein besonderen Fokus auf den Datentyp *Dictionary* legen.

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.
trump[3] [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'
```

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

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

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']
```

```
len(langenscheidt["trumpery"])
```

```
4
```

5.2 Übungen

5.2.1 Übung 3.1: Dictionary erstellen

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

```
# Musterlösung

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

5.2.2 Übung 3.2: Elemente aus Dictionary ansprechen

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

```
# Musterlösung

me["nachname"]
```

```
'van Rossum'
```

5.2.3 Übung 3.3: *Dictionary* nutzen

Nutze `me` um nachstehenden Satz (mit **deinen** *Values*) zu erstellen:

```
# Musterlösung

"Mein name ist "+me["nachname"] +", "+ me["vorname"]+" "+me["nachname"]
```

```
'Mein name ist van Rossum, Guido van Rossum'
```

```
'Mein name ist van Rossum, Guido van Rossum'
```

5.2.4 Übung 3.4: Key ergänzen

Ergänze die Dictionary `me` durch einen Eintrag „groesse“ mit (d)einer Grösse.

```
# Musterlösung

me["groesse"] = 181
```

5.2.5 Übung 3.5: Dictionary mit List

Erstelle eine neue Dictionary `people` mit den *Keys* „vornamen“, „nachnamen“ und „groesse“ und jeweils 3 Einträgen pro *Key*.

```
# Musterlösung

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

5.2.6 Übung 3.6: Einträge abrufen

Rufe den **ersten** Vornamen deiner *Dict* auf. Dazu musst du dein Wissen über Listen und Dictionaries kombinieren.

```
# Musterlösung

people["vornamen"][0]
```

```
'Christopher'
```

5.2.7 Übung 3.7: Einträge abrufen

Rufe den **dritten** Nachname deiner *Dict* auf.

```
# Musterlösung

people["nachnamen"][2]
```

```
'Kantereit'
```

5.2.8 Übung 3.8: Mittelwert berechnen

Berechne den Mittelwert aller grössen in deiner Dict

```
# Musterlösung

sum(people["groessen"])/len(people["groessen"])
```

```
172.66666666666666
```

KAPITEL 6

Aufgabe 4: Tabellarische Daten

6.1 Theorie

Schauen wir uns nochmals die *Dictionary* `people` aus der letzten Übung an. Diese ist ein Spezialfall einer Dictionary: Jeder Eintrag besteht aus einer Liste von gleich vielen Werten. Wie bereits erwähnt, kann es in einem solchen Fall sinnvoll sein, die Dictionary als Tabelle darzustellen.

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

```
import pandas as pd # Was diese Zeile beudeutet lernen wir später

people_df = pd.DataFrame(people)

people_df
```

```
      vornamen  nachnamen  groessen
0  Christopher      Annen       174
1      Henning        May       182
2      Severin  Kantereit       162
```

6.2 Übungen

6.2.1 Übung 4.1: von einer *Dictionary* zu einer *DataFrame*

Importiere `pandas` und nutze die Funktion `DataFrame` um `people` in eine DataFrame umzuwandeln (siehe dazu das beispiel oben). Weise den Output der Variable `people_df` zu und schaue es dir im *Variable Explorer* an.

```
# Musterlösung

import pandas as pd

people_df = pd.DataFrame(people)
```

6.2.2 Übung 4.2: *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 und suche anschliessend die erstellte csv-Datei.

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

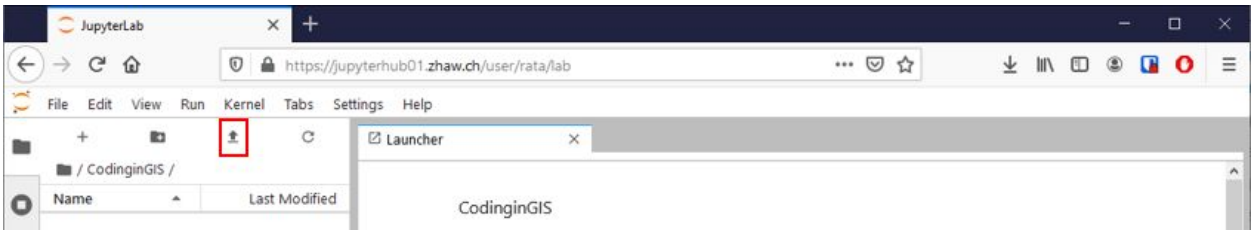
6.2.3 Übung 4.3: CSV als *DataFrame* importieren

Genau so einfach ist es eine csv zu importieren. Lade hier die Datei „zeckenstiche.csv“ (Rechtsklick –> Ziel speichern unter) herunter und speichere es im aktuellen Arbeitsverzeichnis ab. Importiere mit folgendem Code die Datei „zeckenstiche.csv“. Schau dir `zeckenstiche` nach dem importieren im „Variable Inspector“ an.

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

Achtung!

- Wenn du auf dem JupyterHub Server arbeitest dann ist dein Arbeitsverzeichnis ebenfalls *auf dem Server*. Das heisst, du musst „zeckenstiche.csv“ auf den Server hochladen. Dies kannst du mit dem Button „Upload Files“ im Tab „File Browser“ bewerkstelligen (s.u.).



- Der Code (`pd.read_csv("zeckenstiche.csv")`) funktioniert nur, wenn „zeckenstiche.csv“ im aktuellen Arbeitsverzeichnis (*Current Working Directory*) abgespeichert ist. Wenn du nicht sicher bist, wo dein aktuelles Arbeitsverzeichnis liegt, kannst du dies mit der Funktion `os.getcwd()` (**get c**urrent **w**orking **d**irectory) herausfinden (s.u.).

```
import os
os.getcwd()
```

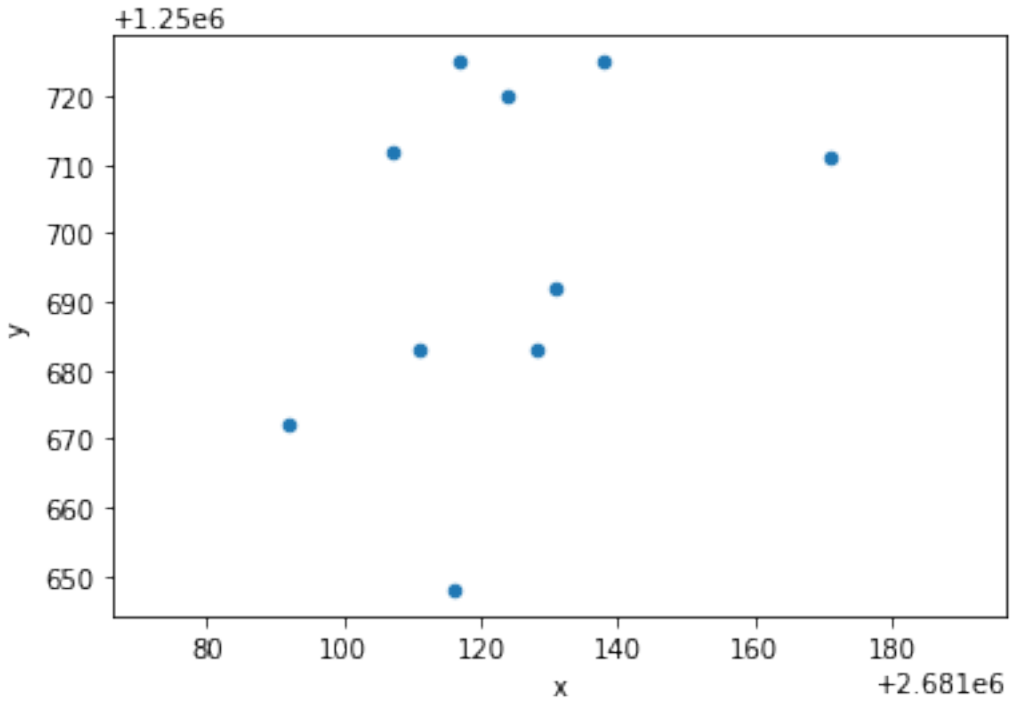
6.2.4 Übung 4.4: Koordinaten räumlich darstellen

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. Überlege dir, was die zweite Zeile bewirkt und warum dies sinnvoll ist.

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

fig.axis("equal")
```

```
(2681088.05, 2681174.95, 1250644.15, 1250728.85)
```




```
# Musterlösung

# fig.axis("equal") sorgt dafür, dass die Skala der beiden Achsen
# (x und y) gleich sind. Dies ist desshalb sinnvoll, da es sich um
# räumliche Koordinaten handelt und die Distanzen in Richtung "Nord-Süd"
# (y-Achse) sowie in "West-Ost" (x-Achse) die gleiche Skala haben (Meter)
# https://matplotlib.org/3.1.3/api/_as_gen/matplotlib.pyplot.axis.html
```

6.2.5 Übung 4.5: Einzelne Spalte selektieren

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

```
# Musterlösung

zeckenstiche["ID"]
```

```
0     2550
1    10437
2     9174
3     8773
4     2764
5     2513
6     9185
7    28521
8    26745
9    27391
Name: ID, dtype: int64
```

6.2.6 Übung 4.6: Neue Spalte erstellen

Auch das Erstellen einer neuen Spalte ist identisch mit der Erstellung eines neuen *Dictionary* Eintrags. Erstelle eine neue Spalte „Stichtyp“ mit dem Wert „Zecke“ auf jeder Zeile (s.u.).

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

```
zeckenstiche
```

	ID	accuracy	x	y	Stichtyp
0	2550	439.128951	2681116	1250648	Zecke
1	10437	301.748542	2681092	1250672	Zecke
2	9174	301.748542	2681128	1250683	Zecke
3	8773	301.748542	2681111	1250683	Zecke
4	2764	301.748529	2681131	1250692	Zecke
5	2513	301.748529	2681171	1250711	Zecke
6	9185	301.748542	2681107	1250712	Zecke
7	28521	301.748542	2681124	1250720	Zecke
8	26745	301.748542	2681117	1250725	Zecke
9	27391	301.748542	2681138	1250725	Zecke

KAPITEL 7

Einleitung zu diesem Block

Letzte Woche habt ihr Jupyter Labs kennen gelernt und erste Kontakte mit Python durch Jupyter Labs gehabt, dazu habt ihr *conda* verwendet. Diese Woche widmen wir etwas mehr Zeit, *conda* zu beherrschen und lernen zudem mehr über Module und Functions.

Übungsziele

- Conda beherrschen
  - neue *Environment* erstellen
  - *Modules* in eine *Environment* installieren
  - *Jupyter Lab* in einer *Environment* nutzen
- *Functions* kennenlernen und beherrschen
- *Function* auf eine ganze Spalte einer DataFrame anwenden können.

KAPITEL 8

Conda cheat sheet

In der folgenden Tabelle werden die Einzelschritte in der Verwendung von Conda näher beschrieben. Wichtig ist vor allem, wann dieser Schritt nötig ist und wie er ausgeführt wird. Um die Tabelle kompakt zu halten werden gewisse Details als Fussnote verlinkt.

Schritt	Wann ist dies nötig?	Details zum Vorgehen / Befehl für die Konsole[1]
1. Conda installieren (installiert das Program *conda*)	einmalig (ist nicht nötig, wenn ArcGIS Pro installiert ist)	Miniconda (empfohlen) oder anaconda herunterladen und installieren
2. Systemvariable setzen (vermittelt der Konsole, wo das Programm *conda* installiert ist)	einmalig und nur, wenn folgender Befehl in der Konsole eine Fehlermeldung verursacht: `conda --version`	Pfad zur *conda*-installation[2] in die Umgebungsvariable „Path“ einfügen[3]
3. *Virtual environment* erstellen (erstellt eine neue Arbeitsumgebung)	einmal pro Projekt nötig (wobei eine environment auch wiederverwendet werden kann)	in der Konsole: `conda create --name codingingis`
4. *Virutal environment* aktivieren (schaltet den „Bearbeitungsmodus“ ein)	jedes mal nöig wenn ein **Erweiterung installiert** oder **jupyter lab gestartet** werden soll	in der Konsole[4]: `activate codingingis`
5. Jupyter lab installieren (fügt der virtuellen Umgebung diese IDE hinzu)	1x pro *environment*	in der Konsole[5]: `conda install -c conda-forge jupyterlab`
6. Jupyter lab starten (startet die IDE „Jupyter Lab“)	jedes mal, wenn am Projekt gearbeitet wird	in der Konsole[5]: `jupyter lab`
7. Jupyter lab (JL) beenden (beendet „JupyterLab“ in der Console)	wenn ihr die Konsole wieder braucht	Während JL läuft, ist die Konsole blockiert. Um JL zu beenden und die Konsole freizugeben: Tastenkombination `CTRL + C`
8. weitere Module[6] installieren (fügt der *environment* zB `pandas` hinzu)	jedes mal nötig, wenn ein Modul in einer Environment fehlt[7]	in der Konsole[5][8]: `conda install -c conda-forge pandas`

[1] Mit Konsole ist unter Windows *cmd* gemeint (Windowstaste > cmd). Unter Linux wird bash, auf Mac der Terminal verwendet.

[2] Wenn *conda* von ArcGIS Pro verwendet wird, befindet sich die *conda* installation vermutlich hier: *C:\Program Files\ArcGIS\Pro\bin\Python\Scripts*. Prüfen, ob dieser Folder existiert und dort `conda.exe` vorhanden ist.

[3] Windowstaste > Umgebungsvariable für dieses Konto bearbeiten > Zeile „Path“ auswählen (doppelklick) > Neu > Pfad zur conda installation hinzufügen > mit OK bestätigen > cmd neu starten > `conda --version` nochmals eingeben.

[4] Unter Linux: `conda activate codingingis`

[5] Falls die richtige environment noch nicht aktiviert ist, muss dies zuerst noch erfolgen (z.B `activate codingingis`).

[6] In Coding in GIS I - III brauchen wir die Module `pandas`, `matplotlib`, `geopandas` und `descartes`

[7] Dies macht sich bemerkbar duch die Fehlermeldung `ModuleNotFoundError: No module named 'pandas'`

[8] Falls Jupyter Labs läuft und dadurch die Konsole blockiert ist, gibt es folgende Möglichkeiten:

1. Jupyter Labs beenden (CTRL + C) > Modul installieren > Jupyter Lab nochmal starten
2. einen neue Konsole starten > *environment* aktivieren > Modul installieren
3. den Terminal innerhalb von Jupyter Labs verwenden (File > New > Terminal) und dort die *environment* aktivieren und Modul installieren

KAPITEL 9

Python Modules

9.1 Vergleich R vs. Python

Der Umgang mit Modulen ist in Python in vielerlei Hinsicht ähnlich wie in R. An dieser stelle möchten wir die Unterschiede in einem Direktvergleich beleuchten. Dafür verwenden wir **ein fiktives Modul** namens `maler`, in Anlehnung an die Analogie des Haus bauens mit Spezialisten (siehe Vorlesungsfolien). Nehmen wir an, dieses Modul existiert als Python Modul wie auch als R Library.

9.1.1 Erweiterung installieren

In R ist die Installation einer *Library* selbst ein R-Befehl und wird innerhalb von R ausgeführt. Wenn wir keine Quelle angeben, woher die Library heruntergeladen werden soll, wird eine Default-Quelle verwendet, die im System hinterlegt ist (z.B. „https://cloud.r-project.org“).

In Python ist dies leider etwas komplizierter, es braucht für die Installation einer Python *Library* eine Zusatzsoftware wie zum Beispiel `conda` (siehe dazu das Kapitel *Conda cheat sheet*). Es gibt auch noch andere Wege, wie zum Beispiel `pip`, aber diese lassen wir der Einfachheit an dieser Stelle weg.

in R*:

```
install.packages("maler")
```

In Python**:

```
conda install -c conda-foge maler
```

9.1.2 Erweiterung laden

Um eine Erweiterung nutzen zu können, müssen wir diese sowohl in R wie auch in Python in die aktuelle Session importieren. In R und Python sehen die Befehle folgendermassen aus:

in R:

```
library(maler)
```

in Python:

```
import maler
```

9.1.3 Erweiterung verwenden

Um eine Funktion aus einer *Library* in R zu verwenden, kann ich diese *Function* direkt aufrufen. In Python hingegen muss ich entsprechende Erweiterung der *Function* mit einem Punkt voranstellen.

Das ist zwar umständlicher, dafür aber weniger Fehleranfällig. Angenommen, zwei leicht unterschiedliche Funktionen heissen beide `wand_bemalen()`. Die eine stammt aus der Erweiterung `maler`, die andere aus der Erweiterung `maurer`. Wenn die Funktion in R aufgerufen wird ist nicht klar, aus welcher Library die Funktion verwendet werden soll. In Python ist im nachstehenden Beispiel unmissverständlich, dass `wand_bemalen()` aus dem Modul `maler` gemeint ist.

in R:

```
wand_bemalen()
```

in Python:

```
maler.wand_bemalen()
```

9.2 Python Eigenheiten

In Python gibt es in Bezug auf die Verwendung von Modulen ein paar Eigenheiten, die wir aus der R Welt nicht kennen. Es ist wichtig diese Eigenheiten zu kennen, denn man trifft sie immer wieder an.

9.2.1 Modul mit Alias importieren

Da es umständlich sein kann, jedesmal `maler.wand_bemalen()` voll auszuschreiben, können wir dem Modul beim Import auch einen „Alias“ vergeben. Für gewisse populäre Module haben sich solche Aliasse eingebürgert. Beispielsweise wird `pandas` meist mit dem Alias `pd` importiert. Es es ist sinnvoll, sich an diese Konventionen zu halten. Übertragen auf unser `maler` beispiel sieht der Import mit einem Alias folgendermassen aus:

```
import maler as m     # importiert "maler" mit dem Alias "m"
m.wand_bemalen()      # nun wird "m." vorangestellt statt "maler."
```

9.2.2 Einzelne *Function* importieren

Es gibt noch die Variante, explizit eine spezifische *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 maler import wand_bemalen # importiert nur die Funktion "wand_bemalen"
wand_bemalen()                 # das voranstellen von "maler." ist nun nicht nötig
```

9.2.3 Alle *Functions* importieren

Zusätzlich ist es möglich, **alle** *Functions* aus einem Modul so zu importieren, dass der Modulname nicht mehr erwähnt werden muss. Diese Notation wird nicht empfohlen, aber es ist wichtig sie zu kennen.

```
from maler import * # importier alle Funktionen (*) von "maler"
wand_bemalen()      # das voranstellen von "maler." ist nun nicht nötig
```

KAPITEL 10

Aufgabe 5: *Function* Basics

10.1 Theorie

Ein Grundprinzip von Programmieren ist „DRY“ (*Don’t repeat yourself*). Wenn unser Script sehr viele gleiche oder sehr ähnliche Codezeilen enthält ist das ein Zeichen dafür, dass man besser eine *Function* schreiben sollte. Das hat viele Vorteile: Unter anderem wird der Code wird lesbarer, einfacher zu warten und kürzer.

Um mit Python gut zurecht zu kommen ist das schreiben von eigenen *Functions* unerlässlich. Sie sind 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.

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. Beispielsweise könnten wir der *Function* unseren Vornamen übergeben, damit wir persönlich gegrüsst werden:

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

Nun können wir der Function ein Argument übergeben. In folgendem Beispiel ist `vorname` ein Parameter, „Guido“ ist sein Argument.

```
sag_hallo(vorname = "Guido")
```

```
'Hallo Guido!'
```

Wir können auch eine *Function* gestalten, die mehrere Parameter annimmt. Beispielweise könnte `sag_hallo()` zusätzlich noch einen Parameter `nachname` erwarten:

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

```
sag_hallo(vorname = "Guido", nachname = "van Rossum")
```

```
'Hallo Guido van Rossum!'
```

10.2 Übungen

10.2.1 Übung 5.1: Erste *Function* erstellen

Erstelle eine Function, die `gruezi` heisst, einen Nachnamen als Input annimmt und per Sie grüsst.

```
# Musterlösung

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

```
# Das Resultat soll in etwa folgendermassen aussehen:
gruezi(nachname = "van Rossum")
```

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

10.2.2 Übung 5.2: *Function* erweitern

Erstelle eine neue Funktion `gruezi2` welche im Vergleich zu `gruezi` einen weiteren Parameter namens `anrede` annimmt.

Das Resultat soll in etwa folgendermassen aussehen:

```
# Musterlösung

def gruezi2(nachname, anrede):
    return "Guten Tag, " + anrede + " "+nachname
```

```
gruezi2(nachname = "van Rossum", anrede = "Herr")
```

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

10.2.3 Übung 5.3: Zahlen summieren

Erstelle eine Funktion `add` die zwei Zahlen summiert.

```
# Musterlösung

def add(zahl1, zahl2):
    return zahl1 + zahl2
```

```
# Das Resultat sollte folgendermassen aussehen:
add(zahl1 = 2, zahl2 = 10)
```

```
12
```

10.2.4 Übung 5.4: Quadratzahl

Erstelle eine Funktion `square`, welche den Input quadriert.

Tipp: „Quadireren“ heisst ja „mit sich selbst multiplizieren“. In Python können zwei Zahlen mit $\star$ multipliziert werden.

```
# Musterlösung

def square(zahl):
    return zahl*zahl
```

```
# Das Resultat sollte folgendermassen aussehen:
square(zahl = 5)
```

```
25
```

10.2.5 Übung 5.5: Meter in Fuss konvertieren

Erstelle eine Funktion `meter_zu_fuss`, die eine beliebige Zahl von Meter in Fuss konvertiert. Zur Errinnerung: 30.48 cm ergeben 1 Fuss.

```
# Musterlösung

def meter_zu_fuss(meter):
    return meter/0.3048
```

```
# Das Resultat sollte folgendermassen aussehen:

meter_zu_fuss(meter = 1.80)
```

```
5.905511811023622
```

KAPITEL 11

Aufgabe 6: *Function* Advanced

11.1 Theorie

11.1.1 Standart-Werte

Man kann für einzelne (oder alle) Parameter auch Standardwerte festlegen. Das sind Werte die dann zum Zug kommen, wenn der Nutzer der Funktion das entsprechende Parameter leer lässt. Schauen wir dazu nochmals `sag_hallo()` an.

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

Um diese Funktion zu nutzen müssen dem Parameter `vorname` ein Argument übergeben, sonst erhalten wir eine Fehlermeldung.

```
sag_hallo()
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-2-92896a02d815> in <module>
----> 1 sag_hallo()

TypeError: sag_hallo() missing 1 required positional argument: 'vorname'
```

Wenn wir möchten, dass gewisse Parameter auch ohne Argument auskommen, dann könnnen wir einen Standartwert festlegen. So wird der Parameter optional. Bespielsweise könnte `sag_hallo()` einfach *Hallo Du!* zurück geben, wenn kein Vorname angegeben wird. Um dies zu erreichen, definieren wird den Standartwert bereits innerhalb der Klammer, und zwar folgendermassen:

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

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
# Wenn "vorname" nicht angegeben wird:
sag_hallo()
```

```
'Hallo Du!'
```

Wichtig

Wenn mehrere Parameter in einer Funktion definiert werden, dann kommen die optionalen Parameter **immer zum Schluss**.

11.1.2 Reihenfolge der Argumente

Wenn die Argumente in der gleichen Reihenfolge eingegeben werden, wie sie in der *Function*-Definiert sind, müssen die Parameter **nicht** spezifiziert werden (z.B: `anrede=`, `nachname=`).

```
def gruezi2(nachname, anrede):
    return "Guten Tag, " + anrede + " "+nachname

gruezi2("van Rossum", "Herr")
```

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

Wenn wir die Reihenfolge missachten, ist der Output unserer Funktion fehlerhaft:

```
gruezi2("Herr","van Rossum")
```

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

Aber wenn die Parameter der Argumente spezifiziert werden, können wir sie in jeder beliebigen Reihenfolge auflisten:

```
gruezi2(anrede = "Herr", nachname = "van Rossum")
```

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

11.1.3 Funktionen auf mehreren Zeilen

Bisher waren unsere Funktionen sehr kurz und einfach und wir benötigten dafür immer nur zwei Zeilen: Die erste Zeile begann die *Function*-Definition (`def..`) und die zweite Zeile retournierte bereits die Lösung `return(...)`. Zwischen diesen beiden Komponenten haben wir aber viel Platz, den wir uns zu Nutze machen können. Wir können hier Kommentare hinzufügen wie auch unsere Funktion in Einzelschritte aufteilen um den Code lesbarer zu machen.

```
def gruezi2(nachname, anrede):
    # Wozu ist diese Funktion da?
    # Diese Funktion soll Menschen freundlich grüssen

    gruss = "Guten Tag, " + anrede + " "+nachname
    return gruss
```

11.1.4 Globale und Lokale Variabeln

Innerhalb einer *Function* können nur die Variabeln verwendet werden, die der *Function* als Argumente übergeben (oder innerhalb der Funktion erstellt) werden. Diese nennt man „lokale“ Variabeln, sie sind lokal in der *Function* vorhanden. Im Gegensatz dazu stehen „globale“ Variabeln, diese sind Teil der aktuellen Session.

Versuchen wir das mit einem Beispiel zu verdeutlichen. Angenommen wir definieren gobal die Variabeln `nachname` und `anrede`:

```
# Wir definieren globale Variabeln
vorname = "Guido"

# Nun erstellen wir eine Function, welche diese Variabel ("vorname") nutzen soll:
def sag_hallo(vorname):
    return "Hallo " + vorname

# Wenn wir jetzt aber die Function ausführen wollen, entsteht die Fehlermeldung,
# dass "vorname" fehlt (obwohl wir vorname ja schon definiert haben)
sag_hallo()
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-8-bd26d99d0230> in <module>
      8 # Wenn wir jetzt aber die Function ausführen wollen, entsteht die␣
↪Fehlermeldung,
      9 # dass "vorname" fehlt (obwohl wir vorname ja schon definiert haben)
---> 10 sag_hallo()

TypeError: sag_hallo() missing 1 required positional argument: 'vorname'
```

11.1.5 Lambda-Function

Mit dem Begriff `lambda` kann eine *Function* verkürzt geschrieben werden. Wir werden dies im Unterricht kaum verwenden, es ist aber doch gut davon gehört zu haben. Nachstehend wird die Funktion `sag_hallo()` in der bekannten, wie auch in der verkürzten Form definiert.

Herkömmliche Weise:

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

Verkürzt mit `lambda`:

```
sag_hallo = lambda vorname: "Hallo "+vorname
```

11.2 Übungen

11.2.1 Übung 6.1: Multiplizieren

Erstelle eine Funktion namens `times`, die zwei Zahlen miteinander multipliziert.

```
# Musterlösung

def times(x,y):
    return x*y
```

```
times(2,2)
```

```
4
```

11.2.2 Übung 6.2: Optionale Parameter

Die eben erstellte Funktion `times` benötigt 2 Argumente (die miteinander multipliziert werden). Wandle den einen in Parameter einen optionalen Parameter um (mit dem Defaultwert `1`).

Zusatzaufgabe: Was passiert, wenn du den ersten Parameter in einen optionalen Parameter umwandelst?

```
# Musterlösung

def times(x,y = 1):
    return x*y
```

```
times(3)
```

```
3
```

```
# Musterlösung
# (Zusatzaufgabe)

def times(x  = 1 ,y):
    return x*y
```

```
  File "<ipython-input-10-e0d2091c9b0f>", line 1
    def times(x  = 1 ,y):
              ^
SyntaxError: non-default argument follows default argument
```

11.2.3 Übung 6.3: BMI

Erstelle eine Funktion namens `bmi`, die aus Grösse und Gewicht einen BodyMassIndex berechnet ($BMI = \frac{m}{l^2}$, m: Körpermasse in Kilogramm, l: Körpergrösse in Meter). Das Resultat soll etwa folgendermassen aussehen:

```
# Musterlösung

def bmi(groesse_m, gewicht_kg):
    return gewicht_kg / (groesse_m*groesse_m)
```

```
bmi(groesse_m=1.8, gewicht_kg=88)
```

```
27.160493827160494
```

11.2.4 Übung 6.4 Mittelwert

Erstelle eine Funktion `mean()`, welche den Mittelwert aus einer Liste (`List`) von Zahlen berechnet. Das Resultat sollte folgendermassen aussehen:

Tipp: Nutze dazu `sum()` und `len()` analog *Übung 2.6: Mittelwert berechnen*.

```
# Musterlösung

def mean(zahlen):
    return sum(zahlen)/len(zahlen)
```

```
meine_zahlen = [50, 100,550,1000]
mean(meine_zahlen)
```

```
425.0
```

11.2.5 Übung 6.5 Grad Celsius in Farenheit

Erstelle eine Funktion `celsius_zu_farenheit`, welche eine beliebige Zahl von Grad Celsius in Grad Kelvin konvertiert. Zur Erinnerung: *Temperatur in °F = Temperatur in °C x 1,8 + 32*.

```
# Musterlösung

def celsius_in_farenheit(celsius):
    return celsius*1.8+32
```

Das Resultat sollte folgendermassen aussehen:

```
celsius_in_farenheit(celsius = 25)
```

```
77.0
```

11.2.6 Übung 6.6 Lambda Function

Schreibe die letzte Funktion `celsius_zu_farenheit` in der *lambda* Notation.

```
# Musterlösung

celsius_in_farenheit2 = lambda celsius: celsius*1.8+32
```

KAPITEL 12

Aufgabe 7: Zufallszahlen generieren

12.1 Theorie

Im Block „*Datenqualität und Unsicherheit*“ hattet ihr auch mit Zufallszahlen und Simulationen auseinandergesetzt. Programmiersprachen sind für eine solche Anwendung sehr gut geeignet, und deshalb werden wir in diesem Abschnitt eine Erweiterung zur Erstellung von Zufallszahlen kennenlernen. Diese Erweiterung lautet `random` und ist teil der „Python Standard Library“, was bedeutet das wir dieses Erweiterung bereits installiert ist, und wir sie nicht installieren müssen um sie zu nutzen.

```
import random
```

Innerhalb vom `random` gibt es zahlreiche Funktionen um Zufallszahlen zu generieren, je nach dem was unsere Anforderungen an die Zufallszahl ist. Zum Beispiel könnte eine Anforderung sein, dass die Zahl innerhalb von einem bestimmten Bereich liegt (z.B. „*generiere eine Zufallszahl zwischen 1 und 10*“). Oder aber, dass sie eine ganze Zahl sein muss. Weiter könnte die Anforderung sein, dass sie aus einer bestimmten Verteilung kommen sollte, zum Beispiel einer Normalverteilung. In diesem letzten Fall müssen wir den Mittlwert sowie die Standartabweichung unserer Verteilung angeben.

Um eine ganzzahlige Zufallszahl zwischen 0 und 10 zu generieren, können wir die Funktion `randrange()` nutzen:

```
random.randrange(start = 0, stop = 10)
```

```
0
```

```
random.randrange(start = 0, stop = 10)
```

```
1
```

```
random.randrange(start = 0, stop = 10)
```

```
1
```

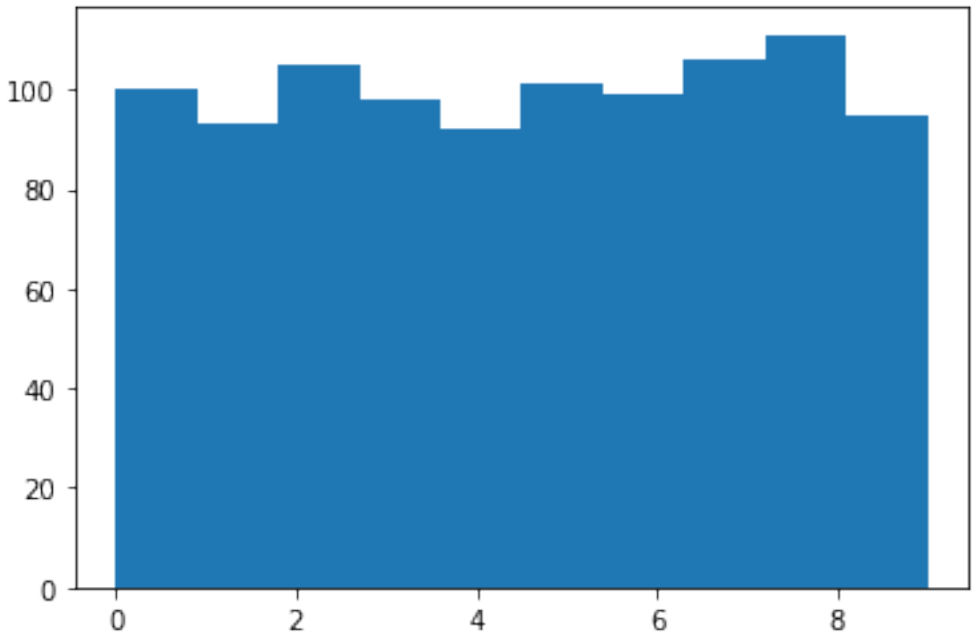
Wenn wir auf diese Weise mit `randrange()` immer wieder neue Zufallszahlen generieren fällt irgendwann auf, dass die Verteilung der Zahlen ziemlich gleichmässig ist. Es ist also gleich wahrscheinlich eine 10 zu bekommen eine eine 0 oder eine 5. Die Zahlen kommen also aus einer „uniformen“ Verteilung. Dies lässt sich auch sehr schön visualisieren. Ich generiere in den folgenden Codezeilen 1'000 zufallszahlen zwischen 0 und 10 mit der Funktion `randrange`.

```
from matplotlib import pyplot as plt
```

```
fig, ax = plt.subplots()
a = [random.randrange(0,10) for x in range(0,1000)]

ax.hist(a)

plt.show()
```



Die Funktion `randrange(0,10)` generiert nur ganzzahlige Zufallszahlen. Wenn wir aber eine Zufallszahl mit Nachkommastellen haben möchten, müssen wir die Funktion `uniform()` verwenden.

Um Zufallszahlen aus einer „Normalverteilung“ zu bekommen müssen wir die Funktion `normalvariate` nutzen. Hier müssen wir, wie Eingangs erwähnt, den Mittelwert und die Standartabweichung dieser Verteilung angeben. Tatsächlich können wir bei dieser Variante keine Minimum- und Maximumwerte festlegen. Theoretisch könnte der Generator jeden erdenklichen Zahlenwert rausspucken, am wahrscheinlichsten ist jedoch eine Zahl nahe am angegebenen Mittelwert.

```
# mu = Mittelwert, sigma = Standartabweichung
random.normalvariate(mu = 5, sigma = 2)
```

```
8.818586486923941
```

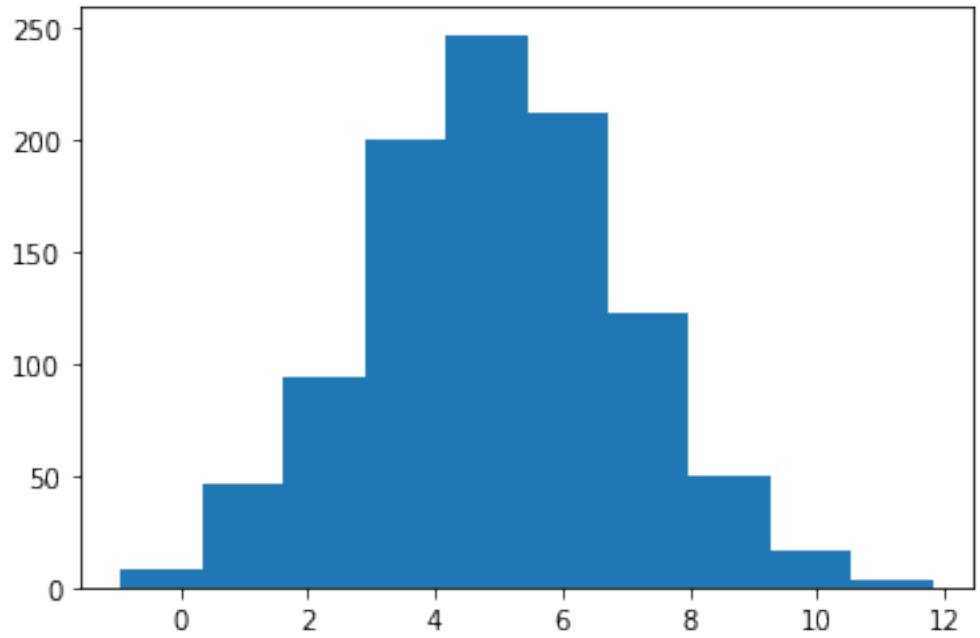
```
# mu = Mittelwert, sigma = Standartabweichung
random.normalvariate(mu = 5, sigma = 2)
```

```
4.632207532391158
```

```
# mu = Mittelwert, sigma = Standartabweichung
random.normalvariate(mu = 5, sigma = 2)
```

```
3.3760146418651114
```

Wenn wir die obige Funktion 1000x laufen lassen und uns das Histogramm der generierten Zahlen anschauen, dann zeichnet sich folgendes Bild ab.



12.2 Übungen

Nun wollen wir diesen Zufallszahlengenerator `random` nutzen um eine Funktion zu entwickeln, welche einen beliebigen Punkt (mit einer x-/y-Koordinate) zufällig in einem definierten Umkreis verschiebt. Unser Fernziel ist es, den simulierten Datensatz aus „Datenqualität und Unsicherheit“ zu rekonstruieren (siehe unten). Der erste Schritt dorthin ist es, einen gemeldeten Punkt (rot in Abb. 12.1) in einem definierten Umkreis zu verschieben.

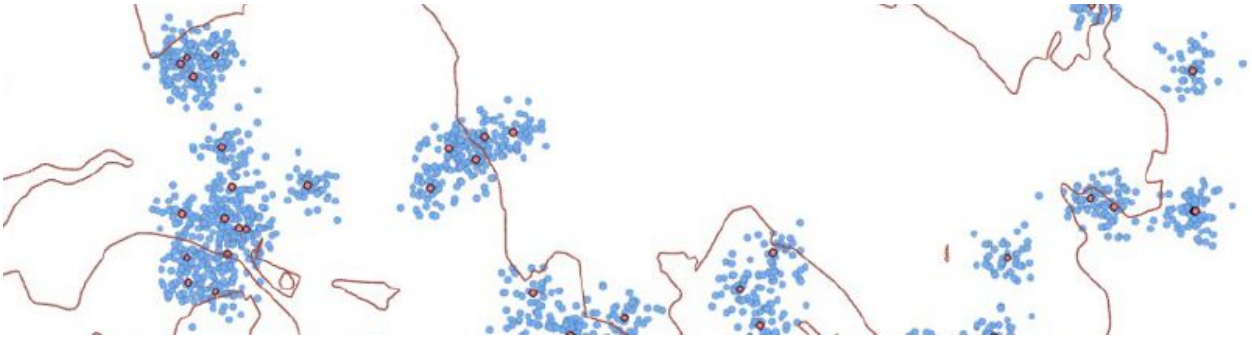


Abb. 12.1: Ausschnitt der simulierten Zeckenstiche. Der rote Punkt stellt jeweils der gemeldete Zeckenstich dar, die blaue Punktwolke drum herum sind simulierte Punkte welche die Ungenauigkeit der Daten wiederspiegelt.

Das Ziel dieser Übung ist es also, dass wir eine Funktion entwickeln, die uns einen zufälligen Punkt in der Nähe eines

Ursprungspunktes vorschlägt. Unser Vorgehen: Wir addieren jedem Koordinatenwert (`x`/`y`) des Ursprungspunktes einen Zufallswert, zum Beispiel zwischen -100 bis +100.

12.2.1 Übung 7.1: Zufallszahlen aus Uniformverteilung

Bevor wir mit Koordinaten arbeiten wollt ihr euch zuerst mit dem Modul `random` vertraut machen. Importiere das Modul `random` und generiere eine Zufallszahl zwischen -100 und +100 aus einer uniformen Verteilung sowie aus einer Normalverteilung mit Mittelwert 100 und Standartabweichung 20.

```
# Musterlösung

import random

random.uniform(-100,100)

random.normalvariate(100,20)
```

```
115.36479372811071
```

12.2.2 Übung 7.2: Dummykoordinaten erstellen

Nun wollen wir uns den Koordinaten zuwenden. Erstelle als erstes zwei Dummykoordinaten `x_start` und `y_start` mit jeweils dem Wert `0`. Diese sollen als „Ursprungskoordinaten“ dienen.

```
# Musterlösung

x_start = 0
y_start = 0
```

12.2.3 Übung 7.3: Zufallswerte generieren

Generiere nun eine Zufallszahl, die aus einer Normalverteilung stammt und die *in etwa* zwischen -100 und +100 liegt. Weise diese Zahl der Variabel `x_offset` zu. Generiere danach eine zweite Zufallszahl (auf die gleiche Art) und weise diese `y_offset` zu.

Tipp: Überlege dir, welcher *Mittelwert* Sinn macht um Werte zwischen -100 und +100 zu bekommen. Welche Zahl liegt zwischen -100 und +100?

Überlege dir als nächstes, welche Standartabweichung sinnvoll wäre. Zur Erinnerung: Etwa 68% der Werte liegen inerhalb von +/- 1 Standartabweichung (SD), 95% innerhalb von +/- 2 SD, 99% innerhalb von 3 SD (siehe unten):

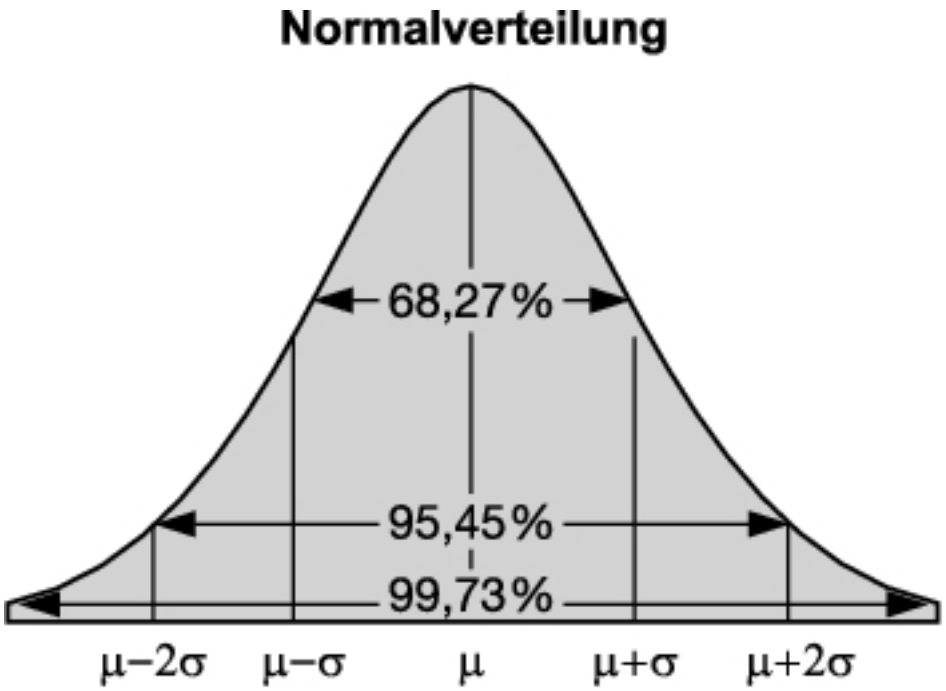


Abb. 12.2: Normalverteilung und die Anteile innerhalb von 1 Standartabweichung (Mittelwert μ minus Standartabweichung σ), 2 Standartabweichungen ($\mu - 2 \times \sigma$) und 2 Standartabweichungen ($\mu - 3 \times \sigma$). Quelle: cobocards

```
# Musterlösung

# Normalverteilte Werte mit Mittelwert 0 und Standartabweichung 100
# Achtung: bei dieser Standartabweichung sind ca 30% der Werte > 100!
x_offset = random.normalvariate(0,100)
y_offset = random.normalvariate(0,100)

x_offset
y_offset
```

```
145.71995856903638
```

12.2.4 Übung 7.4: Zufallswerte addieren

Addiere nun die Zufallszahlen `x_offset` und `y_offset` **jeweils** zu den Dummykoordinaten `x_start` und `y_start` und weise diese neuen Koordinaten `x_neu` und `y_neu` zu. Die neuen Werte stellen die leicht verschobenen Ursprungskoordinaten dar. In meinem Fall sind diese um 10.2 Meter nach Osten (positiver Wert) bzw. 4.4 Meter nach Süden (negativer Wert) verschoben worden.

```
# Musterlösung

x_neu = x_start+x_offset
y_neu = y_start+y_offset
```

```
x_neu
```

```
10.246170309600945
```

```
y_neu
```

```
-4.443904000288846
```

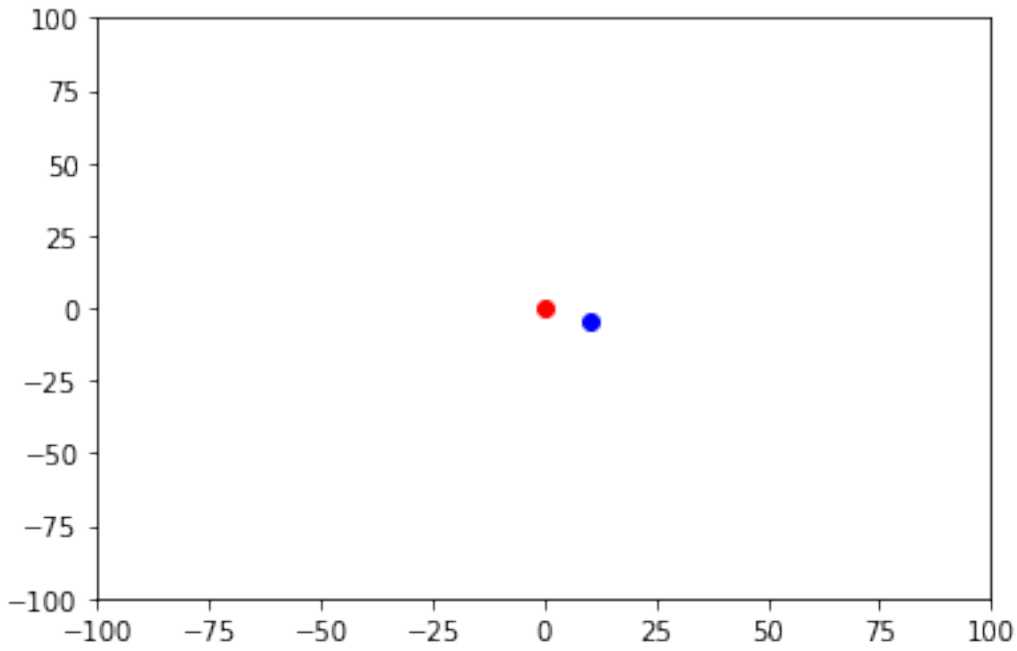
Visuell betrachtet sieht das folgendermassen aus:

```
from matplotlib import pyplot as plt
import pandas as pd


plt.scatter(x_start,y_start, color = "red") # ursprung
plt.scatter(x_neu,y_neu, color = "blue")    # neu

plt.gca().set_xlim([-100,100])
plt.gca().set_ylim([-100,100])
```

```
(-100.0, 100.0)
```



12.2.5 Übung 7.5: Arbeitsschritte in eine *Function* verwandeln

Nun haben wir das zufällige Verschieben eines Einzelpunktes am Beispiel einer Dummykoordinaten (`0/0`) durchgespielt. In der nächsten Aufgabe (*Aufgabe 8: Funktionen in DataFrames*) werden wir *alle* unsere Zeckenstichkoordinaten auf diese Weise zufällig verschieben um einen Simulierten Zeckenstichdatensatz ähnlich wie Abb. 12.1 zu erhalten.

Dafür brauchen wir die eben erarbeiteten Einzelschritte als Funktion, um diese auf alle Zeckenstiche anwenden zu können. **Erstelle jetzt eine Funktion namens `offset_coordinate` welche als Input eine `x` oder `y`-Achsenwert annimmt und eine leicht verschobene Wert zurück gibt.** Integriere die Standartabweichung der Verteilung als optionalen Parameter mit einem Standartwert von 100.

```
# Musterlösung

def offset_coordinate(old, distance = 100):
    new = old + random.normalvariate(0,distance)

    return(new)


offset_coordinate(x_start, 100)
```

```
42.93099869002511
```

12.2.6 Übung 7.6: Output visualisieren

Nun ist es wichtig, dass wir unser Resultat visuell überprüfen. Im Beispiel unten wende ich die in der letzten Übung erstellte Funktion `offset_coordinate()` 1'000x auf die Dummykoordinate an. Nutze *deine* Funktion `offset_coordinate` um eine Visualisierung gemäss unten stehendem beispiel zu machen.

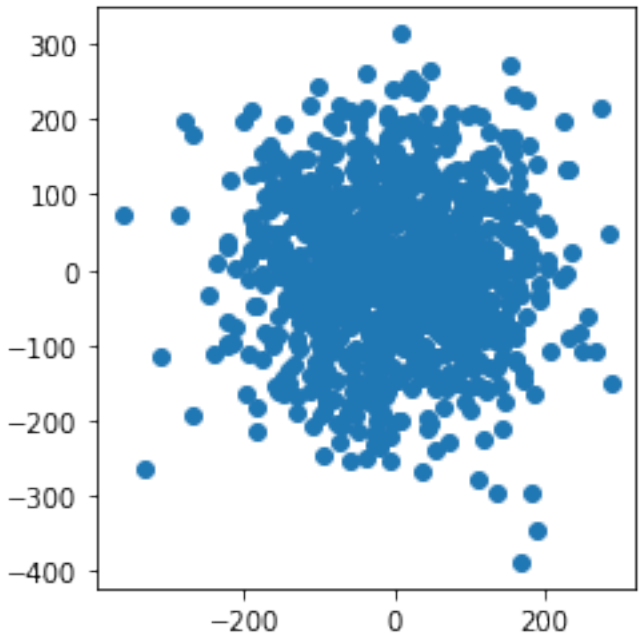
```
from matplotlib import pyplot as plt
import pandas as pd

x_neu_list = [offset_coordinate(x_start, 100) for i in range(1,1000)]
y_neu_list = [offset_coordinate(y_start, 100) for i in range(1,1000)]

fig = plt.scatter(x_neu_list,y_neu_list)

plt.axis("scaled")
```

```
(-394.6142029599762, 322.2338142629571, -422.78943256027486, 348.9299646737648)
```



KAPITEL 13

Aufgabe 8: Funktionen in *DataFrames*

13.1 Theorie

In dieser Aufgabe haben wir das Ziel, die in der letzten Aufgabe (*Aufgabe 7: Zufallszahlen generieren*) erstellte Funktion `offset_coordinate()` auf alle Zeckenstich-Koordinaten anwenden. Bildlich gesprochen: Wir nehmen unsere Zeckenstichdatensatz und schütteln ihn **einmal** durch. So erhalten wir einen Datensatz ähnlich wie in Abb. 12.1 mit dem Unterschied, dass jede Zeckenstichmeldung nicht eine *Wolke* von simulierten Punkten enthält, sondern nur einen einzelnen Punkt.

Nutze hier die Datei „zeckenstiche.csv“ von letzter Woche (du kannst auch sie hier erneut runterladen, Rechtsklick –> Ziel speichern unter). Erstelle ein neues Notebook und nutze nachstehenden Code um die nötigen Module und Functions zu haben:

```
import pandas as pd

def offset_coordinate(old, distance = 100):
    import random
    new = old + random.normalvariate(0,distance)

    return(new)

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

zeckenstiche
```

```
      ID    accuracy        x        y
0   2550  439.128951  2681116  1250648
1  10437  301.748542  2681092  1250672
2   9174  301.748542  2681128  1250683
3   8773  301.748542  2681111  1250683
4   2764  301.748529  2681131  1250692
5   2513  301.748529  2681171  1250711
6   9185  301.748542  2681107  1250712
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
7  28521  301.748542  2681124  1250720
8  26745  301.748542  2681117  1250725
9  27391  301.748542  2681138  1250725
```

13.2 Übungen

13.2.1 Übung 8.1: Spalten selektieren

Mache dich nochmals damit vertraut, einzelne Spalten zu selektieren. Schau dir *Aufgabe 4: Tabellarische Daten* nochmals an wenn du nicht mehr weisst wie das geht.

```
# Musterlösung

zeckenstiche["x"]
zeckenstiche["y"]
```

```
0    1250648
1    1250672
2    1250683
3    1250683
4    1250692
5    1250711
6    1250712
7    1250720
8    1250725
9    1250725
Name: y, dtype: int64
```

13.2.2 Übung 8.2: Neue Spalten erstellen

Mache dich nochmals damit vertraut, wie man neue Spalten erstellt. Schau dir *Aufgabe 4: Tabellarische Daten* nochmals an wenn du nicht mehr weisst wie das geht. Erstelle ein paar neue Spalten nach dem Beispiel unten um die Hangriffe zu üben. Lösche die Spalten im Anschluss wieder mit `del zeckenstiche['test1']` etc.

```
# Musterlösung

zeckenstiche["test1"] = "test1"

zeckenstiche["test2"] = 10

zeckenstiche["test3"] = [1,2,3,4,5,6,7,8,9,10]
```

```
zeckenstiche
```

```
      ID    accuracy        x        y  test1  test2  test3
0   2550  439.128951  2681116  1250648  test1     10      1
1  10437  301.748542  2681092  1250672  test1     10      2
2   9174  301.748542  2681128  1250683  test1     10      3
3   8773  301.748542  2681111  1250683  test1     10      4
4   2764  301.748529  2681131  1250692  test1     10      5
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
5   2513  301.748529  2681171  1250711  test1     10      6
6   9185  301.748542  2681107  1250712  test1     10      7
7  28521  301.748542  2681124  1250720  test1     10      8
8  26745  301.748542  2681117  1250725  test1     10      9
9  27391  301.748542  2681138  1250725  test1     10     10
```

```
# Musterlösung

del zeckenstiche['test1']
del zeckenstiche['test2']
del zeckenstiche['test3']
```

13.2.3 Übung 8.3: `apply`

`pandas` kennt eine ganze Familie von Methoden, um Spalten zu Manipulieren und Daten zu Aggregiren (`apply`, `map`, `mapapply`, `assign`). Es würde den Rahmen von diesem Kurs sprengen, die alle im Detail durch zu gehen, es lohnt sich aber sehr sich mit diesen zu befassen wenn man in sich näher mit Python befassen möchte.

Im unseren Fall brauchen wir lediglich die Methode `apply` um die Funktion `offset_coordinate()` auf die Zeckenstichkoordinaten anzuwenden. Dabei gehen wir wie folgt for:

```
zeckenstiche["x"].apply(offset_coordinate)
#\_______1_______/ \_2_/\_______3_________/

# 1. Spalte selektieren (["x"])
# 2. Methode "apply" aufrufen
# 3. Function übergeben
```

```
0    2.681103e+06
1    2.681260e+06
2    2.680922e+06
3    2.681004e+06
4    2.681170e+06
5    2.681156e+06
6    2.681099e+06
7    2.681075e+06
8    2.681302e+06
9    2.681219e+06
Name: x, dtype: float64
```

Verwende dieses Schema um auch `offset_coordinate` auf die `y` Spalte anzuwenden und speichere den Output dieser beiden Operationen als neue Spalten `x_sim` sowie `y_sim`. Die *DataFrame* `zeckenstiche` sollte danach wie folgt aussehen:

```
# Musterlösung

zeckenstiche["x_sim"] = zeckenstiche["x"].apply(offset_coordinate)
zeckenstiche["y_sim"] = zeckenstiche["y"].apply(offset_coordinate)
```

```
zeckenstiche
```

```
     ID    accuracy        x        y         x_sim         y_sim
0  2550  439.128951  2681116  1250648  2.681176e+06  1.250578e+06
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
1  10437  301.748542  2681092  1250672  2.681181e+06  1.250765e+06
2   9174  301.748542  2681128  1250683  2.681097e+06  1.250550e+06
3   8773  301.748542  2681111  1250683  2.681153e+06  1.250734e+06
4   2764  301.748529  2681131  1250692  2.681139e+06  1.250829e+06
5   2513  301.748529  2681171  1250711  2.680992e+06  1.250706e+06
6   9185  301.748542  2681107  1250712  2.680891e+06  1.250516e+06
7  28521  301.748542  2681124  1250720  2.681070e+06  1.250717e+06
8  26745  301.748542  2681117  1250725  2.680957e+06  1.250664e+06
9  27391  301.748542  2681138  1250725  2.681155e+06  1.250616e+06
```

13.2.4 Übung 8.4: Zusätzliche Parameter

In *Übung 8.3: apply* haben wir unsere Funktion `offset_coordinate` aufgerufen, ohne den Parameter `distance` zu spezifizieren. Dies war möglich, weil wir für `distance` einen Defaultwert festgelegt hat (100 Meter). Wir können aber auch zusätzliche Parameter kommagetrennt nach der Funktion angeben. Dies sieht folgendermassen aus:

```
zeckenstiche["x"].apply(offset_coordinate,distance = 200)
```

```
0    2.680879e+06
1    2.680790e+06
2    2.681067e+06
3    2.680968e+06
4    2.681098e+06
5    2.681232e+06
6    2.681181e+06
7    2.681110e+06
8    2.681393e+06
9    2.681152e+06
Name: x, dtype: float64
```

Nuzte diese Möglichkeit, um die den Offset (`distance`) auf lediglich etwa 10 Meter zu reduzieren.

```
# Musterlösung

zeckenstiche["x_sim"] = zeckenstiche["x"].apply(offset_coordinate, distance = 10)
zeckenstiche["y_sim"] = zeckenstiche["y"].apply(offset_coordinate, distance = 10)
```

13.2.5 Übung 8.5: Simulation visualisieren

Um die Original `x`/`y`-Werte sowie die Simulierten Daten 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.

Bei den roten Punkten handelt es sich um die Original-Zeckenstichen, bei den blauen um die simulierten (leicht verschoben) Zeckenstiche. Visualisiere deine eigenen Zeckenstiche auf diese Weise.

```
from matplotlib import pyplot as plt
```

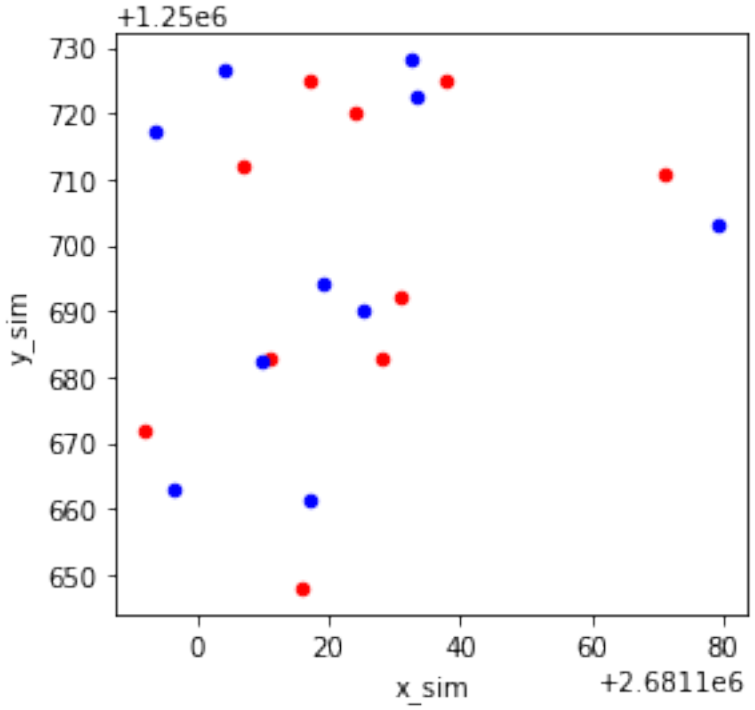
```
basemap = zeckenstiche.plot.scatter("x", "y", color = "red")
zeckenstiche.plot.scatter("x_sim", "y_sim", ax = basemap, color = "blue")
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
plt.axis("scaled")

plt.show()
```



13.2.6 Übung 8.6: Genauigkeitsangaben der Punkte mitberücksichtigen.

In *Übung 8.4: Zusätzliche Parameter* haben wir alle Punkte um etwa die gleiche Distaz (+/- 10m) verschoben. Wenn wir unsere *DataFrame* „zeckensiche“ genau anschauen, steht uns eine Genauigkeitsangabe pro Punkt zur Verfügung: Die Spalte `accuracy`. Diese Spalte ist eine Genauigkeitsangabe über den gemeldeten Zeckenstich. Sie sagt etwas darüber aus, wie sicher der/die Nutzer*in bei der Standortsangabe war (z.B. „*Diese Meldung ist etwa auf 300 Meter genau*“). Wir können diese Genauigkeitsangabe auch nutzen um den offset *pro Punkt* zu bestimmen.

Nutze die Spalte `accuracy` als Argument des Parameters `distance` in der Funktion `offset_coordinate` um genau dies zu erreichen. Visualisiere nun die Daten. Was ist hier passiert?

```
zeckenstiche
```

	ID	accuracy	x	y	x_sim	y_sim
0	2550	439.128951	2681116	1250648	2.681117e+06	1.250661e+06
1	10437	301.748542	2681092	1250672	2.681096e+06	1.250663e+06
2	9174	301.748542	2681128	1250683	2.681110e+06	1.250682e+06
3	8773	301.748542	2681111	1250683	2.681125e+06	1.250690e+06
4	2764	301.748529	2681131	1250692	2.681119e+06	1.250694e+06
5	2513	301.748529	2681171	1250711	2.681179e+06	1.250703e+06
6	9185	301.748542	2681107	1250712	2.681094e+06	1.250717e+06
7	28521	301.748542	2681124	1250720	2.681104e+06	1.250726e+06
8	26745	301.748542	2681117	1250725	2.681133e+06	1.250728e+06
9	27391	301.748542	2681138	1250725	2.681134e+06	1.250722e+06

```
# Musterlösung

zeckenstiche["x_sim"] = zeckenstiche["x"].apply(offset_coordinate, distance =␣
↪zeckenstiche["accuracy"])
zeckenstiche["y_sim"] = zeckenstiche["y"].apply(offset_coordinate, distance =␣
↪zeckenstiche["accuracy"])

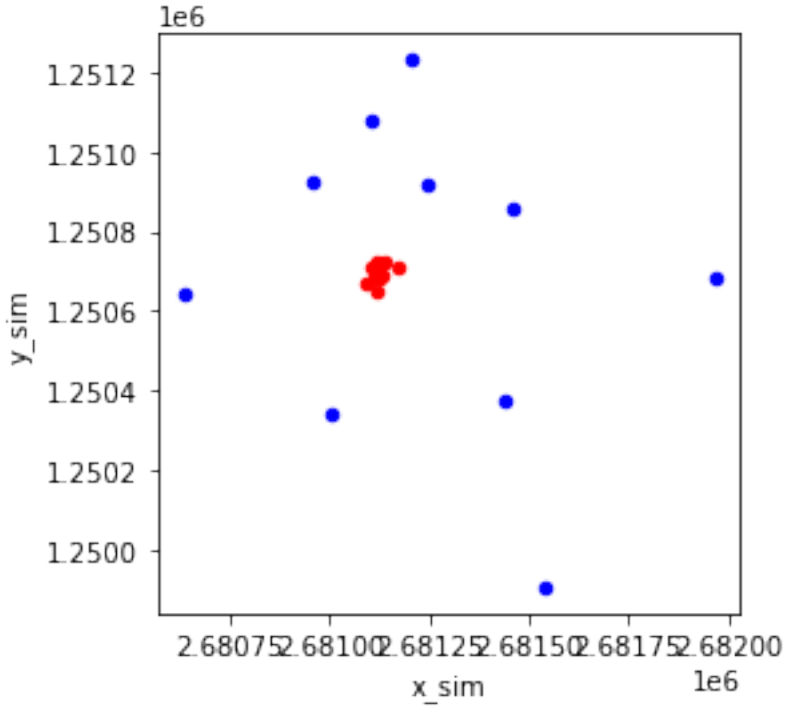
zeckenstiche
```

	ID	accuracy	x	y	x_sim	y_sim
0	2550	439.128951	2681116	1250648	2.681967e+06	1.250683e+06
1	10437	301.748542	2681092	1250672	2.681103e+06	1.251077e+06
2	9174	301.748542	2681128	1250683	2.681440e+06	1.250376e+06
3	8773	301.748542	2681111	1250683	2.681004e+06	1.250341e+06
4	2764	301.748529	2681131	1250692	2.681461e+06	1.250858e+06
5	2513	301.748529	2681171	1250711	2.680960e+06	1.250923e+06
6	9185	301.748542	2681107	1250712	2.680637e+06	1.250646e+06
7	28521	301.748542	2681124	1250720	2.681248e+06	1.250918e+06
8	26745	301.748542	2681117	1250725	2.681541e+06	1.249906e+06
9	27391	301.748542	2681138	1250725	2.681204e+06	1.251231e+06

```
# Musterlösung
basemap = zeckenstiche.plot.scatter("x", "y", color = "red")
zeckenstiche.plot.scatter("x_sim", "y_sim", ax = basemap, color = "blue")

plt.axis("scaled")

plt.show()
```



KAPITEL 14

Einleitung zu diesem Block

Heute 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 (✓ haben wir gelöst in *Übung 7.5: Arbeitsschritte in eine Function verwandeln*)
- Schritt 2: Alle Punkte einer DataFrame zufällig verschieben (1 „Run“, ✓ haben wir gelöst in *Übung 8.3: apply*)
- Schritt 3: Alle Punkte einer DataFrame mehrfach zufällig verschieben (z.B. 50 „Runs“, werden wir heute IN PLATZHALTER lösen)
- Schritt 4: Anteil der Punkte im Wald pro „Run“ ermitteln (werden wir heute in PLATZHALTER lösen)

Übungsziele

- Ihr kennt For-Loops und könnt sie anwenden
- Ihr verwendet eure erste räumliche Operation «Spatial Join» und wisst, dass es hier eine ganze Palette an weiteren Operatoren gibt
- Ihr könnt eine (Geo-) DataFrame nach Gruppe Zusammenfassen
- Ihr lernt weitere Visualisierungstechniken kennen

KAPITEL 15

Aufgabe 9: *For Loop* Einführung

15.1 Theorie

15.1.1 Die Grundform

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. Auf den ersten Blick sieht eine For Loop aus wie eine *Function* definition (siehe *Aufgabe 5: Function Basics* und *Aufgabe 6: Function Advanced*). Im folgenden Beispiel seht ihr ein minimales Beispiel einer *For Loop*.

```
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 ihr beliebig benennen könnt. Im obigen Beispiel lautet diese `platzhalter`
- Nach dem Platzhalter kommt der Begriff `in`. Dieser Begriff kommt zwingend nach dem Platzhalter.
- Nach `in` wird der „Iterator“ festgelegt, also worüber der For-Loop iterieren soll (hier: über eine `List` mit den Werten `[0,1,2]`).
- Danach kommt ein Doppelpunkt `:` der Zeigt: „Nun legen wir gleich fest was im For-Loop passieren soll“ (ähnlich wie in einer *Function*)
- Auf einer neuen Zeile wird eingerückt festgelegt, was in der *For-Loop* passieren soll. In unserem Fall wird etwas Nonsens in die Konsole ausgespukt

- Achtung: `return()` gibt's in For-Loops nicht. Ich nutze hier deshalb *print*[1], damit in der Konsole etwas erscheint.

15.1.2 Der Iterator

Im obigen Beispiel haben wir über eine *List* iterierte, wir haben also eine Liste als Iterator verwendet. Es gibt aber noch andere „Dinge", über die wir iterieren können. Angenommen wir wollen das gleiche machen wir oben, aber nicht mit den Zahlen 0, 1 und 2 sondern von 0 bis 100 oder 100 bis 1'000. Es wäre ganz schön mühsam, alle Zahlen von 0 bis 100 manuell in einer Liste zu erfassen. Zu diesem Zwecken können wir die Funktion `range` verwenden. Mit `range(3)` erstellen wir einen Iterator mit den werten 0, 1 und 2. Mit `range(100,1001)` erhalten wir die Werte von 100 bis 1'000. Um den gleichen Loop wie oben mit `range` zu erstellen ersetzen wir einfach `[0,1,2]` mit `range(3)`:

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

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

15.1.3 Der Platzhalter

Die Platzhaltervariabel liegt immer zwischen `for` und `in`, der Name dieser Variabel könnt ihr frei wählen. Ich habe sie im obigen Beispiel `platzhalter` genannt. Speziell an dieser Variabel ist, dass sie während der Dauer des *Loops* ihren Wert verändert. Mehr dazu in *Aufgabe 10: For Loop Basics*.

15.2 Übungen

15.2.1 Übung 9.1

Kopiere den ersten der beiden *Loops* und lasse ihn bei dir laufen. Spiele mit den Werten rum um ein Gefühl für *For Loops* zu bekommen: Ergänze die Liste mit weiteren Zahlen, verändere den Namen der Platzhaltervariabel und verändere den Text, der in `print` herausgegeben wird.

15.2.2 Übung 9.2: Erste For-Loop erstellen

Konstruieren eine Liste bestehend aus 3 Namen und nenne diese Liste `namen`. Erstelle danach einen *For Loop*, wo jede Person in der Liste gegrüsst wird. Nutze dafür `print`.

```
# Der Output könnte etwa so aussehen (ich nutze zur Begrüssung "Ciao")

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

[1] Mit `print` können wir Variabeln in die Konsole „ausdrucken" lassen. Innerhalb von `print` können dazu verschiedene Variablen kommagetrennt aufgeführt werden, ohne sie mit + verbinden zu müssen wir damals in *Aufgabe 5: Function Basics*.

15.2.3 Übung 9.3: For-Loop mit `range()`

Kopiere den zweiten *For Loop* (der mit `range`) und spiele hier mit den Werten herum. Verändere den *For Loop* so, dass er die über die Werte von 0 - 10 iteriert und von -5 bis +5.

15.2.4 Übung 9.4: *For Loop* mit Quadieren

Bis jetzt haben unsere *Loops* nicht viel Arbeiten müssen. Erstelle nun einen *For Loop*, welcher für die Werte -5 bis +5 folendes ausgibt:

```
Das Quadrat von -5 ist 25
Das Quadrat von -4 ist 16
...
```

15.2.5 Übung 9.5: *For Loop* ohne Platzhaltervariabel

Bisher haben wir die Platzhaltervariabel immer in unserem *Loop* wiederverwendet. Das müssen wir aber gar nicht, wir können den *For Loop* einfach nutzen um etwas x mal zu wiederholen. Erstellen einen *For Loop* der folgende Sätze 5x wiederholt ausgibt:

```
Who likes to party?
We like to party!
Who likes to party?
....
```

Tipp: Nutze dafür zwei verschiedene `print` Befehle auf zwei Zeilen.

KAPITEL 16

Aufgabe 10: *For Loop* Basics

16.1 Theorie

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. Aber `return()` gibt es wie bereits erwähnt bei *For-Loops* nicht. Nehmen wir folgendes Beispiel[1]:

```
for rolle in ["bitch","lover","child","mother","sinner","saint"]:
    liedzeile = "I'm a "+ rolle
    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` aber jetzt anschauen ist dort nur das Resultat aus der letzten Durchführung gespeichert. Das gleiche gilt auch für die Variabel `rolle`.

```
liedzeile
```

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

```
rolle
```

```
'saint'
```

[1] Übrigens: Auch bei Loops haben wir nach der Erstellung (nach `for`) viel Platz um unseren Loop zu konstruieren, bisher war das einfach nicht nötig.

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 hinzuzufügen.

```
refrain = []

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

refrain
```

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

16.2 Übungen

16.2.1 Übung 10.1: Grüsse speichern

Nehmen wir nochmals das Beispiel aus *Übung 9.2: Erste For-Loop erstellen*. Erstelle nochmal ein Loop, wo drei Personen aus einer Liste gegrüsst werden. Diesmal sollen aber die drei Grüsse in einer Liste (z.B. `mylist`) gespeichert werden.

```
# Das Resultat sieht dann so aus:
mylist
```

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

16.2.2 Übung 10.2: Ganze Strophe speichern

Der im Beispiel verwendete Refrain aus dem Lied „Bitch“ von Meredith Brooks besteht bis auf zwei Zeilen aus Wiederholungen. Versuche mit zwei verschiedenen *For Loops* den ganzen Refrain in einer Liste zu speichern. Die beiden Teile die vom Muster Abweichen („*I do not feel ashamed*“ und „*You know you wouldn't want it any other way*“) kannst du auch ausserhalb der Loops in die Listen einfügen (`append`).

```
# Das Resultat sieht dann so aus:
refrain
```

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

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
"I'm a saint",
'I do not feel ashamed',
"I'm your hell",
"I'm your dream",
"I'm nothing in between",
"You know you wouldn't want it any other way"]
```

KAPITEL 17

Aufgabe 11: *For Loops* Advanced

17.1 Theorie

In diesem Kapitel kommen noch zwei Aspekte von *For Loops*, die als „*Advanced*“ eingestuft werden können aber in der Praxis sehr nützlich sind. Dabei geht es um einerseits um verschachtelte *For Loops* und zum andere um eine verkürzte Schreibweise von *For Loops*.

17.1.1 Verschachtelte *For Loops*

Wir können verschiedene *For Loops* auch ineinander verschachteln (englisch: *nested loops*). Das ist vor allem dann nützlich, wenn alle Kombinationen aus zwei Datensätzen miteinander verrechnet werden müssen. Angenommen du willst die drei Mitglieder deiner Band (bestehend aus *Il Buono*, *Il Brutto*, *Il Cattivo*) deinen Eltern vorstellen und auch umgekehrt deine Eltern deiner Band vorstellen. Für so was eignen sich zwei verschachtelte vor Loops hervorragend:

```
eltern = ["Papa", "Mama"]
band = ["Il Buono", "Il Brutto", "Il Cattivo"]

for bandmitglied in band:
    for elternteil in eltern:
        print(elternteil, "das ist",bandmitglied)
        print(bandmitglied, "das ist",elternteil)
        print("---")
```

```
Papa das ist Il Buono
Il Buono das ist Papa
---
Mama das ist Il Buono
Il Buono das ist Mama
---
Papa das ist Il Brutto
Il Brutto das ist Papa
---
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
Mama das ist Il Brutto
Il Brutto das ist Mama
---
Papa das ist Il Cattivo
Il Cattivo das ist Papa
---
Mama das ist Il Cattivo
Il Cattivo das ist Mama
---
```

17.1.2 Verkürzte Schreibweise

Es ist äusserst häufig der Fall, dass wir den Output aus einem Loop in einer Liste abspeichern wollen. Wie das geht haben wir ja bereits in *Aufgabe 10: For Loop Basics* gelernt:

```
rollen = ["bitch","lover","child","mother","sinner","saint"]

refrain = []
for rolle in rollen:
    liedzeile = "I'm a "+ rolle
    refrain.append(liedzeile)
```

Nur ist das ein *bisschen* umständlich, weil wir dafür viele Zeilen Code brauchen um etwas eigentlich ganz simples zu bewerkstelligen. Es gibt deshalb dafür auch eine verkürzte Schreibweise, welche ich in der letzten Woche bereits einmal verwendet habe (siehe *Übung 7.6: Output visualisieren*). Der obige Loop hat in der verkürzten Schreibweise die folgende Form:

```
refrain = ["I'm a "+ rolle for rolle in rollen]
```

Diese verkürzte Schreibweise heisst in Python *list comprehension* und sie ist äusserst praktisch, wenn man sie beherrscht. Das Beherrschen ist aber nicht zentral, es reicht schon wenn ihr eine solche Schreibweise wieder erkennt und richtig interpretieren könnt (im Sinne von „*Aha, hier wird also in einem Loop eine Liste erstellt*“). In der folgenden Darstellung seht ihr farblich, welche Elemente sich in der verkürzten Schreibweise wo wiederfinden und welche Elemente gar nicht wiederverwendet werden.

```
rollen = ["bitch","lover","child","mother","sinner","saint"]

refrain = []
for rolle in rollen:
    liedzeile = "I'm a "+ rolle
    refrain.append(liedzeile)
```

```
refrain = ["I'm a "+ rolle for rolle in rollen]
```

17.2 Übungen

17.2.1 Übung 11.1: Nested Loop: Bellen und Fauchen

Erstelle zwei Listen bestehend aus 3 Hundenamen (`hunde`) und 3 Katzenamen (`katzen`). Erstelle einen verschachtelter *For Loop*, wo jeder Hund jede Katze anbellt und jede Katze jeden Hund anfaucht.

```
Bruno bellt Greta an
Greta faucht Bruno an
Berta bellt Greta an
....
```

17.2.2 Übung 11.2: Nested Loop: Multiplikation

Erstelle einen verschachtelten Loop, wo alle Kombinationen von 0 bis 9 miteinander addiert werden.

KAPITEL 18

Aufgabe 12: GIS in Python

18.1 Theorie

18.1.1 Tabellen *ohne* Raumbezug

Unsere Zeckenstiche haben wir bisher immer auf nachstehende Weise importiert. Jeder Zeckenstich hat offensichtlich eine genaue Koordiate, aber ein *richtiger* Geodatensatz kann man diese DataFrame noch nicht nennen. *Wir* wissen ja, dass mit den Spalten `x` und `y` Koordinaten in der Schweiz gemeint sind, Python hingegen weiss das (noch) nicht. Der Raumbezug fehlt noch, und das wird irgendwann problematisch: Denn wir wollen jetzt pro Zeckenstich ermitteln, ob er sich im Wald befindet oder nicht. Das ist eine explizit räumliche Abfrage, was nur mit explizit räumlichen Geodaten beantwortet werden kann.

```
import pandas as pd
zeckenstiche = pd.read_csv("zeckenstiche.csv")
zeckenstiche
```

```
      ID    accuracy        x        y
0   2550  439.128951  2681116  1250648
1  10437  301.748542  2681092  1250672
2   9174  301.748542  2681128  1250683
3   8773  301.748542  2681111  1250683
4   2764  301.748529  2681131  1250692
5   2513  301.748529  2681171  1250711
6   9185  301.748542  2681107  1250712
7  28521  301.748542  2681124  1250720
8  26745  301.748542  2681117  1250725
9  27391  301.748542  2681138  1250725
```

```
type(zeckenstiche)
```

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

18.1.2 Tabellen *mit* Raumbezug

Glücklicherweise können wir unsere *Zeckenstich-Dataframe* mit nur einem Zusatzmodul und wenigen Zeilen code in eine räumliche *DataFrame* konvertieren. Aktuell handelt es sich um das Modul ***geo**pandas*, was aus unserer *DataFrame* eine ***Geo**DataFrame* macht. Mit diesem Modul erhält die *DataFrame* im Wesentlichen

- eine Zusatzspalte `geometry` mit der Geometrie als räumliches Objekt
- ein Attribut „crs“ welches das Koordinatensystem der Geometriespalte enthält.

Beides müssen wir bei der Erstellung der *GeoDataFrame* aber festlegen. Die `geometry` Spalte ist relativ offensichtlich, es handelt sich dabei um die `x` und `y` Spalten, welche die Kordianten enthalten. Was aber ist das „Koordinatensystem“ unserer Daten? Das ist Information, welche euch vom Datenersteller geliefert werden muss. Man man das Koordinatensytem aber auch anhand der Koordinaten erahnen. Es handelt sich dabei um Werte im Bereich von 2’600’000 (x) und 1’200’000 (y), und da wir wissen das die Daten aus der Schweiz stammen ist es naheliegend, dass es sich um das neue Schweizer Koordinatensystem CH1903+ / LV95 handelt mit dem EPSG Code 2056. Diesen Code können wir der Funktion `gpd.GeoDataFrame` als Argument (`crs =` , **C**oordinate **R**eference **S**ystem) übergeben, um das Koordinatensystem festzulegen.

```
import geopandas as gpd

zeckenstiche_gpd = gpd.GeoDataFrame(zeckenstiche, geometry=gpd.points_from_
↪xy(zeckenstiche['x'], zeckenstiche['y'], crs = 2056))
zeckenstiche_gpd
```

	ID	accuracy	x	y	geometry
0	2550	439.128951	2681116	1250648	POINT (2681116.000 1250648.000)
1	10437	301.748542	2681092	1250672	POINT (2681092.000 1250672.000)
2	9174	301.748542	2681128	1250683	POINT (2681128.000 1250683.000)
3	8773	301.748542	2681111	1250683	POINT (2681111.000 1250683.000)
4	2764	301.748529	2681131	1250692	POINT (2681131.000 1250692.000)
5	2513	301.748529	2681171	1250711	POINT (2681171.000 1250711.000)
6	9185	301.748542	2681107	1250712	POINT (2681107.000 1250712.000)
7	28521	301.748542	2681124	1250720	POINT (2681124.000 1250720.000)
8	26745	301.748542	2681117	1250725	POINT (2681117.000 1250725.000)
9	27391	301.748542	2681138	1250725	POINT (2681138.000 1250725.000)

```
type(zeckenstiche_gpd)
```

```
geopandas.geodataframe.GeoDataFrame
```

```
# Das Attribut `crs` wurde aufgrund vom EPSG Code richtig erkannt:
zeckenstiche_gpd.crs.name
```

```
'CH1903+ / LV95'
```

18.1.3 Räumliche Operationen

Was hat das nun bewirkt, was bringt uns diese *Geo* Erweiterung? Ein kleiner, aber offensichlicher Vorteil ist das veränderte Verhalten von der `.plot()` Methode. Da `zeckenstiche_gpd` ein räumliches Objekt ist, macht `.plot()` automatisch einen räumliche Darstellung der Daten und sorgt auch dafür, dass die beiden Achsen gleich skaliert sind. So ist`fig.axis("equal")` nicht mehr nötig wie es im Scatterplot in der *Übung 4.4: Koordinaten räumlich darstellen* der Fall war.

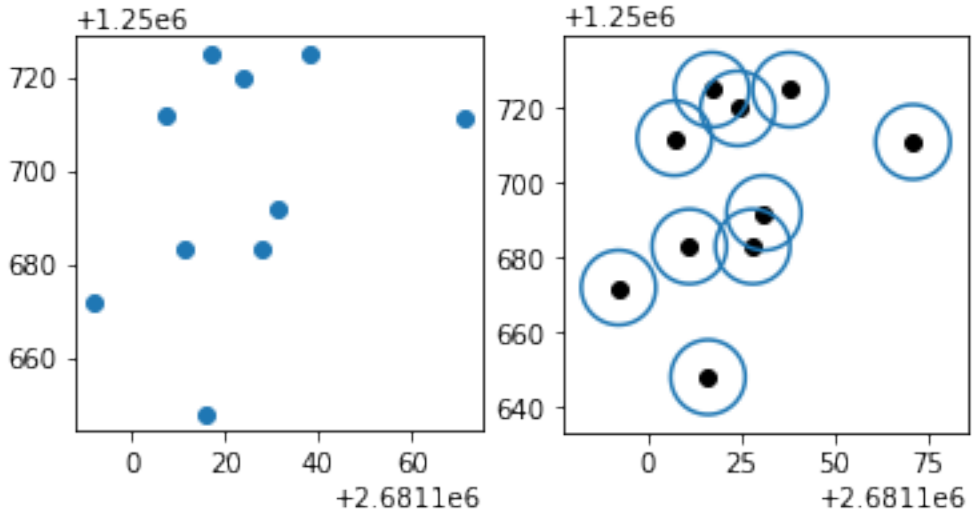
Die Vorteile geht aber noch weiter. Mit *GeoPandas* sind nun andere räumliche Operationen möglich, die wir aus ArcGIS kennen aber mit einfachen *DataFrames* noch nicht möglich waren. Mit `.buffer()` können wir zum Beispiel einen Buffer um unsere Punkte machen.

```
from matplotlib import pyplot as plt    # dieser Teil wird nur benötigt, weil die␣
↪beiden
fig, (ax1, ax2) = plt.subplots(1, 2)    # Plots nebeinenader stehen sollen

zeckenstiche_gpd.plot(ax = ax1)         # "ax = ax1" kann auch weggelassen weren

buffered = zeckenstiche_gpd.buffer(10) # macht ein Buffer mit 10m Distanz
buffered.boundary.plot(ax = ax2)        # mit .boundary erhalte ich die Polygonumrisse
zeckenstiche_gpd.plot(ax = ax2, color = "black")
```

```
<AxesSubplot:>
```



Im Prinzip stehen uns jetzt auch alle anderen Vektor Operationen zur Verfügung, die wir schon aus ArcGIS kennen. Mit `unary_union`[1] können wir aus unseren Einzelpunkten ein *Multipolygon* erstellen und darüber ein *Convex Hull*[2] oder ein *Envelope*[3] rechnen.

```
# Macht ein Union der Zeckenstiche (ein Multipoint-Objekt)
# (dies ist nötig, weil convex_hull / envelope Multipoint Objekte benötigen)
zeckenstiche_union = zeckenstiche_gpd["geometry"].unary_union

# Berechnet den Convex Hull und speichert den Ouput als Polygon
```

(Fortsetzung auf der nächsten Seite)

[1] Dieser Befehl lautet in ArcGIS Union

[2] Convex Hull (Konvexe Hülle) ist eine „Rahmen“ um alle Punkte, wo die Innenwinkel immer kleiner sind als 180°). Dieser Befehl lautet in ArcGIS Minimum Bounding Geometry (Option *Convex Hull*)

[3] Envelope ist ebenfalls ein „Rahmen“ um alle Punkte, was aber quadratisch und am Koordinatensystem orientiert ist. Auch diese Operation lautet in ArcGIS unter dem Tool Minimum Bounding Geometry (Option *Envelope*)

(Fortsetzung der vorherigen Seite)

```
my_convex_hull = zeckenstiche_union.convex_hull

# Konvertiert das Polygon in eine GeoSeries
# (GeoSeries sind praktischer als Polygone)
my_convex_hull = gpd.GeoSeries(my_convex_hull)

# Berechnet das Envelope und speichert den Output als Polygon
my_envelope = zeckenstiche_union.envelope

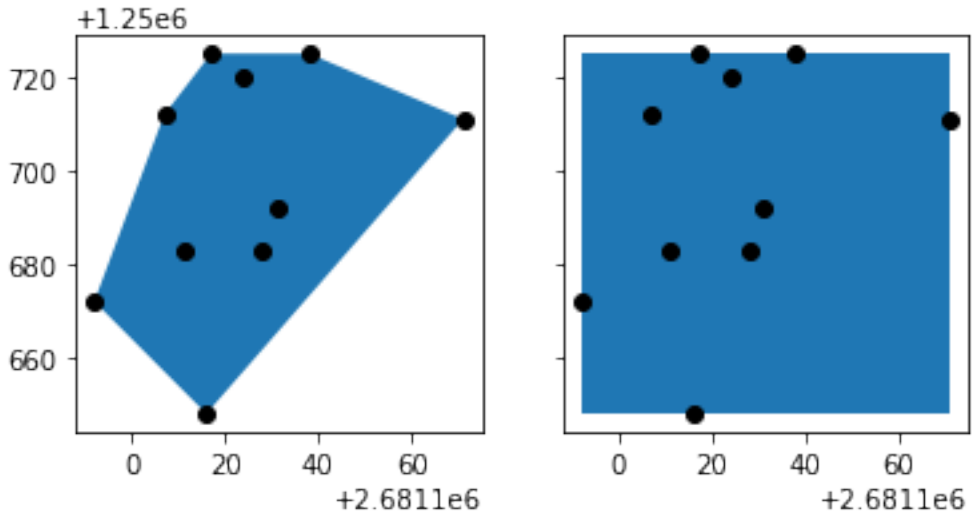
# Konvertiert das Polygon in eine GeoSeries
# (GeoSeries sind praktischer als Polygone)
my_envelope = gpd.GeoSeries(my_envelope)

# Bereitet wieder die beiden Subplots vor
fig, (ax1, ax2) = plt.subplots(1, 2,sharex=True, sharey = True)

# Erstellt den linken Plot
my_convex_hull.plot(ax = ax1)
zeckenstiche_gpd.plot(ax = ax1, color = "black")

# Erstellt den rechten Plot
my_envelope.plot(ax = ax2)
zeckenstiche_gpd.plot(ax = ax2, color = "black")
```

```
<AxesSubplot:>
```



Achtung

In Geopandas gibt es drei Fundamentale Datentypen:

- **Geometrien**: Einzelne Objekte[4] der folgenden Typen
  - Points / Multi-Points
  - Lines / Multi-Lines

[4] Die Geometrien in Geopandas sind eigentlich Objekte vom Modul *Shapely*. Shapely wiederum ist ein Python Modul, welches mit *Geopandas* mit-installiert und mit-importiert wird.

  - Polygons / Multi-Polygons

- **GeoSeries**: Eine Serie von Geometrien, gleichbedeutend wie eine Spalte in einer Tabelle
- **GeoDataFrame**: Eine Tabelle, welche über eine Geometrie-Spalte (GeoSeries) verfügt

Die verschiedenen Operationen in Geopandas erwarten teilweise unterschiedlichen Input, deshalb müssen wir teilweise zwischen *Geometrien*, *Geoseries* und *GeoDataFrames* hin- und her konvertieren. Geopandas ist noch in Arbeit und ich hoffe, dass das Package in der Zukunft noch etwas einfacher in der Handhabung wird.

Aktuell brauchen zum Beispiel `unary_union`, `convex_hull` und `envelope` alle *GeoSeries* als Input. Der Output den drei Operationen eine *Geometrie*. Die Operation `buffer` nimmt sowohl ganze *GeoDataFrames* wie auch *GeoSeries* als Input an. Die Overlay Operationen mit `overlay` funktionieren aktuell nur mit „GeoDataFrames". *Geopandas* ist aber gut dokumentiert, und es lohnt sich bei Unklarheiten immer das Handbuch zu konsultieren: geopandas.org

```
# Nachstehender Code ist nur nötig, weil die Overlay
# Operationen GeoDataFrames als Input benötigen.
buffered_gdf = gpd.GeoDataFrame(geometry = buffered, crs = 2056)
my_convex_hull_gdf = gpd.GeoDataFrame(geometry = my_convex_hull, crs = 2056)

# Bereitet die drei Subplots vor:
fig, (ax1, ax2, ax3) = plt.subplots(1, 3, sharex=True, sharey = True, figsize = (10,
↪8))

################################################
my_convex_hull_gdf.plot(ax = ax1)               #
ax1.set_title("Das Minimum Convex Polygon")     # Plot links
ax1.set_axis_off()                              #
################################################
buffered_gdf.plot(ax = ax2)                     #
ax2.set_title("Die gebufferten Punkte")         # Plot mitte
ax2.set_axis_off()                              #
################################################

# Overlay Operation
my_difference = gpd.overlay(my_convex_hull_gdf,buffered_gdf, how='difference')

################################################
my_difference.plot(ax = ax3)                    #
ax3.set_title("Die Differenz der ersten beiden")# # Plot rechts
ax3.set_axis_off()                              #
################################################
```

18.2 Übungen

18.2.1 Übung 12.1: *DataFrame* zu *GeoDataFrame*

Importiere *Geopandas* und wandle `zeckenstiche` in eine *GeoDataFrame* um (`zeckenstiche_gpd`). Vergiss nicht, das Koordinatensystem festzulegen!

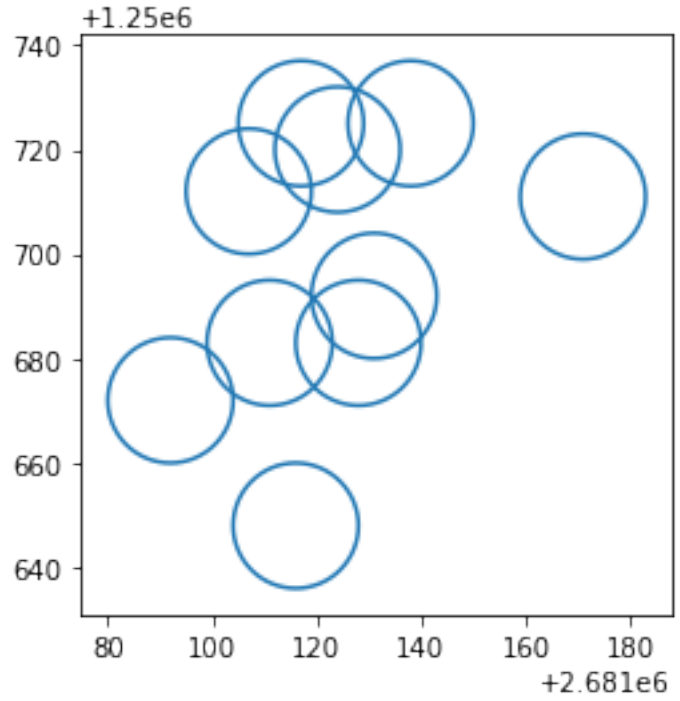
18.2.2 Übung 12.2: Punkte Buffern

Buffere die Zeckenstiche um eine Distanz von 12 Meter und speichere den Output in der Variabel `zeckenstiche_buffer`. Visualisiere die gebufferten Punkte in einem Plot.

18.2.3 Übung 12.3: Umrisse visualisieren

Extrahiere die Umrisse von `zeckenstiche_buffer` und speichere diese in `zeckenstiche_buffer_outline`. Visualisiere anschliessend diese Umrisse.

```
<AxesSubplot:>
```



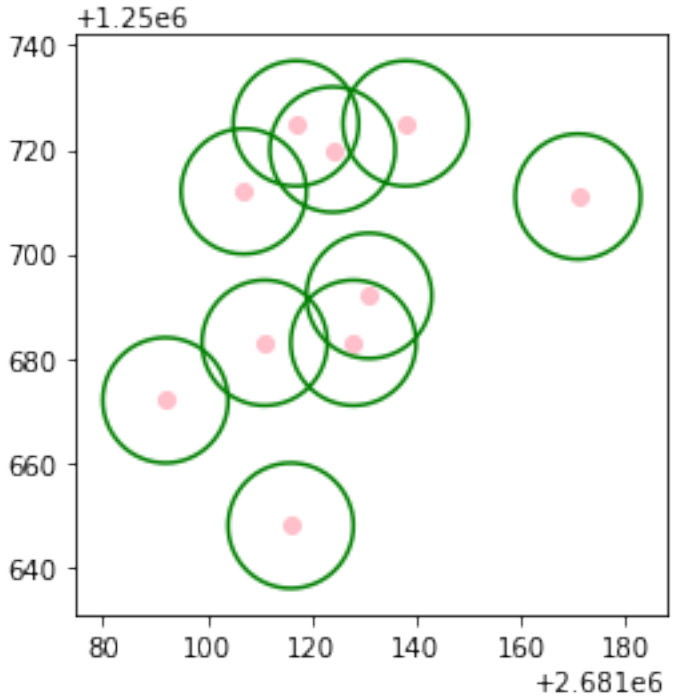
18.2.4 Übung 12.4: Layers überlagern

Nutze nachstehenden Code um zwei Datensätze im gleichen Plot darzustellen.

```
from matplotlib import pyplot as plt
fig, ax = plt.subplots()

zeckenstiche_buffer_outline.plot(ax = ax, color = "green")
zeckenstiche_gpd.plot(ax = ax, color = "pink")
```

```
<AxesSubplot:>
```



18.2.5 Übung 12.5 Envelope

Berechne das „Envelope“ von `zeckenstiche_gpd` anhand der obigen Beispielen. Speichere den Output als `zeckenstiche_envelope`.

Tipp: Denk daran, dass du zuerst noch einen Union machen musst (siehe *Räumliche Operationen*)

18.2.6 Übung 12.6: Overlay Operation

Führe verschiedene Overlay Operationen zwischen `zeckenstiche_envelope` und `zeckenstiche_buffer` durch. Schaue dir dazu die entsprechende Geopandas Hilfeseite an. Beispielsweise würden sich *Union* und *Symetrical Difference* gut anbieten.

Tipp:

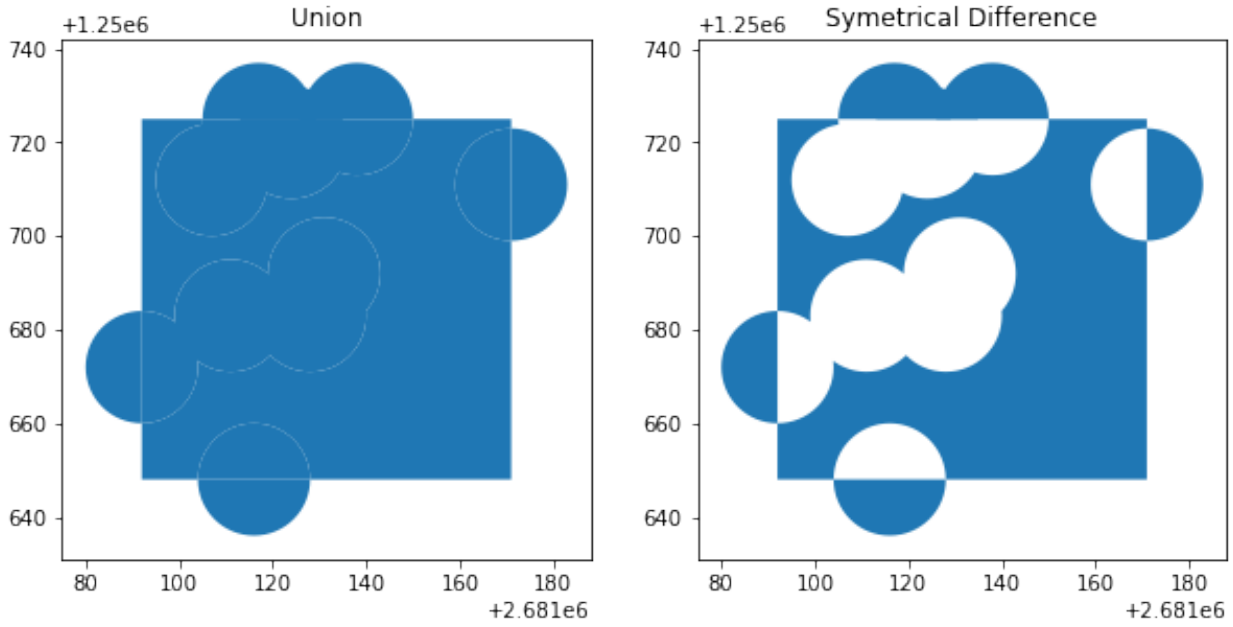
- `zeckenstiche_envelope` musst zu zuerst noch in eine GeoSeries umwandeln. Den Ouput davon kannst du in eine GeoDataFrame konvertieren
- `zeckenstiche_buffer` sollte schon eine GeoSeries sein, diese kannst du direkt in eine GeoDataFrame konvertieren
- beim Konvertieren in eine GeoDataFrame kannst du jeweils direkt das Koordinatensystem (`crs = `) korrekt setzen.

```
# Das Resultat sollte folgendermassen aussehen

fig, (ax1, ax2) = plt.subplots(1,2, figsize = (10,8))

my_union.plot(ax = ax1)
ax1.set_title("Union")
my_symmdiff.plot(ax = ax2)
ax2.set_title("Symetrical Difference")
```

```
Text(0.5, 1.0, 'Symetrical Difference')
```



18.2.7 Übung 12.7: Geopandas als Shapefile / Geopackage abspeichern

Jetzt wo wir unsere Zeckenstiche als Geodaten aufbereitet haben können wir diese auch in gängige Geodaten abspeichern. Genau wie wir in *Übung 4.2: DataFrame in csv umwandeln* eine *DataFrame* mit `.to_csv` in eine csv exportiert haben, gibt es für GeoDataFrames die Methode `.to_file`. Exportiere `zeckenstiche_gpd` mit dieser Methode in eine Shapefile.

Zusatzaufgabe: Shapefiles sind eigentlich ein ganz schreckliches Format (siehe switchfromshapefile.org). Viel praktischer sind an dieser Stelle *Geopackages* (nicht zu verwechseln mit ArcGIS Pro Packages). Ihr könnt `monte_carlo_df` auch als *Geopackage* exportieren, dafür müsst ihr dem Output die Erweiterung „gpkg“ geben und zusätzlich `driver = GPKG` festlegen.

KAPITEL 19

Übung: Zeckenstich Simulation mit Loop

Nun geht es weiter mit unserer Zeckenstich Monte-Carlo Simulation. Schritte 1 und 2 haben wir bereits erleidgt. Nun packen wir Schritt 3 an:

Ladet dafür die nötigen Module (`pandas` und `random`), holt euch die Funktion `offset_point()` und importiert den Datensatz `zeckenstiche.csv`. Tipp: Importiert mit `head(5)` nur die ersten 5 Zeile aus dem csv, das macht die die Entwicklung des Loops leichter.

```
import pandas as pd
import random

def offset_coordinate(old, distance = 100):
    new = old + random.normalvariate(0,distance)
    return(new)

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

19.1 Übung 1: Mit For-Loop `zeckenstiche` mehrfach verschieben

Um euer Gedächnis etwas aufzufrischen: Letzte Woche hatten wir mit `apply()` sowie unserer eigenen *Function* `offset_coordinate` alle Koordinaten einer *DataFrame* verschoben und die neuen Daten als eine neue DataFrame abgespeichert.

```
zeckenstiche_sim = pd.DataFrame()

zeckenstiche_sim["ID"] = zeckenstiche["ID"]

zeckenstiche_sim["x"] = zeckenstiche["x"].apply(offset_coordinate)
zeckenstiche_sim["y"] = zeckenstiche["y"].apply(offset_coordinate)
zeckenstiche_sim
```

```
      ID             x             y
0   2550  2.681095e+06  1.250709e+06
1  10437  2.681090e+06  1.250710e+06
2   9174  2.681140e+06  1.250544e+06
3   8773  2.681019e+06  1.250675e+06
4   2764  2.681160e+06  1.250921e+06
5   2513  2.681187e+06  1.250772e+06
6   9185  2.680996e+06  1.250715e+06
7  28521  2.681196e+06  1.250716e+06
8  26745  2.681015e+06  1.250865e+06
9  27391  2.681308e+06  1.250607e+06
```

Kombiniere dies nun mit deinem Wissen über Loops, um die Punkte der *DataFrame* nicht einmal, sondern 5 mal zu verschieben. Dazu brauchst du 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).

19.2 Übung 2: DataFrames 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`?

```
type(zeckenstiche)
```

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

```
type(monte_carlo)
```

```
list
```

```
len(zeckenstiche)
```

```
10
```

```
len(monte_carlo)
```

```
5
```

Worauf ich hinaus will: `zeckenstiche_sim` ist eine *DataFrame* und `monte_carlo` ist eine Liste von *DataFrames*. 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 *DataFrame* zusammenführen. Führe die Funktion aus und speichere den Output als `monte_carlo_df`.

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

19.3 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.scatter()`.

KAPITEL 20

Übung: 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. In der Todo-Liste sind wir nun bei Schritt 4:

- Schritt 1: Einen Einzelpunkt zufällig verschieben ✓
- Schritt 2: Alle Punkte einer DataFrame zufällig verschieben (1 „Run“) ✓
- Schritt 3: Alle Punkte einer DataFrame mehrfach zufällig verschieben (z.B. 50 „Runs“) ✓
- **Schritt 4: Anteil der Punkte im Wald pro „Run“ ermitteln 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**

Lade dafür den Datensatz „wald.gpkg“ von Moodle herunter und verschiebt es in eure *Working directory*. Importiere „wald.gpkg“ 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.gpkg") # oder .shp, je nach dem wie␣
↪ihr es gespeichert habt

wald = gpd.read_file("wald.gpkg")


wald
```

```
/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
```

```
---------------------------------------------------------------------------
CPLE_OpenFailedError                      Traceback (most recent call last)
fiona/_shim.pyx in fiona._shim.gdal_open_vector()

fiona/_err.pyx in fiona._err.exc_wrap_pointer()

CPLE_OpenFailedError: monte_carlo_df.gpkg: No such file or directory

During handling of the above exception, another exception occurred:

DriverError                               Traceback (most recent call last)
<ipython-input-2-9ff8e131362a> in <module>
      2 import geopandas as gpd
      3
----> 4 monte_carlo_df = gpd.read_file("monte_carlo_df.gpkg") # oder .shp, je nach␣
↪dem wie ihr es gespeichert habt
      5
      6 wald = gpd.read_file("wald.gpkg")

/opt/hostedtoolcache/Python/3.7.9/x64/lib/python3.7/site-packages/geopandas/io/file.
↪py in _read_file(filename, bbox, mask, rows, **kwargs)
     94
     95     with fiona_env():
---> 96         with reader(path_or_bytes, **kwargs) as features:
     97
     98             # In a future Fiona release the crs attribute of features will

/opt/hostedtoolcache/Python/3.7.9/x64/lib/python3.7/site-packages/fiona/env.py in␣
↪wrapper(*args, **kwargs)
    398     def wrapper(*args, **kwargs):
    399         if local._env:
--> 400             return f(*args, **kwargs)
    401         else:
    402             if isinstance(args[0], str):

/opt/hostedtoolcache/Python/3.7.9/x64/lib/python3.7/site-packages/fiona/__init__.py␣
↪in open(fp, mode, driver, schema, crs, encoding, layer, vfs, enabled_drivers, crs_
↪wkt, **kwargs)
    255         if mode in ('a', 'r'):
    256             c = Collection(path, mode, driver=driver, encoding=encoding,
--> 257                            layer=layer, enabled_drivers=enabled_drivers,␣
↪**kwargs)
    258         elif mode == 'w':
    259             if schema:

/opt/hostedtoolcache/Python/3.7.9/x64/lib/python3.7/site-packages/fiona/collection.py␣
↪in __init__(self, path, mode, driver, schema, crs, encoding, layer, vsi, archive,␣
↪enabled_drivers, crs_wkt, ignore_fields, ignore_geometry, **kwargs)
    162             if self.mode == 'r':
    163                 self.session = Session()
--> 164                 self.session.start(self, **kwargs)
    165             elif self.mode in ('a', 'w'):
    166                 self.session = WritingSession()

fiona/ogrext.pyx in fiona.ogrext.Session.start()

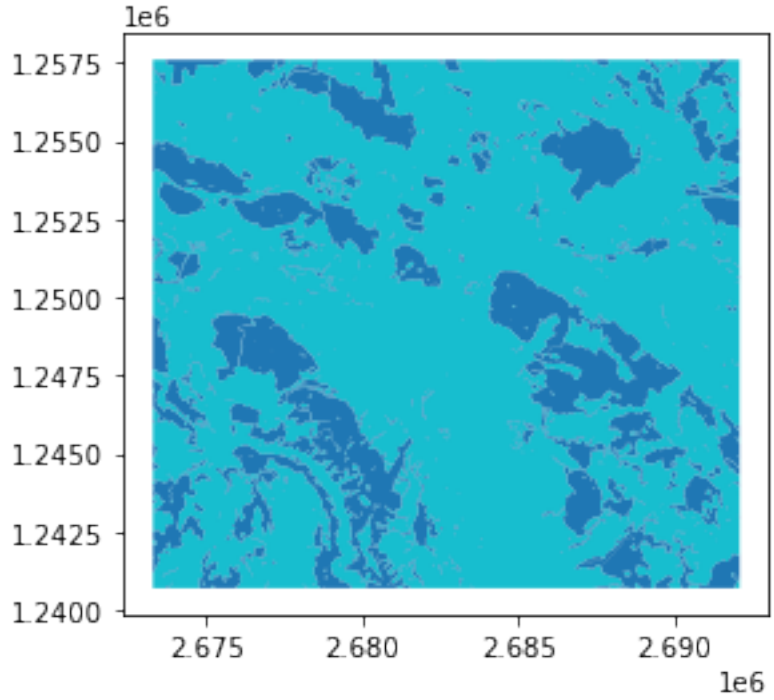
fiona/_shim.pyx in fiona._shim.gdal_open_vector()
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

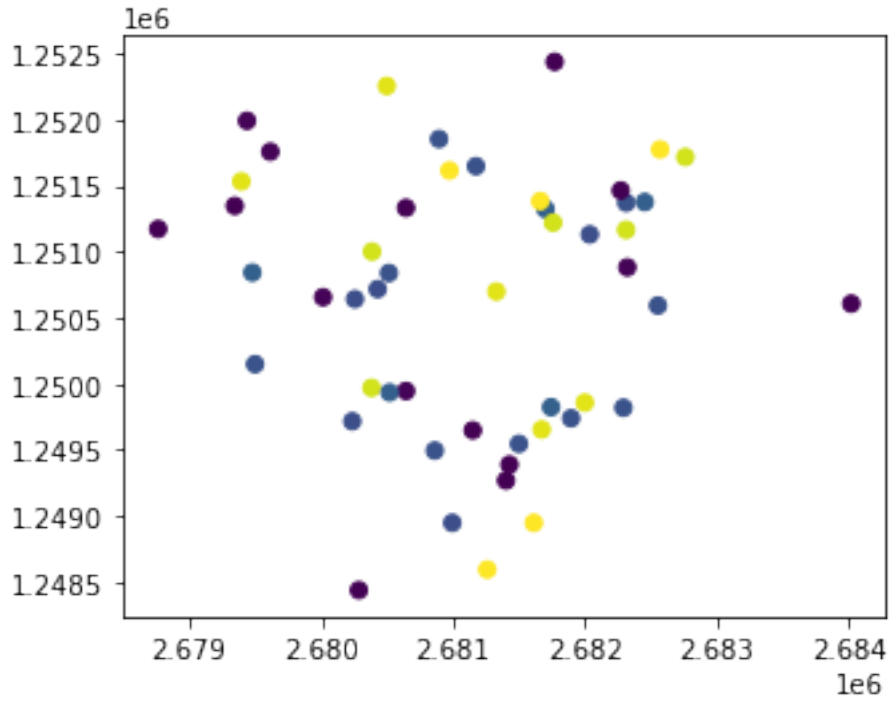
```
DriverError: monte_carlo_df.gpkg: No such file or directory
```

```
# hierfür braucht ihr das modul "descartes"
wald.plot(column = "Wald_text")
```




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

```
<AxesSubplot:>
```



20.1 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_sjoin = gpd.sjoin(monte_carlo_df,wald)

monte_carlo_sjoin.head()
```

```
      ID             x             y  Run_Nr                          geometry  \
0   2550  2.679342e+06  1.251346e+06       0  POINT (2679342.334 1251346.270)
1  10437  2.681741e+06  1.249824e+06       0  POINT (2681740.565 1249824.010)
2   9174  2.681497e+06  1.249546e+06       0  POINT (2681497.002 1249546.406)
3   8773  2.682034e+06  1.251131e+06       0  POINT (2682033.926 1251130.509)
4   2764  2.680012e+06  1.250656e+06       0  POINT (2680011.649 1250655.533)

   index_right  Wald     Shape_Leng    Shape_Area Wald_text
0            0     0  947316.853401  2.380876e+08      nein
1            0     0  947316.853401  2.380876e+08      nein
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
2             0      0  947316.853401  2.380876e+08       nein
3             0      0  947316.853401  2.380876e+08       nein
4             0      0  947316.853401  2.380876e+08       nein
```

20.2 Übung 2: Anteil der Punkte pro „Gruppe“

Jetzt wirds etwas kniffelig. Um die Anzahl Punkte innerhalb / ausserhalb vom Wald zu zählen, brauchen wir die `groupby` und `size` aus der Pandas Bibliothek. Es würde den Rahmen von diesem Kurs sprengen, euch die Einzelschritte im Detail zu erläutern, deshalb gebe ich euch den fertigen Code den ihr auf euren Datensatz anwenden könnt.

```
anteile = monte_carlo_sjoin.groupby(["Run_Nr", "Wald_text"]).size().to_frame("Anzahl")
anteile
```

```
                   Anzahl
Run_Nr Wald_text
0      ja               1
       nein             9
1      nein            10
2      ja               3
       nein             7
3      ja               1
       nein             9
4      ja               2
       nein             8
```

Wir sehen in der obigen Tabelle für jeden Run (Spalte „Run_Nr“) die Anzahl Werte im Wald („ja“) und ausserhalb („nein“). Beachte

- das die Summe aus „ja“ + „nein“ pro Run gleich gross ist, da wir ja immer gleich viele Zeckenstiche pro Run haben.
- das auch alle Zeckenstiche in einer Gruppe landen können (also alle innerhalb oder alle ausserhalb des Waldes)

Im Nächsten Schritt „pivotiere“ ich die Tabelle so, dass „ja“ und „nein“ einzelne Spalten sind.

```
anteile_pivot = anteile.pivot_table(index = "Run_Nr", columns = "Wald_text", values =
↪"Anzahl", fill_value = 0)
# mit fill_value = 0 spezifiziere ich, dass der Wert "0" sein soll wenn
# in einem Run kein Wert in "ja" oder "nein" vorhanden sind
# (sprich: wenn alle Stiche entweder innerhalb oder ausserhalb
# des Waldes gelandet sind)

anteile_pivot
```

```
Wald_text  ja  nein
Run_Nr
0           1     9
1           0    10
2           3     7
3           1     9
4           2     8
```

20.3 Übung 3: Anteil *im Wald* pro Run ermitteln

Berechnet den Anteil im Wald indem du die die Spalte „ja“ mit der Summe aus den Spalten „Ja“ + „Nein“ dividierst. Nutze dafür die Eckigen Klammern (`[/]`) sowie die Splatennamen. Speichere den Output als `anteil_ja`.

```
anteil_ja = anteile_pivot["ja"]/(anteile_pivot["ja"]+anteile_pivot["nein"])

anteil_ja
```

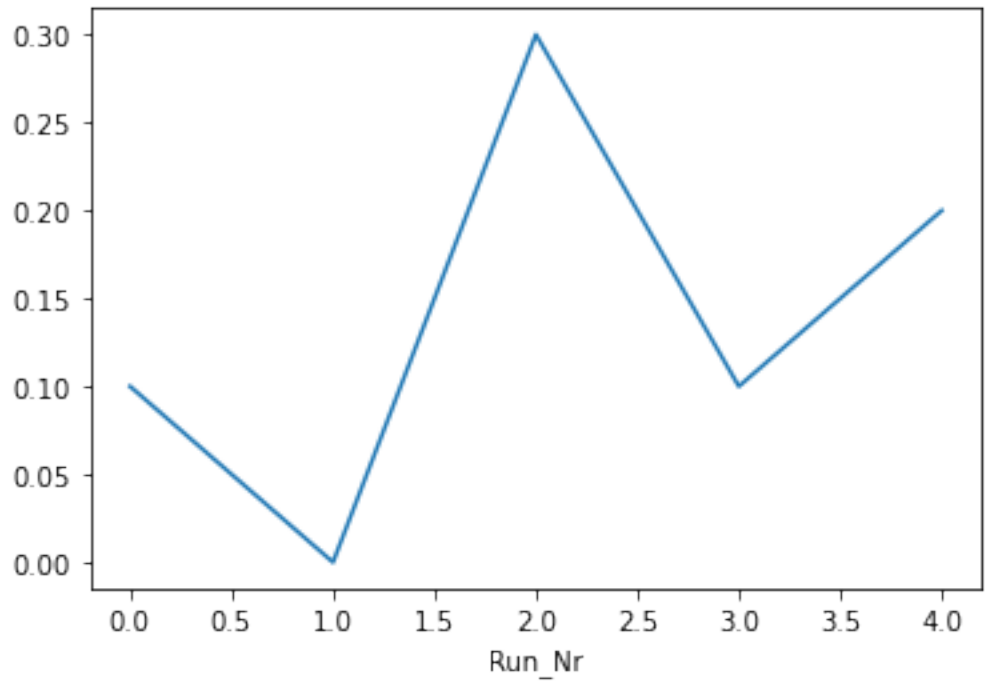
```
Run_Nr
0    0.1
1    0.0
2    0.3
3    0.1
4    0.2
dtype: float64
```

20.4 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()`.

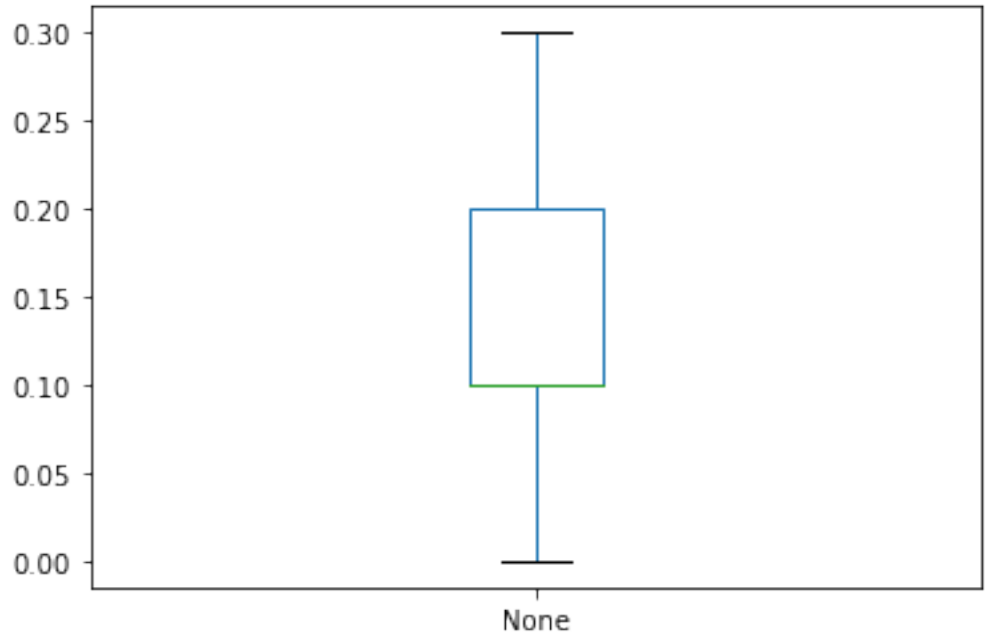
```
anteil_ja.plot()
```

```
<AxesSubplot:xlabel='Run_Nr'>
```



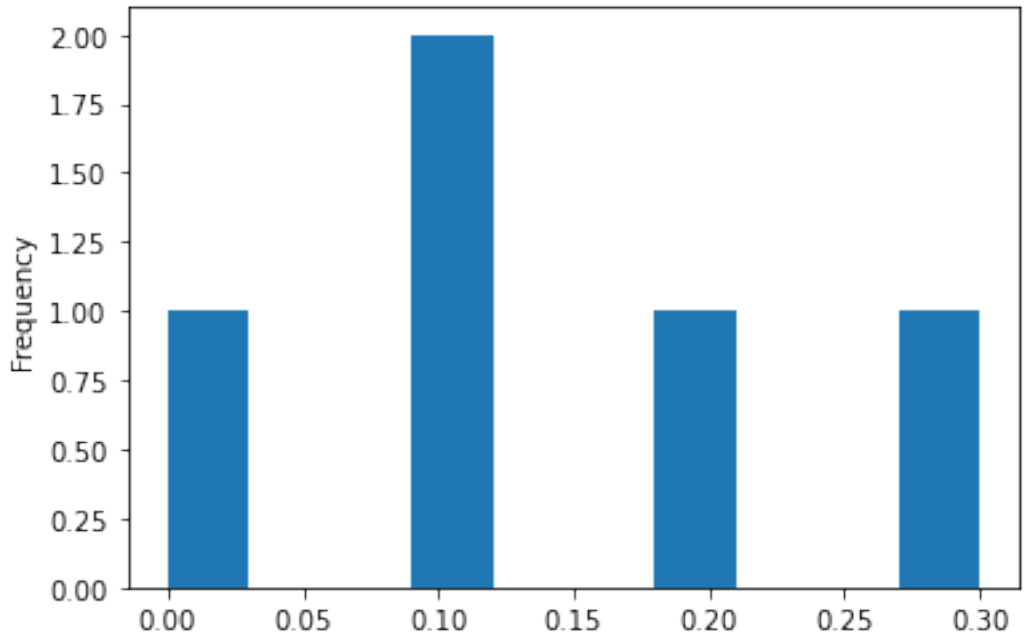
```
anteil_ja.plot.box()
```

```
<AxesSubplot:>
```




```
anteil_ja.plot.hist()
```

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



KAPITEL 21

Basic shortcuts for Jupyter lab

- **Alt + Enter**: Run current cell
- **ESC**: takes users into command mode view while ENTER takes users into cell mode view.
- **A**: inserts a cell above the currently selected cell. Before using this, make sure that you are in command mode (by pressing ESC).
- **B**: inserts a cell below the currently selected cell. Before using this make sure that you are in command mode (by pressing ESC).
- **D + D**: Pressing D two times in a quick succession in command mode deletes the currently selected cell.
- **M**: to change current cell to a markdown cell,
- **Y**: to change it to a code cell and R to change it to a raw cell.
- **CTRL + B**: Jupyter lab has two columns design. One column is for launcher or code blocks and another column is for file view etc. To increase workspace while writing code, we can close it. CTRL + B is the shortcut for toggling the file view column in the Jupyter lab.
- **SHIFT + MIt**: merges multiple selected cells into one cell.
- **CTRL + SHIFT +** –: It splits the current cell into two cells from where your cursor is.
- **SHIFT+J or SHIFT + DOWN**: It selects the next cell in a downward direction. It will help in making multiple selections of cells.
- **SHIFT + K or SHIFT + UP**: It selects the next cell in an upwards direction. It will help in making multiple selections of cells.
- **CTRL + /**: It helps you in either commenting or uncommenting any line in the Jupyter lab. For this to work, you don't even need to select the whole line. It will comment or uncomment line where your cursor is. If you want to do it for more that one line then you will need to first select all the line and then use this shortcut.