
Coding in GIS

Nils Ratnaweera

05.10.2020

1	Einleitung zu diesem Block	3
2	Aufgabe 1: Primitive Datentypen	5
2.1	Theorie	5
2.2	Übungen	7
3	Komplexe Datentypen	9
4	Aufgabe 2: Listen	11
4.1	Theorie	11
4.2	Übungen	12
5	Aufgabe 3: Dictionaries	15
5.1	Theorie	15
5.2	Übungen	16
6	Aufgabe 4: Tabellarische Daten	19
6.1	Theorie	19
6.2	Übungen	19
7	Einleitung zu diesem Block	23
8	Conda cheat sheet	25
9	Python Modules	27
9.1	Vergleich R vs. Python	27
9.2	Python Eigenheiten	28
10	Aufgabe 5: <i>Function Basics</i>	31
10.1	Theorie	31
10.2	Übungen	32
11	Aufgabe 6: <i>Function Advanced</i>	35
11.1	Theorie	35
11.2	Übungen	38
12	Aufgabe 7: Zufallszahlen generieren	41
12.1	Theorie	41

12.2	Übungen	43
13	Aufgabe 8: Funktionen in <i>DataFrames</i>	49
13.1	Theorie	49
13.2	Übungen	50
14	Einleitung zu diesem Block	57
15	Aufgabe 9: <i>For Loop</i> Einführung	59
15.1	Theorie	59
15.2	Übungen	60
16	Aufgabe 10: <i>For Loop</i> Basics	63
16.1	Theorie	63
16.2	Übungen	64
17	Aufgabe 11: <i>For Loops</i> Advanced	67
17.1	Theorie	67
17.2	Übungen	69
18	Aufgabe 12: Zeckenstich Simulation	71
18.1	Theorie	71
18.2	Übungen	72
19	Aufgabe 13: GIS in Python	75
19.1	Theorie	75
19.2	Übungen	80
20	Aufgabe 14: Spatial Joins	85
20.1	Theorie	85
20.2	Übungen	87
21	Einleitung	89
22	Aufgabe 15: Vereinigen aller Einzelteile	91
22.1	Theorie	91
22.2	Übungen	92
23	Abgabe Leistungsnachweis	95
23.1	Zusatzaufträge	95
24	Basic shortcuts for Jupyter lab	97

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 diesen Kurs ist eine Offenheit, neue Tools und Ansätze kennen zu lernen, die Bereitschaft für lösungsorientiertes 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](https://github.com/ratnanil/codingingis).

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. Benennt 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
-

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, wird dies folgendermassen formuliert:

```
# 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

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

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

```
# Musterlösung

vorname+" "+nachname
```

```
'Guido van Rossum'
```

2.2.3 Übung 1.3

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

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

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

```
# Musterlösung  
  
blond = False
```

2.2.6 Übung 1.6

Erstelle eine Variabel `einwohner` mit der Einwohnerzahl der Schweiz (8'603'900, per 31. Dezember 2019). Erstelle eine zweite Variabel `flaeche` (ohne Umlaute!) mit der Flächengrösse der Schweiz (41'285 km²). Berechne nun die Einwohnerdichte.

```
# Musterlösung  
  
einwohner = 8603900  
flaeche = 41285  
  
dichte = einwohner/flaeche  
  
dichte
```

```
208.40256751846917
```

2.2.7 Übung 1.7

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
```

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	<code>hexerei = [3,2,1]</code>
Dict	List	werden über ein Schlüsselwort abgerufen	<code>langenscheidt = { ↳ "trump": ↳ "erdichten"}</code>
DataFrame	Dataframe	Tabellarische Daten	<code>pd. ↳ DataFrame(langenscheidt)</code>

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}`

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

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

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

```
# Musterlösung

vornamen[0]
```

```
'Christopher'
```

4.2.3 Übung 2.3

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

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

```
# Musterlösung  
sum(groessen)
```

695

4.2.5 Übung 2.5

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

```
# Musterlösung  
len(groessen)
```

4

4.2.6 Übung 2.6

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

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

```
# Musterlösung  
min(groessen)  
max(groessen)
```

182

Aufgabe 3: Dictionaries

5.1 Theorie

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

Ähnlich wie eine Liste, ist eine Dictionary ein Behälter, in dem 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“.



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

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

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

```
# Musterlösung
```

```
me["nachname"]
```

```
'van Rossum'
```

5.2.3 Übung 3.3

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

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

```
# Musterlösung
```

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

5.2.5 Übung 3.5

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

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

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

```
# Musterlösung  
people["nachnamen"][2]
```

```
'Kantereit'
```

5.2.8 Übung 3.8

Berechne den Mittelwert aller grössen in deiner Dict

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

```
172.66666666666666
```

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 bedeutet 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

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

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

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 current working directory**) herausfinden (s.u.).

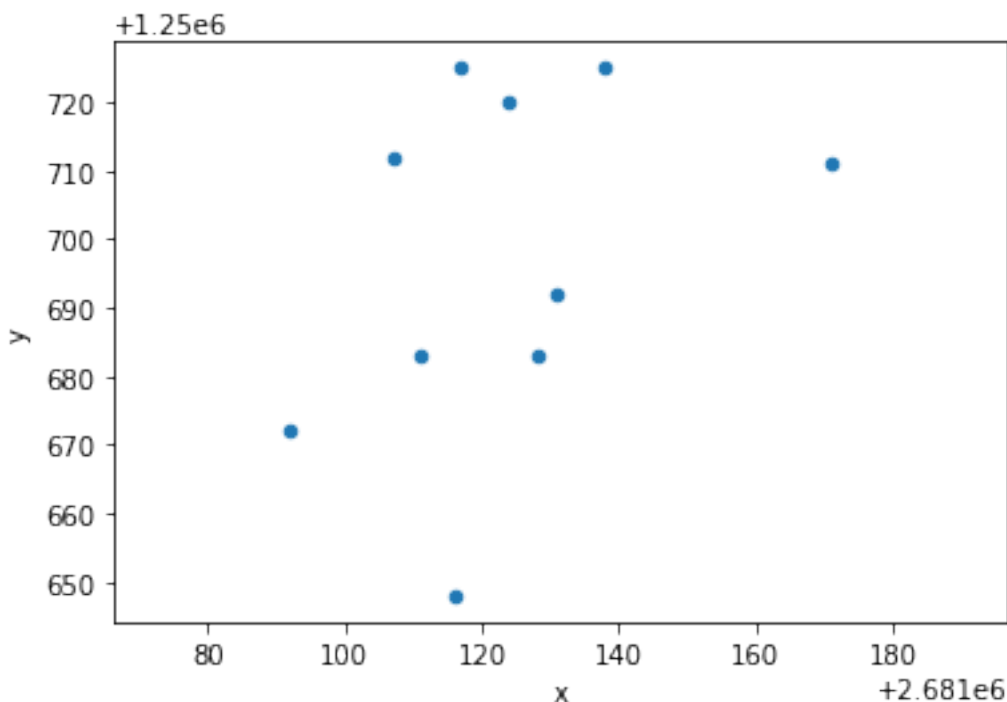
```
import os
os.getcwd()
```


6.2.4 Übung 4.4

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 deshalb 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

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

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

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.
-

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 Programm <i>conda</i>)	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 <i>conda</i> installiert ist)	einmalig und nur, wenn folgender Befehl in der Konsole eine Fehlermeldung verursacht: <code>conda --version</code>	Pfad zur <i>conda</i> -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: <code>conda create --name <u> </u></code> → <code>codingingis</code>
4. Virtual environment aktivieren (schaltet den „Bearbeitungsmodus“ ein)	jedes mal nötig wenn ein Erweiterung installiert oder jupyter lab gestartet werden soll	in der Konsole ⁴ : <code>activate codingingis</code>
5. Jupyter lab installieren (fügt der virtuellen Umgebung diese IDE hinzu)	1x pro <i>environment</i>	in der Konsole ⁵ : <code>conda install -c conda- →forge jupyterlab</code>
6. Jupyter lab starten (startet die IDE „Jupyter Lab“)	jedes mal, wenn am Projekt gearbeitet wird	in der Konsole ⁵ : <code>jupyter lab</code>
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 Module⁶ installieren (fügt der <i>environment</i> zB pandas hinzu)	jedes mal nötig, wenn ein Modul in einer Environment fehlt ⁷	in der Konsole ^{5,8} : <code>conda install -c conda- →forge pandas</code>

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 durch 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

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 Hausbauens 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-forge 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 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
```

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 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 begrüßt 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. Beispielsweise 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

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

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

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

Erstelle eine Funktion square, welche den Input quadriert.

Tipp: „Quadrieren“ heisst ja „mit sich selbst multiplizieren“. In Python können zwei Zahlen mit * multipliziert werden.

```
# Musterlösung

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

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

```
25
```

10.2.5 Übung 5.5

Erstelle eine Funktion `meter_zu_fuss`, die eine beliebige Zahl von Meter in Fuss konvertiert. Zur Erinnerung: 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
```

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 können wir einen Standardwert festlegen. So wird der Parameter optional. Beispielsweise könnte `sag_hallo()` einfach *Hallo Du!* zurück geben, wenn kein Vorname angegeben wird. Um dies zu erreichen, definieren wir den Standardwert 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 Variablen

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

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

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

# Nun erstellen wir eine Function, welche diese Variable ("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

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

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

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

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*.

```
# 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

Erstelle eine Funktion `celsius_zu_fahrenheit`, welche eine beliebige Zahl von Grad Celsius in Grad Kelvin konvertiert. Zur Erinnerung: $Temperatur\ in\ ^\circ F = Temperatur\ in\ ^\circ C \times 1,8 + 32$.

```
# Musterlösung
```

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

Das Resultat sollte folgendermassen aussehen:

```
celsius_in_fahrenheit(celsius = 25)
```

```
77.0
```

11.2.6 Übung 6.6

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

```
# Musterlösung
```

```
celsius_in_fahrenheit2 = lambda celsius: celsius*1.8+32
```


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 Mittelwert sowie die Standardabweichung 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
```

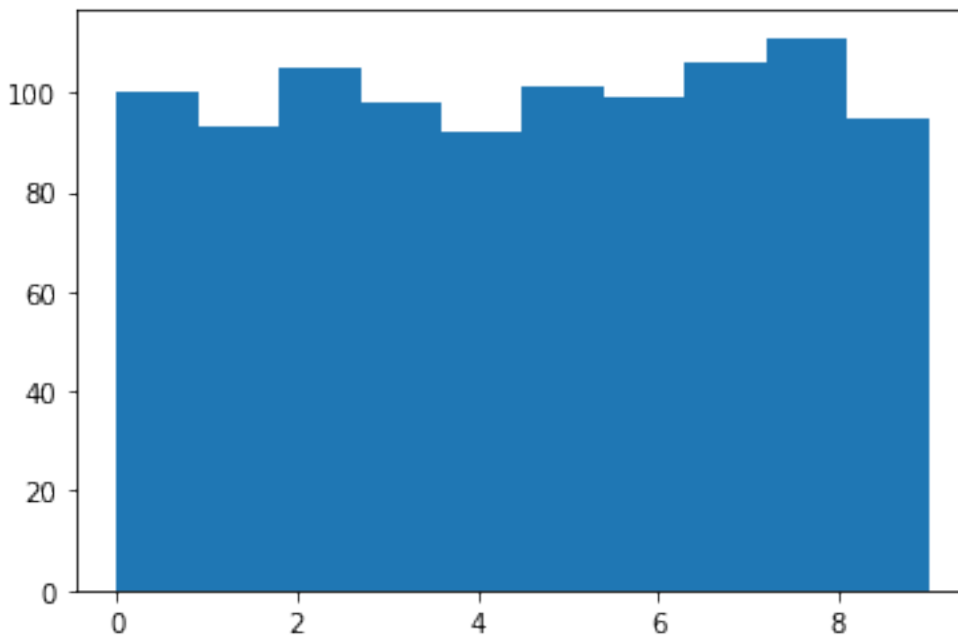
Wenn wir auf diese Weise mit `randrange()` immer wieder neue Zufallszahlen generieren fällt irgendwann auf, dass die Verteilung der Zahlen ziemlich gleichmässig ist. Es ist also gleich wahrscheinlich eine 10 zu bekommen eine 0 oder eine 5. Die Zahlen kommen also aus einer „uniformen“ Verteilung. Dies 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 Standardabweichung 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 = Standardabweichung
random.normalvariate(mu = 5, sigma = 2)
```

```
8.818586486923941
```

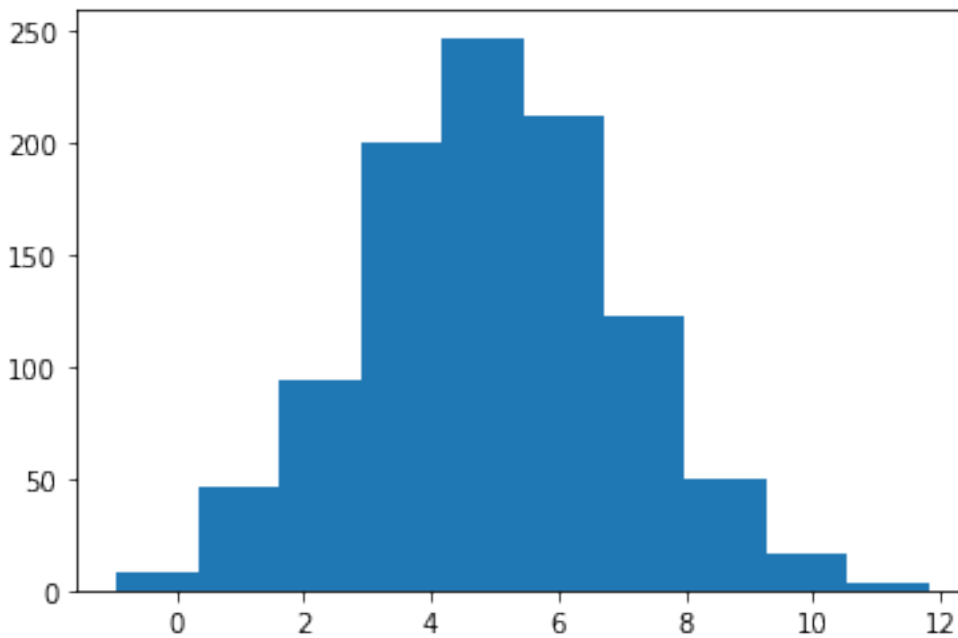
```
# mu = Mittelwert, sigma = Standardabweichung
random.normalvariate(mu = 5, sigma = 2)
```

```
4.632207532391158
```

```
# mu = Mittelwert, sigma = Standardabweichung  
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.

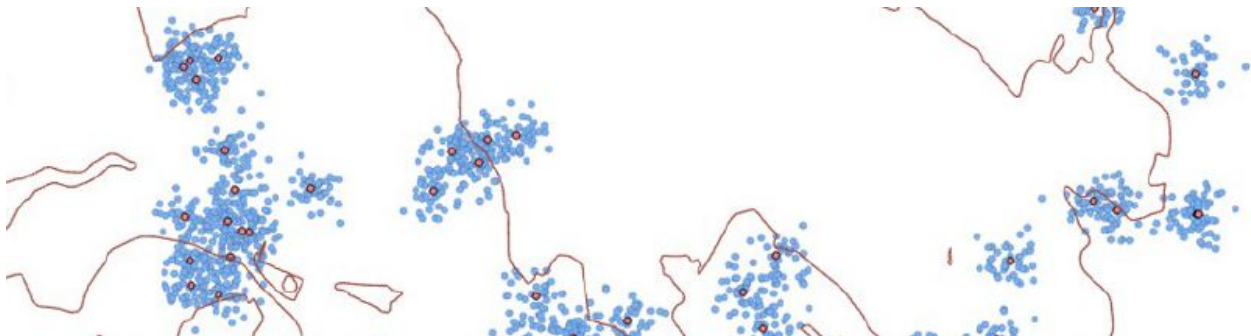


Abb. 12.1: Ausschnitt der simulierten Zeckenstiche. Der rote Punkt stellt jeweils der gemeldete Zeckenstich dar, die blaue Punktwolke drum herum sind simulierte Punkte welche die Ungenauigkeit der Daten widerspiegelt.

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

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 Standardabweichung 20.

```
# Musterlösung

import random

random.uniform(-100,100)

random.normalvariate(100,20)
```

```
115.36479372811071
```

12.2.2 Übung 7.2

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

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 Standardabweichung sinnvoll wäre. Zur Erinnerung: Etwa 68% der Werte liegen innerhalb von +/- 1 Standardabweichung (SD), 95% innerhalb von +/- 2 SD, 99% innerhalb von 3 SD (siehe unten):

Normalverteilung

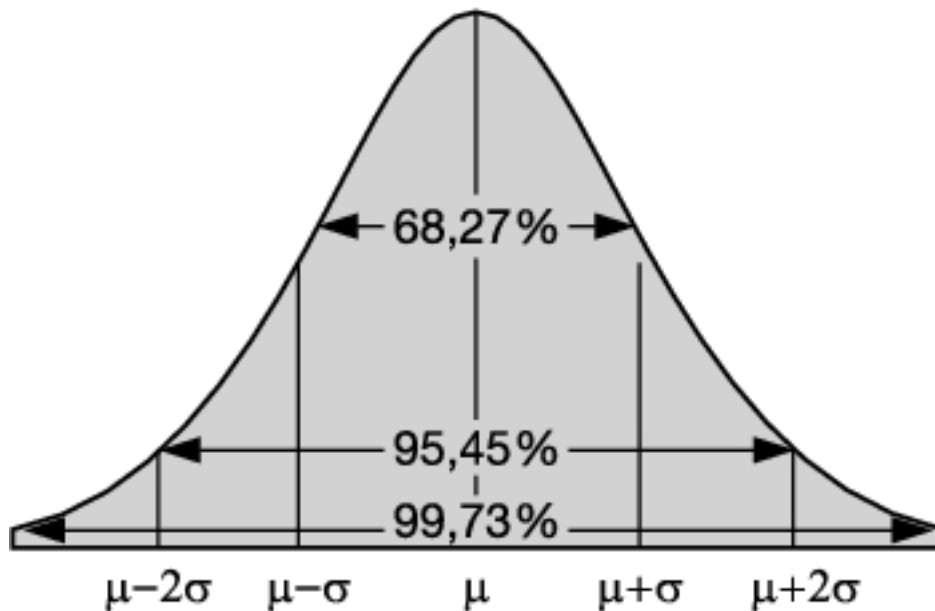


Abb. 12.2: Normalverteilung und die Anteile innerhalb von 1 Standardabweichung (Mittelwert μ minus Standardabweichung σ), 2 Standardabweichungen ($\mu - 2 \times \sigma$) und 2 Standardabweichungen ($\mu + 2 \times \sigma$). Quelle: [cobocards](#)

Musterlösung

```
# Normalverteilte Werte mit Mittelwert 0 und Standardabweichung 100
# Achtung: bei dieser Standardabweichung 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

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
```

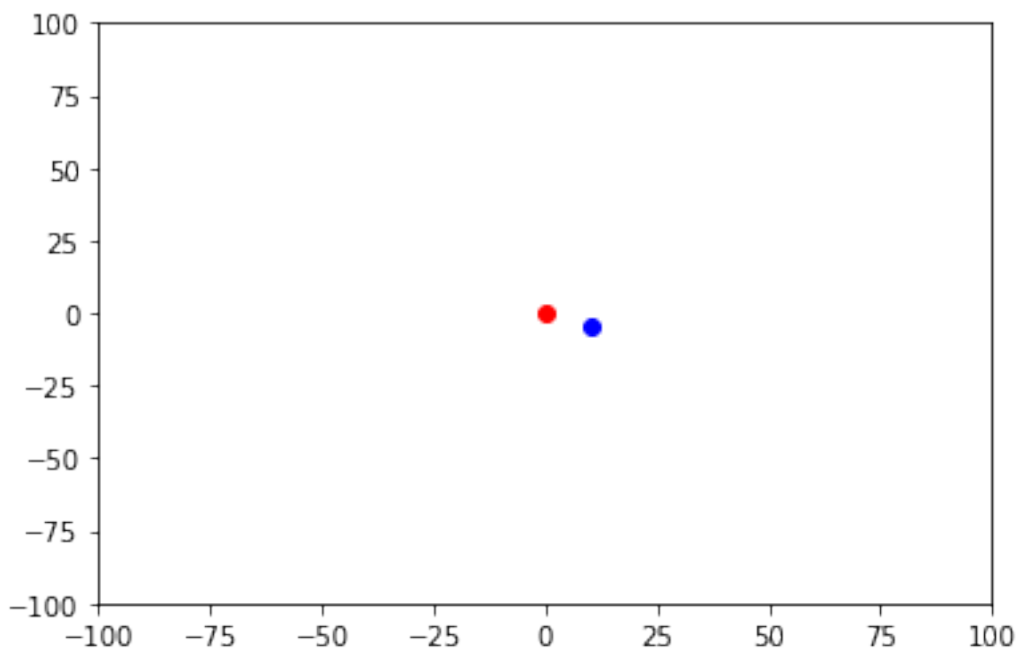
Visuell betrachtet sieht das folgendermassen aus:

```
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

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 Standardabweichung der Verteilung als optionalen Parameter mit einem Standardwert 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

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 Dummykoordinaten an. Nutze *deine* Funktion `offset_coordinate` um eine Visualisierung gemäss unten stehendem Beispiel zu machen.

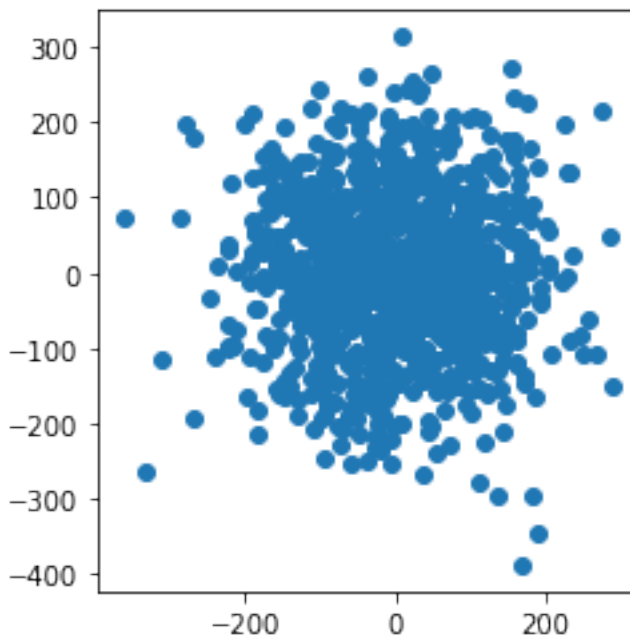
```
from matplotlib import pyplot as plt
import pandas as pd

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)
```



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

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

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 Handgriffe 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

pandas kennt eine ganze Familie von Methoden, um Spalten zu Manipulieren und Daten zu Aggregieren (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    2.681083e+06
1    2.681003e+06
2    2.681029e+06
3    2.681081e+06
4    2.680961e+06
5    2.681265e+06
6    2.681172e+06
7    2.681042e+06
8    2.681215e+06
9    2.681242e+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	2.681117e+06	1.250574e+06

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

1	10437	301.748542	2681092	1250672	2.681134e+06	1.250594e+06
2	9174	301.748542	2681128	1250683	2.681187e+06	1.250720e+06
3	8773	301.748542	2681111	1250683	2.680981e+06	1.250706e+06
4	2764	301.748529	2681131	1250692	2.681215e+06	1.250447e+06
5	2513	301.748529	2681171	1250711	2.681219e+06	1.250753e+06
6	9185	301.748542	2681107	1250712	2.681054e+06	1.250713e+06
7	28521	301.748542	2681124	1250720	2.680971e+06	1.250857e+06
8	26745	301.748542	2681117	1250725	2.680961e+06	1.250584e+06
9	27391	301.748542	2681138	1250725	2.681136e+06	1.250857e+06

13.2.4 Übung 8.4

In *Übung 8.3* 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    2.681188e+06
1    2.680950e+06
2    2.681172e+06
3    2.681134e+06
4    2.681120e+06
5    2.681122e+06
6    2.681187e+06
7    2.680915e+06
8    2.681272e+06
9    2.680991e+06
Name: x, dtype: float64
```

Nutze 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

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-Zeckenstiche, bei den blauen um die simulierten (leicht verschoben) Zeckenstiche. Visualisiere deine eigenen Zeckenstiche auf diese Weise.

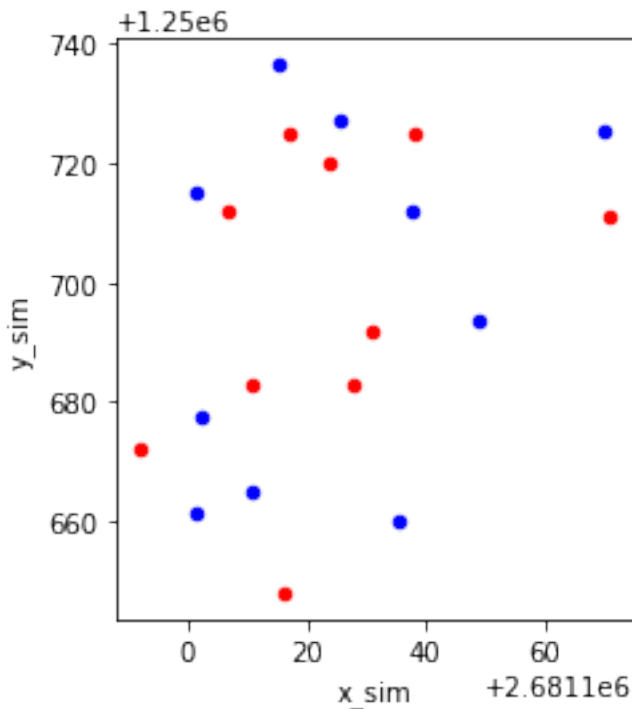
```
from matplotlib import pyplot as plt
```

```
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

In [Übung 8.4](#) haben wir alle Punkte um etwa die gleiche Distanz (+/- 10m) verschoben. Wenn wir unsere *DataFrame* „zeckenstiche“ 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	2.681111e+06	1.250665e+06
1	10437	301.748542	2681092	1250672	2.681101e+06	1.250661e+06
2	9174	301.748542	2681128	1250683	2.681136e+06	1.250660e+06
3	8773	301.748542	2681111	1250683	2.681102e+06	1.250677e+06
4	2764	301.748529	2681131	1250692	2.681149e+06	1.250694e+06
5	2513	301.748529	2681171	1250711	2.681170e+06	1.250726e+06
6	9185	301.748542	2681107	1250712	2.681102e+06	1.250715e+06
7	28521	301.748542	2681124	1250720	2.681126e+06	1.250727e+06
8	26745	301.748542	2681117	1250725	2.681115e+06	1.250736e+06
9	27391	301.748542	2681138	1250725	2.681138e+06	1.250712e+06

```
# Musterlösung

zeckenstiche["x_sim"] = zeckenstiche["x"].apply(offset_coordinate, distance = 1000000)
zeckenstiche["y_sim"] = zeckenstiche["y"].apply(offset_coordinate, distance = 1000000)

zeckenstiche
```

	ID	accuracy	x	y	x_sim	y_sim
0	2550	439.128951	2681116	1250648	2.681423e+06	1.250413e+06
1	10437	301.748542	2681092	1250672	2.681859e+06	1.250376e+06
2	9174	301.748542	2681128	1250683	2.680826e+06	1.251268e+06
3	8773	301.748542	2681111	1250683	2.680721e+06	1.251250e+06
4	2764	301.748529	2681131	1250692	2.681222e+06	1.250063e+06
5	2513	301.748529	2681171	1250711	2.680925e+06	1.250269e+06
6	9185	301.748542	2681107	1250712	2.681698e+06	1.250340e+06
7	28521	301.748542	2681124	1250720	2.680745e+06	1.249504e+06
8	26745	301.748542	2681117	1250725	2.680811e+06	1.251524e+06
9	27391	301.748542	2681138	1250725	2.681458e+06	1.250806e+06

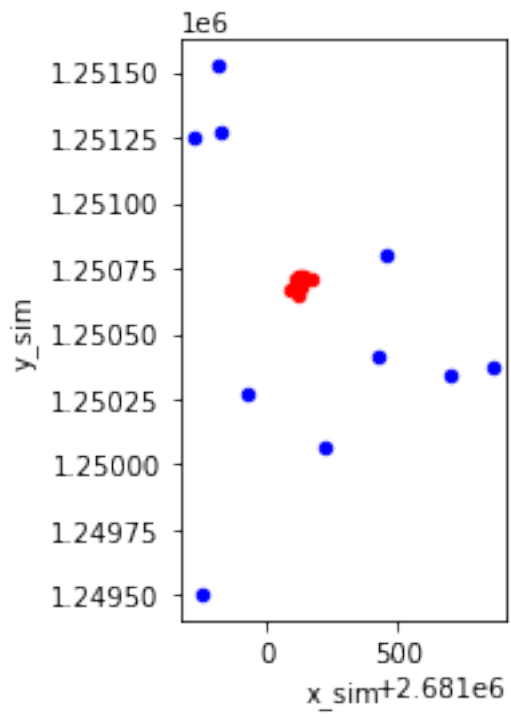
```
# Musterlösung

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

plt.axis("scaled")

plt.show()

# Was ist hier passiert?
# Offensichtlich sind die Zeckenstiche nur wenige Meter voneinander entfernt.
# Der offset von mehreren hundert meter ist viel weiter, als diese Distanz.
```



Einleitung zu diesem Block

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
-

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 Nonsense in die Konsole ausgespuckt

- 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* iteriert, 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 wie 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 Zweck 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 Platzhaltervariable liegt immer zwischen `for` und `in`, der Name dieser Variable könnt ihr frei wählen. Ich habe sie im obigen Beispiel `platzhalter` genannt. Speziell an dieser Variable 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 Platzhaltervariable und verändere den Text, der in `print` herausgegeben wird.

15.2.2 Übung 9.2

Konstruiere eine Liste bestehend aus 3 Namen und nenne diese Liste `namen`. Erstelle danach einen *For Loop*, wo jede Person in der Liste begrüßt 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 Variablen in die Konsole „ausdrucken“ lassen. Innerhalb von `print` können dazu verschiedene Variablen kommasetrennt aufgeführt werden, ohne sie mit `+` verbinden zu müssen wir damals in *Aufgabe 5: Function Basics*.

15.2.3 Übung 9.3

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

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

Bisher haben wir die Platzhalternvariable 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 beiden Sätze 5x wiederholt:

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

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

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 Beispiel¹:

```
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 Variable `liedzeile` aber jetzt anschauen ist dort nur das Resultat aus der letzten Durchführung gespeichert. Das gleiche gilt auch für die Variable `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 vor 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

Nehmen wir nochmals das Beispiel aus *Übung 9.2*. Erstelle nochmal ein Loop, wo drei Personen aus einer Liste begrüßt 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

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

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
'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"]
```

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 *for 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*). 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

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

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

Aufgabe 12: Zeckenstich Simulation

18.1 Theorie

Jetzt wo ihr *For Loops* im Griff habt, nutzen wir dieses Instrument um unsere eigenen Zeckenstichdaten zu Simulieren. Das heisst, wir generieren fiktive Zeckenstiche auf der Basis der Zeckenstichmeldungen in `zeckenstiche.csv`. Es geht jetzt darum, das zufällige verschieben der Zeckenstichpunkte nicht nur einmal durchzuführen wie in [Übung 8.3](#), sondern diesen Prozess 5 mal mithilfe eines Loops zu wiederholen.

Um euer Gedächtnis etwas aufzufrischen: Letzte Woche hatten wir in [Übung 8.3](#) mit `apply()` sowie unserer Funktion `offset_coordinate` alle Koordinaten einer *DataFrame* verschoben und diese Simulierten Daten in unserer *DataFrame* abgespeichert. Schau dir diese Übung und die Musterlösung nochmals an wenn du sie nicht mehr ganz im Kopf hast.

Starte mit einem neuen Jupyter Notebook und lade darin die nötigen Module (`pandas` und `random`), erstellt euch die Funktion `offset_point()` und importiert den Datensatz `zeckenstiche.csv` (s.u.).

```
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")
```

18.2 Übungen

18.2.1 Übung 12.1

Nutze nun dein Wissen über Loops um den Schritt in *Übung 8.3* nicht nur einmal, sondern 5 mal durch zu führen. Kümmere dich zu diesem Zeitpunkt noch nicht um das Abspeichern des Outputs, es reicht wenn die Simulation 5x durchgeführt wird.

18.2.2 Übung 12.2

Nun wollen wir dafür sorgen, dass die simulierten Daten aus der letzten Übung auch irgendwo abgespeichert werden. Sorge dafür, dass der Output in einer Liste mit dem Namen `monte_carlo` gespeichert wird.

Tip: Nutze hier die Anweisungen aus *Aufgabe 10: For Loop Basics* respektive *Übung 10.2*.

18.2.3 Übung 12.3

Schau dir `monte_carlo` an. Wenn du alles richtig gemacht hast, sollte `monte_carlo` eine Liste bestehend aus 5 *DataFrames* sein. Überprüfe dies mit `type(monte_carlo)`, `len(monte_carlo)` und `type(monte_carlo[0])`. Vergleiche auch die *DataFrames* selbst: Fällt dir etwas auf?

```
type(monte_carlo)      # Was für ein Datentyp hat "monte_carlo"?
len(monte_carlo)       # Wie viele Elemente hat "monte_carlo"?
type(monte_carlo[0])   # Was haben diese Elemente für Datentypen?

monte_carlo[0].head(2) # Vergleiche die ersten beiden Zeilen von der ersten DataFrame.
↪ ...
monte_carlo[1].head(2) # ...mit den den ersten beiden Zeilen der zweiten DataFrame
```

18.2.4 Übung 12.4

Scharfen Beobachtern mag aufgefallen sein, dass die simulierten Koordinaten zwischen allen *DataFrames* identisch ist. Dieses Verhalten hat auch mich sehr verwirrt, siehe dazu [meine Frage auf Stackoverflow](#). Es geht über den Umfang dieses Kurses hinaus die Gründe für dieses Verhalten zu erläutern.

Wichtig ist an dieser Stelle vor allem, dass wir das Problem der identischen Zufallsvariablen lösen, dafür braucht ihr nachstehender Code.

```
monte_carlo = []
for i in range(5):
    zeckenstiche["x_sim"] = zeckenstiche["x"].apply(offset_coordinate)
    zeckenstiche["y_sim"] = zeckenstiche["y"].apply(offset_coordinate)
    monte_carlo.append(zeckenstiche.copy()) # hier liegt die Lösung des Problems (
↪ copy)
```

18.2.5 Übung 12.5

Zu einem späteren Zeitpunkt werden wir wissen wollen, welcher simulierte Zeckenstich zu welcher Iteration gehört. Speichere die „Laufnummer“ der Iteration (*i* in meinem Beispiel) als Spalte des *DataFrames*.

18.2.6 Übung 12.6

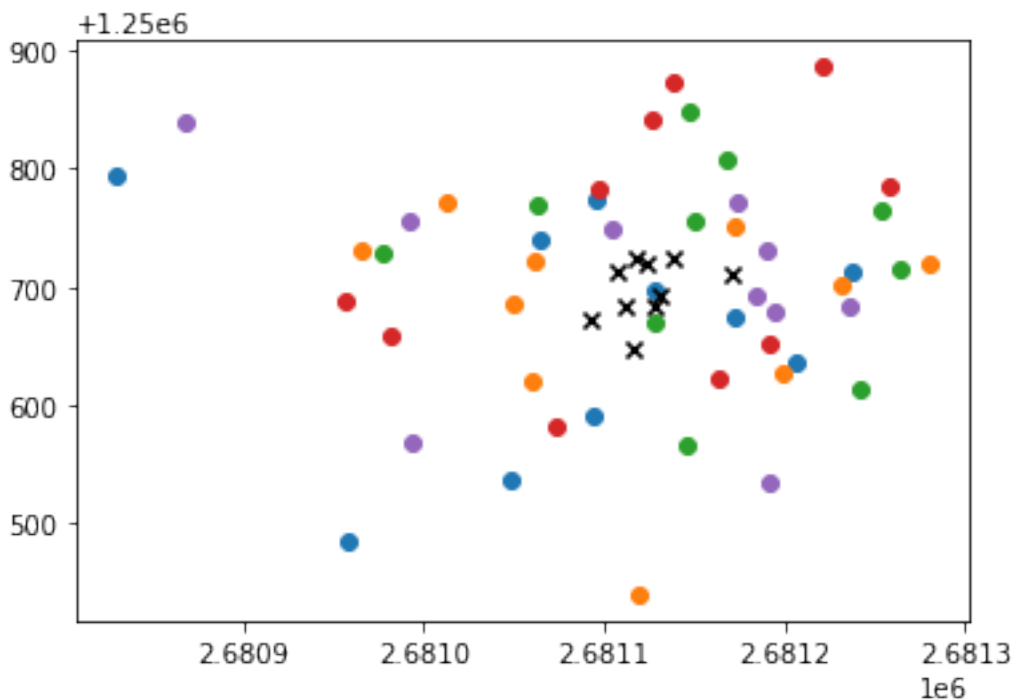
Wir können die simulierten Daten auch mit einem *For Loop* in einem Plot darstellen. Nutze den folgenden Code um deine Daten zu visualisieren.

```
from matplotlib import pyplot as plt

for simulation in monte_carlo:
    plt.scatter(simulation["x_sim"], simulation["y_sim"])

plt.scatter(zeckenstiche["x"], zeckenstiche["y"], color = "black", marker = "x")
```

```
<matplotlib.collections.PathCollection at 0x7fdbf7744b10>
```



18.2.7 Übung 12.7

Aktuell ist *monte_carlo* noch eine Liste bestehend aus 5 *DataFrames* mit jeweils den gleichen Spalten und jeweils 10 Zeilen. Um aus dieser Liste von *DataFrames* eine einzelne *DataFrame* mit 50 Zeilen zu machen können wir die Funktion `pd.concat`¹ nutzen. Wende diese Funktion auf *monte_carlo* an und speichere den Output in der Variabel *monte_carlo_df*.

¹ Dies ist der gleiche Schritt, wie ihr in Übung 2 in „Datenqualität und Unsicherheit“ mit dem ArcGIS Geoprocessing Tool „Merge“ durchgeführt habt.

```
monte_carlo_df = pd.concat(monte_carlo)

monte_carlo_df.head()
```

	ID	accuracy	x	y	x_sim	y_sim	Run
0	2550	439.128951	2681116	1250648	2.681172e+06	1.250675e+06	0
1	10437	301.748542	2681092	1250672	2.681129e+06	1.250697e+06	0
2	9174	301.748542	2681128	1250683	2.681237e+06	1.250714e+06	0
3	8773	301.748542	2681111	1250683	2.681093e+06	1.250592e+06	0
4	2764	301.748529	2681131	1250692	2.681048e+06	1.250537e+06	0

Aufgabe 13: GIS in Python

19.1 Theorie

Manche mag das jetzt überraschen, andere haben es vielleicht schon gemerkt: Mit **GIS** hatte unseres bisheriges Wirken in Python eigentlich wenig zu tun. Unsere Zeckenstiche haben zwar x/y-Koordinaten, aber diese haben wir bisher gleich behandelt wie alle anderen Spalten.

Anders gesagt: Wir wissen ja, dass mit den Spalten `x` und `y` Koordinaten in der Schweiz gemeint sind, Python hingegen wusste das bisher (noch) nicht. Der konkrete Raumbezug fehlt also noch, und das wird irgendwann problematisch: Denn bald wollen wir pro simulierten Zeckenstich ermitteln, ob er sich im Wald befindet oder nicht. Das ist eine explizit räumliche Abfrage, welche nur mit explizit räumlichen Geodaten beantwortet werden kann.

Was wir später mit den simulierten Zeckenstiche machen wollen, spielen wir an dieser Stelle mit den Zeckenstichreports (`zeckenstiche.csv`) durch.

```
# Unsere Zeckenstiche hatten wir auf folgende Weise importiert:  
  
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

19.1.1 GeoDataFrames

Glücklicherweise können wir unsere Zeckenstich-*Dataframe* mit nur einem Zusatzmodul und wenigen Zeilen code in eine **räumliche** *DataFrame* konvertieren. Mit dem Modul *geopandas* erstellen wir aus unserer *pandas DataFrame* eine *geopandas GeoDataFrame*. Mit dieser Erweiterung 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.

Bei *geometry* handelt es sich Punkt-Geometrien die sich durch die *x* und *y* Koordinaten zusammensetzen. Für das Attribut *crs* möchte ich kurz etwas ausholen:

19.1.2 Koordinatensystem

Das Koordinatensystem gibt unseren *x/y*-Zahlenwerten einen konkreten Raumbezug auf dem Planeten. Sie macht aus ihnen Koordinaten *in der Schweiz*. Wie lautet aber das „Koordinatensystem“ unserer Daten?

Das ist Information, welche euch vom Datenersteller geliefert werden muss. Man kann das Koordinatensystem aber auch anhand der Koordinaten errahnen. Es handelt sich dabei um Werte im Bereich von 2.6 Mio auf der einen und 1.2 Mio auf der anderen Achse. Da wir wissen dass die Daten aus der Schweiz stammen kann man mit etwas Erfahrung sagen, dass es sich um Daten im neuen Schweizer Koordinatensystem **CH1903+ / LV95** handeln muss. Der EPSG Code dieses Koordinatensystems lautet 2056 und diesen Code können wir in der Funktion *gpd.GeoDataFrame* nutzen, um das Korrekte Koordinatensystem (engl. **Coordinate Reference System**) zu zuweisen (s.u.)

```
import geopandas as gpd

zeckenstiche_gpd = gpd.GeoDataFrame(zeckenstiche, # die Basis-DataFrame / die
    ↪ "Attributtabelle"

                                # die Geometrie Spalte:
                                geometry=gpd.points_from_xy(zeckenstiche['x'],
    ↪ zeckenstiche['y']),

                                crs = 2056) # EPSG Code des Koordinatensystem

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
```

```
# Vergleich mit "zeckenstiche"
type(zeckenstiche)
```



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

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

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

19.1.3 Räumliche Operationen

Was hat das nun bewirkt, was bringt uns diese *Geo* Erweiterung? Ein kleiner, aber offensichtlicher Vorteil ist das veränderte Verhalten von der `.plot()` Methode. Da `zeckenstiche_gpd` ein räumliches Objekt ist, macht `.plot()` automatisch eine räumliche Darstellung der Daten, `.plot.scatter(...)` ist also nicht mehr nötig, wie es *vorher* der Fall war¹.

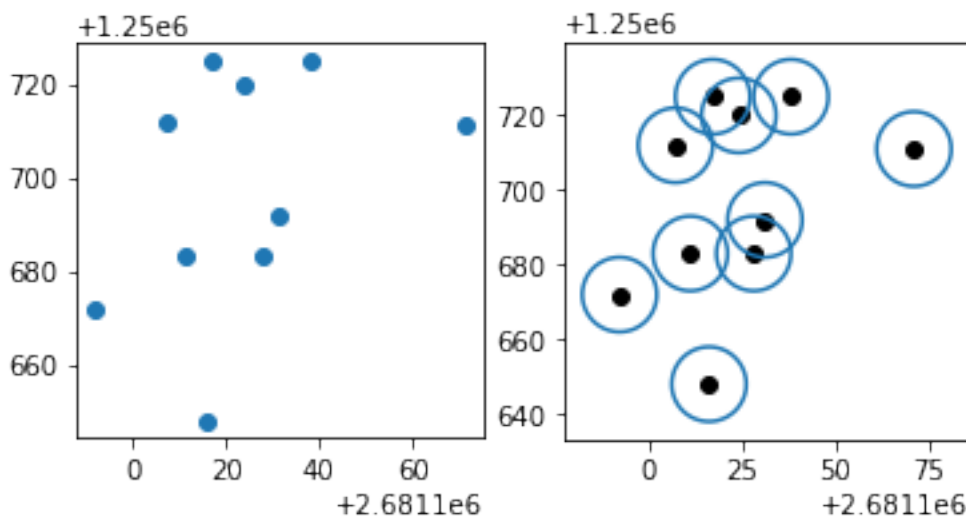
Die Vorteile gehen aber noch viel weiter. Mit *GeoPandas* sind nun auch räumliche Operationen möglich, die wir bereits aus ArcGIS kennen aber mit einfachen *DataFrames* noch nicht möglich waren. Mit `.buffer()` können wir zum Beispiel einen Buffer um unsere Punkte machen.

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

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

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.

¹ Zudem sorgt `.plot` 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* der Fall war.

Mit `unary_union`² können wir aus unseren Einzelpunkten ein *Multipolygon* erstellen und darüber ein *Convex Hull*³ oder ein *Envelope*⁴ 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 Output als Polygon
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

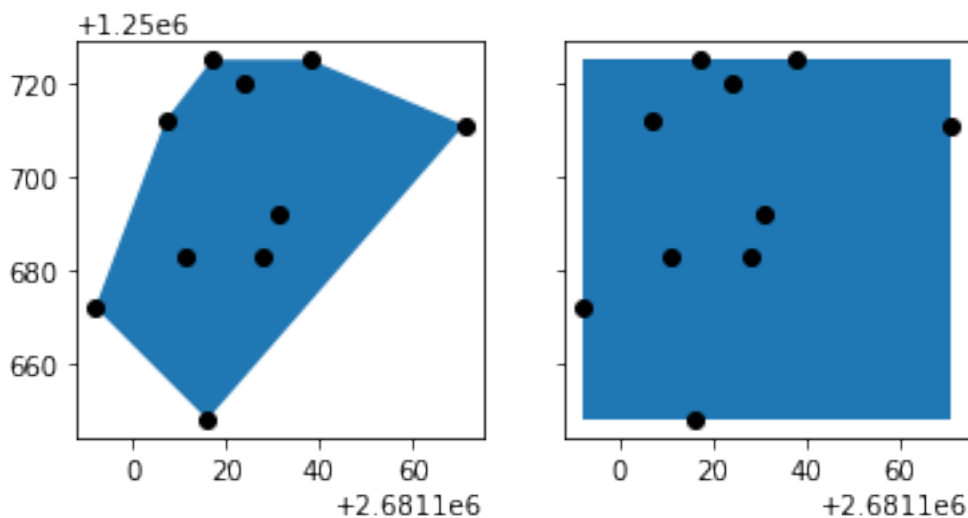
# Konvertiert das Polygon in eine GeoSeries
# (GeoSeries sind praktischer als Polygone)
my_envelope = gpd.GeoSeries(my_envelope)

# 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:>



² Dieser Befehl lautet in ArcGIS [Union](#)

³ 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*)

⁴ 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*)

Wichtig!

Geopandas kennt drei Fundamentale Datentypen:

- **Geometrien:** Einzelne Objekte⁵ der folgenden Typen
 - Points / Multi-Points
 - Lines / Multi-Lines
 - 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

GeoDataFrame					
	ID	accuracy	x	y	geometry
0	2550	439.128951	2681116	1250648	POINT (2681116.000 1250648.000)
1	10437	301.748542	2681092	1250672	POINT (2681128.000 1250683.000)
2	9174	301.748542	2681128	1250683	POINT (2681111.000 1250683.000)
3	8773	301.748542	2681111	1250683	POINT (2681131.000 1250692.000)
4	2764	301.748529	2681131	1250692	

Abb. 19.1: Die drei Datentypen in Geopandas.

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 der drei Operationen ist jedoch 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. Zudem hilft es, die Datentypen regelmässig mit `type` zu kontrollieren.

⁵ 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.

```
#####
# Diese 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() #
#####
```

Das Minimum Convex Polygon



Die gebufferten Punkte



Die Differenz der ersten beiden



19.2 Übungen

19.2.1 Übung 13.1

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

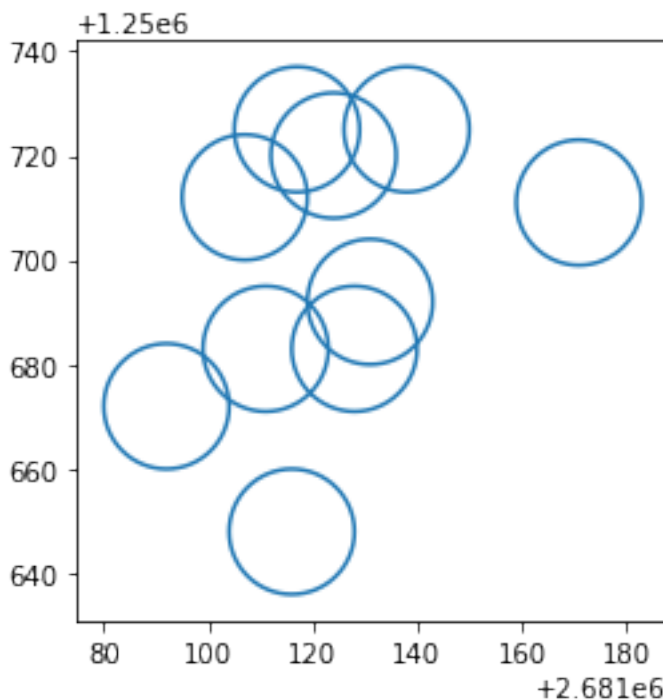
19.2.2 Übung 13.2

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.

19.2.3 Übung 13.3

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

<AxesSubplot:>



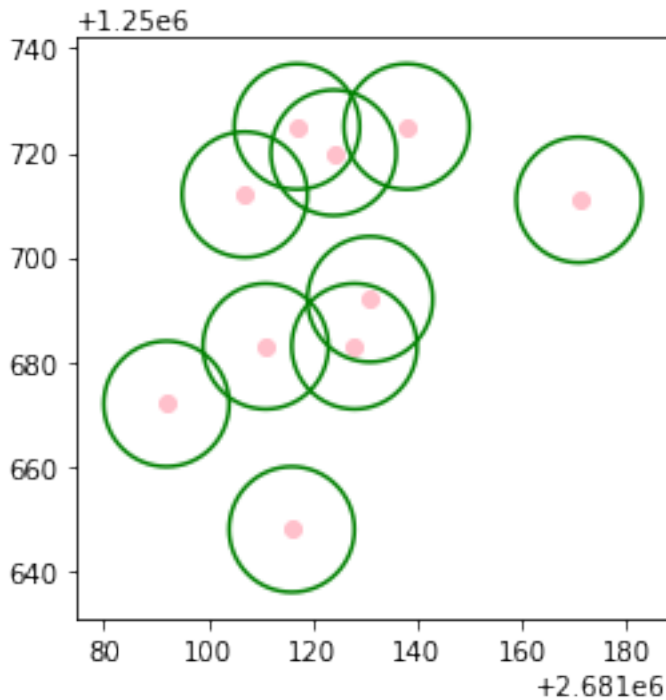
19.2.4 Übung 13.4

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:>



19.2.5 Übung 13.5

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](#))

19.2.6 Übung 13.6

Führe verschiedene Overlay Operationen zwischen `zeckenstiche_envelope` und `zeckenstiche_buffer` durch. Schaue dir dazu die entsprechende [Geopandas Hilfeseite](#) an. Beispielsweise würden sich *Union* und *Symmetrical Difference* gut anbieten.

Tipp:

- `zeckenstiche_envelope` musst zu zuerst noch in eine GeoSeries umwandeln. Den Output 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
```

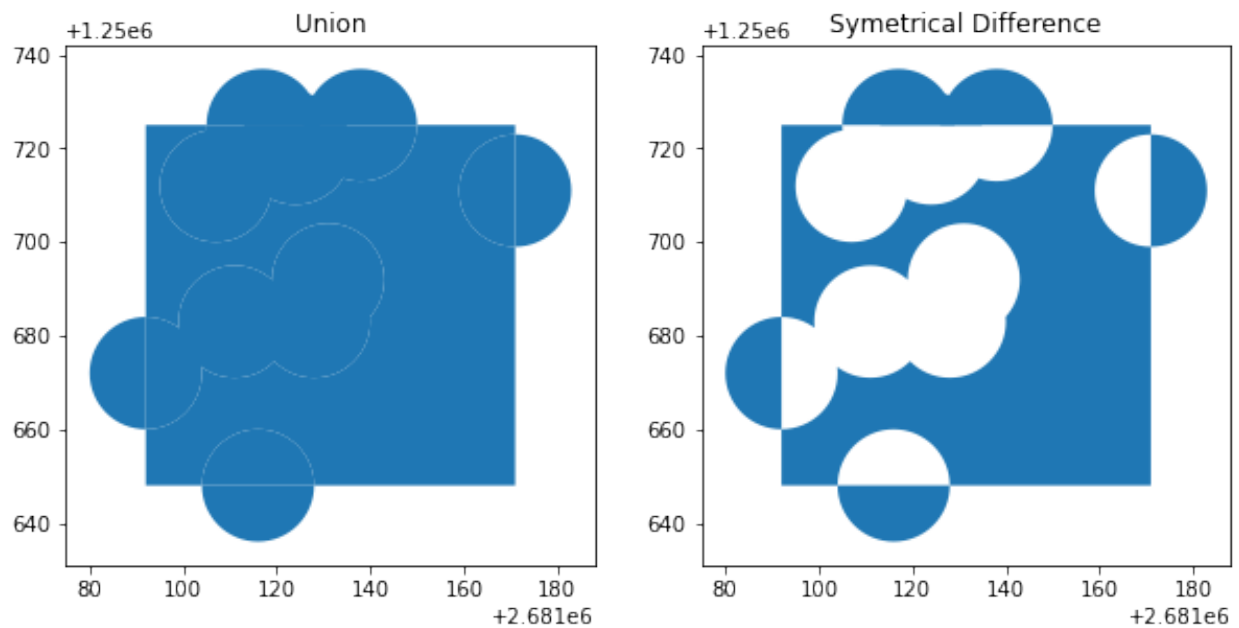
(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
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')
```



Aufgabe 14: Spatial Joins

20.1 Theorie

In *Aufgabe 13: GIS in Python* habt ihr euch mit den GIS-Funktionalitäten von `geopandas` vertraut gemacht. Eine ganz zentrale Funktion in GIS sind die sogenannten „Spatial Joins“. Dabei werden Attribute von einem Geodatensatz auf einen anderen Geodatensatz aufgrund einer räumlichen Beziehung der beiden Datensätze übertragen. Konkret auf unsere Zeckenstiche bedeutet dies: Jedem Zeckenstich sollte die Eigenschaft „Wald: ja“ / „Wald: nein“ aus `wald` zugewiesen werden. Am einfachsten lässt sich dies in einer Darstellung erklären:

Angenommen ihr habt folgende Ausgangslage:

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

zeckenstiche_gpd = gpd.GeoDataFrame(zeckenstiche,
                                   geometry=gpd.points_from_xy(zeckenstiche['x'],
↳zeckenstiche['y']),
                                   crs = 2056)

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

```
/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
```

In Python wird ein *Spatial Join* zwischen `zeckenstiche` und `wald` wie folgt durchgeführt (wichtig ist dabei auch die Reihenfolge der Argumente: `left_df` bestimmt den Geometrietyp des Outputs):

```
gpd.sjoin(left_df = zeckenstiche_gpd, right_df = wald)
```

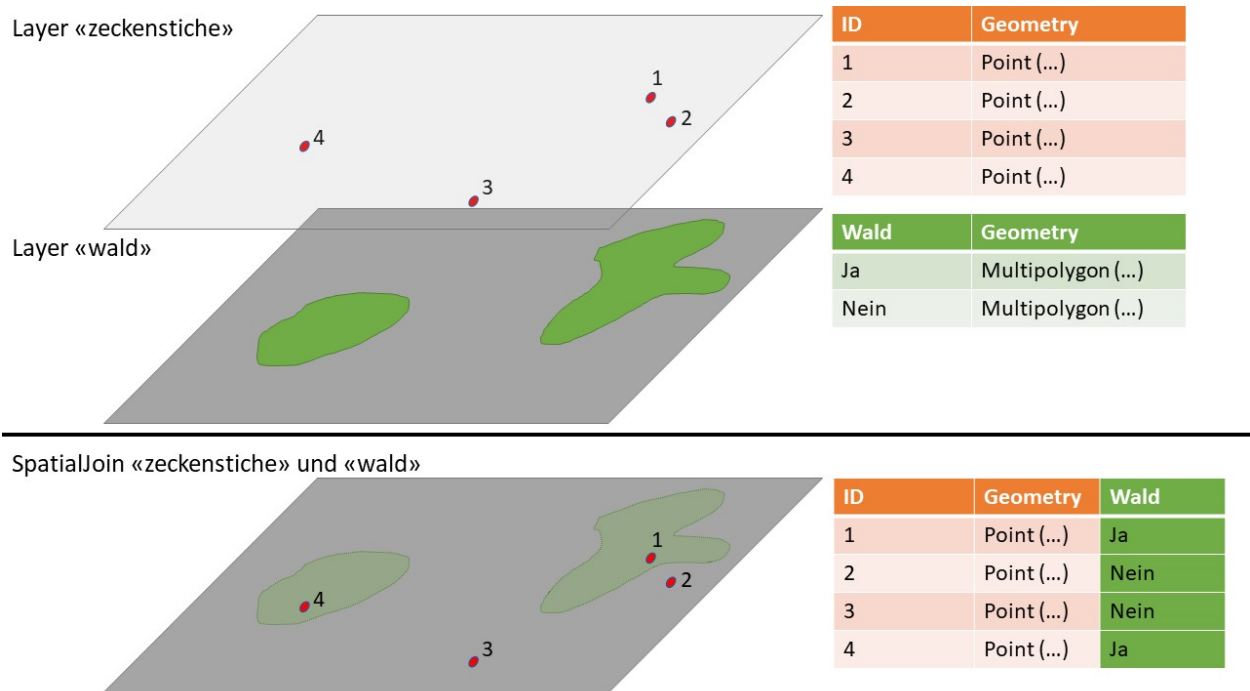


Abb. 20.1: „Spatial Join“ zwischen zeckenstiche und wald. In diesem Spatial Join wurde die Geometrie von zeckenstiche übernommen, das heisst das Resultat des Joins ist ein Punkt-Layer.

	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)

	index_right	Shape_Area	Wald_text
0	0	2.380876e+08	nein
1	0	2.380876e+08	nein
2	0	2.380876e+08	nein
3	0	2.380876e+08	nein
4	0	2.380876e+08	nein
5	1	7.963237e+07	ja
6	1	7.963237e+07	ja
7	1	7.963237e+07	ja
8	1	7.963237e+07	ja
9	1	7.963237e+07	ja

20.2 Übungen

20.2.1 Übung 14.1

Lade hier das File „wald.gpkg“ herunter (Rechtsklick > Speichern unter) und speichere es in deiner Workings Directory. Importiere den Datensatz und speichere ihn in der Variable `wald`.

Schau dir `wald` an (mit `type`, `.plot()` etc.)

20.2.2 Übung 14.2

Führe einen `SpatialJoin` zwischen `wald` und `zeckenstiche` durch. Vertausche die Reihenfolge (`left_df`, `right_df`) und schaue dir den Output an. Was ist hier passiert?

Im Leistungsnachweis von *Coding in GIS* werden wir die Übung 2 - 3 aus „Datenqualität und Unsicherheit“ in Python umsetzen, aber mit selbst simulierten Daten.

In der Übung geht es um folgendes: Wir wissen das die Lagegenauigkeit der Zeckenstichmeldungen mit einer gewissen Unsicherheit behaftet sind. Um diese Unsicherheit bei der Frage „*Welcher Anteil der der Zeckenstiche befinden sich im Wald?*“ zu berücksichtigen, führen wir eine *Monte Carlo Simulation* durch. In diesem Ansatz simulieren wir Zeckenstichmeldungen durch zufälliges Verschieben ihrer Koordinaten. Für jede Iteration der Verschiebung („*Run*“) berechnen wir den Anteil der Meldungen *im Wald*. Die Verteilung dieser Werte beantwortet die Frage („*Welcher Anteil der der Zeckenstiche befinden sich im Wald?*“) unter Berücksichtigung der Unsicherheit.

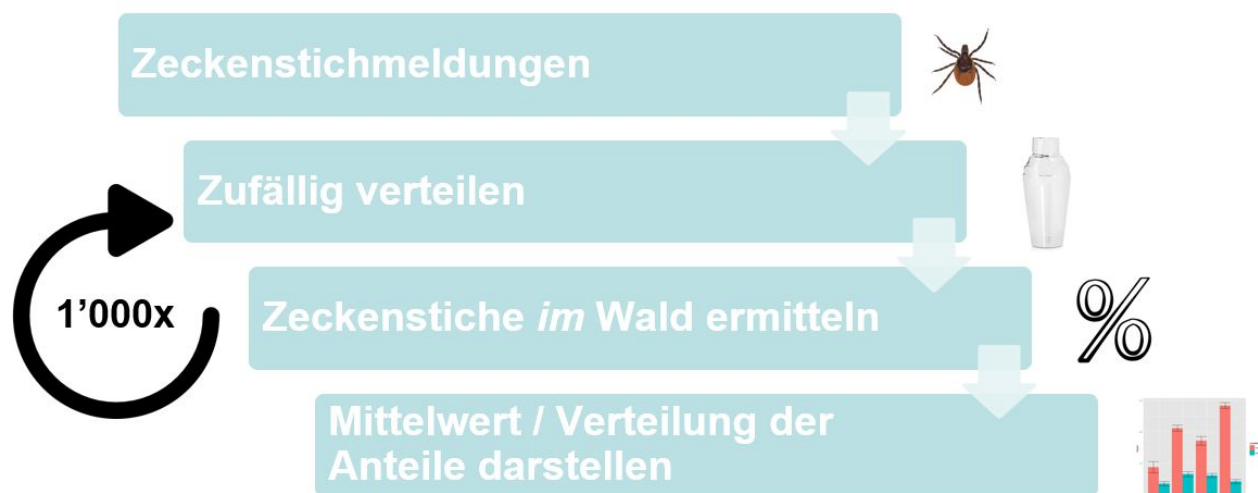


Abb. 21.1: Der generelle Workflow userer Monte Carlo Simulation

Um diese Komplexe Aufgabe lösen zu können müssen wir sie in Teilaufgaben zerlegen. Wir haben in den drei Blöcken (*Coding in GIS I - III*) darauf hingearbeitet, alle Voraussetzungen zur Lösung dieser Teilaufgaben zu erfüllen. Siehe dazu die nachstehende Tabelle:

Tab. 21.1: Teilaufgaben für das Lösen der Monte Carlo Simulation

Teilaufgabe	Status	Voraussetzung
Einen Einzelpunkt zufällig verschieben	✓ haben wir gelöst in <i>Übung 7.5</i>	<i>Aufgabe 5: Function Basics</i> und <i>Aufgabe 6: Function Advanced</i>
Alle Punkte einer DataFrame zufällig verschieben	✓ haben wir gelöst in <i>Übung 8.3</i>	<i>Aufgabe 7: Zufallszahlen generieren</i> und <i>Aufgabe 8: Funktionen in DataFrames</i>
Wiederholung von Schritt 2 (zum Beispiel 50-mal)	✓ haben wir gelöst in <i>Aufgabe 12: Zeckenstich Simulation</i>	<i>Aufgabe 9: For Loop Einführung</i> und <i>Aufgabe 10: For Loop Basics</i>
Für jeden simulierten Punkt ermitteln, ob er sich im Wald / ausserhalb des Waldes befindet	✓ haben wir gelöst in <i>Übung 14.2</i>	<i>Aufgabe 13: GIS in Python</i>
Anteil der Punkte im Wald pro Run ermitteln	werden wir in <i>Aufgabe 15: Vereinigen aller Einzelteile</i> lösen	Coding in GIS I - III

Aufgabe 15: Vereinigen aller Einzelteile

22.1 Theorie

Um an diesen Punkt zu kommen habt ihr euch diverse Werkzeuge aneignen müssen (*Functions, For Loops, GIS*) und habt dazu Übungen gelöst. Die nachstehenden Code Blöcke nutzt Elemente aus diesen Übungen um auf den gleichen Stand zu kommen, mit der ihr im Themenblock „Datenqualität und Unsicherheiten“ in die Übung gestartet seid. Ihr hattet damals (a) Simulierte Zeckenstich-Daten und (b) einen Layer Waldinformation im Projektgebiet.

Kopiere die Code Blöcke in ein frisches Notebook und versuche den Code zu verstehen - sie sollten dir sehr bekannt vorkommen. Schau, ob du gewisse Teile an deine Umgebung anpassen musst.

Lade die benötigten Module (siehe *Erweiterung laden*):

```
# Benötigten Module laden #####
import pandas as pd
import geopandas as gpd
from matplotlib import pyplot as plt
#####
```

```
/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
```

Definiere die benötigte *Function* (siehe *Übung 7.5*):

```
# Notwendige Functions definieren #####
def offset_coordinate(old, distance = 100):
    import random
    new = old + random.normalvariate(0,distance)
    return(new)
#####
```

Importiere die notwendigen Datensätze (siehe *Übung 4.3* und *Übung 14.1*):

```
# Daten Importieren #####
zeckenstiche = pd.read_csv("zeckenstiche.csv")
wald = gpd.read_file("wald.gpkg")
#####
```

Simuliere die Zeckenstiche (siehe *Übung 12.2*):

```
# Daten Simulieren #####
monte_carlo = []
for i in range(5):
    zeckenstiche["x_sim"] = zeckenstiche["x"].apply(offset_coordinate)
    zeckenstiche["y_sim"] = zeckenstiche["y"].apply(offset_coordinate)
    zeckenstiche["Run"] = i
    monte_carlo.append(zeckenstiche.copy())

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

22.2 Übungen

Wir wollen nun den Anteil der Zeckenstiche pro „Simulationsrunde“, sprich pro *Loop* Iteration, ermitteln. Diese ist in `monte_carlo_df` mit `Run` gekennzeichnet. Dafür müssen wir noch folgendes machen:

1. Die *DataFrame* `monte_carlo_df` in eine *GeoDataFrame* konvertieren
2. Für jeden simulierten Punkt ob er sich im Wald befindet oder nicht
3. Der Anteil der Zeckenstiche im Wald *pro Run* ermitteln
4. Verteilung dieser berechneten Anteile visualisieren

22.2.1 Übung 15.1

Konvertiere `monte_carlo_df` in eine *GeoDataFrame* und speichere den Output als `monte_carlo_gpd`. Setze dabei auch das Korrekte Koordinatensystem. Schaue dir nochmal [Aufgabe 13: GIS in Python](#) an wenn du dir unsicher bist, wie das geht.

22.2.2 Übung 15.2

Ermittle für jeden simulierten Punkt, ob er im Wald ist oder nicht und speichere den Output als `monte_carlo_join`. Schaue dir nochmal [Aufgabe 14: Spatial Joins](#) an wenn du dir unsicher bist, wie das geht. Schau dir anschliessend `monte_carlo_join` an. Sortiere die **GeoDataFrame* nach „Run“ (`monte_carlo_join.sort_values("Run")`) um anschliessend nach „ID“ und versuche so, die Daten zu verstehen.

22.2.3 Übung 15.3

Nun wollen wir von jeder Iteration (bzw. jedem „Run“) wissen, wie viele der der Simulierten Punkte sich im Wald und wie viel sich ausserhalb des Waldes befinden. Dafür müssen wir `monte_carlo_join` nach „Run“ und „Wald_text“ gruppieren (mit `.groupby` und anschliessend die Anzahl der Werte berechnen (mit `.size`). Führe diese Schritte bei dir aus und schau dir den Output an.

```
monte_carlo_join.groupby(["Run", "Wald_text"]).size()
```

```
Run  Wald_text
0    ja          5
    nein         5
1    ja          3
    nein         7
2    ja          7
    nein         3
3    ja          8
    nein         2
4    ja         10
dtype: int64
```

22.2.4 Übung 15.4

Das ist schon mal nahe dran an was wir wollen, wir haben nun aber für jede Iteration zwei Zeilen: Eine für „Ja“ (im Wald) und eine für „Nein“ (nicht im Wald). Für die kommenden Schritte wäre es aber praktischer, wenn wir pro Iteration eine Zeile hätten und in zwei Spalten die Anzahl („Ja“ und „Nein“) die Anzahl Werte im Wald und ausserhalb des Waldes. Dies erreichen wir durch den Zusatz `.unstack()`. Führe diesen Befehl bei dir aus und schau dir den Output an.

```
monte_carlo_join.groupby(["Run", "Wald_text"]).size().unstack()
```

```
Wald_text    ja  nein
Run
0           5.0  5.0
1           3.0  7.0
2           7.0  3.0
3           8.0  2.0
4          10.0  NaN
```

22.2.5 Übung 15.5

In meiner Simulation sind im Run Nummer 4 alle Zeckenstiche im Wald gelandet, deshalb steht dort in der Spalte „Nein“ der Wert NaN („not a number“). Mit `fill_value` kann festgelegt werden, was in einem solchen Fall passieren soll. In unserem Fall ist ein `fill_value` von 0 sinnvoll. Ergänze dies in deinem Code und speichere den Output als `mont_carlo_results`.

```
mont_carlo_results = monte_carlo_join.groupby(["Run", "Wald_text"]).size().
↳ unstack(fill_value = 0)

mont_carlo_results
```

```
Wald_text  ja  nein
Run
0           5    5
1           3    7
2           7    3
3           8    2
4          10    0
```

22.2.6 Übung 15.6

Wir haben nun die *Anzahl* der Werte pro Kategorie und Iteration, aussagekräftiger wäre aber deren *Anteil* $ja_anteil = ja / (ja+nein)$. Erstelle eine neue Spalte „ja_anteil“ wo der Anteil der Punkte im Wald enthalten ist.

```
# Das Resultat sollte folgendermassen aussehen:
mont_carlo_results
```

```
Wald_text  ja  nein  ja_anteil
Run
0           5    5         0.5
1           3    7         0.3
2           7    3         0.7
3           8    2         0.8
4          10    0         1.0
```

22.2.7 Übung 15.7

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 `boxplot()`.

Abgabe Leistungsnachweis

Folgendes müsst ihr für den Leistungsnachweis für Coding in GIS abgeben:

- Inhalt: Ein Report der Analyse der Zeckenstichdaten gemäss *Der generelle Workflow unserer Monte Carlo Simulation*
 - Der Inhalt ist Gegeben durch *Aufgabe 15: Vereinigen aller Einzelteile* inkl. der *Zusatzaufträge*
 - Integriert erklärender Text mit Markdown Zellen. Hier müsst ihr keine Romane schreiben
- Form:
 - Ein Jupyter-Notebook-File (*.ipynb)
 - Ein html-Export des Files:
 - * **Wichtig:** Zuerst alle Zellen ausführen (*Run > Run all cells*)
 - * Export via *File > Export Notebook as > Export Notebook to html*

23.1 Zusatzaufträge

23.1.1 Auftrag 1

Um den ganzen *Workflow* zu entwickeln haben wir es bisher einfach gehalten. Wir arbeiteten mit nur 10 Zeckenstichen und 5 Iterationen („Runs“).

- Führe die ganze Analyse mit dem ganzen Datensatz `zeckenstiche_full.csv` (1'076 Meldungen) durch den du [hier herunterladen](#) kannst (Rechtsklick > Speichern Unter)
- Lasse die Simulation 50-mal statt nur 5-mal laufen

23.1.2 Auftrag 2

Wir simulieren die Zeckenstiche, indem wir die Original-Meldungen mit der Funktion `offset_coordinate` um einen Zufallswert verschieben (siehe [Übung 7.5](#)). Dieser Zufallswert stammt aus einer Verteilung mit Mittelwert 0 und Standardabweichung 100. Variiere in deiner Analyse die Standardabweichung und nutze nebst 100 zwei andere Werte, Beispielsweise 10 und 1'000. Schau dir [Übung 8.4](#) nochmals an wenn du nicht mehr weisst, wie das geht.

Tipp: Für Profis

Es ist natürlich nicht nötig, den *For Loop* 3x zu kopieren. Man kann auch einen verschachtelten *For Loop* konstruieren, wo der äussere *Loop* über die Radien iteriert und der innere *Loop* der bisherige *Loop* darstellt. Wichtig dabei ist, dass die Liste `monte_carlo` vor dem äusseren Loop konstruiert wird und im DataFrame jeweils gespeichert wird, was für ein Radius verwendet wurde. Diese Spalte kann danach in `groupby` (siehe unten) mitverwendet werden.

```
results = monte_carlo_join2.groupby(["Radius", "Run", "Wald_text"]).size().unstack(fill_
↪value = 0).assign(ja_anteil = lambda x: x.ja/(x.ja+x.nein))

results.boxplot(column = "ja_anteil", by = "Radius", color = "blue")
```

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 + M:** 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 than one line then you will need to first select all the line and then use this shortcut.