

---

# **Coding in GIS**

**Nils Ratnaweera**

**Sep 14, 2020**



# CODING IN GIS I

<b>1</b>	<b>Einleitung zu diesem Block</b>	<b>3</b>
<b>2</b>	<b>Primitive Datentypen</b>	<b>5</b>
2.1	Boolean . . . . .	5
2.2	String . . . . .	5
2.3	Integer . . . . .	6
2.4	Float . . . . .	6
2.5	Übung 1: Variablen erstellen . . . . .	6
2.6	Übung 2: String verbinden . . . . .	6
2.7	Übung 3: Numerische Variablen 1 . . . . .	7
2.8	Übung 3: Numerische Variablen 2 . . . . .	7
2.9	Übung 4: Boolesche Variablen . . . . .	7
<b>3</b>	<b>Zusammengesetzte Datentypen</b>	<b>9</b>
3.1	List . . . . .	9
3.2	Dict . . . . .	10
3.3	DataFrame . . . . .	10
3.4	Weitere Strukturen . . . . .	11
3.5	Übung 1: Lists . . . . .	12
3.6	Übung 2: Elemente aus Liste ansprechen . . . . .	12
3.7	Übung 3: Liste ergänzen . . . . .	12
<b>4</b>	<b>Dictionaries</b>	<b>13</b>
4.1	Übung 1: Dictionary erstellen . . . . .	14
4.2	Übung 2: Elemente aus Dictionary ansprechen . . . . .	14
4.3	Übung 3: Key ergänzen . . . . .	14
4.4	Übung 4: Mehrere <i>Values</i> pro <i>Key</i> . . . . .	14
4.5	Übung 5: Dictionary mit List . . . . .	15
<b>5</b>	<b>Tabellarische Daten</b>	<b>17</b>
5.1	Übung 1: von einer <i>Dictionary</i> zu einer <i>DataFrame</i> . . . . .	17
5.2	Übung 2: <i>DataFrame</i> in csv umwandeln . . . . .	17
5.3	Übung 3: CSV als <i>DataFrame</i> importieren . . . . .	17
5.4	Übung 4: Koordinaten räumlich darstellen . . . . .	18
5.5	Übung 5: Einzelne Spalte selektieren . . . . .	18
5.6	Übung 6: Neue Spalte erstellen . . . . .	19
<b>6</b>	<b>Einleitung zu diesem Block</b>	<b>21</b>
<b>7</b>	<b>Virtual Environments mit Conda</b>	<b>23</b>

<b>8</b>	<b>Python Modules</b>	<b>25</b>
8.1	Erweiterung installieren . . . . .	25
8.2	Erweiterung laden . . . . .	25
8.3	Erweiterung verwenden . . . . .	26
8.4	Modul mit Alias importieren . . . . .	26
8.5	Einzelne <i>Function</i> importieren . . . . .	26
8.6	Alle <i>Functions</i> importieren . . . . .	27
<b>9</b>	<b>Function Basics</b>	<b>29</b>
9.1	Übung 1: Erste <i>Function</i> erstellen . . . . .	30
9.2	Übung 2: <i>Function</i> erweitern . . . . .	30
9.3	Übung 3: Default-Werte festlegen . . . . .	30
<b>10</b>	<b>Function Advanced</b>	<b>31</b>
10.1	Standart-Werten . . . . .	31
10.2	Reihenfolge der Argumente . . . . .	31
10.3	Globale und Lokale Variablen . . . . .	32
<b>11</b>	<b>Einen Einzelpunkt zufällig verschieben</b>	<b>33</b>
11.1	Übung 1: Zufallswerte generieren . . . . .	34
11.2	Übung 2: Zufallswerte addieren . . . . .	34
11.3	Übung 3: Arbeitsschritte in eine <i>Function</i> verwandeln . . . . .	34
11.4	Übung 4: Output visualisieren . . . . .	34
<b>12</b>	<b>Übung Punkte einer <i>DataFrame</i> zufällig verschieben</b>	<b>37</b>
12.1	Übung 1: Alle Zeckenstiche zufällig verschieben . . . . .	37
12.2	Übung 2: Neue <i>GeoDataFrame</i> mit simulierten Punkten erstellen . . . . .	38
12.3	Übung 4: Mehrere <i>DataFrames</i> visualisieren . . . . .	38
12.4	Übung 5: Genauigkeitsangaben der Punkte mitberücksichtigen. . . . .	38
<b>13</b>	<b>Einleitung zu diesem Block</b>	<b>41</b>
<b>14</b>	<b>Übung: For Loops (Teil I)</b>	<b>43</b>
14.1	Übung 1: Erste For-Loop erstellen . . . . .	43
14.2	Übung 2: For-Loop mit <code>range()</code> . . . . .	43
<b>15</b>	<b>Übung: For Loops (Teil II)</b>	<b>45</b>
15.1	Übung: Output aus For-Loop speichern . . . . .	46
<b>16</b>	<b>Übung: Zeckenstich Simulation mit Loop</b>	<b>47</b>
16.1	Übung 1: Mit For-Loop <code>zeckenstiche</code> mehrfach verschieben . . . . .	48
16.2	Übung 2: <i>DataFrames</i> aus Simulation zusammenführen . . . . .	48
16.3	Übung 3: Simulierte Daten visualisieren . . . . .	49
<b>17</b>	<b>Übung: GIS in Python</b>	<b>51</b>
17.1	Übung 1: <i>DataFrame</i> zu <i>GeoDataFrame</i> . . . . .	52
17.2	Übung 2: Koordinatensystem festlegen . . . . .	53
17.3	Übung 3: Zeckenstiche als Shapefile exportieren . . . . .	54
17.4	Übung 4 (Optional): Export als Geopackage . . . . .	54
<b>18</b>	<b>Übung: Waldanteil berechnen</b>	<b>55</b>
18.1	Übung 1: Wald oder nicht Wald? . . . . .	57
18.2	Übung 2: Anteil der Punkte pro “Gruppe” . . . . .	57
18.3	Übung 3: Anteil <i>im Wald</i> pro Run ermitteln . . . . .	58
18.4	Übung 3: Mittelwerte Visualisieren . . . . .	59

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

Das github-repo ist hier [ratnanil/codingingis](https://github.com/ratnanil/codingingis).

---

**Dieses Buch ist sowohl online als auch pdf version verfügbar:**

- online: <https://ratnanil.github.io/codingingis>
  - pdf: <https://github.com/ratnanil/codingingis/raw/master/codingingis.pdf>
- 

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



## 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. Diesen findet ihr hier:

[jupyterhub01.zhaw.ch](https://jupyterhub01.zhaw.ch)

Ihr könnt euch mit eurem ZHAW Kürzel (ohne “@students.zhaw.ch”) und Passwort einloggen.

---

### 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
-





## PRIMITIVE DATENTYPEN

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
Boolean	Logical	ja / nein
String	Character	Textinformation
Integer	Integer	Zahl ohne Nachkommastelle
Float	Double	Zahl mit Nachkommastelle

### 2.1 Boolean

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

```
regen = True
```

```
sonne = False
```

```
type(sonne)
```

```
bool
```

### 2.2 String

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

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

```
type(stadt)
```

```
str
```

Strings können mit + miteinander verbunden werden

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

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

## 2.3 Integer

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

```
bern_einwohner = 133115
```

```
type(bern_einwohner)
```

```
int
```

## 2.4 Float

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

```
bern_temp = 22.5
```

```
type(bern_temp)
```

```
float
```

## 2.5 Übung 1: Variablen erstellen

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

## 2.6 Übung 2: String verbinden

“Klebe” die beiden Variablen zusammen.

## 2.7 Übung 3: Numerische Variablen 1

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

## 2.8 Übung 3: Numerische Variablen 2

Ermittle deine Grösse in Meter auf der Basis von `groesse`. Was ist das für ein Datentyp?

## 2.9 Übung 4: Boolsche Variablen

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



## ZUSAMMENGESETZTE 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
List	Vector	Mehrere Werte, Abrufen über Zahlen
Dict	List	Mehrere Werte, Abrufen über Schlüsselwörter
DataFrame	Dataframe	Tabellarische Daten

### 3.1 List

Eine List:

- speichert die Reihenfolge, in der die Werte eingegeben werden
- kann unterschiedliche Datentypen enthalten
- wird mit eckigen Klammern ( [ und ] ) erstellt

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

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

```
hexerei[0]
```

```
3
```

```
type(hexerei)
```

```
list
```

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

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

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

```
int
```

```
# Der Inhalt vom dritten Wert ist vom Typ "List"
type(chaos[2])
```

```
list
```

## 3.2 Dict

Eine Dict:

- ist kurz für *Dictionary*
- speichert die Reihenfolge, in der die Werte eingegeben werden **nicht**
- kann unterschiedliche Datentypen enthalten
- wird mit geschweiften Klammern ({ und }) erstellt
- speichert für jeden Wert ("Value") einen Schlüssel ("Key"), mit dem er abgerufen werden kann: dict\_name = {"Key" : "Value"}

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

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

```
'Suppe'
```

Auch hier sind verschiedene Datentypen möglich:

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

```
type(menue)
```

```
dict
```

## 3.3 DataFrame

Bei den bisherigen Datentypen handelte es sich um Strukturen, die in der Standardinstallation von Python enthalten sind. Tabellarische Strukturen sind in Python, im Gegensatz zu R, nicht standardmäßig vorhanden. Dazu brauchen wir eine Erweiterung zu Python: Was es sich damit auf sich hat und wie diese installiert wird erfahren wir später. An dieser Stelle möchte ich nur die Struktur *DataFrame* vorstellen. *DataFrames* sind tabellarische Daten. Man kann sie auch als "Spezialfall" einer *Dict* betrachten, wo alle Werte (also *Keys*) die gleiche Länge haben.

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

```
type(menue2)
```

```
dict
```

Beachte, dass nun aus den *Key* Spaltennamen werden:

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

```
menue_df = pd.DataFrame(menue2)
menue_df
```

	Typ	Beschreibung	Preis
0	Vorspeise	Suppe	7.5
1	Hauptspeise	Gratin mit Spinat	32
2	Dessert	Himbeerglacé	10.50

```
type(menue_df)
```

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

## 3.4 Weitere Strukturen

In Python gibt es noch weitere zusammengesetzte Datentypen wie *Tuples* und *Sets*. Diese spielen in unserem Kurs aber eine untergeordnete Rolle. Ich erwähne sie an dieser Stelle zwei häufig gennante 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 `(( und ))` erstellt (`mytuple = (2, 2, 1)`).
- *Sets*
  - sind ähnlich wie *Dicts*, verfügen nicht über `keys` und `values`
  - jeder Wert wird nur 1x gespeichert (Duplikate werden automatisch gelöscht)
  - Sie werden mit geschweiften Klammern `{ und }` erstellt.

## 3.5 Übung 1: Lists

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

## 3.6 Übung 2: Elemente aus Liste ansprechen

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

## 3.7 Übung 3: Liste ergänzen

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

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



## DICTIONARIES

In den letzten Übungen haben wir die wichtigsten Datentypen kennengelernt. Nun wollen wir einen besonderen Fokus auf den Datentyp *Dictionary* legen.

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



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'
```

### 4.1 Übung 1: Dictionary erstellen

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

### 4.2 Übung 2: Elemente aus Dictionary ansprechen

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

```
me["vorname"]
```

```
'Guido'
```

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

```
'van Rossum'
```

### 4.3 Übung 3: Key ergänzen

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

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

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

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

### 4.4 Übung 4: Mehrere *Values* pro *Key*

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
```

## 4.5 Übung 5: Dictionary mit List

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



## TABELLARISCHE DATEN

Schauen wir uns nochmals die *Dictionary* `people` an 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]}
```

### 5.1 Übung 1: von einer *Dictionary* zu einer *DataFrame*

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

### 5.2 Übung 2: *DataFrame* in csv umwandeln

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

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

### 5.3 Übung 3: CSV als *DataFrame* importieren

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

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

---

#### Achtung!

- Wenn du auf dem JupyterHub Server arbeitest dann ist dein Arbeitsverzeichnis ebenfalls *auf dem Server*. Das heisst, du musst "zeckenstiche.csv" auf den Server hochladen. Dies kannst du mit dem Button "Upload Files" im Tab "File Browser" bewerkstelligen.
- der obige Code funktioniert nur, wenn "zeckenstiche.csv" im aktuellen Arbeitsverzeichnis (*Current Working Directory*) abgespeichert ist. Wenn du nicht sicher bist, wo dein aktuelles Arbeitsverzeichnis liegt, kannst du dies mit der Funktion `os.getcwd()` (get **c**urrent**w**orking**d**irectory) herausfinden (s.u.).

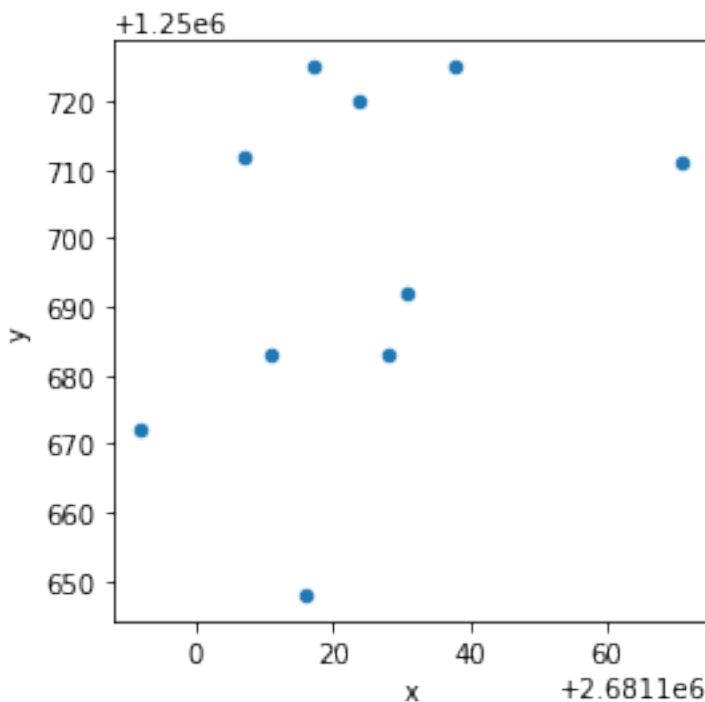
```
import os
os.getcwd()
```

## 5.4 Übung 4: Koordinaten räumlich darstellen

Die *DataFrame* *zeckenstiche* beinhaltet x und y Koordinaten für jeden Unfall in den gleichnamigen Spalten. Wir können die Stiche mit einem Scatterplot räumlich visualisieren. Führe dazu folgenden Code aus. Überlege dir, was die zweite Zeile bewirkt und warum dies sinnvoll ist.

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

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



## 5.5 Übung 5: Einzelne Spalte selektieren

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

## 5.6 Übung 6: Neue Spalte erstellen

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

```
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 gehabt. Um den Einstieg möglichst einfach zu halten habt ihr bisher auf dem ZHAW Jupyter-Lab Server (JupyterHub) gearbeitet. Diese Woche wollen wir versuchen, dass ihr lokal auf euren eigenen Rechnern arbeitet. So lernt ein Programm kennen, welches sich ausschliesslich aus der Commandozeile bedienen lässt. Ihr lernt zudem, wie man durch das setzen der Systemvariabel ein solches Programm in der Kommandozeile verfügbar macht.

Wenn dieser Schritt erfolgt ist lernt ihr Python Module besser kennen und wie sich deren Handhabung von R unterscheidet. Später schauen wir uns Funktionen genauer an: Was sie brauchen, wie man sie erstellt und vor allem wie man sie nutzt.

---

### **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.
-



## VIRTUAL ENVIRONMENTS MIT CONDA

Dafür müsst ihr das Konzept von “Virtual Environments”, welches in der Vorlesung erläutert wurde, (halbwegs) verstanden haben. Sollte dies nicht der Fall sein, meldest du dich am besten beim Dozenten oder schaust dir nochmals die Vorlesungsfolien an.

### Schritt 1: Conda installieren

Normalerweise wäre nun der erste Schritt, *conda* herunterzuladen und installieren (ich empfehle [miniconda](#), nicht [anaconda](#)). Wenn ihr aber ArcGIS Pro installiert habt, ist dieser Schritt nicht nötig.

### Schritt 2: Systemvariable setzen

Wir wollen das Programm *conda* von der Kommandozeile aus (cmd) bedienen. Dafür müssen wir evt. die Systemvariable zuerst noch setzen. Um zu Prüfen ob dies nötig ist, kannst du die Konsole starten (Windowstaste > cmd) und folgende Zeile einfügen

```
conda --version
```

Wenn ihr eine Versionsnummer könnt ihr zum nächsten Schritt übergehen. Ansonsten müsst ihr die Systemvariable *PATH* zuerst noch setzen. Siehe dazu die Vorlesungsfolie. Überprüfe mit *conda --version* ob das setzen der Systemvariable erfolgreich war.

### Schritt 3: Virtual Environment erstellen und aktivieren

Nun kannst du mit *conda* eine neue Virtual Environment erstellen. Gibt dazu folgenden Befehl in die Konsole ein:

```
conda create --name codingingis
```

Die Frage Proceed ([y]/n) kannst du mit *y* bestätigen. Anschliessend kannst du mit dem Befehl *activate codingingis* die neu erstellte Environment “einschalten”.

### Schritt 4: Module installieren

Nun kannst du die benötigten Module installieren. Beispielsweise ist Jupyter Labs ebenfalls ein Python Modul, welches sich über *conda* installieren lässt. Dazu braucht es folgenden Befehl:

```
conda install -c conda-forge jupyterlab
```

Ersetze nun *jupyterlab* mit dem namen der anderen Module, die du für den Unterricht noch brauchst. Starte mal mit folgenden *pandas*, *matplotlib*, *geopandas*, *descartes*, fehlende Module kannst du später immer noch nach installieren.

Nun kannst du Jupyter Lab mit folgendem Befehl starten

```
jupyter lab
```



## PYTHON MODULES

Ähnlich wie R basiert Python auf Erweiterungen: Diese Erweiterungen heissen in R *Libraries* oder *Packages*, in Python werden sie *Modules* genannt. Sie sind dazu da, gewisse Teilbereiche unseres Arbeitsprozesses zu vereinfachen. Eine Analogie dazu: Um ein Haus zu bauen sind wir auf verschiedene Spezialisten / Spezialistinnen angewiesen: Wir brauchen zum Beispiel eine Malerin oder einen Maler. Im Telefonbuch sind seitenweise Maler\*innen aufgelistet, und jede\*r arbeitet etwas anders. Um eine spezifische Malerin anzuheuern müssen wir zuerst den Kontakt herstellen und die Vertragsmodalitäten vereinbaren. Erst dann können wir sie in unseren Arbeitsprozess (Haus bauen) einbinden. Um diese Analogie auf unser Projekt zu übertragen: Das “Haus bauen” ist unser Forschungsprojekt (z.B. eine Bachelorarbeit). Ein “Telefonbuch”, wo die Spezialisten erfasst sind nennt ein *Repository*.

### 8.1 Erweiterung installieren

Den Erstkontakt mit der Malerin zu erstellen und die Vertragsmodalitäten zu vereinbaren bedeutet, die Erweiterung zu installieren. In R ist die Installation einer *Library* ein R-Befehl und wird innerhalb von R ausgeführt. In Python ist dies leider etwas komplizierter, es braucht für die Installation einer Python library eine Zusatzsoftware. Diese lernen wir zu einem späteren Zeitpunkt kennen, schauen wir uns jetzt nur mal den R-Code zur Installation eines Packages an:

in R:

```
install.packages("malerin")
```

In Python\*:

```
conda install -c conda-forge malerin
```

\* in Python gibt es noch andere Wege, Erweiterungen zu installieren (z.B. mit `pip`). Wir lassen sie der Einfachheit an dieser Stelle aber weg.

In diesem Beispiel heisst die Erweiterung `malerin`. Das *Repository* geben wir in R meistens nicht an, weil in RStudio bereits eine Default-Adresse hinterlegt ist.

### 8.2 Erweiterung laden

Treiben wir an dieser Stelle die Analogie etwas weiter: Der Erstkontakt mit der Malerin ist also erstellt und alle Vertragsmodalitäten sind vereinbart. Nun wollen wir an einem bestimmten Tag mit ihr arbeiten. Dafür müssen wir sie zuerst auf die Baustelle bestellen. Übersetzt auf Programmieren bedeutet dies, wir müssen die Erweiterung in unsere Session laden. In R und Python sehen die Befehle folgendermassen aus:

in R:

```
library(malerin)
```

in Python:

```
import malerin
```

### 8.3 Erweiterung verwenden

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

in R:

```
wand_bemalen()
```

in Python:

```
malerin.wand_bemalen()
```

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

### 8.4 Modul mit Alias importieren

Da es umständlich sein kann, jedesmal `malerin.wand_bemalen()` voll auszuschreiben, können wir beim Importieren dem Modul auch einen “Alias” vergeben. Dies kann beispielsweise folgendermassen aussehen:

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

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

### 8.5 Einzelne *Function* importieren

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

```
from malerin import wand_bemalen  
wand_bemalen()
```

## 8.6 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 und ist hier nur erwähnt, falls ihr diese Schreibweise mal antrifft.

```
from malerin import *  
wand_bemalen()
```

---

### Das wichtigste in Kürze

- Erweiterungen heissen in Python *Modules*
  - Vor der erstmaligen Nutzung muss ein *Module* installiert werden
  - Vor der Verwendung in einer Session muss ein *Module* importiert werden
  - Ein *Module* wird in Python mit `import modulename` geladen
  - Module müssen für **jede Session** importiert werden!
  - Eine *Function* aus einem *Module* wird folgendermassen aufgerufen: `modulname.function()`
  - Ein Modul kann auch mit einem Alias importiert werden `import modul as modulalias`
  - Eine Einzelne *Function* kann aus einem Modul explizit importiert werden: `from malerin import wand_bemalen`
  - Mit einem Platzhaltersymbol `*` können auch alle *Functions* aus einem Modul importiert werden: `from malerin import *`
-





## FUNCTION BASICS

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 muss man Fehler nur an einer Stelle beheben. Deshalb: 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 einen Parameter übergeben. In folgendem Beispiel ist `vorname` ein Parameter, “Guido” ist sein Argument.

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

```
'Hallo Guido!'
```

Den Output können wir wie gewohnt einer neuen Variabel zuweisen:

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

```
'Hallo Guido!'
```

## 9.1 Übung 1: Erste *Function* erstellen

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

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

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

## 9.2 Übung 2: *Function* erweitern

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

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

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

## 9.3 Übung 3: Default-Werte festlegen

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

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

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

## FUNCTION ADVANCED

### 10.1 Standart-Werten

Wenn Parameter Default-Argumente zugewiesen werden, dann werden sie für den Nutzer automatisch zu optionalen Parametern. Wenn eine neue *Function* erstellt wird, kommen die Optionalen Parameter immer am Schluss. Also so:

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

Wenn wir versuchen die optionalen Parameter an den Anfang zu stellen, meldet sich Python mit einem Error:

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

**Error:**

```
File "<ipython-input-12-b48869b8c585>", line 1
    def gruezi(anrede = "Herr oder Frau", nachname):
        ^
SyntaxError: non-default argument follows default argument
```

### 10.2 Reihenfolge der Argumente

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

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

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

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

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

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

Aber wenn die Parameter der Argumente spezifiziert werden, ist die Reihenfolge wiederum egal:

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

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

### 10.3 Globale und Lokale Variablen

Innerhalb einer *Function* können nur die Variablen verwendet werden, die der *Function* als Argumente übergeben werden (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. Folgende Vorgehensweise erzeugt also eine Fehlermeldung:

```
# Wir definieren globale Variablen
nachname = "van Rossum"
anrede = "Herr"

def gruezi(anrede = "Herr oder Frau", nachname):
    # nun versuchen wir in der Funktion auf diese
    # globalen Variablen zu zu greifen
    return("Guten Tag, "+anrede+" "+nachname)

gruezi()
```

Dies erzeugt jedoch eine Fehlermeldung:

#### Error:

```
-----
TypeError                                Traceback (most recent call last)
<ipython-input-16-fc94d019a607> in <module>()
      2 anrede = "Herr"
      3
----> 4 gruezi()

TypeError: gruezi() missing 1 required positional argument: 'nachname'
```

## EINEN EINZELPUNKT ZUFÄLLIG VERSCHIEBEN

Im Hinblick auf die kommende Woche entwickeln wir in dieser Übung eine *Function*, welches x/y Koordinaten zufällig in einem definierten Umkreis verschiebt.

Um diese Function zu entwickeln nehmen wir die Koordinaten der alten Sternwarte Bern beziehungsweise deren Gedenktafel (s.u.).

```
x = 2600000  
y = 1200000  
accuracy = 60
```



Fig. 11.1: Links: Koordinatensystem der Schweiz, mit dem (alten) Referenzwert Sternwarte Bern. Quelle: lv95.bve.be.ch rechts: Gedenktafel an die Alte Sternwarte Bern. Quelle: aiub.unibe.ch

Das Ziel ist es also, dass wir eine *Function* haben die uns einen Zufälligen Punkt in der Nähe der alten Sternwarte vorschlägt.

## 11.1 Übung 1: Zufallswerte generieren

Es sind verschiedene Methoden denkbar, wie wir ein neues  $x,y$  Koordinatenpaar aus den bestehenden Koordinaten bestimmen. Die einfachste Variante ist es wohl, zu jedem Achsenwert ( $x/y$ ) einzeln einen Zufallswert zu addieren, z.B. von -100 bis +100.

Wie generiert man aber Zufallszahlen in Python? Versuche dies selbst mittels deiner Lieblingssuchmaschine herauszufinden.

## 11.2 Übung 2: Zufallswerte addieren

Addiere nun die Zufallszahlen zu den Dummy-Werten um `x_neu` und `y_neu` zu erhalten.

```
print(x_neu, y_neu)
```

```
2600035.897171546 1200122.8831768164
```

## 11.3 Übung 3: Arbeitsschritte in eine *Function* verwandeln

Jetzt sind die Einzelschritte zur Verschiebung eines Punktes klar. Da wir dies für viele Punkte machen müssen, ist es sinnvoll, aus den Arbeitsschritten eine *Function* zu erstellen. Erstelle eine Function `point_offset` welche als Input eine  $x$  oder  $y$  Koordinate annimmt und eine leicht verschobene Koordinate zurück gibt. Wenn du möchtest kannst du die Distanz der Verschiebung als optionalen Parameter (mit *Default* = 100) definieren.

## 11.4 Übung 4: Output visualisieren

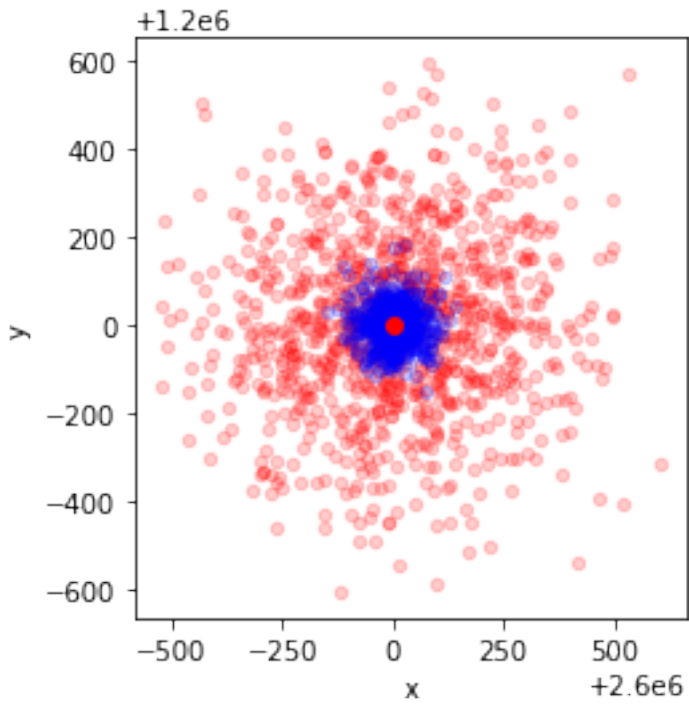
Nun ist es wichtig, dass wir unser Resultat visuell überprüfen. Im Beispiel unten wende ich die *Function* `offset_coordinate()` 1000x auf die Koordinaten der alten Sternwarte an. Für diesen Schritt gebe ich euch den fertigen Code, da ihr die dafür benötigten Techniken noch nicht gelernt habt. Füge diesen Code in dein Script ein und führe ihn aus. Allenfalls musst du den Code leicht an deine Situation anpassen.

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

fig = pd.DataFrame({"x": [offset_coordinate(x, 200) for rand in range(1,1000)], "y": [
    offset_coordinate(y, 200) for rand in range(1,1000)]}).plot.scatter("x", "y", color = "red",
    alpha = 0.2)

pd.DataFrame({"x": [offset_coordinate(x, 50) for rand in range(1,500)], "y": [offset_
    coordinate(y, 50) for rand in range(1,500)]}).plot.scatter("x", "y", color = "blue",
    alpha = 0.2, ax = fig)

plt.scatter(x,y, c = "red")
fig.set_aspect("equal", "box")
plt.show()
```







## ÜBUNG PUNKTE EINER *DATAFRAME* ZUFÄLLIG VERSCHIEBEN

Nun wollen wir unsere Funktion `offset_coordinate()` auf alle Zeckenstich-Koordinaten anwenden. Bildlich gesprochen: Wir nehmen unsere Zeckenstichdatensatz und schütteln ihn einmal durch.

Nutze hier die Datei “zeckenstiche.csv” von letzter Woche (du kannst auch sie [hier erneut runterladen](#), Rechtsklick → Ziel speichern unter).

```
import pandas as pd
import random

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

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

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

### 12.1 Übung 1: Alle Zeckenstiche zufällig verschieben

Nun können wir die Funktion `offset_coordinate()` auf die gesamte Spalten `x` und `y` der *DataFrame* zeckenstiche anwenden. Nutze eckige Klammern um die entsprechende Spalte zu wählen.

## 12.2 Übung 2: Neue GeoDataFrame mit simulierten Punkten erstellen

Um die neuen, verschobenen Koordinaten abzuspeichern erstellen wir zuerst eine neue, leere *DataFrame* (z.B. `zeckenstiche_sim`) und fügen den Output als neue Spalten dieser *DataFrame* hinzu. Auch die "ID" könnt ihr als neue Spalte hinzufügen, so behalten wir den Bezug zum ursprünglichen Datensatz.

## 12.3 Übung 4: Mehrere *DataFrames* visualisieren

Um zwei *DataFrames* 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.

## 12.4 Übung 5: Genauigkeitsangaben der Punkte mitberücksichtigen.

Bisher haben wir alle Punkte um die gleiche Distanz verschoben. Wenn wir unsere *DataFrame* "zeckensiche" genau anschauen, steht uns eine Genauigkeitsangabe pro Punkt zur Verfügung: Die Spalte "accuracy".

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

Diese Spalte sagt was darüber aus, wie sicher der/die Nutzer\*in bei der Standortsangabe war. Wir können diese Angabe auch nutzen um den offset *pro Punkt* festzulegen.

```
zeckenstiche_sim["x"] = offset_coordinate(zeckenstiche["x"], distance = zeckenstiche[
↪ "accuracy"])

zeckenstiche_sim["y"] = offset_coordinate(zeckenstiche["y"], distance = zeckenstiche[
↪ "accuracy"])

zeckenstiche_sim
```

	ID	x	y
0	2550	2.680691e+06	1.250421e+06
1	10437	2.680800e+06	1.250516e+06
2	9174	2.680836e+06	1.250527e+06
3	8773	2.680819e+06	1.250527e+06
4	2764	2.680839e+06	1.250536e+06
5	2513	2.680879e+06	1.250555e+06

(continues on next page)

(continued from previous page)

6	9185	2.680815e+06	1.250556e+06
7	28521	2.680832e+06	1.250564e+06
8	26745	2.680825e+06	1.250569e+06
9	27391	2.680846e+06	1.250569e+06



## EINLEITUNG ZU DIESEM BLOCK

Heute wollen wir uns weiter mit den Zeckenstichdaten befassen. Wir werden im Wesentlichen ein Teil der Übung aus “Datenqualität und Unsicherheit” in Python rekonstruieren.

In der Übung geht es um folgendes: Wir wissen das die Lagegenauigkeit der Zeckenstichmeldungen mit einer gewissen Unsicherheit behaftet sind. Um die Frage “Welcher Anteil der der Zeckenstiche befinden sich im Wald?” unter Berücksichtigung dieser Unsicherheit beantworten zu können, führen wir eine (Monte Carlo) Simulation durch. In dieser Simulation verändern wir die Position der Zeckenstichmeldungen zufällig und berechnen den Anteil der Zeckenstiche im Wald. Das zufällige Verschieben und berechnen wiederholen wir beliebig lange und bekommen für jede Wiederholung einen leicht unterschiedlichen Prozentwert. Die Verteilung dieser Prozentwerte ist die Antwort auf die ursprüngliche Frage (“Welcher Anteil...”) unter Berücksichtigung der Unsicherheit.

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

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

---

### 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
-



## ÜBUNG: FOR LOOPS (TEIL I)

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

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

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

- `for` legt fest, dass eine For-Loop beginnt
- Nach `for` kommt eine Platzhalter-Variabel, die ihr beliebig benennen könnt. Im obigen Beispiel lautet diese `platzhalter`
- Nach dem Platzhalter kommt der Begriff `in`
- 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” (dies erinnert an die Konstruktion 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.

### 14.1 Übung 1: Erste For-Loop erstellen

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

### 14.2 Übung 2: For-Loop mit `range()`

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





## ÜBUNG: FOR LOOPS (TEIL II)

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

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

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

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

```
liedzeile
```

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

```
schlagwort
```

```
'saint'
```

Das verrät uns etwas über die Funktionsweise des *For-Loops*: Bei jedem Durchgang werden die Variablen immer wieder überschrieben. Wenn wir also den Output des ganzen For-Loops abspeichern wollen, müssen wir dies etwas vorbereiten. Dafür erstellen wir unmittelbar 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 hinzu zu fügen.

```
strophe = []

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

strophe
```

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

### 15.1 Übung: Output aus For-Loop speichern

Erstelle einen *For-Loop*, wo in jeder Iteration einen Output in einer Liste gespeichert wird.

## ÜBUNG: ZECKENSTICH SIMULATION MIT LOOP

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

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

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

```
import pandas as pd
import random

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

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



**Gregory R. Hancock**  
@GregoryRHancock

If someone does a Monte Carlo simulation study using Python, can we call it a “Monte Python” study?  
Please?

2:25 PM · Oct 3, 2019 · [Twitter for iPhone](#)

**79** Retweets   **14** Quote Tweets   **710** Likes

Fig. 16.1: Quelle [twitter.com](#)

## 16.1 Übung 1: Mit For-Loop zeckenstiche mehrfach verschieben

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

```
zeckenstiche_sim = pd.DataFrame()

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

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

	ID	x	y
0	2550	2.681065e+06	1.250530e+06
1	10437	2.681276e+06	1.250637e+06
2	9174	2.681193e+06	1.250783e+06
3	8773	2.681000e+06	1.250637e+06
4	2764	2.681140e+06	1.250633e+06
5	2513	2.681163e+06	1.250824e+06
6	9185	2.681241e+06	1.250691e+06
7	28521	2.681120e+06	1.250881e+06
8	26745	2.681096e+06	1.250827e+06
9	27391	2.681176e+06	1.250775e+06

Kombiniere dies nun mit deinem Wissen über Loops, um die Punkte der *DataFrame* nicht einmal, sondern 5 mal zu verschieben. Dazu brauchst du vor dem Loop eine leere Liste (z.B. `monte_carlo = []`) damit du den Output aus jedem Loop mit `append()` abspeichern kannst. Erstelle auch eine neue Spalte `Run_Nr` mit der Nummer der Durchführung (die du vom Platzhalter erhältst).

## 16.2 Übung 2: DataFrames aus Simulation zusammenführen

Schau dir die Outputs an.

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

```
type(zeckenstiche)
```

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

```
type(monte_carlo)
```

```
list
```

```
len(zeckenstiche)
```

```
10
```

```
len(monte_carlo)
```

```
5
```

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

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

## 16.3 Übung 3: Simulierte Daten visualisieren

Exploriere nun `monte_carlo_df`. Was ist es für ein Datentyp? Was hat es für Spalten? Visualisiere den Datensatz räumlich mit `monte_carlo_df.plot.scatter()`.



## ÜBUNG: GIS IN PYTHON

Bis jetzt haben wir noch nicht mit eigentlich Geodaten gearbeitet. Die  $x / y$  Werte der Zeckenstiche repräsentieren zwar Zeckenstiche in der Schweiz (sie sind also im Schweizer Koordinatensystem), dies ist aber nur uns bewusst (Python weiss davon nichts). Der Raumbezug fehlt noch, und den stellen wir an dieser Stelle her. Warum? Weil wir im nächsten Schritt unserer Todo Liste (s.u.) berechnen müssen, wie viele Zeckenstiche sich im Wald befinden. Das ist eine räumliche Abfrage, die sich ohne räumliche Objekte nicht bewerkstelligen lässt.

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

Um mit Geodaten in Python arbeiten zu können, müssen wir ein neues Modul importieren. Im Grunde genommen sind Vektordaten nicht mehr als Tabellen mit einer zusätzlichen “Geometrie”-Spalte. Dementsprechend müssen wir pandas nur ein bisschen erweitern um mit Vektordaten arbeiten zu können, und diese “Geo”-Erweiterung lautet: geopandas.

```
# Vorbereitung der Arbeitsumgebung
# (nur wenn ihr in einer neuen Session startet)

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

def offset_coordinate(old, distance = 1000):
    new = old + random.normalvariate(0,distance)
    return (new)

monte_carlo = []
for i in range(5):
    zeckenstiche_sim = pd.DataFrame()

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

    zeckenstiche_sim["x"] = zeckenstiche["x"].apply(offset_coordinate)
    zeckenstiche_sim["y"] = zeckenstiche["y"].apply(offset_coordinate)
    zeckenstiche_sim["Run_Nr"] = i
    monte_carlo.append(zeckenstiche_sim)

monte_carlo_df = pd.concat(monte_carlo)
```

## 17.1 Übung 1: *DataFrame* zu *GeoDataFrame*

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

```
import geopandas as gpd
```

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

```
/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
```

```
type(monte_carlo_df)
```

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

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

```
monte_carlo_df = gpd.GeoDataFrame(monte_carlo_df)
```

```
type(monte_carlo_df)
```

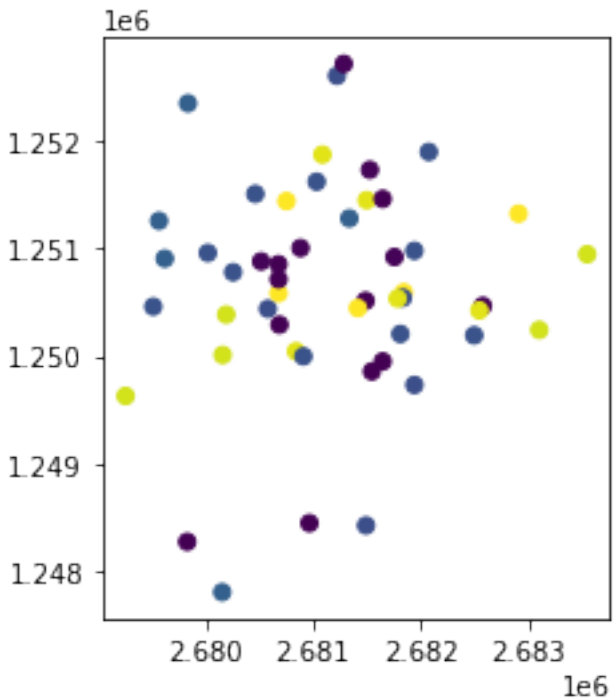
```
geopandas.geodataframe.GeoDataFrame
```

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

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

```
<AxesSubplot:>
```





## 17.2 Übung 2: Koordinatensystem festlegen

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

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

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

```
monte_carlo_df.crs
```

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

## 17.3 Übung 3: Zeckenstiche als Shapefile exportieren

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

## 17.4 Übung 4 (Optional): Export als Geopackage

Shapefiles sind ein ganz schreckliches Format (siehe [switchfromshapefile.org](https://switchfromshapefile.org)). Viel praktischer sind an dieser Stelle zum Beispiel *Geopackages*. Ihr könnt `monte_carlo_df` auch mit folgender Codezeile als *Geopackage* exportieren.

```
monte_carlo_df.to_file("monte_carlo_df.gpkg", layer = "monte_carlo_simulation", driver_  
↳= "GPKG")
```

## ÜBUNG: WALDANTEIL BERECHNEN

Nun sind wir so weit, dass wir 50 Simulation der Zeckenstiche mit zufällig verschobenen Punkten vorbereitet haben. Wir haben also die gleiche Ausgangslage, mit der ihr den Themenblock “Datenqualität und Unsicherheiten” gestartet habt. In der Todo-Liste sind wir nun bei Schritt 4:

- Schritt 1: Einen Einzelpunkt zufällig verschieben ✓
- Schritt 2: Alle Punkte einer DataFrame zufällig verschieben (1 “Run”) ✓
- Schritt 3: Alle Punkte einer DataFrame mehrfach zufällig verschieben (z.B. 50 “Runs”) ✓
- **Schritt 4: Anteil der Punkte im Wald pro “Run” ermitteln 1. Für jeden Simulierten Punkt zu bestimmen ob er innerhalb oder ausserhalb des Waldes liegt 2. Den Anteil der Punkte im Wald pro Simulation zu bestimmen**

Lade dafür den Datensatz “wald.gpkg” von Moodle herunter und verschiebe es in eure *Working directory*. Importiere “wald.gpkg” mit `pd.read_file()` und speichere es als Variable `wald`.

```
import pandas as pd
import geopandas as gpd

monte_carlo_df = gpd.read_file("monte_carlo_df.gpkg") # oder .shp, je nach dem wie_
↳ ihr es gespeichert habt

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

wald
```

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

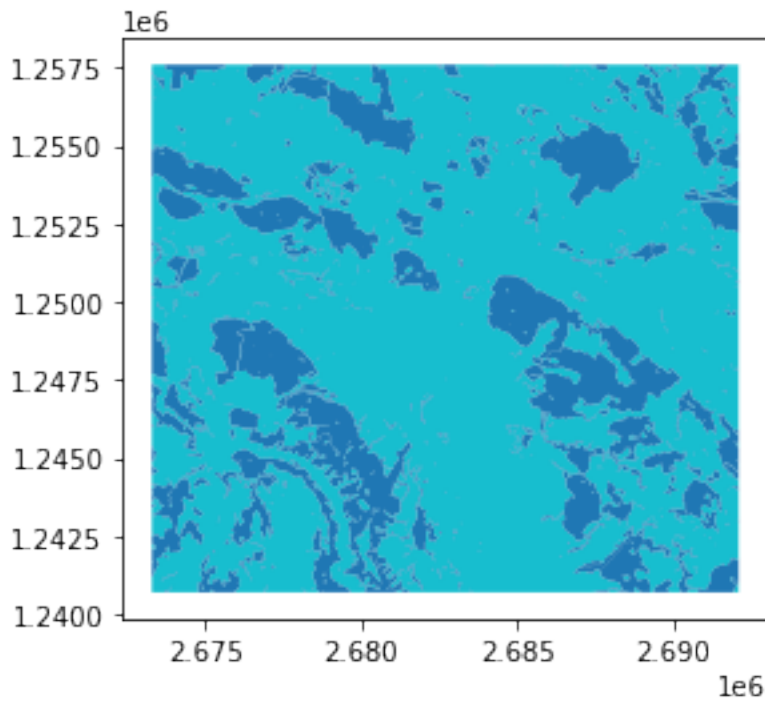
	Wald	Shape_Leng	Shape_Area	Wald_text	\
0	0	947316.853401	2.380876e+08	nein	
1	1	921225.341854	7.963237e+07	ja	

	geometry
0	MULTIPOLYGON Z (((2692100.000 1256542.253 276....
1	MULTIPOLYGON Z (((2689962.355 1245335.250 644....

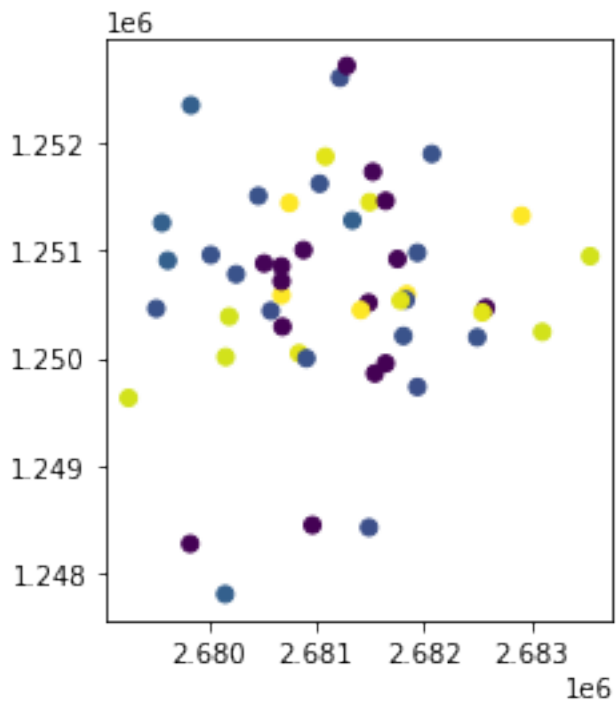
```
# hierfür braucht ihr das modul "descartes"
wald.plot(column = "Wald_text")
```

```
<AxesSubplot:>
```



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

```
<AxesSubplot:>
```



## 18.1 Übung 1: Wald oder nicht Wald?

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

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

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

```
monte_carlo_sjoin = gpd.sjoin(monte_carlo_df, wald)

monte_carlo_sjoin.head()
```

	ID	x	y	Run_Nr	geometry \
0	2550	2.681646e+06	1.249950e+06	0	POINT (2681646.093 1249949.520)
1	10437	2.679616e+06	1.250905e+06	0	POINT (2679615.976 1250905.268)
2	9174	2.680460e+06	1.251506e+06	0	POINT (2680459.591 1251505.910)
3	8773	2.680576e+06	1.250438e+06	0	POINT (2680576.211 1250438.321)
4	2764	2.680881e+06	1.251003e+06	0	POINT (2680880.858 1251003.114)

	index_right	Wald	Shape_Leng	Shape_Area	Wald_text
0	0	0	947316.853401	2.380876e+08	nein
1	0	0	947316.853401	2.380876e+08	nein
2	0	0	947316.853401	2.380876e+08	nein
3	0	0	947316.853401	2.380876e+08	nein
4	0	0	947316.853401	2.380876e+08	nein

## 18.2 Übung 2: Anteil der Punkte pro “Gruppe”

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

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

		Anzahl
Run_Nr	Wald_text	
0	ja	2
	nein	8
1	ja	1
	nein	9
2	ja	1
	nein	9
3	ja	1

(continues on next page)

(continued from previous page)

	nein	9
4	ja	3
	nein	7

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

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

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

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

anteile_pivot
```

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

## 18.3 Übung 3: Anteil *im Wald* pro Run ermitteln

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

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

anteil_ja
```

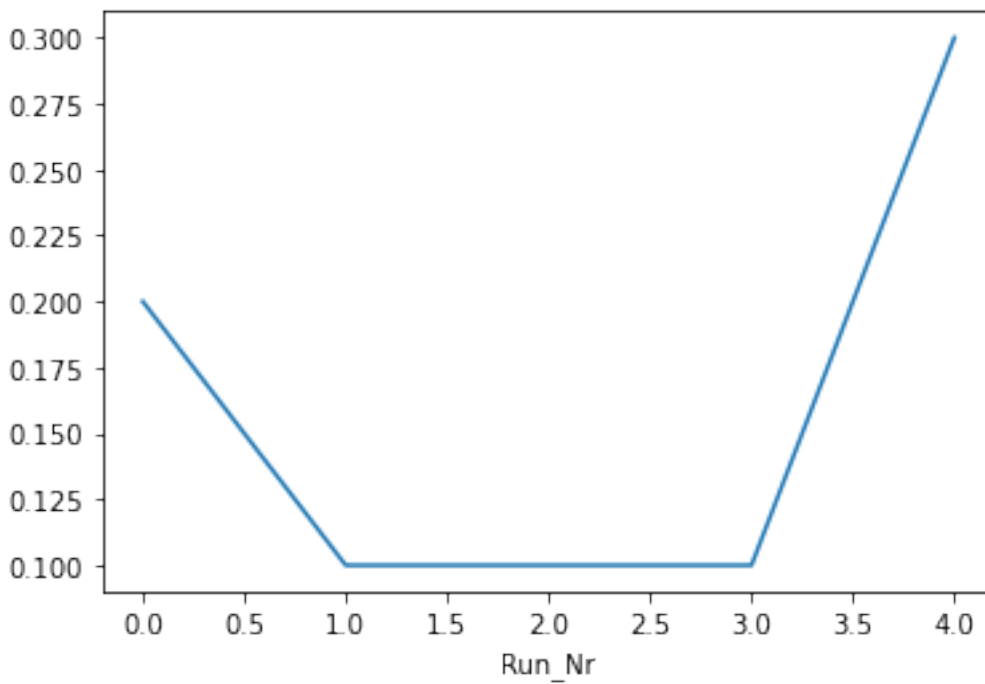
```
Run_Nr
0    0.2
1    0.1
2    0.1
3    0.1
4    0.3
dtype: float64
```

## 18.4 Übung 3: Mittelwerte Visualisieren

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

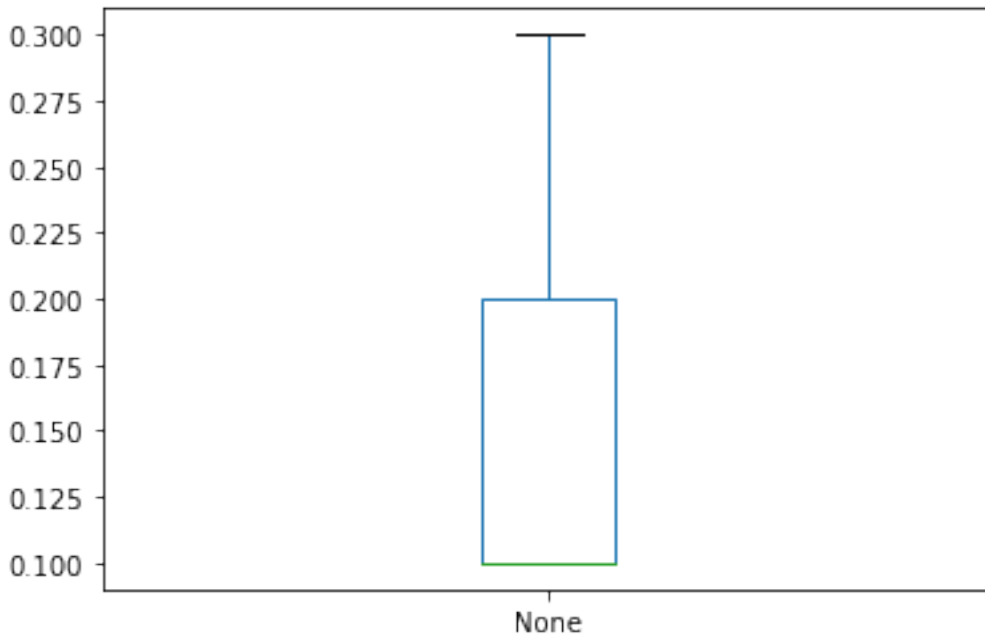
```
anteil_ja.plot()
```

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



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

```
<AxesSubplot:>
```



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

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

