Coding in GIS

Nils Ratnaweera

Sep 23, 2020

CODING IN GIS I

Dieser kurze Kurs ist Bestandteil des übergreifenden Moduls "Angewandte Geoinformatik" der Zürcher Hochschule für Angewandte Wissenschaften (ZHAW). Er soll einen Einstieg in die Programmierwelt von Python bieten und spezifisch zeigen wie man räumliche Fragestellungen mit frei verfügbarer Software lösen kann.

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

Dieses Buch auch als pdf version verfügbar

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

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

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

---

Noch ein paar Hinweise zur Handhabung dieses Dokumentes:

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

CHAPTER

ONE

EINLEITUNG ZU DIESEM BLOCK

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

jupyterhub01.zhaw.ch

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

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

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

Übungsziele

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

---

CHAPTER

TWO

AUFGABE 1: PRIMITIVE DATENTYPEN

2.1 Theorie

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

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

2.1.1 Boolean

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

```
regen = True # "es regnet"

sonne = False # "die Sonne scheint nicht"

type(sonne)
```

```
bool
```

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

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

```
True
```

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

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

```
False
```

2.1.2 String

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

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

type(stadt)
```

```
str
```

Strings können mit + miteinander verbunden werden

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

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

2.1.3 Integer

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

```
bern_einwohner = 133115

type(bern_einwohner)
```

```
int
```

2.1.4 Float

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

```
bern_flaeche = 51.62

type(bern_flaeche)
```

```
float
```

2.2 Übungen

2.2.1 Übung 1.1: Variablen erstellen

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

```
# Musterlösung

vorname = "Guido"
nachname = "van Rossum"

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

```
str
```

2.2.2 Übung 1.2: String verbinden

"Klebe" die beiden Variabeln mit einem Leerschlag dazwischen zusammen.

```
# Musterlösung

vorname+" "+nachname
```

```
'Guido van Rossum'
```

2.2.3 Übung 1.3: Zahl ohne Nachkommastelle

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

```
# Musterlösung

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

```
int
```

2.2.4 Übung 1.4: Zahl mit Nachkommastelle

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

```
# Musterlösung

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

```
float
```

2.2.5 Übung 1.5: Boolsche Variabeln

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

```
# Musterlösung

blond = False
```

2.2.6 Übung 1.6: Einwohnerdichte

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

```
# Musterlösung

einwohner = 8603900
flaeche = 41285

dichte = einwohner/flaeche

dichte
```

```
208.40256751846917
```

2.2.7 Übung 1.7: BMI

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

```
# Musterlösung

gewicht_kg = 85
groesse_m = groesse_cm/100

gewicht_kg/(groesse_m*groesse_m)
```

```
25.10633270321361
```

CHAPTER

THREE

KOMPLEXE DATENTYPEN

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

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

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

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

CHAPTER

FOUR

AUFGABE 2: LISTEN

4.1 Theorie

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

Eine *List* wird folgendermassen erstellt:

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

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

```
hexerei[0]
```

```
3
```

```
type(hexerei)
```

```
list
```

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

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

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

```
int
```

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

type(chaos[2])
```

```
list
```

4.2 Übungen

4.2.1 Übung 2.1: Lists

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

```
# Musterlösung

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

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

4.2.2 Übung 2.2: Elemente aus Liste ansprechen

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

```
# Musterlösung

vornamen[0]
```

```
'Christopher'
```

4.2.3 Übung 2.3: Liste ergänzen

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

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

```
# Musterlösung

nachnamen.append("Huck")

groessen.append(177)
```

4.2.4 Übung 2.4: Summen berechnen

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

```
# Musterlösung

sum(groessen)
```

```
695
```

4.2.5 Übung 2.5: Anzahl Werte ermitteln

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

```
# Musterlösung

len(groessen)
```

```
4
```

4.2.6 Übung 2.6: Mittelwert berechnen

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

```
# Musterlösung

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

```
173.75
```

4.2.7 Übung 2.7: Minimum-/Maximumwerte

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

```
# Musterlösung

min(groessen)
max(groessen)
```

```
182
```

CHAPTER

FIVE

AUFGABE 3: DICTIONARIES

5.1 Theorie

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

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


III. v/i. 5. trumpf ausspielen, trumpfen, stechen.
trump³ [trʌmp] v/t. ~ up erdichten, zs.-schwindeln, sich aus den Fingern saugen; 'trumped-up [trʌmpt] adj. erfunden, erlogen, falsch: ~ charges.
trump·er·y ['trʌmpəri] I. s. 1. Plunder m, Ramsch m, Schund m; 2. fig. Gewäsch n, Quatsch m; II. adj. 3. Schund..., Kitsch..., kitschig, geschmacklos; 4. fig. billig, nichtssagend: ~ arguments.
trum·pet ['trʌmpit] I. s. 1. ♪ Trom'pete f: to blow one's own ~ fig. sein eigenes Lob singen; the last ~ die Posaune des Jüngsten Gerichts; 2.

In Python würde man diese *Dictionary* folgendermassen erstellen:

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

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

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

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

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

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

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

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

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

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

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

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

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

```
4
```

5.2 Übungen

5.2.1 Übung 3.1: Dictionary erstellen

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

```
# Musterlösung

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

5.2.2 Übung 3.2: Elemente aus Dictionary ansprechen

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

```
# Musterlösung

me["nachname"]
```

```
'van Rossum'
```

5.2.3 Übung 3.3: *Dictionary* nutzen

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

```
# Musterlösung

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

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

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

5.2.4 Übung 3.4: Key ergänzen

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

```
# Musterlösung

me["groesse"] = 181
```

5.2.5 Übung 3.5: Dictionary mit List

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

```
# Musterlösung

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

5.2.6 Übung 3.6: Einträge abrufen

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

```
# Musterlösung

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

```
'Christopher'
```

5.2.7 Übung 3.7: Einträge abrufen

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

```
# Musterlösung

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

```
'Kantereit'
```

5.2.8 Übung 3.8: Mittelwert berechnen

Berechne den Mittelwert aller grössen in deiner Dict

```
# Musterlösung

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

```
172.66666666666666
```

CHAPTER

SIX

AUFGABE 4: TABELLARISCHE DATEN

6.1 Theorie

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

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

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

people_df = pd.DataFrame(people)

people_df
```

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

6.2 Übungen

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

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

6.2.2 Übung 4.2: *DataFrame* in csv umwandeln

In der Praxis kommen Tabellarische Daten meist als "csv" Dateien daher. Wir können aus unserer eben erstellten DataFrame sehr einfach eine csv Datei erstellen. Führe das mit folgendem Code aus und suche anschliessend die erstellte csv-Datei.

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

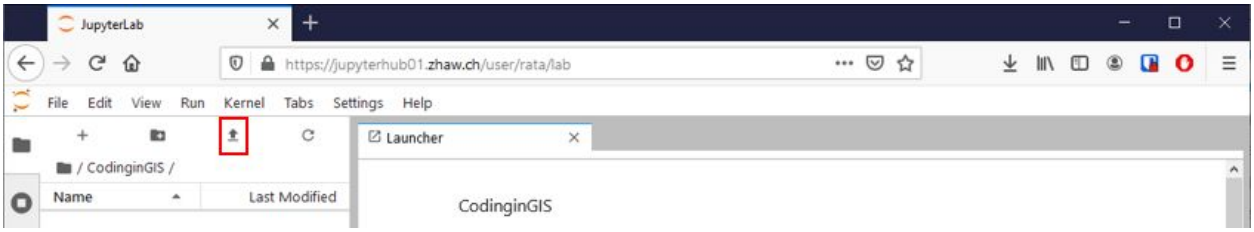
6.2.3 Übung 4.3: CSV als *DataFrame* importieren

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

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

Achtung!

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



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

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

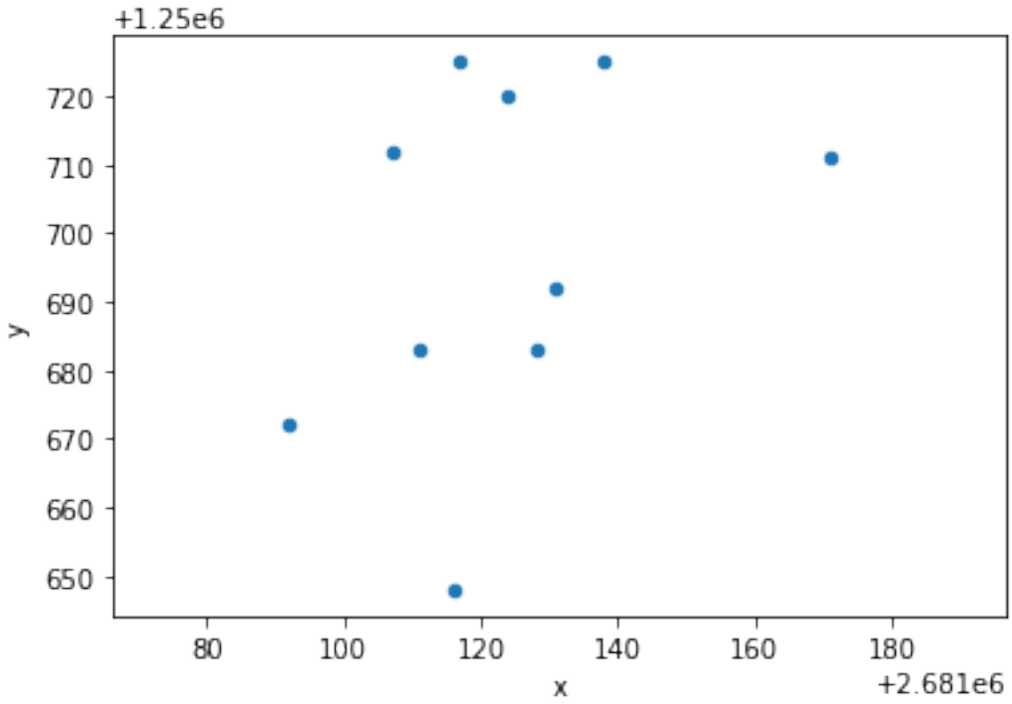
6.2.4 Übung 4.4: Koordinaten räumlich darstellen

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

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

fig.axis("equal")
```

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



6.2.5 Übung 4.5: Einzelne Spalte selektieren

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

6.2.6 Übung 4.6: Neue Spalte erstellen

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

```
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
```

CHAPTER

SEVEN

EINLEITUNG ZU DIESEM BLOCK

Letzte Woche habt ihr Jupyter Labs kennen gelernt und erste Kontakte mit Python gehabt. Um den Einsteig möglichst einfach zu halten hattet ihr bisher auf dem ZHAW Jupyter-Lab Server (JupyterHub) gearbeitet.

Diese Woche wollen wir versuchen, dass ihr *lokal* auf euren eigenen Rechnern arbeitet. Es ist extrem hilfreich, eine eigene, lokale installation von Python zu haben die man gut verwalten kann. Es ist aber auch nicht ganz trivial, und einige von euch müsst allenfalls bei der Server-basierten Lösung bleiben.

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.

CHAPTER

EIGHT

AUFGABE 5: VIRTUAL ENVIRONMENTS / CONDA

Um *conda* zu verstehen und nutzen zu können, müsst ihr das Konzept von "Virtual Environments" (siehe Vorlesung) einigermassen verstanden haben. Sollte dies nicht der Fall sein, meldest du dich am besten beim Dozenten oder schaust dir nochmals die Vorlesungsfolien an.

Diese nächsten Schritte zur Nutzung von *conda* wurden bereits in der Vorlesung beschrieben. Sie sind aber an dieser stelle nochmal erläutert und mit ein paar zusätzlichen Angaben (z.B. Links) erweitert.

Schritt 1: Conda installieren

Normalerweise ist der erste Schritt, *conda* herunterzuladen und installieren (ich empfehle dazu miniconda, nicht anaconda). Wenn ihr aber ArcGIS Pro installiert habt, ist dieser Schritt nicht nötig (da *conda* mit ArcGIS mitinstalliert wird).

Schritt 2: Systemvariable setzen

Conda ist eine Software, die sich nur von der Kommandozeile (Windows: *cmd*) aus bedienen lässt. Dafür muss *cmd* aber wissen, WO conda installiert ist. Um zu Prüfen, ob dies der fall ist kannst du die Konsole starten (Windowstaste > cmd) und folgende Zeile einfügen

```
conda --version
```

Wenn du eine Versionsnummer siehst kannst du zum nächsten Schritt übergehen. Ansonsten mussst du die Systemvariable `PATH` zuerst noch mit dem Pfad zur *conda* installation ergänzen. 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. Anschlissend 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
```

CHAPTER

NINE

PYTHON MODULES

9.1 Erweiterung installieren

In R ist die Installation einer *Library* selbst 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 (`conda`).

in R:

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

In Python*:

```
conda install -c conda-foge 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 *Reposoritory* geben wir in R meistens nicht an, weil in RStudio bereits eine Default-Adresse hinterlegt ist.

9.2 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(malerin)
```

in Python:

```
import malerin
```

9.3 Erweiterung verwenden

Um eine geladene *Function* in R zu verwenden, 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, eine andere Erweiterung (z.b. `maurer`) hat ebenfalls eine *Function* `wand_bemalen()`. Dann ist in R nicht klar, welche Erweiterung gemeint ist, was zu Missverständnissen führen kann. In Python ist im obigen Beispiel unmissverständlich, dass ich `wand_bemalen()` aus dem Modul `malerin` meine.

9.4 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.

9.5 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()
```

9.6 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()
```

CHAPTER

TEN

AUFGABE 6: *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 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 gegrüsst 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!'
```

10.1.1 Übungen

10.1.2 Übung 6.1: 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'
```

10.1.3 Übung 6.2: *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'
```

10.1.4 Übung 6.3: Default-Werte festlegen

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

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

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

CHAPTER

ELEVEN

AUFGABE 7: *FUNCTION* ADVANCED

11.1 Theorie

11.1.1 Standart-Werte

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
```

11.1.2 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'
```

11.1.3 Globale und Lokale Variabeln

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

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

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

Nun erstellen wir eine Function, welche diese Variabeln nutzen soll:

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

Wenn wir jetzt aber die Function ausführen wollen, entsteht eine Fehlermeldung.

```
gruezi()
```

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'
```

11.1.4 Lambda-Function

Mit dem Begriff `lambda` kann das erstellen einer Funktion verkürzt werden. Nehmen wir nochmal die *Function* `gruezi()`:

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

Mit `lambda` kann die *Function* verküzrt geschrieben werden, was in verschiedenen Situation nützlich sein kann. Vor allem ist es gut, diese Notation zu kennen, da man ab und zu drüber stolpert.

```
gruezi = lambda nachname, anrede: "Guten Tag, "+anrede +" "+ nachname

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

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

11.2 Übungen

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 enwickeln 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.

11.2.1 Übung 7.1: 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.

11.2.2 Übung 7.2: Zufallswerte addieren

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

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

```
2599945.5170354503 1199905.9056325185
```

11.2.3 Übung 7.3: 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.

11.2.4 Übung 7.4: Output visualisieren

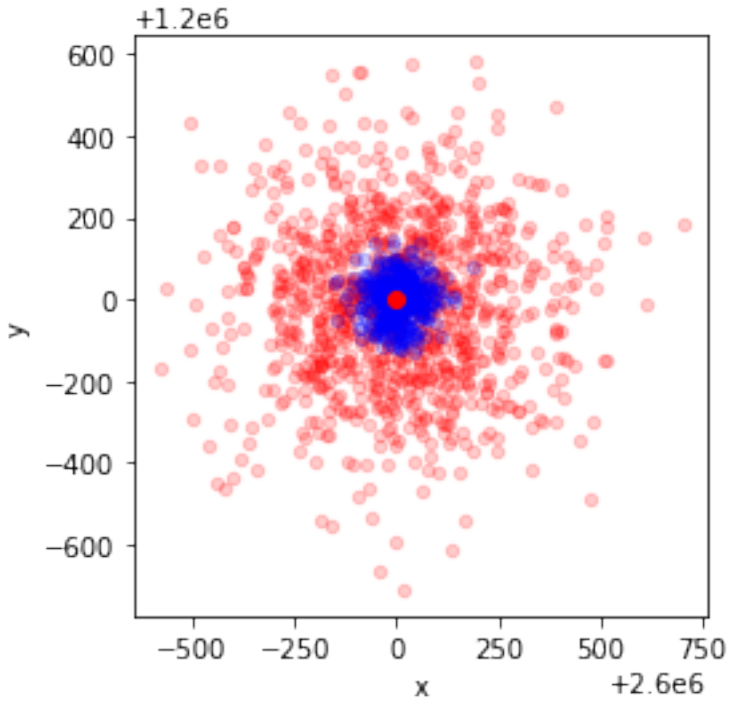
Nun ist es wichtig, dass wir unser Resultat visuell überprüfen. Im Beispiel unten wende ich die *Function* `offset_coordinate()` 1000x auf die Koordianten der alten Sternwarte an. Für diesen Schritt gebe ich euch den fertigen Code, da ihr die dafür benötigten Techniken noch nicht gelernt habt. Füge diesen 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()
```


+1.2e6
600
400
200
0
−200
−400
−600
y
−500
−250
0
250
500
750
x
+2.6e6

CHAPTER

TWELVE

AUFGABE 8: *FUNCTION* IN *DATAFRAMES*

12.1 Theorie

Nun wollen wir usere Function `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
```

12.2 Übungen

12.2.1 Übung 8.1: Alle Zeckenstiche zufällig verschieben

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

12.2.2 Übung 8.2: Neue GeoDataFrame mit simulierten Punkten erstellen

Um die neuen, verschobenen Koordianten 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.

12.2.3 Übung 8.4: 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-Zeckenstichen, bei den blauen um die simulierten (leicht verschoben) Zeckenstiche.

12.2.4 Übung 8.5: 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  2.681000e+06  1.249945e+06
1  10437  2.681012e+06  1.250189e+06
2   9174  2.681048e+06  1.250200e+06
3   8773  2.681031e+06  1.250200e+06
4   2764  2.681051e+06  1.250209e+06
5   2513  2.681091e+06  1.250228e+06
6   9185  2.681027e+06  1.250229e+06
7  28521  2.681044e+06  1.250237e+06
8  26745  2.681037e+06  1.250242e+06
9  27391  2.681058e+06  1.250242e+06
```

CHAPTER

THIRTEEN

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

CHAPTER

FOURTEEN

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 Nonsens in die Konsole ausgespukt. 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 ersetzte `[0,1,2]` mit `range(3)`.

CHAPTER

FIFTEEN

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 for dem For-Loop einen leeren Behälter, zum Beispiel eine leere Liste (`strophe = []`). Nun können wir innerhalb des *Loops* `append()` nutzen, um den Output von einem Durchgang dieser Liste hinzu zu fügen.

```
strophe = []

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

strophe
```

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

15.1 Übung: Output aus For-Loop speichern

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

CHAPTER

SIXTEEN

ÜBUNG: ZECKENSTICH SIMULATION MIT LOOP

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

- 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")
```

Fig. 16.1: Quelle twitter.com

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

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

```
zeckenstiche_sim = pd.DataFrame()

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

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

	ID	x	y
0	2550	2.681144e+06	1.250707e+06
1	10437	2.680972e+06	1.250628e+06
2	9174	2.681266e+06	1.250686e+06
3	8773	2.681057e+06	1.250625e+06
4	2764	2.681228e+06	1.250693e+06
5	2513	2.681127e+06	1.250677e+06
6	9185	2.680979e+06	1.250748e+06
7	28521	2.681120e+06	1.250709e+06
8	26745	2.680885e+06	1.250724e+06
9	27391	2.681244e+06	1.250772e+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()`.

CHAPTER

SEVENTEEN

ÜBUNG: GIS IN PYTHON

Bis jetzt haben wir noch nicht mit eigentlich Geodaten gearbeitet. Die x / y Werte der Zeckenstiche repräsentatieren 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)

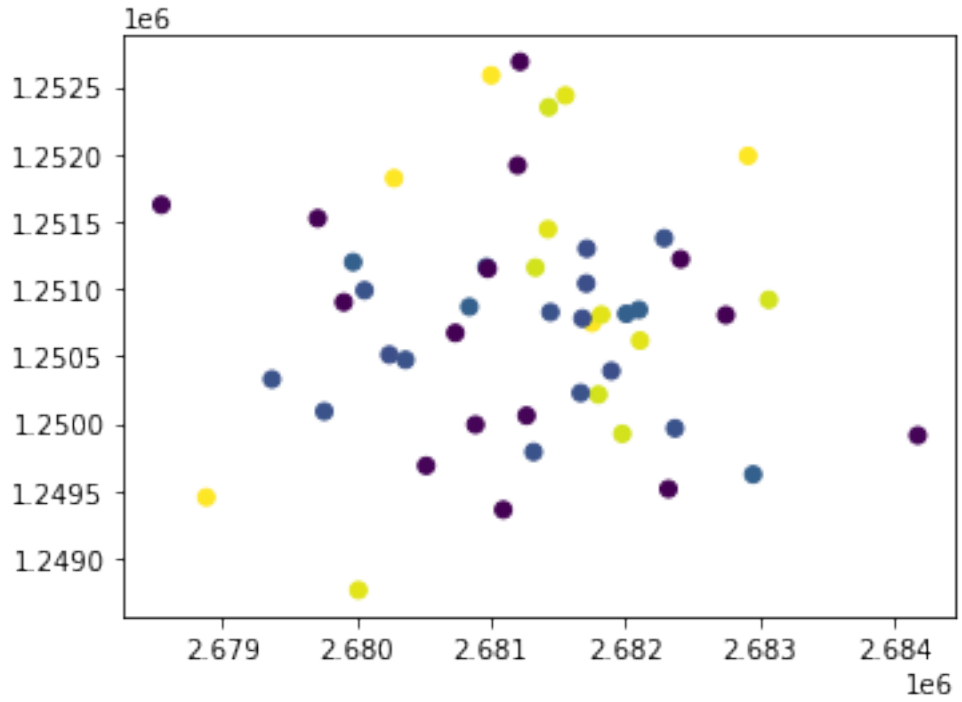
type(monte_carlo_df)
```

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

```
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). Viel praktischer sind an dieser Stelel 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")
```

CHAPTER

EIGHTEEN

ÜBUNG: WALDANTEIL BERECHNEN

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

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

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

```
import pandas as pd
import geopandas as gpd

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

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


wald
```

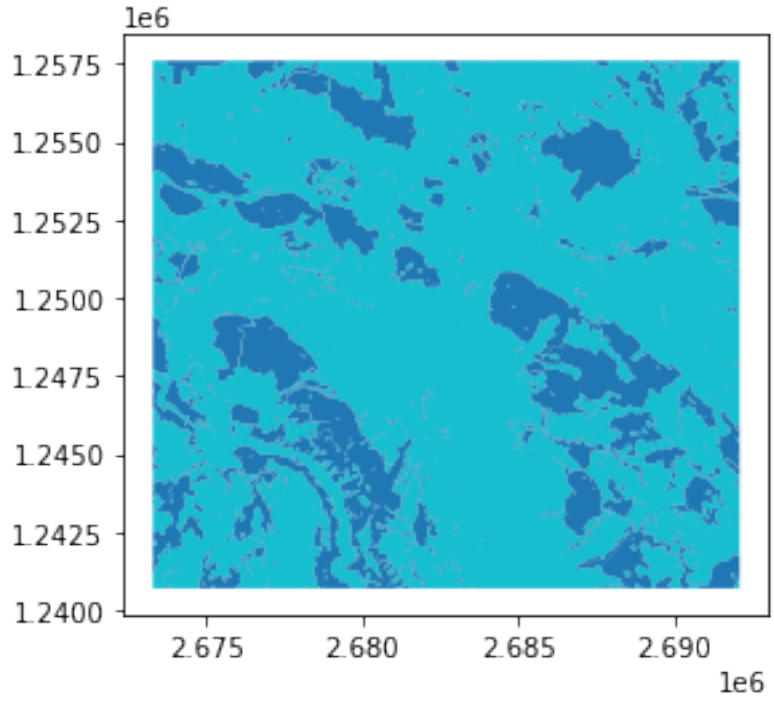
```
/opt/hostedtoolcache/Python/3.7.9/x64/lib/python3.7/site-packages/geopandas/_compat.
↪py:88: UserWarning: The Shapely GEOS version (3.8.0-CAPI-1.13.1 ) is incompatible␣
↪with the GEOS version PyGEOS was compiled with (3.8.1-CAPI-1.13.3). Conversions␣
↪between both will be slow.
  shapely_geos_version,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

```
   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....
```

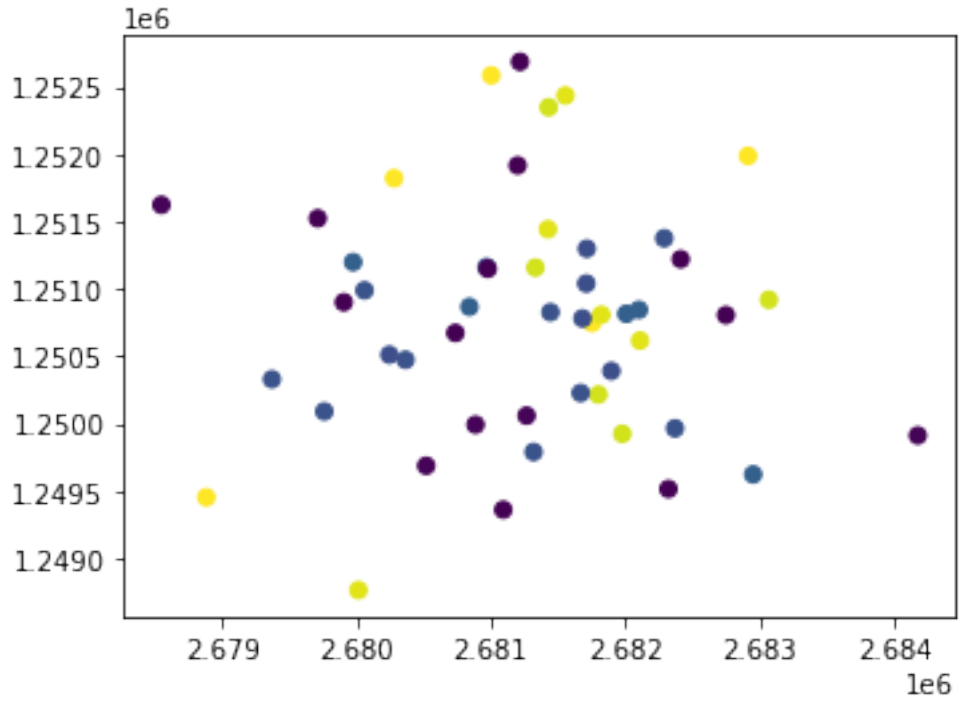
```
# hierfür braucht ihr das modul "descartes"
wald.plot(column = "Wald_text")
```

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

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

monte_carlo_sjoin.head()
```

```
      ID             x             y  Run_Nr                          geometry  \
0   2550  2.679907e+06  1.250900e+06       0  POINT (2679906.592 1250899.871)
1  10437  2.679976e+06  1.251198e+06       0  POINT (2679975.685 1251197.595)
3   8773  2.681895e+06  1.250390e+06       0  POINT (2681895.192 1250389.933)
4   2764  2.681265e+06  1.250058e+06       0  POINT (2681264.845 1250058.434)
5   2513  2.681198e+06  1.251915e+06       0  POINT (2681198.323 1251915.280)

   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
3            0     0  947316.853401  2.380876e+08      nein
4            0     0  947316.853401  2.380876e+08      nein
5            0     0  947316.853401  2.380876e+08      nein
```

18.2 Übung 2: Anteil der Punkte pro "Gruppe"

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

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

```
                  Anzahl
Run_Nr Wald_text
0      ja              3
       nein            7
1      ja              2
       nein            8
2      ja              4
       nein            6
3      ja              5
```

(continues on next page)

(continued from previous page)

```
       nein            5
4      ja              2
       nein            8
```

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

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

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

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

anteile_pivot
```

```
Wald_text  ja  nein
Run_Nr
0           3     7
1           2     8
2           4     6
3           5     5
4           2     8
```

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
```

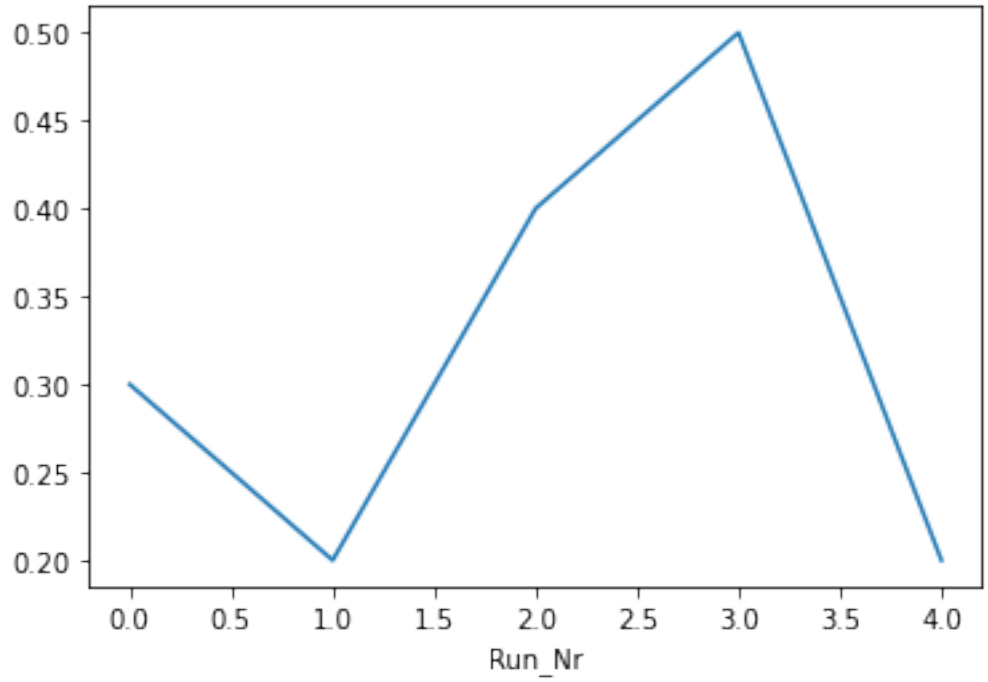
```
Run_Nr
0    0.3
1    0.2
2    0.4
3    0.5
4    0.2
dtype: float64
```

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()`.

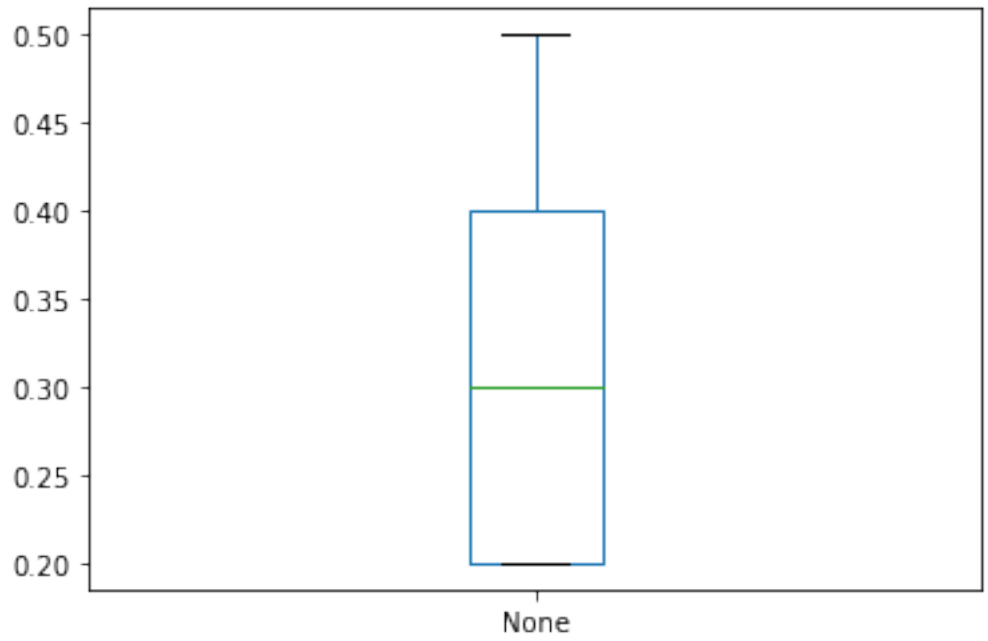
```
anteil_ja.plot()
```

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



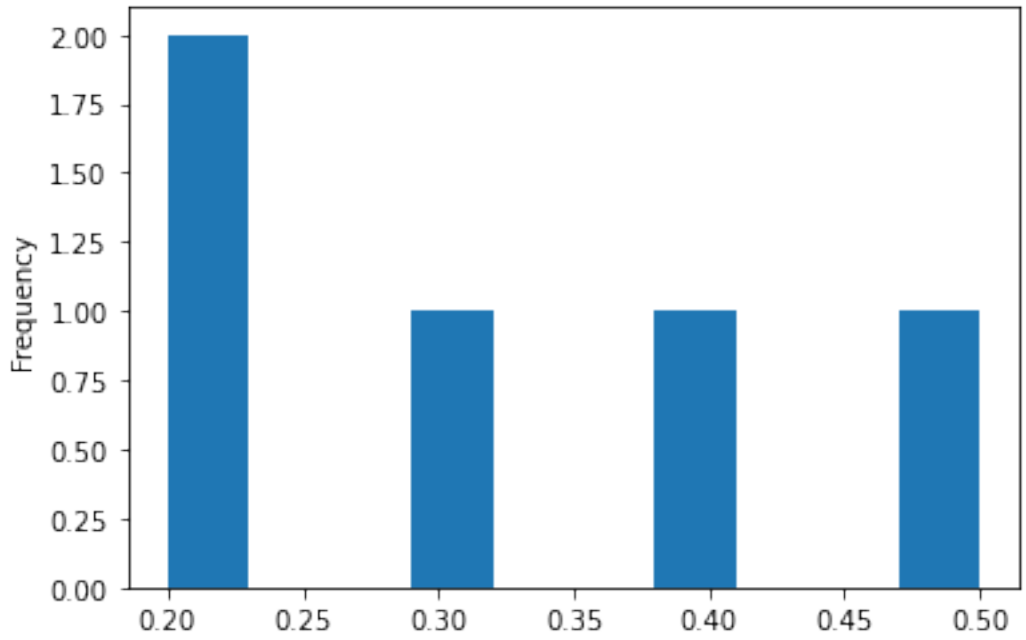

```
anteil_ja.plot.box()
```

```
<AxesSubplot:>
```




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

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



CHAPTER

NINETEEN

BASIC SHORTCUTS FOR JUPYTER LAB

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