
Coding in GIS

Nils Ratnaweera

Sep 07, 2020

CODING IN GIS I

1	Einleitung zu diesem Block	3
2	Input: Primitive Datentypen	5
2.1	Boolean	5
2.2	String	5
2.3	Integer	6
2.4	Float	6
2.5	Weitere Datentypen	6
3	Input: Zusammengesetzte Datentypen	7
3.1	List	7
3.2	Dict	8
3.3	DataFrame	8
3.4	Tuples	9
3.5	Sets	10
4	Aufgabe A: Lists	11
4.1	Übung 1: Variablen erstellen	11
4.2	Übung 2: Lists	12
4.3	Übung 3: Elemente aus Liste ansprechen	12
4.4	Übung 4: Liste ergänzen	12
5	Aufgabe B: Dictionaries	13
5.1	Übung 1: Dictionary	14
5.2	Übung 6: Elemente aus Dictionary ansprechen	14
5.3	Übung 2: Key ergänzen	14
5.4	Input	15
5.5	Übung 3: Dictionary mit List	15
5.6	Übung 4: Functions	15
6	Input: Python Modules	17
6.1	Erweiterung installieren	17
6.2	Erweiterung laden	17
6.3	Erweiterung verwenden	18
6.4	Modul mit Alias importieren	18
6.5	Einzelne <i>Function</i> importieren	18
6.6	Alle <i>Functions</i> importieren	18
7	Aufgabe: Tabellarische Daten	21
7.1	Übung 1: von einer <i>Dictionary</i> zu einer <i>DataFrame</i>	21
7.2	Übung 2: <i>DataFrame</i> in csv umwandeln	21

7.3	Übung 3: CSV als <i>DataFrame</i> importieren	21
7.4	Übung 4: Koordinaten räumlich darstellen	22
7.5	Übung 5: Einzelne Spalte selektieren	22
7.6	Übung 6: Neue Spalte erstellen	23
8	Einleitung zu diesem Block	25
9	Input: <i>Functions</i> Teil I	27
10	Aufgabe: <i>Functions</i>	29
10.1	Übung 1: Erste <i>Function</i> erstellen	29
10.2	Übung 2: <i>Function</i> erweitern	29
10.3	Übung 3: Default-Werte festlegen	29
10.4	Übung 4: Python-File in Module konvertieren	30
11	Input: <i>Functions</i> Teil II	31
11.1	<i>Functions</i> mit Default Werten	31
11.2	Reihenfolge der Argumente	31
11.3	Globale und Lokale Variablen	32
12	Einen Einzelpunkt zufällig verschieben	33
12.1	Übung 1	33
12.2	Übung 2: Zufallswert addieren	34
12.3	Übung 3: Arbeitsschritte in eine <i>Function</i> verwandeln	35
12.4	Output visualisieren	35
12.5	Optionale Übung für <i>Hochmotivierte</i> (sehr anspruchsvoll)	36
13	Punkte einer <i>GeoDataFrame</i> zufällig verschieben	39
13.1	Übung 1: Alle Zeckenstiche zufällig verschieben	39
13.2	Übung 2: Neue <i>GeoDataFrame</i> mit simulierten Punkten erstellen	40
13.3	Übung 3: ID übernehmen	40
13.4	Übung 4: Mehrere <i>GeoDataFrames</i> visualisieren	40
14	Einleitung zu diesem Block	43
15	For Loops	45
15.1	Übung 1: Erste For-Loop erstellen	46
15.2	Übung 2: For-Loop mit <code>range()</code>	46
15.3	Input: Output aus For-Loop	46
15.4	Übung 3: Output aus For-Loop speichern	47
16	Zeckenstich Simulation mit Loop	49
16.1	Übung 1: Mit For-Loop Monte Carlo Simulation durchführen	50
16.2	Übung 2: <i>GeoDataFrames</i> aus Simulation zusammenführen	50
16.3	Übung 3: Simulierte Daten visualisieren	51
17	Aufgabe: GIS in Python	53
17.1	Übung 1: <i>DataFrame</i> zu <i>GeoDataFrame</i>	53
17.2	Übung 2: Koordinatensystem festlegen	54
17.3	Übung 3: Zeckenstiche als Shapefile exportieren	55
18	Waldanteil berechnen	57
18.1	Übung 1: Wald oder nicht Wald?	57
18.2	Übung 2: Anteil der Punkte im Wald	58
18.3	Übung 3: Mittelwerte Visualisieren	58

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](#).

Online und pdf

Dieses Buch ist als online version hier gehostet: <https://ratnanil.github.io/codingingis>. Es kann aber auch als pdf [hier](#) heruntergeladen werden.

Typographie

Noch ein kurzer Hinweis zur Typografie dieses Dokumentes:

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

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

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

Für R Nutzer

- hier wird ein Bezug zu R gemacht
-

EINLEITUNG ZU DIESEM BLOCK

Übungsziele

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

INPUT: PRIMITIVE DATENTYPEN

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

Für R Nutzer

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

2.1 Boolean

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

```
regen = True
```

```
sonne = False  
  
type(sonne)
```

```
bool
```

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 Weitere Datentypen

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

INPUT: ZUSAMMENGESETZTE DATENTYPEN

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

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
```

```
```{admonition} Für R Nutzer
:class: tip
Eine Python List entspricht eines R Vectors
```
```

```
File "<ipython-input-7-c5ec95084d82>", line 1
    ```{admonition} Für R Nutzer
 ^
SyntaxError: invalid syntax
```

## 3.2 Dict

Eine Dict:

- 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
- Ein Spezialfall einer Dict:

- values sind Lists von gleicher Länge
- alle Werte in einer List bestehen aus dem gleichen Datentyp
- jeder key ist ein Spaltenname

```
import pandas as pd
```

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

menue2
```

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

```
type(menue2)
```

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

```
menue_df = pd.DataFrame(menue2)

menue_df
```

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

```
type(menue_df)
```

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

## 3.4 Tuples

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. Tuples werden mit runden Klammern ( ( und ) ) erstellt.

Dieser Datentyp spielt in unserem Kurs *Coding in GIS* eine untergeordnete Rolle.

```
gewinner = (1, 2, 3)
```

```
type(gewinner)
```

```
tuple
```

### 3.5 Sets

Sets sind ähnlich wie Dicts, jedoch kommt jeder Wert nur 1x vor. Zudem verfügen Sets nicht über keys und values. Sie werden mit geschweiften Klammern ({ und }) erstellt.

Dieser Datentyp spielt in unserem Kurs *Coding in GIS* eine untergeordnete Rolle.

```
hexerei2 = {1, 2, 3, 1, 2, 2, 1, 3}
```

```
hexerei2
```

```
{1, 2, 3}
```

```
type(hexerei2)
```

```
set
```

## AUFGABE A: LISTS

### 4.1 Übung 1: Variablen erstellen

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

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

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

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

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

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

blond = False
```

## 4.2 Übung 2: Lists

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

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

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

## 4.3 Übung 3: Elemente aus Liste ansprechen

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

---

### Für R Nutzer

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

---

## 4.4 Übung 4: Liste ergänzen

Listen können durch der Method `append` ergänzt werden.

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

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

```
nachnamen.append("Huck")

groessen.append(177)
```



## AUFGABE B: DICTIONARIES

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 Kommage- trennt aufgeführt.

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

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

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

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

---

### Für R Nutzer

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

---

## 5.1 Übung 1: Dictionary

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

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

## 5.2 Übung 6: Elemente aus Dictionary ansprechen

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

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

```
'Guido'
```

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

```
'van Rossum'
```

## 5.3 Übung 2: Key ergänzen

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

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

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

---

### Für R Nutzer

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

---

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

## 5.4 Input

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']
```

## 5.5 Übung 3: Dictionary mit List

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

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

## 5.6 Übung 4: Functions

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

```
len(groessen)
```

```

NameError Traceback (most recent call last)
<ipython-input-11-97619b63240a> in <module>
----> 1 len(groessen)

NameError: name 'groessen' is not defined
```

```
sum(groessen)
```

```
695
```

```
len(vornamen)
```

```
4
```

```
sum(vornamen)
```

## INPUT: 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*.

### 6.1 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:

| R                                              | python                                                                           |
|------------------------------------------------|----------------------------------------------------------------------------------|
| <code>install.packages("malerinWanger")</code> | <code># Module können nicht innerhalb<br/># von Python installiert werden</code> |

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

### 6.2 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:

| R                                    | python                            |
|--------------------------------------|-----------------------------------|
| <code>library(malerlinWanger)</code> | <code>import malerinWanger</code> |

## 6.3 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:

| R                           | python                                    |
|-----------------------------|-------------------------------------------|
| <code>wand_bemalen()</code> | <code>malerinWanger.wand_bemalen()</code> |

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 `malerinWanger` meine.

## 6.4 Modul mit Alias importieren

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

```
import malerinWanger 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.

## 6.5 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 malerinWanger import wand_bemalen
wand_bemalen()
```

## 6.6 Alle *Functions* importieren

Zusätzlich ist es möglich, **alle** *Functions* aus einem Modul zu importieren. Diese Notation wird nicht empfohlen und ist hier nur erwähnt, weil ihr allenfalls diese Schreibweise antreffen werdet.

```
from malerinWanger 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 malerinWanger import wand_bemalen`
  - Mit einem Platzhaltersymbol `*` können auch alle *Functions* aus einem Modul importiert werden: `from malerinWanger import *`
-





## AUFGABE: TABELLARISCHE DATEN

Schauen wir uns nochmals die *Dictionary* `people` an. Diese ist ein Spezialfall einer *Dictionary*: Jeder Eintrag besteht aus einer Liste von gleich vielen Werten. Wie bereits erwähnt, kann es in einem solchen Fall sinnvoll sein, die *Dictionary* als Tabelle darzustellen. Dafür brauchen wir das Modul `pandas`, welches wir mit dem Alias `pd` importieren.

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

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

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

### 7.2 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")
```

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

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

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

**Hinweis:** Dieser Code funktioniert nur, wenn “zeckenstiche.csv” im aktuellen Arbeitsverzeichnis (*Current Working Directory*) abgespeichert wird. Wenn du nicht sicher bist, wo dein aktuelles Arbeitsverzeichnis liegt, kannst du das Modul `os` laden und dies mit der Funktion `os.getcwd()` (get *current working directory*) herausfinden.

```
import os

os.getcwd()
```

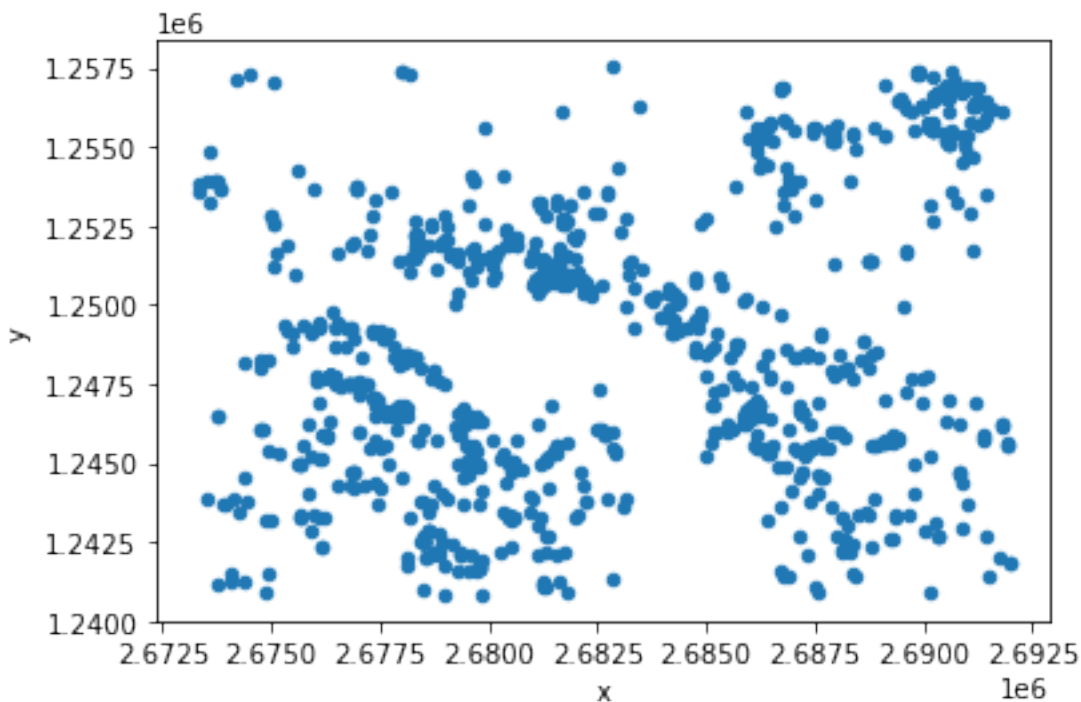
```
'/home/runner/work/codingingis/codingingis'
```

## 7.4 Übung 4: Koordinaten räumlich darstellen

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

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

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



## 7.5 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.

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

```
0 0
1 1
2 2
3 3
4 4
...
1071 1071
1072 1072
```

(continues on next page)

(continued from previous page)

```

1073 1073
1074 1074
1075 1075
Name: ID, Length: 1076, dtype: int64

```

## 7.6 Übung 6: Neue Spalte erstellen

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

```

zeckenstiche["Stichtyp"] = "Zecke"

zeckenstiche

```

```

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

[1076 rows x 5 columns]

```



## EINLEITUNG ZU DIESEM BLOCK

Letzte Woche haben wir einen weiten Bogen gespannt: Von einfachen (primitiven) Datentypen bis hin zu Tabellarischen. In dieser und der nächsten Übung (Coding in GIS II und III) wollen wir uns weiter mit den Zeckenstichdaten befassen. Wir werden im Wesentlichen ein Teil der Übung aus “Datenqualität und Unsicherheit” in Python rekonstruieren.

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

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

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

---

### Übungsziele

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



## INPUT: *FUNCTIONS* TEIL I

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

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

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

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

```
sag_hallo()
```

```
'Hallo!'
```

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

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

Nun können wir der Function einen Parameter übergeben, damit wir persönlich begrüßt werden. In folgendem Beispiel ist `vorname` ein Parameter, “Guido” ist sein Argument.

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

```
'Hallo Guido!'
```

Den Output können wir wie gewohnt einer neuen Variabel zuweisen:

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

```
'Hallo Guido!'
```



## AUFGABE: *FUNCTIONS*

### 10.1 Übung 1: Erste *Function* erstellen

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

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

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

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

### 10.2 Übung 2: *Function* erweitern

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

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

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

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

### 10.3 Übung 3: Default-Werte festlegen

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

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

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

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

### 10.4 Übung 4: Python-File in Module konvertieren

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

```
import myownmodule

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

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

## INPUT: *FUNCTIONS* TEIL II

### 11.1 *Functions* mit Default 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)
```

nicht so:

```
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
```

### 11.2 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'
```

Andersrum würde ein Fehler entstehen:

```
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'
```

## 11.3 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). Folgendes geht also nicht:

```
nachname = "van Rossum"
anrede = "Herr"

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'
```

## EINEN EINZELPUNKT ZUFÄLLIG VERSCHIEBEN

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

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

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

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

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

```
POINT (2600000 1200000)
```

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

### 12.1 Übung 1

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

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

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



Fig. 12.1: 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

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

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

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

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

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

```
1200000.0
```

## 12.2 Übung 2: Zufallswert addieren

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

Importiere dieses Modul und nutze die genannte Funktion um einen Versatz je für `x` und `y` zu generieren. Addiere diese beiden Zahlen zu `x_alt` resp. `y_alt` und speichere die Outputs als `x_neu` und `y_neu`. Nun können wir

wieder `Point()` verwenden um aus `x_neu`, `y_neu` eine neue Punkt-Geometrie zu erstellen. Speichere diese neue Geometrie unter der Variabel `sternwarte_neu`.

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

import random

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

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

```
POINT (2599915 1200067)
```

## 12.3 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 Funktion `point_offset` welche als Input ein `Point()`-Objekt nimmt und ein leicht verschobenes `Point()`-Objekt zurück gibt. Wenn du möchtest kannst du die Distanz der Verschiebung als optionalen Parameter (wichtig!) definieren.

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

 distance = int(distance)

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

 point_off = Point(x_neu, y_neu)

 return(point_off)

point_offset(sternwarte, 100)
```

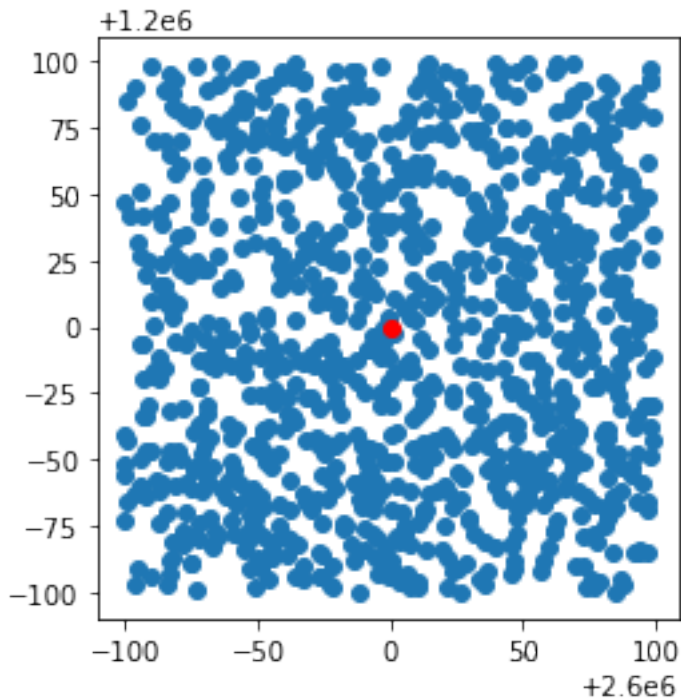
```
<shapely.geometry.point.Point at 0x7f5fb005f350>
```

## 12.4 Output visualisieren

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

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

```
<AxesSubplot:>
```



## 12.5 Optionale Übung für *Hochmotivierte* (sehr anspruchsvoll)

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

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

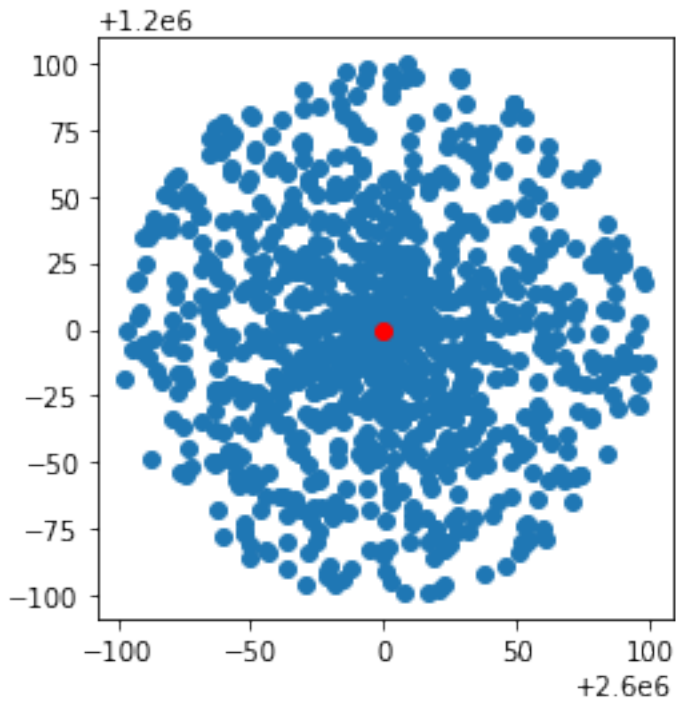
(continues on next page)



(continued from previous page)

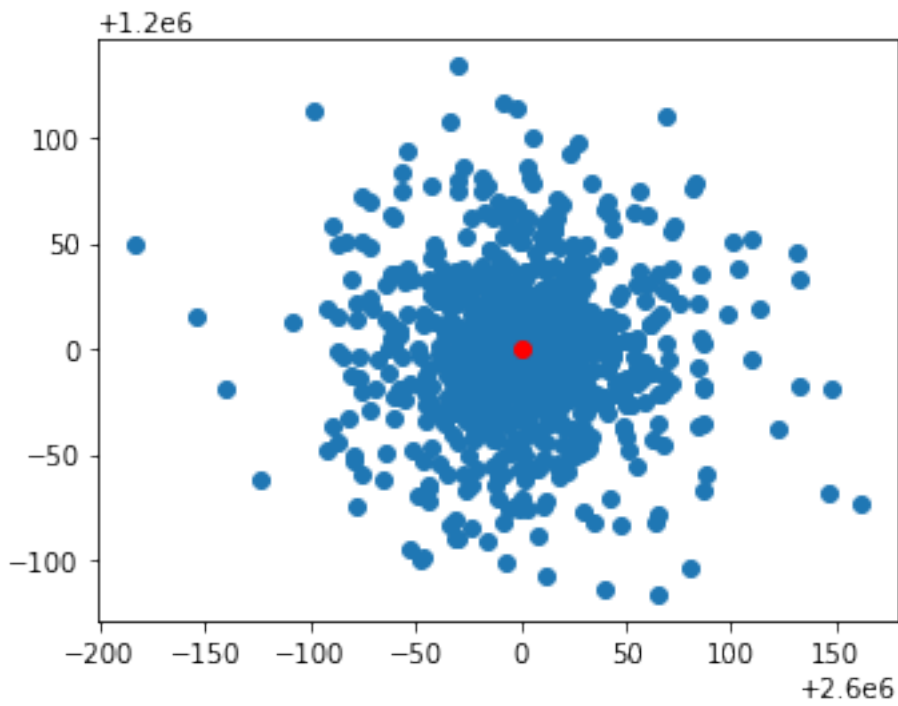
```
ax = gpd.GeoSeries([point_offset2(sternwarte,100, "uniform") for x in range(1000)]).
 ↳plot()
gpd.GeoSeries(sternwarte).plot(ax = ax, color = "red")
```

&lt;AxesSubplot:&gt;



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

&lt;AxesSubplot:&gt;



## PUNKTE EINER GEODATAFRAME ZUFÄLLIG VERSCHIEBEN

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

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

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

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

 distance = int(distance)

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

 point_off = Point(x_neu, y_neu)

 return(point_off)
```

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

### 13.1 Übung 1: Alle Zeckenstiche zufällig verschieben

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

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

```
0 POINT (2678910.000 1240792.000)
1 POINT (2679877.000 1240917.000)
2 POINT (2687471.000 1240809.000)
3 POINT (2674928.000 1240939.000)
4 POINT (2681860.000 1240987.000)
Name: geometry, dtype: geometry
```

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

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

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

## 13.3 Übung 3: ID übernehmen

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

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

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

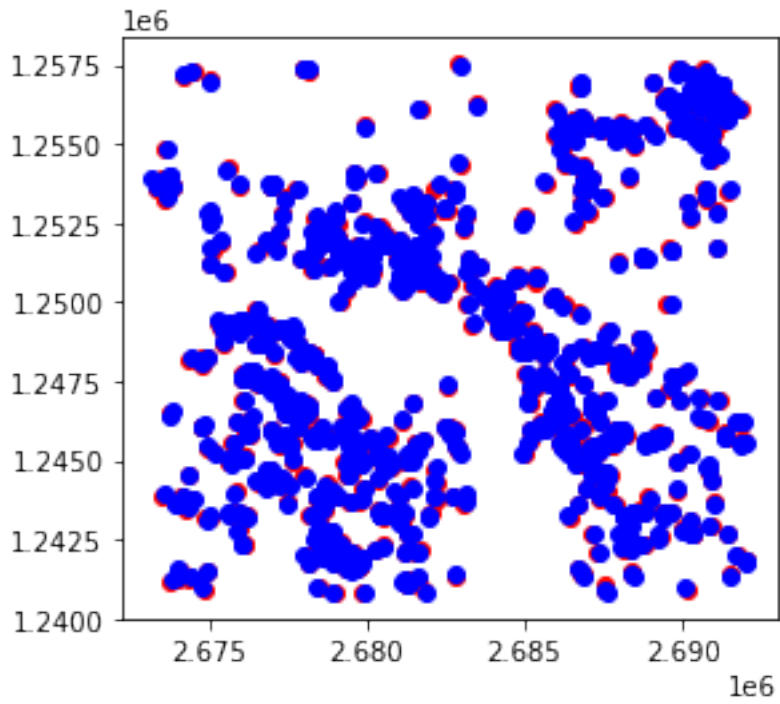
```
 geometry ID
0 POINT (2678886.000 1240850.000) 0
1 POINT (2679931.000 1240826.000) 1
2 POINT (2687632.000 1240850.000) 2
3 POINT (2674761.000 1241000.000) 3
4 POINT (2681870.000 1240833.000) 4
```

## 13.4 Übung 4: Mehrere GeoDataFrames visualisieren

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

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

```
<AxesSubplot:>
```





## **EINLEITUNG ZU DIESEM BLOCK**

!! HIER NOCH EINE EINLEITUNG SCHREIBEN !!

---

### **Übungsziele**

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





## FOR LOOPS

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

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

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

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

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

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

```
platzhalter
```

```
2
```

## 15.1 Übung 1: Erste For-Loop erstellen

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

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

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

## 15.2 Übung 2: For-Loop mit range ()

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

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

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

## 15.3 Input: Output aus For-Loop

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

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

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

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

```
liedzeile
```

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

```
schlagwort
```

```
'saint'
```

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

## 15.4 Übung 3: Output aus For-Loop speichern

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

```
mylist = []

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

mylist
```

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



## ZECKENSTICH SIMULATION MIT LOOP

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

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

Der Code für diese Schritte 1 – 4 lautet folgendermassen:

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

Schritt 2
from shapely.geometry import Point

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

 distance = int(distance)

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

 point_off = Point(x_neu, y_neu)

 return (point_off)

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

Schritt 4
zeckenstiche_sim = gpd.GeoDataFrame()
zeckenstiche_sim["geometry"] = zeckenstiche["geometry"].apply(point_offset)
zeckenstiche_sim["ID"] = zeckenstiche["ID"]
```

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

## 16.1 Übung 1: Mit For-Loop Monte Carlo Simulation durchführen

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

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

## 16.2 Übung 2: GeoDataFrames aus Simulation zusammenführen

Schau dir die Outputs an.

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

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

```
type(zeckenstiche_sim)
```

```
geopandas.geodataframe.GeoDataFrame
```

```
type(monte_carlo)
```

```
list
```

```
len(zeckenstiche_sim)
```

```
1076
```

```
len(monte_carlo)
```

```
50
```

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

## 16.3 Übung 3: Simulierte Daten visualisieren

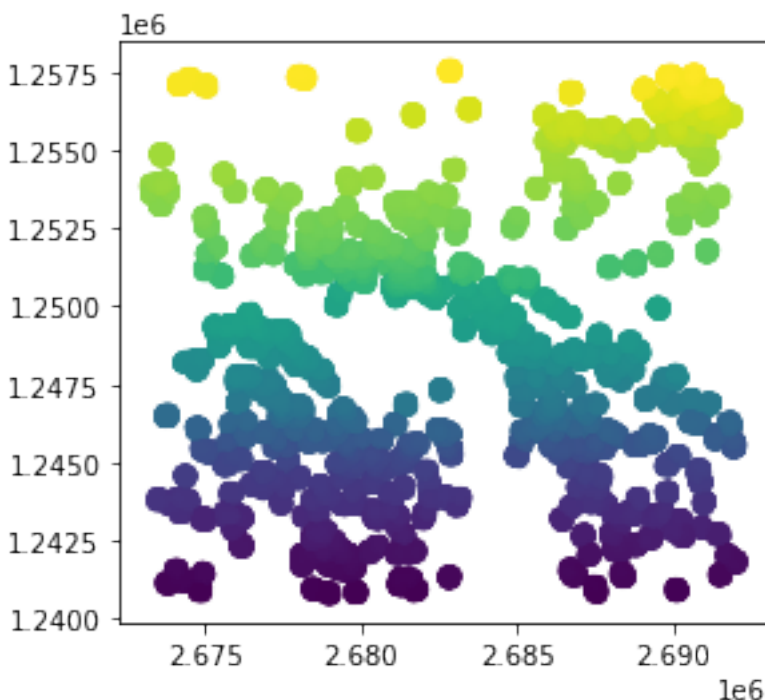
Exploriere nun `monte_carlo_df`? Was ist es für ein Datentyp? Was hat es für Spalten? Visualisiere den Datensatz räumlich mit `monte_carlo_df.plot()`. Optional: Wenn du Punkte mit der gleichen ID einer Farbe zuweisen möchtest, dann erreichst du dies mit der Option `column = "ID"` (das geht natürlich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

```
monte_carlo_df.head()
```

|   | geometry                        | ID | Run_Nr |
|---|---------------------------------|----|--------|
| 0 | POINT (2678989.000 1240777.000) | 0  | 0      |
| 1 | POINT (2679853.000 1240945.000) | 1  | 0      |
| 2 | POINT (2687498.000 1240912.000) | 2  | 0      |
| 3 | POINT (2674795.000 1240847.000) | 3  | 0      |
| 4 | POINT (2681710.000 1240958.000) | 4  | 0      |

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

```
<AxesSubplot:>
```



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



## AUFGABE: GIS IN PYTHON

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

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

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

### 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.

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

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

```
zeckenstiche = gpd.GeoDataFrame(zeckenstiche)
```

```
zeckenstiche
```

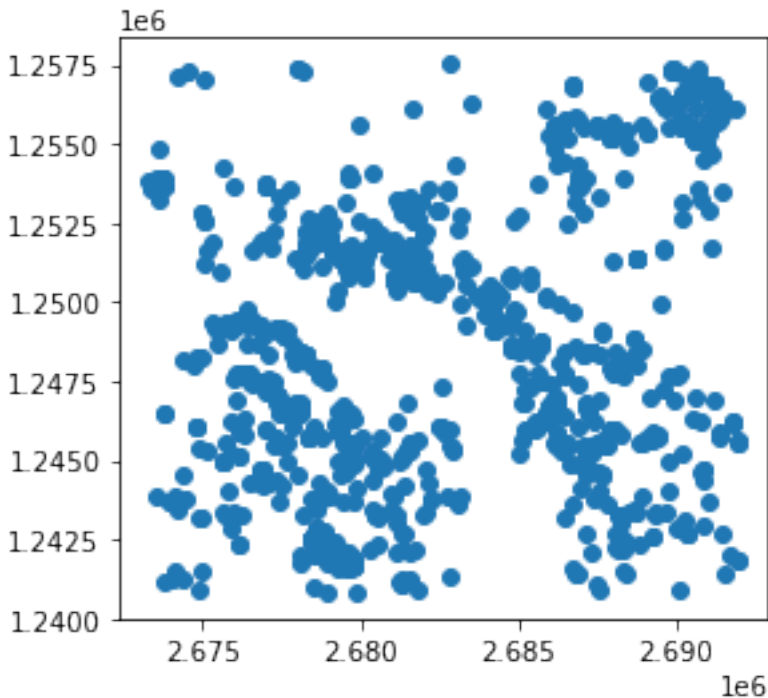
|      | ID   | accuracy   | x         | y         | geometry                        |
|------|------|------------|-----------|-----------|---------------------------------|
| 0    | 0    | 65.617154  | 2678971.0 | 1240824.0 | POINT (2678971.000 1240824.000) |
| 1    | 1    | 257.492052 | 2679837.0 |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

```
[1076 rows x 5 columns]
```

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

```
zeckenstiche.plot()
```

```
<AxesSubplot:>
```



## 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:

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

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

```
zeckenstiche.crs
```

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

(continues on next page)

(continued from previous page)

```
- 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.

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



## WALDANTEIL BERECHNEN

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

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

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

```
import pandas as pd
import geopandas as gpd

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

monte_carlo_df

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

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

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

monte_carlo_sjoin = gpd.sjoin(wald, monte_carlo_df)

monte_carlo_sjoin.head()
```

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

				geometry	index_right	ID	Run_Nr
0	POLYGON	Z	((2689962.355 1245335.250 644.591, 2...		51866	218	48
0	POLYGON	Z	((2689962.355 1245335.250 644.591, 2...		50790	218	47
0	POLYGON	Z	((2689962.355 1245335.250 644.591, 2...		14206	218	13
0	POLYGON	Z	((2689962.355 1245335.250 644.591, 2...		7750	218	7
0	POLYGON	Z	((2689962.355 1245335.250 644.591, 2...		26042	218	24

## 18.2 Übung 2: Anteil der Punkte im Wald

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

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

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

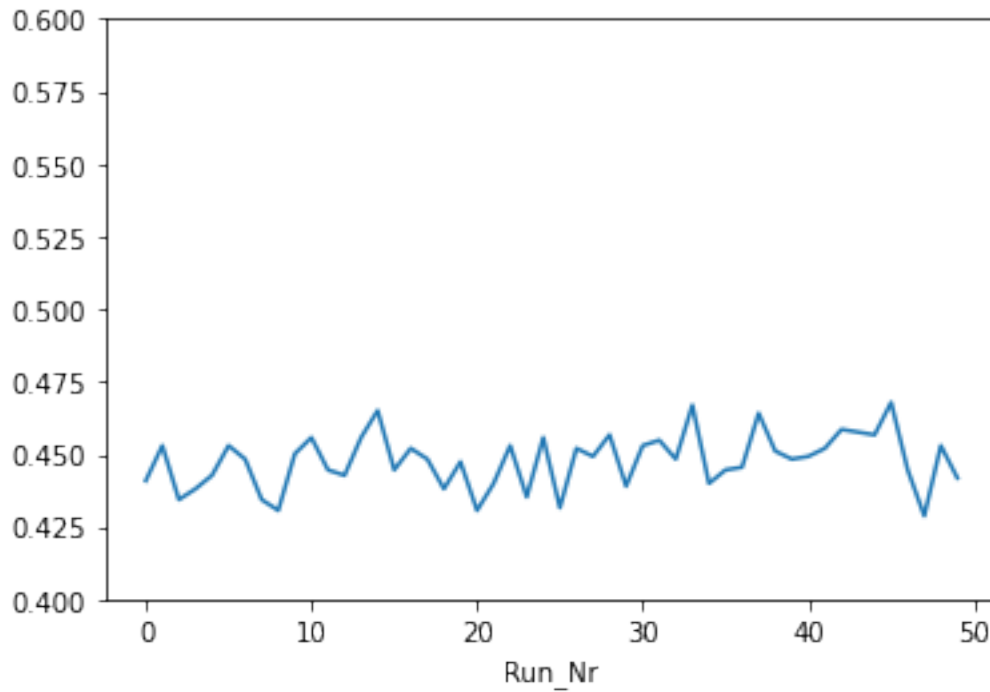
```
mittelwerte = monte_carlo_sjoin.groupby("Run_Nr").size()/1075
```

## 18.3 Übung 3: Mittelwerte Visualisieren

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

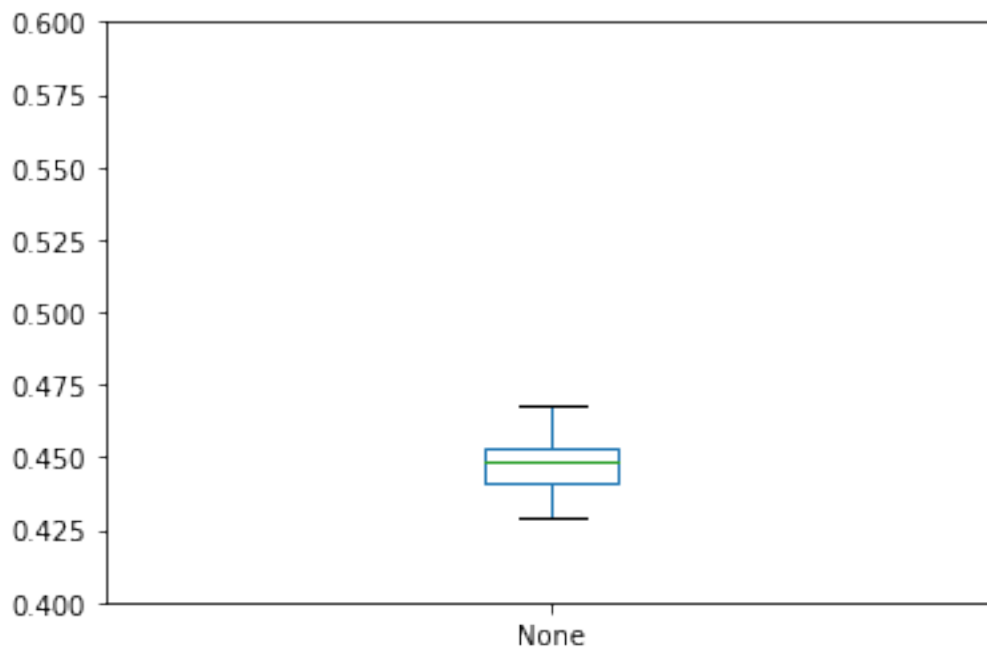
```
mittelwerte.plot().set_ylim(0.4,0.6)
```

```
(0.4, 0.6)
```



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

```
(0.4, 0.6)
```



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

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

