12. A complete example

In [1]:
```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt

import seaborn as sns
```

We have seen now most of the basic features of Pandas including importing data, combining dataframes, aggregating information and plotting it. In this chapter, we are going to re-use these concepts with the real data seen in the introduction chapter (06-DA_Pandas_introduction.ipynb). We are also going to explore some more advanced plotting libraries that exploit to the maximum dataframe structures.

12.1 Importing data

We are importing here two tables provided openly by the Swiss National Science Foundation. One contains a list of all *projects* to which funds have been allocated since 1975. The other table contains a list of all *people* to which funds have been awarded during the same period:

In [7]:
```
# local import
projects = pd.read_csv('Data/P3_GrantExport.csv',sep = ';')
persons = pd.read_csv('Data/P3_PersonExport.csv',sep = ';')

# import from url
#projects = pd.read_csv('http://p3.snf.ch/P3Export/P3_GrantExport.csv',sep =
';')
#persons = pd.read_csv('http://p3.snf.ch/P3Export/P3_PersonExport.csv',sep =
';')
```

We can have a brief look at both tables:

In [8]:
```
projects.head(5)
```

Out[8]:

	Project Number	Project Number String	Project Title	Project Title English	Responsible Applicant	Funding Instrument	Funding Instrument Hierarchy	
0	1	1000-000001	Schlussband (Bd. VI) der Jacob Burckhardt-Biog...	NaN	Kaegi Werner	Project funding (Div. I-III)	Project funding	
1	4	1000-000004	Batterie de tests à l'usage des enseignants po...	NaN	Massarenti Léonard	Project funding (Div. I-III)	Project funding	Psych Scier
2	5	1000-000005	Kritische Erstausgabe der 'Evidentiae contra D...	NaN	Kommission für das Corpus philosophorum medii ...	Project funding (Div. I-III)	Project funding	Komm philoso
3	6	1000-000006	Katalog der datierten Handschriften in der Sch...	NaN	Burckhardt Max	Project funding (Div. I-III)	Project funding	Hanc Alte Dr
4	7	1000-000007	Wissenschaftliche Mitarbeit am Thesaurus Lingu...	NaN	Schweiz. Thesauruskommission	Project funding (Div. I-III)	Project funding	Thesauru

In [9]:
```
persons.head(5)
```

Out[9]:

	Last Name	First Name	Gender	Institute Name	Institute Place	Person ID SNSF	OCRID	Projects as responsible Applicant	Projects as Applicant	Projects as Partner	Projec a Practi Partn
0	a Marca	Davide	male	NaN	NaN	53856	NaN	NaN	NaN	NaN	Na
1	a Marca	Andrea	male	NaN	NaN	132628	NaN	67368	NaN	NaN	Na
2	A. Jafari	Golnaz	female	Universität Luzern	Luzern	747886	NaN	191432	NaN	NaN	Na
3	Aaberg	Johan	male	NaN	NaN	575257	NaN	NaN	NaN	NaN	Na
4	Aahman	Josefin	female	NaN	NaN	629557	NaN	NaN	NaN	NaN	Na

We see that the `persons` table gives information such as the role of a person in various projects (applicant, employee etc.), her/his gender etc. The *project* table on the other side gives information such as the period of a grant, how much money was awarded etc.

What if we now wish to know for example:

- How much money is awarded on average depending on gender?
- How long does it typically take for a researcher to go from employee to applicant status on a grant?

We need a way to *link* the two tables, i.e. create a large table where *each row* corresponds to a single *observation* containing information from the two tables such as: applicant, gender, awarded funds, dates etc. We will now go through all necessary steps to achieve that goal.

12.2 Merging tables

If each row of the persons table contained a single observation with a single person and a single project (the same person would appear of course multiple times), we could just *join* the two tables based e.g. on the project ID. Unfortunately, in the persons table, each line corresponds to a *single researcher* with all projects IDs lumped together in a list. For example:

In [12]:
```
persons.iloc[10041]
```

Out[12]:
```
Last Name
Bodenmann
First Name
Guy
Gender
male
Institute Name                        Lehrstuhl für Klinische Psychologie Kind
er/Jug...
Institute Place
Zürich
Person ID SNSF
47670
OCRID                                                                0000-0003-
0964-6409
Projects as responsible Applicant    46820;56660;62901;109547;115948;128960;1
29627;...
Projects as Applicant                                                112141;1220
90;166348
Projects as Partner
NaN
Projects as Practice Partner
NaN
Projects as Employee
62901
Projects as Contact Person
NaN
Name: 10041, dtype: object
```

In [13]:
```
persons.iloc[10041]['Projects as responsible Applicant']
```

Out[13]:
```
'46820;56660;62901;109547;115948;128960;129627;129699;133004;146775;147634;17
3270'
```

Therefore the first thing we need to do is to split those strings into actual lists. We can do that by using classic Python string splitting. We simply `apply` that function to the relevant columns. We need to take care of rows containing NaNs on which we cannot use `split()`. We create two series, one for applicants, one for employees:

In [14]:
```
projID_a = persons['Projects as responsible Applicant'].apply(lambda x: x.sp
lit(';') if not pd.isna(x) else np.nan)
projID_e = persons['Projects as Employee'].apply(lambda x: x.split(';') if n
ot pd.isna(x) else np.nan)
```

In [15]:
```
projID_a
```

Out[15]:
```
0                                               NaN
1                                           [67368]
2                                          [191432]
3                                               NaN
4                                               NaN
                              ...                 
110811    [52821, 143769, 147153, 165510, 183584]
110812                                          NaN
110813                                          NaN
110814                                          NaN
110815                                          NaN
Name: Projects as responsible Applicant, Length: 110816, dtype: object
```

In [17]:
```
projID_a[10041]
```

Out[17]:
```
['46820',
 '56660',
 '62901',
 '109547',
 '115948',
 '128960',
 '129627',
 '129699',
 '133004',
 '146775',
 '147634',
 '173270']
```

Now, to avoid problems later we'll only keep rows that are not NaNs. We first add the two series to the dataframe and then remove NaNs:

In [18]:
```
pd.isna(projID_a)
```

Out[18]:
```
0          True
1         False
2         False
3          True
4          True
          ...  
110811    False
110812     True
110813     True
110814     True
110815     True
Name: Projects as responsible Applicant, Length: 110816, dtype: bool
```

In [19]:
```
applicants = persons.copy()
applicants['projID'] = projID_a
applicants = applicants[~pd.isna(projID_a)]

employees = persons.copy()
employees['projID'] = projID_e
employees = employees[~pd.isna(projID_e)]
```

Now we want each of these projects to become a single line in the dataframe. Here we use a function that we haven't used before called `explode` which turns every element in a list into a row (a good illustration of the variety of available functions in Pandas):

In [20]:
```
applicants = applicants.explode('projID')
employees = employees.explode('projID')
```

In [21]:
```
applicants.head(5)
```

Out[21]:

	Last Name	First Name	Gender	Institute Name	Institute Place	Person ID SNSF	OCRID	Projects as responsible Applicant	Projects as Applicant	Projects as Partner
1	a Marca	Andrea	male	NaN	NaN	132628	NaN	67368	NaN	NaN
2	A. Jafari	Golnaz	female	Universität Luzern	Luzern	747886	NaN	191432	NaN	NaN
7	Aapro	Matti S.	male	Clinique de Genolier F.M.H. Oncologie-Hématolo...	Genolier	3268	NaN	8532;9513	8155	NaN
7	Aapro	Matti S.	male	Clinique de Genolier F.M.H. Oncologie-Hématolo...	Genolier	3268	NaN	8532;9513	8155	NaN
11	Aas	Gregor	male	Lehrstuhl für Pflanzenphysiologie Universität ...	Bayreuth	36412	NaN	52037	NaN	NaN

So now we have one large table, where each row corresponds to a *single* applicant and a *single* project. We can finally do our merging operation where we combined information on persons and projects. We will do two such operations: one for applicants using the `projID_a` column for merging and one using the `projID_e` column. We have one last problem to fix:

In [22]:
```
applicants.loc[1].projID
```

Out[22]: '67368'

In [23]:
```
projects.loc[1]['Project Number']
```

Out[23]: 4

We need the project ID in the persons table to be a *number* and not a *string*. We can try to convert but get an error:

In [24]:
```
applicants.projID = applicants.projID.astype(int)
employees.projID = employees.projID.astype(int)
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-24-fca9460da04e> in <module>
----> 1 applicants.projID = applicants.projID.astype(int)
      2 employees.projID = employees.projID.astype(int)

~/miniconda3/envs/danalytics/lib/python3.8/site-packages/pandas/core/generic.
py in astype(self, dtype, copy, errors)
   5696         else:
   5697             # else, only a single dtype is given
-> 5698             new_data = self._data.astype(dtype=dtype, copy=copy, erro
rs=errors)
   5699             return self._constructor(new_data).__finalize__(self)
   5700

~/miniconda3/envs/danalytics/lib/python3.8/site-packages/pandas/core/internal
s/managers.py in astype(self, dtype, copy, errors)
    580
    581     def astype(self, dtype, copy: bool = False, errors: str = "rais
e"):
--> 582         return self.apply("astype", dtype=dtype, copy=copy, errors=er
rors)
    583
    584     def convert(self, **kwargs):

~/miniconda3/envs/danalytics/lib/python3.8/site-packages/pandas/core/internal
s/managers.py in apply(self, f, filter, **kwargs)
    440                 applied = b.apply(f, **kwargs)
    441             else:
--> 442                 applied = getattr(b, f)(**kwargs)
    443             result_blocks = _extend_blocks(applied, result_blocks)
    444

~/miniconda3/envs/danalytics/lib/python3.8/site-packages/pandas/core/internal
s/blocks.py in astype(self, dtype, copy, errors)
    623             vals1d = values.ravel()
    624             try:
--> 625                 values = astype_nansafe(vals1d, dtype, copy=True)
    626             except (ValueError, TypeError):
    627                 # e.g. astype_nansafe can fail on object-dtype of str
ings

~/miniconda3/envs/danalytics/lib/python3.8/site-packages/pandas/core/dtypes/c
ast.py in astype_nansafe(arr, dtype, copy, skipna)
    872         # work around NumPy brokenness, #1987
    873         if np.issubdtype(dtype.type, np.integer):
--> 874             return lib.astype_intsafe(arr.ravel(), dtype).reshape(ar
r.shape)
    875
    876         # if we have a datetime/timedelta array of objects

pandas/_libs/lib.pyx in pandas._libs.lib.astype_intsafe()

ValueError: invalid literal for int() with base 10: ''
```

It looks like we have a row that doesn't conform to expectation and only contains ''. Let's try to figure out what happened. First we find the location with the issue:

In [25]:
```
applicants[applicants.projID=='']
```

Out[25]:

	Last Name	First Name	Gender	Institute Name	Institute Place	Person ID SNSF	OCRID	Projects as responsible Applicant	Projects as Applicant	Proje Part
50947	Kleinewefers	Henner	male	Séminaire de politique économique, d'économie ...	Fribourg	10661	NaN	8;	NaN	N
62384	Massarenti	Léonard	male	Faculté de Psychologie et des Sciences de l'Ed...	Genève 4	11138	NaN	4;	NaN	N

Then we look in the original table:

In [26]:
```
persons.loc[50947]
```

Out[26]:
```
Last Name                                                         Kle
inewefers
First Name
Henner
Gender
male
Institute Name                     Séminaire de politique économique, d'éco
nomie ...
Institute Place
Fribourg
Person ID SNSF
10661
OCRID
NaN
Projects as responsible Applicant
8;
Projects as Applicant
NaN
Projects as Partner
NaN
Projects as Practice Partner
NaN
Projects as Employee
NaN
Projects as Contact Person
NaN
Name: 50947, dtype: object
```

Unfortunately, as is often the case, we have a misformatting in the original table. The project as applicant entry has a single number but still contains the `;` sign. Therefore when we split the text, we end up with `['8','']`. Can we fix this? We can for example filter the table and remove rows where `projID` has length 0:

In [30]:
```
applicants = applicants[applicants.projID.apply(lambda x: len(x) > 0)]
employees = employees[employees.projID.apply(lambda x: len(x) > 0)]
```

Now we can convert the `projID` column to integer:

In [31]:
```
applicants.projID = applicants.projID.astype(int)
employees.projID = employees.projID.astype(int)
```

Finally we can use `merge` to combine both tables. We will combine the projects (on 'Project Number') and persons table (on 'projID_a' and 'projID_e'):

In [32]:
```
merged_appl = pd.merge(applicants, projects, left_on='projID', right_on='Pro
ject Number')
merged_empl = pd.merge(employees, projects, left_on='projID', right_on='Proj
ect Number')
```

In [33]:
```
applicants.head(5)
```

Out[33]:

	Last Name	First Name	Gender	Institute Name	Institute Place	Person ID SNSF	OCRID	Projects as responsible Applicant	Projects as Applicant	Projects as Partner
1	a Marca	Andrea	male	NaN	NaN	132628	NaN	67368	NaN	NaN
2	A. Jafari	Golnaz	female	Universität Luzern	Luzern	747886	NaN	191432	NaN	NaN
7	Aapro	Matti S.	male	Clinique de Genolier F.M.H. Oncologie-Hématolo...	Genolier	3268	NaN	8532;9513	8155	NaN
7	Aapro	Matti S.	male	Clinique de Genolier F.M.H. Oncologie-Hématolo...	Genolier	3268	NaN	8532;9513	8155	NaN
11	Aas	Gregor	male	Lehrstuhl für Pflanzenphysiologie Universität ...	Bayreuth	36412	NaN	52037	NaN	NaN

12.3 Reformatting columns: time

We now have in those tables information on both scientists and projects. Among other things we now when each project of each scientist has started via the `Start Date` column:

In [34]:
```
merged_empl['Start Date']
```

Out[34]:
```
0         01.04.1993
1         01.04.1993
2         01.04.1993
3         01.04.1993
4         01.04.1993
             ...
127126    01.04.1990
127127    01.04.1991
127128    01.11.1998
127129    01.11.1992
127130    01.10.2008
Name: Start Date, Length: 127131, dtype: object
```

If we want to do computations with dates (e.g. measuring time spans) we have to change the type of the column. Currently it is indeed just a string. We could parse that string, but Pandas already offers tools to handle dates. For example we can use `pd.to_datetime` to transform the string into a Python `datetime` format. Let's create a new `date` column:

In [35]:
```
merged_empl['date'] = pd.to_datetime(merged_empl['Start Date'])
merged_appl['date'] = pd.to_datetime(merged_appl['Start Date'])
```

In [36]:
```
merged_empl.iloc[0]['date']
```

Out[36]:
```
Timestamp('1993-01-04 00:00:00')
```

In [37]:
```
merged_empl.iloc[0]['date'].year
```

Out[37]:
```
1993
```

Let's add a year column to our dataframe:

In [38]:
```
merged_empl['year'] = merged_empl.date.apply(lambda x: x.year)
merged_appl['year'] = merged_appl.date.apply(lambda x: x.year)
```

12.4 Completing information

As we did in the introduction, we want to be able to broadly classify projects into three categories. We therefore search for a specific string ('Humanities', 'Mathematics','Biology') within the 'Discipline Name Hierarchy' column to create a new column called 'Field'^:

In [39]:
```
science_types = ['Humanities', 'Mathematics','Biology']
merged_appl['Field'] = merged_appl['Discipline Name Hierarchy'].apply(
    lambda el: next((y for y in [x for x in science_types if x in el] if y i
s not None),None) if not pd.isna(el) else el)
```

We will use the amounts awarded in our analysis. Let's look at that column:

In [40]:
```
merged_appl['Approved Amount']
```

Out[40]:
```
0                     20120.00
1        data not included in P3
2                    211427.00
3                    174021.00
4                      8865.00
                   ...
74650                150524.00
74651                346000.00
74652                262960.00
74653                449517.00
74654               1433628.00
Name: Approved Amount, Length: 74655, dtype: object
```

Problem: we have rows that are not numerical. Let's coerce that column to numerical:

In [41]:
```
merged_appl['Approved Amount'] = pd.to_numeric(merged_appl['Approved Amount
'], errors='coerce')
```

In [42]:
```
merged_appl['Approved Amount']
```

Out[42]:
```
0          20120.0
1              NaN
2         211427.0
3         174021.0
4           8865.0
           ...
74650     150524.0
74651     346000.0
74652     262960.0
74653     449517.0
74654    1433628.0
Name: Approved Amount, Length: 74655, dtype: float64
```

12.5 Data anaylsis

We are finally done tidying up our tables so that we can do proper data analysis. We can *aggregate* data to answer some questions.

12.5.1 Amounts by gender

Let's see for example what is the average amount awarded every year, split by gender. We keep only the 'Project funding' category to avoid obscuring the results with large funds awarded for specific projects (PNR etc):

In [44]:
```
merged_projects = merged_appl[merged_appl['Funding Instrument Hierarchy'] ==
'Project funding']
```

In [45]:
```
grouped_gender = merged_projects.groupby(['Gender','year'])['Approved Amount
'].mean().reset_index()
grouped_gender
```

Out[45]:

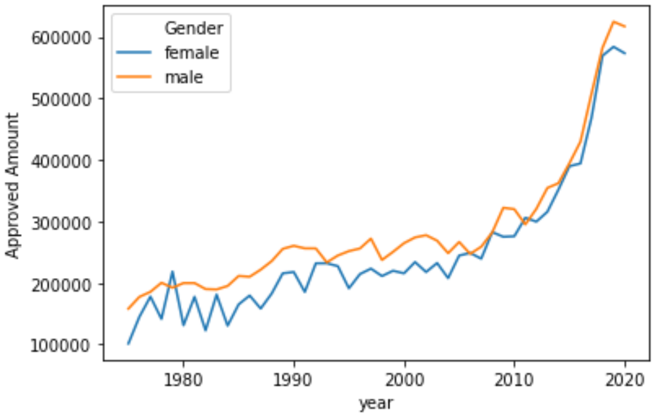
	Gender	year	Approved Amount
0	female	1975.0	101433.200000
1	female	1976.0	145017.750000
2	female	1977.0	177826.157895
3	female	1978.0	141489.857143
4	female	1979.0	218496.904762
...	...	...	...
87	male	2016.0	429717.055907
88	male	2017.0	507521.397098
89	male	2018.0	582461.020513
90	male	2019.0	624826.387985
91	male	2020.0	617256.523404

92 rows × 3 columns

To generate a plot, we use here Seaborn which uses some elements of a grammar of graphics. For example we can assign variables to each "aspect" of our plot. Here x and y axis are year and amount while color ('hue') is the gender. In one line, we can generate a plot that compiles all the information:

In [46]:
```
sns.lineplot(data = grouped_gender, x='year', y='Approved Amount', hue='Gend
er')
```

Out[46]: <matplotlib.axes._subplots.AxesSubplot at 0x122c5d0d0>



There seems to be a small but systematic difference in the average amount awarded.

We can now use a plotting library that is essentially a Python port of ggplot to add even more complexity to this plot. For example, let's split the data also by Field:

In [47]:
```
import plotnine as p9
```

In [48]:
```
grouped_gender_field = merged_projects.groupby(['Gender','year','Field'])['A
pproved Amount'].mean().reset_index()
```

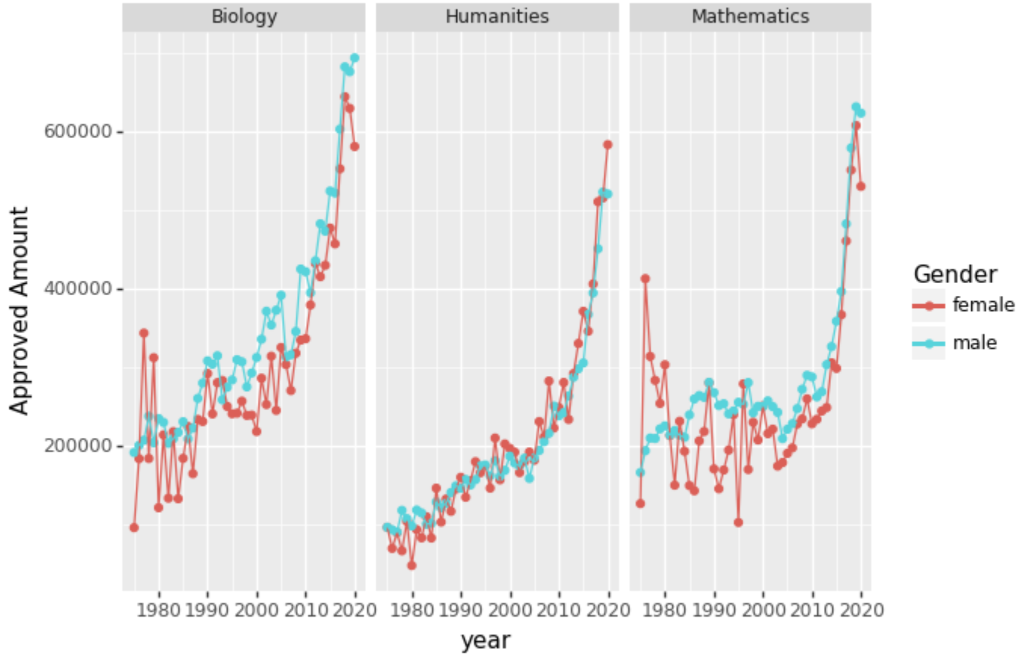
In [49]:
```
grouped_gender_field
```

Out[49]:

	Gender	year	Field	Approved Amount
0	female	1975.0	Biology	95049.000000
1	female	1975.0	Humanities	95451.666667
2	female	1975.0	Mathematics	125762.000000
3	female	1976.0	Biology	183154.200000
4	female	1976.0	Humanities	68590.750000
...	...	...	...	...
271	male	2019.0	Humanities	523397.013072
272	male	2019.0	Mathematics	632188.796040
273	male	2020.0	Biology	694705.243590
274	male	2020.0	Humanities	520925.507246
275	male	2020.0	Mathematics	624141.068182

276 rows × 4 columns

In [50]:
```
(p9.ggplot(grouped_gender_field, p9.aes('year', 'Approved Amount', color='Gender'))
 + p9.geom_point()
 + p9.geom_line()
 + p9.facet_wrap('~Field'))
```



Out[50]: <ggplot: (305412337)>

12.5.2 From employee to applicant

One of the questions we wanted to answer above was how much time goes by between the first time a scientist is mentioned as "employee" on an application and the first time he applies as main applicant. We have therefore to:

1. Find all rows corresponding to a specific scientist
2. Find the earliest date of project

For (1) we can use `groupby` and use the `Person ID SNSF` ID which is a unique ID assigned to each researcher. Once this *aggregation* is done, we can summarize each group by looking for the "minimal" date:

In [51]:
```
first_empl = merged_empl.groupby('Person ID SNSF').date.min().reset_index()
first_appl = merged_appl.groupby('Person ID SNSF').date.min().reset_index()
```

We have now two dataframes indexed by the `Person ID` :

In [52]:
```
first_empl.head(5)
```

Out[52]:

	Person ID SNSF	date
0	1611	1990-01-10
1	1659	1988-01-11
2	1661	1978-01-07
3	1694	1978-01-06
4	1712	1982-01-04

Now we can again merge the two series to be able to compare applicant/employee start dates for single people:

In [53]:
```
merge_first = pd.merge(first_appl, first_empl, on = 'Person ID SNSF', suffix
es=('_appl', '_empl'))
```

In [54]:
```
merge_first
```

Out[54]:

	Person ID SNSF	date_appl	date_empl
0	1659	1975-01-10	1988-01-11
1	1661	1978-01-07	1978-01-07
2	1694	1985-01-01	1978-01-06
3	1712	1982-01-04	1982-01-04
4	1726	1985-01-03	1985-01-03
...	...	...	...
10336	748652	2019-01-12	2019-01-12
10337	748760	2020-01-03	2020-01-03
10338	749430	2020-01-04	2020-01-04
10339	749991	2020-01-03	2020-01-03
10340	750593	2020-01-01	2020-01-01

10341 rows × 3 columns

Finally we merge with the full table, based on the index to recover the other paramters:

In [55]:
```
full_table = pd.merge(merge_first, merged_appl,on = 'Person ID SNSF')
```

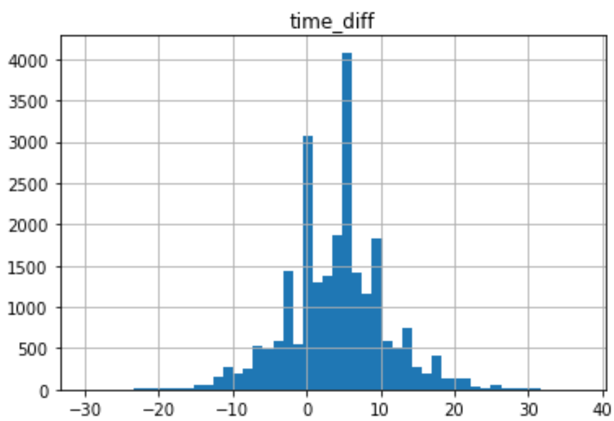
Finally we can add a column to that dataframe as a "difference in dates":

In [56]:
```
full_table['time_diff'] = full_table.date_appl-full_table.date_empl
```

In [57]:
```
full_table.time_diff = full_table.time_diff.apply(lambda x: x.days/365)
```

In [58]:
```
full_table.hist(column='time_diff',bins = 50)
```

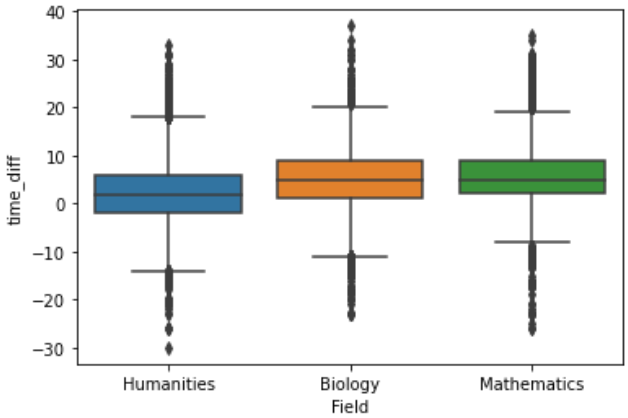
Out[58]:
```
array([[<matplotlib.axes._subplots.AxesSubplot object at 0x12ba24970>]],
      dtype=object)
```



We see that we have one strong peak at $\Delta T == 0$ which corresponds to people who were paid for the first time through an SNSF grant when they applied themselves. The remaining cases have a peak around $\Delta T == 5$ which typically corresponds to the case where a PhD student was payed on a grant and then applied for a postdoc grant ~4-5 years later.

We can go further and ask how dependent this waiting time is on the Field of research. Obviously Humanities are structured very differently

In [60]:
```
sns.boxplot(data=full_table, y='time_diff', x='Field');
```



In [61]:
```
sns.violinplot(data=full_table, y='time_diff', x='Field', );
```