All the techniques of handling ,missing values of numerical as well as categorical data -Days 2 to 3

1. Mean/ Median/Mode replacement
2. Random Sample Imputation
3. Capturing NAN values with a new feature
4. End of Distribution imputation
5. Arbitrary imputation
6. Frequent categories imputation

2.Random Sample Imputation

Aim: Random sample imputation consists of taking random observation from the dataset and we use this observation to replace the nan values.
Instead of imputing missing values with a statistical measure (like the mean or median), Random Sample Imputation replaces missing values with a randomly sampled value from the observed data. This can help preserve the distribution and variability in the dataset.

When should it be used?
It assumes that the data are missing completely at random(MCAR)

In [42]:
```
import pandas as pd
df=pd.read_csv('titanic.csv', usecols=['Age','Fare','Survived'])
df.head()
```

Out[42]:

	Survived	Age	Fare
0	0	22.0	7.2500
1	1	38.0	71.2833
2	1	26.0	7.9250
3	1	35.0	53.1000
4	0	35.0	8.0500

In [43]:
```
df.isnull().sum()
```

Out[43]:
```
Survived      0
Age         177
Fare          0
dtype: int64
```

In [44]:
```
df.isnull().mean()
```

Out[44]:
```
Survived    0.000000
Age         0.198653
Fare        0.000000
dtype: float64
```

In [66]:
```
df['Age'].isnull().sum()
```

Out[66]:
```
177
```

In [82]:
```
df['Age'].dropna() ## drop all missing values
```

Out[82]:
```
0      22.0
1      38.0
2      26.0
3      35.0
4      35.0
       ... 
885    39.0
886    27.0
887    19.0
889    26.0
890    32.0
Name: Age, Length: 714, dtype: float64
```

In [84]:
```
df['Age'].dropna().sample(df['Age'].isnull().sum(),random_state=0) # filled
```

Out[84]:
```
423    28.00
177    50.00
305     0.92
292    36.00
889    26.00
       ... 
539    22.00
267    25.00
352    15.00
99     34.00
689    15.00
Name: Age, Length: 177, dtype: float64
```

In [92]:
```
df[df['Age'].isnull()].index  # find all index of missing value of Age
```

Out[92]:
```
Int64Index([  5,  17,  19,  26,  28,  29,  31,  32,  36,  42,
            ...
            832, 837, 839, 846, 849, 859, 863, 868, 878, 888],
           dtype='int64', length=177)
```

In [ ]:
```

```

In [134]:
```
def impute_nan(df,variable,median):
    df[variable+"_median"]=df[variable].fillna(median) ## fill missing valu
    df[variable+"_random"]=df[variable]
    ## it will have the random sample to fill the missing values
    random_sample=df[variable].dropna().sample(df[variable].isnull().sum(),
    ## random sample and variable have same index in order to marge the dat
    random_sample.index=df[df[variable].isnull()].index
    df.loc[df[variable].isnull(),variable+'_random']=random_sample
```

In [135]:
```
df[df['Age'].isnull()].index
```

Out[135]:
```
Int64Index([  5,  17,  19,  26,  28,  29,  31,  32,  36,  42,
            ...
            832, 837, 839, 846, 849, 859, 863, 868, 878, 888],
           dtype='int64', length=177)
```

In [136]:
```
df['Age_random'].isnull().sum()
```

Out[136]:
```
0
```

In [137]:
```
median=df['Age'].median()
```

In [141]:
```
impute_nan(df,'Age',median) ## calling function
```

In [142]:
```
df.head()
```

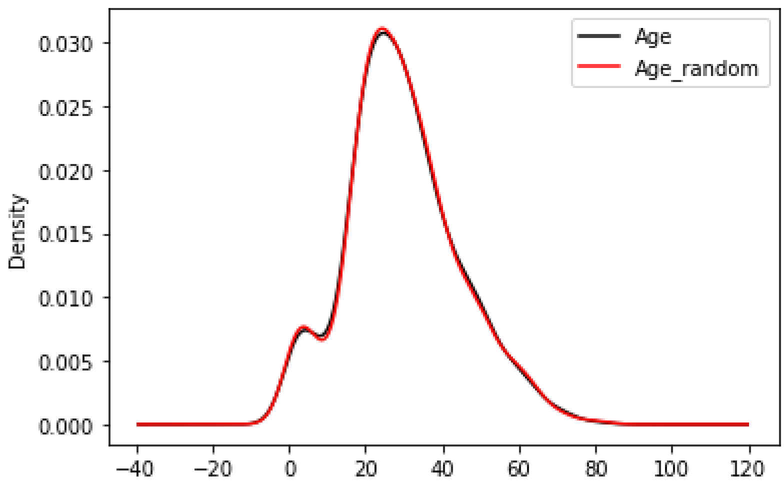
Out[142]:

	Survived	Age	Fare	Age_median	Age_random
0	0	22.0	7.2500	22.0	22.0
1	1	38.0	71.2833	38.0	38.0
2	1	26.0	7.9250	26.0	26.0
3	1	35.0	53.1000	35.0	35.0
4	0	35.0	8.0500	35.0	35.0

In [143]:
```
import matplotlib.pyplot as plt
%matplotlib inline
```

In [154]:
```
fig = plt.figure()
ax = fig.add_subplot(111)
df['Age'].plot(kind='kde', ax=ax, color='black')
#df.Age_median.plot(kind='kde', ax=ax, color='blue')
df.Age_random.plot(kind='kde', ax=ax, color='red')
lines, labels = ax.get_legend_handles_labels()
ax.legend(lines, labels, loc='best')
```

Out[154]:
```
<matplotlib.legend.Legend at 0x27b102434c0>
```



Advantages

Easy To implement

There is less distortion in variance

Disadvantage

Every situation randomness wont work

3.Capturing NAN values with a new feature

It works well if the data are not missing completely at random

Instead of simply imputing missing values, you can create a new feature to indicate the presence or absence of missing data (NaN). This approach can be valuable if the fact that data is missing carries some information.

In [158]:
```
df=pd.read_csv('titanic.csv', usecols=['Age','Fare','Survived'])
df.head()
```

Out[158]:

	Survived	Age	Fare
0	0	22.0	7.2500
1	1	38.0	71.2833
2	1	26.0	7.9250
3	1	35.0	53.1000
4	0	35.0	8.0500

Create a binary indicator column (0 or 1) that marks whether a value is missing or not.

In [171]:
```
import numpy as np
df['Age_NAN']=np.where(df['Age'].isnull(),1,0)
```

In [172]:
```
df.head()
```

Out[172]:

	Survived	Age	Fare	Age_NAN
0	0	22.0	7.2500	0
1	1	38.0	71.2833	0
2	1	26.0	7.9250	0
3	1	35.0	53.1000	0
4	0	35.0	8.0500	0

In [173]:
```
df.Age.median()
```
Out[173]: 29.69911764705882

In [174]:
```
df['Age'].fillna(df.Age.median(),inplace=True)
```

In [176]:
```
df.head()
```
Out[176]:

	Survived	Age	Fare	Age_NAN
0	0	22.0	7.2500	0
1	1	38.0	71.2833	0
2	1	26.0	7.9250	0
3	1	35.0	53.1000	0
4	0	35.0	8.0500	0

Advantages:

1.Easy to implement
2.Captures the importance of missing values

Disadvantages

Creating Additional Features(Curse of Dimensionality)

4.End of Distribution imputation

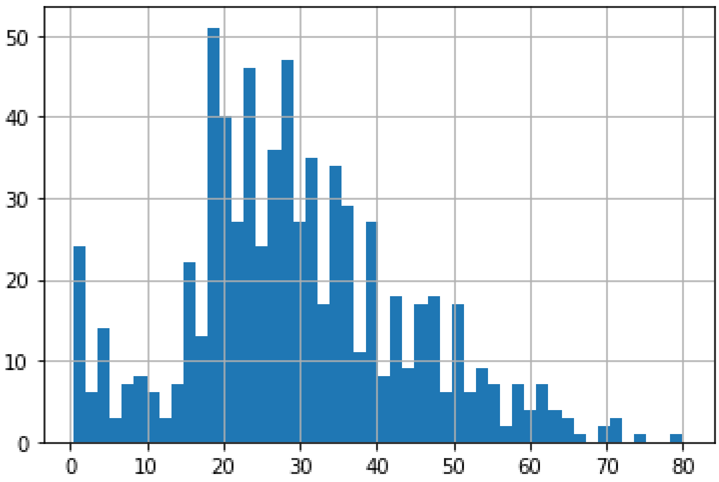
This method involves replacing missing values with a value from the end of the distribution of the data, such as the minimum or maximum value, or a value from a chosen quantile. It's typically used when you want to avoid introducing bias through mean/median imputation but still want to fill in the missing data.

In [177]:
```
df=pd.read_csv('titanic.csv', usecols=['Age','Fare','Survived'])
df.head()
```
Out[177]:

	Survived	Age	Fare
0	0	22.0	7.2500
1	1	38.0	71.2833
2	1	26.0	7.9250
3	1	35.0	53.1000
4	0	35.0	8.0500

In [178]:
```
df.Age.hist(bins=50)
```

Out[178]: <AxesSubplot: >

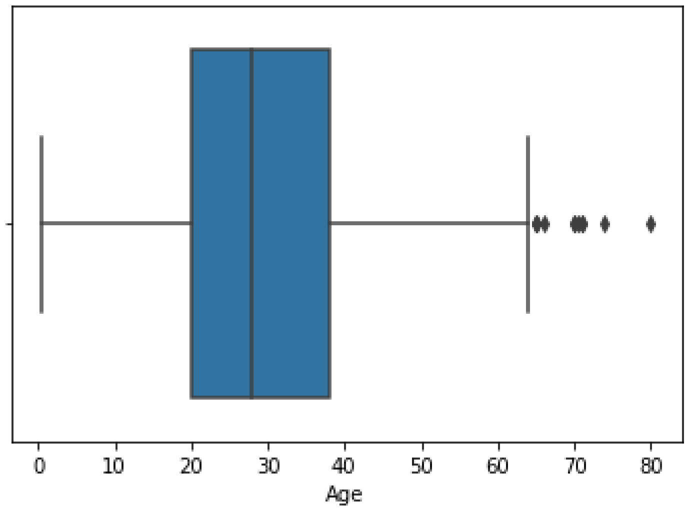


In [185]:
```
extreme=df.Age.mean()+3*df.Age.std()
```

In [182]:
```
import seaborn as sns
sns.boxplot('Age',data=df)
```

```
G:\New folder\lib\site-packages\seaborn\_decorators.py:36: FutureWarning:
Pass the following variable as a keyword arg: x. From version 0.12, the on
ly valid positional argument will be `data`, and passing other arguments w
ithout an explicit keyword will result in an error or misinterpretation.
  warnings.warn(
```

Out[182]: <AxesSubplot: xlabel='Age'>



In [186]:
```
def impute_nan(df,variable,median,extreme):
    df[variable+"_end_distribution"]=df[variable].fillna(extreme)
    df[variable].fillna(median,inplace=True)  ## this for compare
```

In [187]:
```
impute_nan(df,'Age',df.Age.median(),extreme)
```

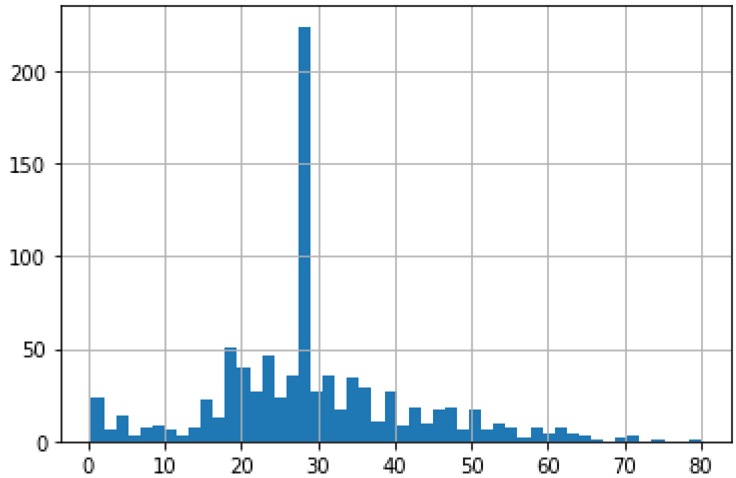
In [188]:
```
df.head()
```

Out[188]:

	Survived	Age	Fare	Age_end_distribution
0	0	22.0	7.2500	22.0
1	1	38.0	71.2833	38.0
2	1	26.0	7.9250	26.0
3	1	35.0	53.1000	35.0
4	0	35.0	8.0500	35.0

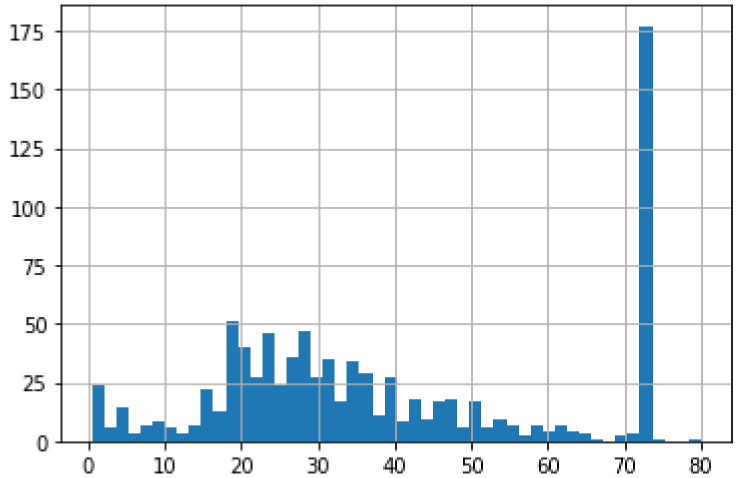
In [192]:
```
df['Age'].hist(bins=50)
```

Out[192]: <AxesSubplot: >



In [193]:
```
df['Age_end_distribution'].hist(bins=50)
```

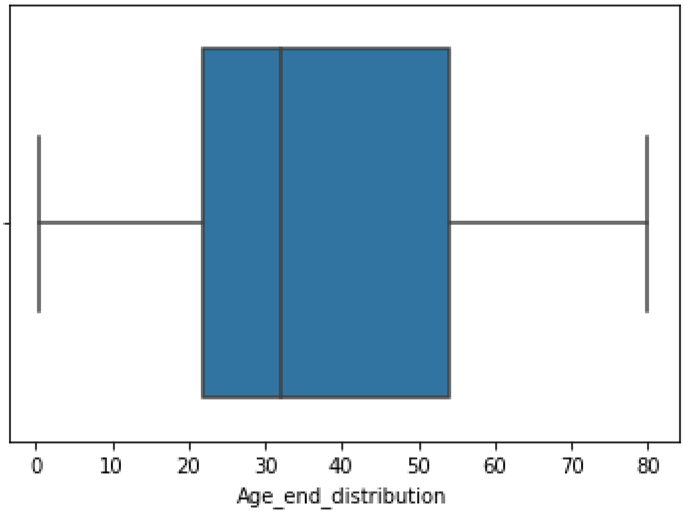
Out[193]: <AxesSubplot: >



In [194]:
```
sns.boxplot('Age_end_distribution',data=df)
```

```
G:\New folder\lib\site-packages\seaborn\_decorators.py:36: FutureWarning: Pass the following variable as a keyword arg: x. From version 0.12, the only valid positional argument will be `data`, and passing other arguments without an explicit keyword will result in an error or misinterpretation.
  warnings.warn(
```

Out[194]: <AxesSubplot: xlabel='Age_end_distribution'>



Now there is No outlier

Advantages:

Avoids the central tendency bias of mean/median imputation. Useful for skewed data or when you want to preserve extremes in the dataset.

Disadvantages:

May artificially distort the distribution by introducing values that aren't representative of the typical data points.

Arbitrary Value Imputation

This technique was derived from kaggle competition It consists of replacing NAN by an arbitrary value.
Replace missing values in categorical data with the most frequent category (mode)

In [1]:
```
import pandas as pd
```

In [4]:
```
df=pd.read_csv("titanic.csv",usecols=["Age","Fare","Survived"])
df.head()
```

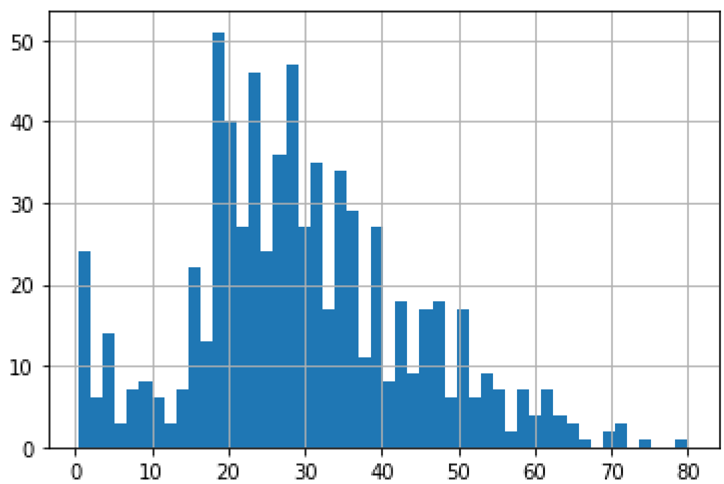
Out[4]:

	Survived	Age	Fare
0	0	22.0	7.2500
1	1	38.0	71.2833
2	1	26.0	7.9250
3	1	35.0	53.1000
4	0	35.0	8.0500

In [5]:
```
def impute_nan(df,variable):
    df[variable+'zero']=df[variable].fillna(0)
    df[variable+'_hundred']=df[variable].fillna(100)
```

In [6]:
```
df['Age'].hist(bins=50)
```

Out[6]: <AxesSubplot: >



In [ ]:
```
## Arbitrsry vslues
## 1. It should be more frequently present
```

Advantages

1.Easy to implement
2.Captures the importance of missingess if there is one

Disadvantages

1.Distorts the original distribution of the variable
2.If missingess is not important, it may mask the predictive power of the original variable by distorting its distribution
3.Hard to decide which value to use

In [ ]:

How To Handle Categroical Missing Values

Handling missing values in categorical data is an important aspect of data preprocessing in machine learning. Categorical features may have missing values due to various reasons, such as non-response in surveys, data entry errors, or not being applicable in certain cases. Below are some effective strategies for handling missing values in categorical data:

1. Frequent Category Imputation (Mode Imputation)

Description: Replace missing values with the most frequent category (the mode) in the column.
Use Cases: This is the most common and simplest approach for categorical data. It works well when the most frequent category is a reasonable replacement for the missing values

In [41]:
```
df=pd.read_csv('loan.csv',usecols=['BsmtQual','FireplaceQu','GarageType','S
```

In [42]:
```
df.head()
```

Out[42]:

	BsmtQual	FireplaceQu	GarageType	SalePrice
0	Gd	NaN	Attchd	208500
1	Gd	TA	Attchd	181500
2	Gd	TA	Attchd	223500
3	TA	Gd	Detchd	140000
4	Gd	TA	Attchd	250000

In [47]:
```
df.shape
```

Out[47]:
```
(1460, 4)
```

In [43]:
```
df.isnull().sum()
```

Out[43]:
```
BsmtQual        37
FireplaceQu    690
GarageType      81
SalePrice        0
dtype: int64
```

In [ ]:

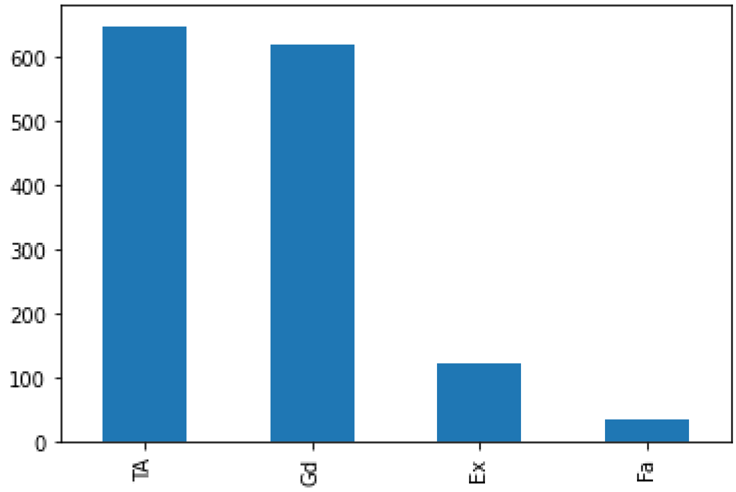
In [45]:
```
df.isnull().mean().sort_values(ascending=True)
```

Out[45]:
```
SalePrice      0.000000
BsmtQual       0.025342
GarageType     0.055479
FireplaceQu    0.472603
dtype: float64
```

compute the frequency with every feature

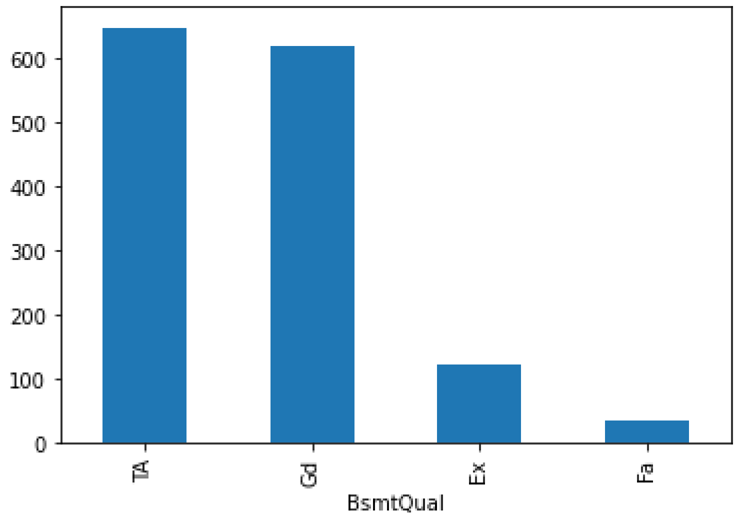
In [56]:
```
df['BsmtQual'].value_counts().plot.bar()
```

Out[56]: <AxesSubplot: >



In [53]:
```
df.groupby(['BsmtQual'])['BsmtQual'].count().sort_values(ascending=False).p
```

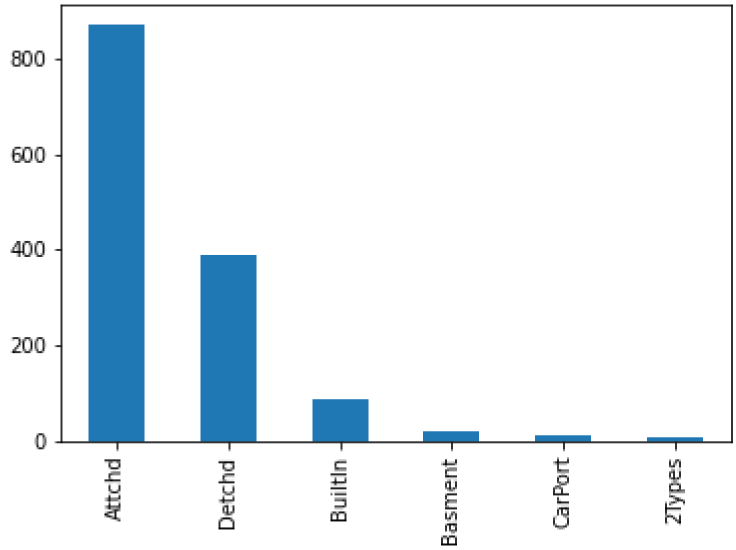
Out[53]: <AxesSubplot: xlabel='BsmtQual'>



In [ ]:

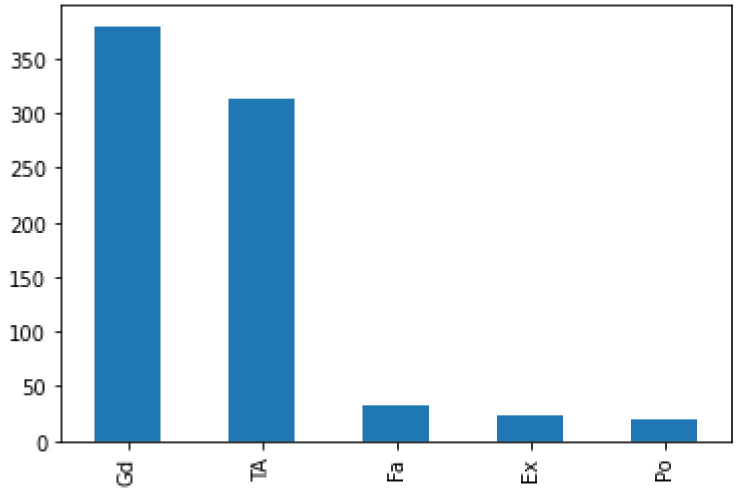
In [58]:
```
df['GarageType'].value_counts().plot.bar()
```

Out[58]: <AxesSubplot: >



In [59]:
```
df['FireplaceQu'].value_counts().plot.bar()
```

Out[59]: <AxesSubplot: >



In [63]:
```
df['GarageType'].value_counts().index[0]
```

Out[63]: 'Attchd'

In [64]:
```
### Relpacing function
def impute_nan(df,variable):
    most_frequent_category=df[variable].value_counts().index[0]
    df[variable].fillna(most_frequent_category,inplace=True)
```

In [ ]:
```
impute_nan(df,'GarageType')
```

In [66]:
```
for feature in ['BsmtQual','FireplaceQu','GarageType']:
    impute_nan(df,feature)
```

In [68]:
```
df.isnull().mean()
```

Out[68]:
```
BsmtQual       0.0
FireplaceQu    0.0
GarageType     0.0
SalePrice      0.0
dtype: float64
```

Advantages

1.Easy To implement

2.Fater way to implement

Disadvantages

1.Since we are using the more frequent labels, it may use them in an over respresented way, if there are many nan's

2.It distorts the relation of the most frequent label

In [ ]:

Adding a variable to capture NAN

In [76]:
```
df=pd.read_csv('loan.csv', usecols=['BsmtQual','FireplaceQu','GarageType','
df.head()
```

Out[76]:

	BsmtQual	FireplaceQu	GarageType	SalePrice
0	Gd	NaN	Attchd	208500
1	Gd	TA	Attchd	181500
2	Gd	TA	Attchd	223500
3	TA	Gd	Detchd	140000
4	Gd	TA	Attchd	250000

In [78]:
```
import numpy as np
df['BsmtQual_var']=np.where(df['BsmtQual'].isnull(),1,0)
```

In [79]:
```
df.head()
```

Out[79]:

	BsmtQual	FireplaceQu	GarageType	SalePrice	BsmtQual_var
0	Gd	NaN	Attchd	208500	0
1	Gd	TA	Attchd	181500	0
2	Gd	TA	Attchd	223500	0
3	TA	Gd	Detchd	140000	0
4	Gd	TA	Attchd	250000	0

In [81]:
```
frequent=df['BsmtQual'].mode()[0]
```

In [82]:
```
df['BsmtQual'].fillna(frequent,inplace=True)
```

In [83]:
```
df.head()
```

Out[83]:

	BsmtQual	FireplaceQu	GarageType	SalePrice	BsmtQual_var
0	Gd	NaN	Attchd	208500	0
1	Gd	TA	Attchd	181500	0
2	Gd	TA	Attchd	223500	0
3	TA	Gd	Detchd	140000	0
4	Gd	TA	Attchd	250000	0

In [84]:
```
df['FireplaceQu_Var']=np.where(df['FireplaceQu'].isnull(),1,0)
frequent=df['FireplaceQu'].mode()[0]
df['FireplaceQu'].fillna(frequent,inplace=True)
```

In [85]:
```
df.head()
```

Out[85]:

	BsmtQual	FireplaceQu	GarageType	SalePrice	BsmtQual_var	FireplaceQu_Var
0	Gd	Gd	Attchd	208500	0	1
1	Gd	TA	Attchd	181500	0	0
2	Gd	TA	Attchd	223500	0	0
3	TA	Gd	Detchd	140000	0	0
4	Gd	TA	Attchd	250000	0	0

Suppose if you have more frequent categories, we just replace NAN with a new category

In [86]:
```
df=pd.read_csv('loan.csv', usecols=['BsmtQual','FireplaceQu','GarageType','
df.head()
```

Out[86]:

	BsmtQual	FireplaceQu	GarageType	SalePrice
0	Gd	NaN	Attchd	208500
1	Gd	TA	Attchd	181500
2	Gd	TA	Attchd	223500
3	TA	Gd	Detchd	140000
4	Gd	TA	Attchd	250000

In [87]:
```
def impute_nan(df,variable):
    df[variable+"newvar"]=np.where(df[variable].isnull(),"Missing",df[varia
```

In [88]:
```
for feature in ['BsmtQual','FireplaceQu','GarageType']:
    impute_nan(df,feature)
```

In [89]:
```
df.head()
```

Out[89]:

	BsmtQual	FireplaceQu	GarageType	SalePrice	BsmtQualnewvar	FireplaceQunewvar	Gar
0	Gd	NaN	Attchd	208500	Gd	Missing	
1	Gd	TA	Attchd	181500	Gd	TA	
2	Gd	TA	Attchd	223500	Gd	TA	
3	TA	Gd	Detchd	140000	TA	Gd	
4	Gd	TA	Attchd	250000	Gd	TA	

In [92]:
```
df=df.drop(['BsmtQual','FireplaceQu','GarageType'],axis=1)
```

In [93]:
```
df.head()
```

Out[93]:

	SalePrice	BsmtQualnewvar	FireplaceQunewvar	GarageTypenewvar
0	208500	Gd	Missing	Attchd
1	181500	Gd	TA	Attchd
2	223500	Gd	TA	Attchd
3	140000	TA	Gd	Detchd
4	250000	Gd	TA	Attchd

In [ ]:

```

```