# Pandas 使用

Pandas 包是基于 Python 平台的数据管理利器，已经成为了 Python 进行数据分析和挖掘时的数据基础平台和事实上的工业标准。本课程将在实战中学习 Pandas 包，学员将学会独立使用 Pandas 包完成数据读入、数据清理、数据准备、图表呈现等工作，为继续学习数据建模和数据挖掘打下坚实基础


## 安装&导入

语言 Python3

```
pip install pandas
```

## 设定系统环境

```
inport pandas as pd
#设定自由列表输出最多为 10 行
pd. options .display.max_rows = 10
# 显示当前 Pandas 版本号
pd. __version__
```

# 1 获取数据

## 1.1 新建数据

| s_id | s_name | s_sex | s_age | s_birthday | s_addr | s_created | s_status |
|---|---|---|---|---|---|---|---|
| 1 | 王 1 | 1 | 16 | 2007-04-18 | 重庆 | 2018-04-18 | 1 |
| 2 | 王 1 | 1 | 16 | 2007-04-18 | 重庆 | 2018-04-18 | 1 |
| 3 | 王小花 | 0 | 20 | 2000-08-11 | 重庆 | 2018-04-18 | 1 |
| 4 | 李四 | 0 | 10 | 2008-03-18 | 重庆 | 2018-04-18 | 0 |
| 5 | 王五 | 0 | 12 | 2002-05-02 | 上海 | 2018-04-18 | 1 |
| 7 | 张三跑 | 0 | 10 | 2008-01-18 | 西安 | 2018-04-18 | 1 |
| 8 | 王麻子 | 0 | 10 | 2008-01-18 | 杭州 | 2018-04-18 | 1 |
| 9 | 小四 | 0 | 10 | 2008-01-18 | 西安 | 2018-04-18 | 0 |
| 10 | 小5 | 0 | 10 | 2008-01-18 | 苏州 | 2018-04-18 | 1 |
| 11 | 小六 | 0 | 10 | 2008-01-18 | 西安 | 2018-04-18 | 1 |
| 15 | 张三跑 | 0 | 10 | 2008-01-18 | 西安 | 2018-04-18 | 1 |

**Series**

它是一种类似于一维数组的对象，是由一组数据(各种 NumPy 数据类型)以及一组与之相关的数据标签(即索引)组成。仅由一组数据也可产生简单的 Series 对象

用值列表生成 Series 时，Pandas 默认自动生成整数索引

```
s = pd.Series([1, 3, 5, 6, 8])
    0  1.0
    1  3.0
    2  5.0
    3  6.0
    4  8.0
```

```
pd.Series(
    data=None :数据列表，字典格式肘直接同肘提供变量名
    name=None : 变量名列表
)
```

**DataFrame**

DataFrame 是 Pandas 中的一个表格型的数据结构，包含有一组有序的列，每列可以是不同的值类型(数值、字符串、布尔型等)，DataFrame 即有行索引也有列索引，可以被看做是由 Series 组成的字典

```
    pd.DataFrame({
    'name':['zs1','zs2','zs3'],
    'sex':[1,1,0],
    'age':18
    })
```

```
pd.DataFrame(
    data=None :数据列表，字典格式肘直接同肘提供变量名
    columns=None : 变量名列表
)

pd.DataFrame (
data = [ [1,'test'] r,[2,'python'],[3,'test'] A [4, 'Hello']],
columns = ['no','info']
)
```

## 1.2 读入文本格式数据文件


pandas. read_csv (filename, encoding)

pandas.read_table（ ）:更通的文本文件读取命令

主要的区别在于默认的 sep="\t",即 tab 符号。

```
pd.read_csv(
filepath_or_buffer :要读入的文件路径
sep = ，, ; :切分隔符
```

header = ' infer ':指定数据中的第几行作为变量名

names = None :自定义变量名列表

index_col = None :将会被用作索引的列名，多列时只能使用序号列表

usecols = None :指定只读入某些列，使用索引列表或者名称列表均可。

[0,1,3],["名次"，"学校名称七"所在地区”]

encoding = None :读入文件的编码方式

utf-8/GBK,中文数据文件最好设定为 utf-8

na_values :指定将被读入为缺失值的数值列表，默认下列数据被读入为缺失值:

''A '#N/A* A '#N/A N/A* A '#NA1A A

'-l.#QNAN* A '-NaN1A '-nan'A 'l.#IND'A '1.#QNAN'A

'N/A* A 'NA。'NULL'A 'NaN'A 'n/a* A 'nan'A 'null'

)：读取 CSV 格式文件，但也可通用于文本文件读取

## 1.3 读取 EXCEL 文件

pd.read_excel（

filepath_or_buf fer :要读入的文件路径

sheet_name:读入的表单，字符串或者数字序号均可，默认读入第一个

```
    )



df2 = pd.read_excel（filename, sheet_name = 0）
```

## 1.4 读入数据库数据

### 1.4.1 配置 MySQL 连接引擎

```
conn = pymysql.connect(
        host = 'localhost',
        user = 'root',
        passwd = '123456',
        db = 'mydb',
        port=3306,
        charset = 'utf8'
    )
```

### 1.4.2 读取数据表

pd.read_sql(

sql :需要执行的 SQL 语句/要读入的表名称

con : 连接引擎名称

index_col = None :将被用作索引的列名称

columns = None :当提供表名称时，需要读入的列名称 list

)

tbl = pd.read_sql ("select code, name from t_hero con = eng)

tbl = pd.read_sql ("select count (*) from t_hero  con = eng)

# 2 保存数据

## 2.1 保存数据至外部文件

df.to_csv(

filepath_or_buf fer :要保存的文件路径

sep =：分隔符

columns :需要导出的变量列表

header = True :指定导出数据的新变量名，可直接提供 list

index = True :是否导出索引

mode = 'w' : Python 写模式,读写方式：r,r+ , w , w+ , a , a+

encoding = 'utf-8' :默认导出的文件编码格式

df2 . to_csv ('temp. txt' )

df.to_excel(

filepath_or_buf fer :要读入的文件路径

sheet_name = 1 Sheetl1 :要保存的表单名称

df2.to_excel(' temp.xlsx'  index = False  sheet_name = data)

## 2.2 保存数据至数据库

df.to_sql(

name :将要存储数据的表名称

```
con : 连接引擎名称
if_exists = 'fail' :指定表已经存在时的处理方式 fail :不做任何处理(不插入新数据)
replace :删除原表并重建新表 append :在原表后插入新数据
index = True :是否导出索引
)
apptbl.to_sql(name=nj t_histrecn r con=engA
if_exists=1 append.1 r index=False)
```

# 3 变量列的基本操作

当我们有了数据源以后，先别急着分析，应该先熟悉数据，只有对数据充分熟悉了，才能更好的进行分析

## 3.1 了解数据基本情况

### 3.1.1 head()与 tail()

当数据表中包含了数据行数过多时，而我们又想看一下每一列数据都是什么样的数据时，就可以把数据表中前几行或后几行数据显示出来进行查看

head()方法返回前 n 行(观察索引值)，显示元素的数量默认是 5，但可以传递自定义数值

tail()方法返回后 n 行观察索引值)，显示元素的数量默认是 5，但可以传递自定义数值

代码如下：

```
#浏览前几条记录
df.head()
df.head(10)
浏览最后几条记录
```

```
df .tail()
```

### 3.1.2 info()

熟悉数据的第一点就是看下数据的类型，不同的数据类型的分析思路是不一样的，比如说数值类型的数据就可以求均值，但是字符类型的数据就没法求均值了

info()方法查看数据表中的数据类型，而且不需要一列一列的查看，info()可以输出整个表中所有列的数据类型

```
df.info()
```

### 3.1.3 shape

熟悉数据的第二点就是看下数据表的大小，即数据表有多少行，多少列

shape()方法会以元组的形式返回行、列数。注意 shape 方法获取行数和列数时不会把索引和列索引计算在内

代码如下：

```
df.shape
```

### 3.1.4 describe()

熟悉数据的第三点就是掌握数值的分布情况，即均值是多少，最值是多少，方差及分位数分别是多少

describe()方法就是可以就可以获取所有数值类型字段的分布值

代码如下：

```
df.describe()
```

## 3.2 修改变量列名

### 3.2.1 columns

df. columns =新的名称 list

df.columns

### 3.2.2 rename()

df.rename（

columns =新旧名称字典：｛旧名称，：新名称，｝

inplace = False :是否直接替换原数据框）

代码如下：

```
df.rename(columns = ｛‘newname':'name','newname2':'name2'｝, inplace = True ）
```

## 3.3 筛选变量列

通过 df.var 或 df[var] 可以选择单列

**注意**：但只适用于已存在的列，只能筛选单列，结果为 Series

代码如下：

```
df.var 单列的筛选结果为 Series
```

通过 df[[var]] 可以选择多列

代码如下：

```
df[[var]] 单列的筛选结果为 DataFream

df[['var1, 'var2']] 多列时，列名需要用列表形式提供（因此可使用列表中的切片操作）多列的筛选结果为 DF
```

## 3.4 删除变量列

```
df.drop（
index / columns =准备删除的行/列标签，多个时用列表形式提供
 inplace = False :是否直接更改原数据框
 ）# pandas0.21 版本以上
```

代码如下：

```
df.drop(columns =['col1','col2'] )
```

```
del df['column-name']
直接删除原数据框相应的一列，建议尽量少用
```

## 3.5 变量类型的转换

### 3.5.1 Pandas 支持的数据类型

具体类型是 Python, Numpy 各种类型的混合，可以比下表分的更细

- float
- int
- string
- bool
- datetime64[nsr]  datetime64[nsr,tz]  timedelta[ns]
- category
- object

df.dtypes :查看各列的数据类型

代码如下：

```
df.dtypes
```

### 3.5.2 在不同数据类型间转换

df.astype(

dtype :指定希望转换的数据类型，可以使用 numpy 或者 python 中的数据类型：int/float/bool/str

copy = True :是否生成新的副本，而不是替换原数据框

errors = ‘raise‘ : 转换出错时是否抛出错误，raise/ ignore )

代码如下：

```
df.astype('str').dtypes
df .column.astype (‘str ')
df.astype('int', errors = 'ignore').dtypes 明确指定转换类型的函数：
```

**旧版本方法**：

pd.to_datetime ()

pd.to_timedelta ()

pd.to_numeric ()

df.to_string()

可以通过 df. apply 来批量进行多列的转换

pd. to_numeric (df .cloumn)

```
df [['cloumn1','cloumn2']].astype (' str' ) . apply (pd. to_numeric) . dtypes
```

## 3.6 变量列的添加

如果想给现有的数据增加新列可以使用如下方法：

### 3.6.1 根据新数据添加

```
df[cloumn] = pd.Series([val,val2,val3],index=[c1,c2,c3])
```

### 3.6.2 根据原数据添加

```
df[cloumn] = df[c2]+df[c3]
```

# 4 索引的使用

**索引的用途**：

用于在分析、可视化、数据展示、数据操作中标记数据行

提供数据汇总、合并、筛选时的关键依据

提供数据重构时的关键依据

**注意事项**：

索引是不可直接修改的，只能增、删、替换

逻辑上索引不应当出现重复值，Pandas 对这种情况不会报错，但显然有潜在风险

## 4.1 建立索引

### 4.1.1 新建数据框时建立索引

所有的数据框默认都已经使用从 0 开始的自然数索引，因此这里的"建立"索引指的是自定义索引

```
df = pd.DataFrame(
{'varl' : 1.0, ' var2 ' : [1,2,3,4],
' var3 ' : ['test', 'python','test', 'hello'] , ' var4' : ' cons'} , index = [0,1,2,6]
```

```
)
```

### 4.1.2 读入数据时建立索引

使用现有的列

```
df = pd. read_csv ("filename",  index_col="column")
```

使用复合列

```
df = pd. read_csv ("filename", index_col=["column1","column2"])
```

### 4.1.3 指定某列为索引列

```
df.set_index(
        keys :被指定为索引的列名，复合索引用 list:格式提供
        drop = True :建立索引后是否删除该列
        append = False :是否在原索引基础上添加索引，默认是直接替换原索引
        inplace = False :是否直接修改原数据框
)
```

```
df_new = df .set_index (['学号'，'性别'],drop = False)
df_new = df . set_index ('学号', append=True, drop=False)
```

## 4.2 将索引还原变量列

```
df.reset_index(
        drop = False :是否将原索引直接删除，而不是还原为变量列
        inplace = False :是否直接修改原数据框
)
```

```
df.set_index (['col1',col2,'col3'])
```

将索引全部还原为变量

```
df.reset_index ()
```

是否删除 index 列

```
df.reset_index (drop=True)
```

# 4.3 引用和修改索引

## 4.3.1 引用索引

注意：索引仍然是有存储格式的，注意区分数值型和字符型的引用方式

```
df.index
```

## 4.3.2 修改索引

**修改索引名**

本质上和变量列名的修改方式相同

**修改索引值**

这里的修改本质上是全部替换

```
dfl.index[3] = 6 #此处无法直接赋值
df1.index = ['a', 'b', 'c', 6]
```

## 4.3.3 更新索引

reindex 则可以使用数据框中不存在的数值建立索引，并据此扩充新索引值对应的索引行/列，同时进行缺失值填充操作

df.reindex(

labels :类数组结构的数值，将按此数值重建索引，非必需

copy = True :建立新对象而不是直接更改原 df/series

缺失数据的处理方式

method :针对已经排序过的索引，确定数据单元格无数据时的填充方法，非必需

pad / ffill:用前面的有效数值填充

backfill / bfill:用后面的有效数值填充

nearest:使用最接近的数值进行填充

fill_value = np.NaN :将缺失值用什么数值替代

limit = None :向前/向后填充时的最大步长

)

```
df.set_index ( '学号' )
df.reindex()
df.reindex([1,2,99,101])
df.reindex ( [l,2,99,101],method = 'ffill' )
df.reindex([1,2,99,101],fill_value="不知道")
df.reindex([1,2,99,101],fill_value="不知道").dtypes
```

# 5 案例的基本操作

## 5.1 案例排序

### 5.1.1 用索引排序

df.sort_index (

level : ( 多重索引时 ) 指定用于排序的级别顺序号/名称

ascending = True :是否为升序排列，多列时以表形式提供

inplace = False :

na_position = 'last' :缺失值的排列顺序，first/last

)

```
df = pd.read_excel ( "stu_data.xlsx", index_col=["学号","性别"] )
df. set_index( ['学号','性别'], inplace = True )
df2.sort_index()
df2.sort_index(ascending = [True,False])
df2.sort_index(level = '学号')
```

### 5.1.2 使用变量值排序

df.sort_values (

by :指定用于排序的变量名，多列时以列表形式提供

ascending = True :是否为升序排列

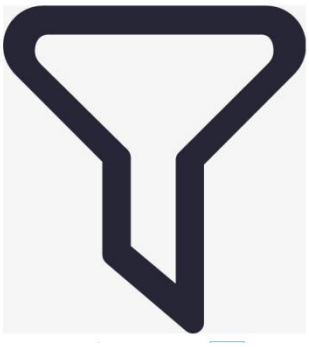
inplace = False :

na_position = 'last' :缺失值的排列顺序，first/last

)

```
df.sort_values(['身高','支出']，0，False)
```

# 5.2 数据选择



## 5.2.1 列选择

**选择列**

当想要获取 df 中某列数据时，只需要在 df 后面的方括号中指明要选择的列即可。如果是一列，则只需要传入一个列名;如果是同时选择多列，则传入多个列名即可（注意：多个列名用一个 list 存放）

```
df[col]
df[[col1 , col2]]
```

Python 中我们把通过传入列名选择数的方式称为普通索引

除了传入具体的列名，我们可以传入具体列的位置，即第几行，对数据进行选取，通过传入位置来获取数据时需要用到 iloc 方法

```
df.iloc[,[0,2]]
```

df.iloc 意为 integer-location,即按照行列序号进行检索,iloc 后面的方括号中逗号之前的部分表示要获取的行的位置，只输入一个冒号，不输入任何数值表示获取所有的行，逗号之后方括号表示要获取的列的位置，列的位置同样也是从 0 开始计数

**选择连续列**

当需要获取连续列时，可以通过 iloc 方法实现

```
df[ : ,0:3]
```

### 5.2.2 案例(行)的选择

获取行的方式主要有 2 种：

一种是**普通索引**，即传入具体索引的名称 **loc 方法**;

另一种是**位置索引**，即传入具体的行数，需要用 **iloc 方法**

```
# 获取某行
df.loc[index_name]
# 获取某几行
df.loc[[index_name1,index_name2]]
# 获取某行
df.iloc[0]
# 获取某行
df.iloc[[0,3]]
# 获取某连续行
df.iloc[0:3]
# 获取满足条件的行
df[df[“支出”]>10]
df[(df[“支出”]>10) & (df[“性别”]=='女')]
```

筛选操作的实质：基于 T/F 值进行筛选

```
select_list = [ True, False, True ]
df.iloc[select_list]
```

### 5.2.3 混合选择

```
#普通索引+普通索引
df.loc[[0,1],['学号','性别']]
```

```
#位置索引+位置索引
df.iloc[[0,1],[0,2]]
#Boolean 索引+普通索引
df[df['支出']>20][['学号','性别','支出']]
#切片索引+切片索引
df.iloc[0:2,0:2]
#切片索引+普通索引
df.loc[1:2,['学号','性别']]
```

df.isin(values) 返回结果为相应的位置是否匹配给出的 values

values 为序列：对应每个具体值

values 为字典：对应各个变量名称

values 为数据框：同时对应数值和变量名称

```
df.col.isin([1,3,5])
df[ df.col.isin([1,3,5])]
df[ df.col.isin([‘val1','val2'])]
df[ df.index.isin([‘val1','val2'])]
```

# 6 变量变换

## 6.1 计算新变量

### 6.1.1 新变量为常数

df['vamame'] = value

```
df .newvar = 1 #注意该命令不会报错!
df.head（ ）
df[‘col'] =1
```

### 6.1.2 基于原变量做简单四则运算

df['var'] = df['oldvar'] *100

df[‘var'] = df.oldvar * 100

```
df [‘new_val'] = df .总分 + df .名次 + 1
```

```
inport numpy
df.new_val = numpy.sqrt (df .col) # numpy 内置函数自动支持 serial 格式数据
Import math
df[“new_val”] = math.sqrt(df.col) # 报错
```

### 6.1.3 基于一个原变量做函数运算

df.apply（

func : 希望对行/列执行的函数表达式

axis = 0 :针对行还是列进行计算

0/ ' index':针对每列进行计算

1/ ' columns ':针对每行进行计算

)

**简化的用法**

df [' varname ' ] = df. oldvar. apply （函数表达式）

```
import math
df[ ' new3 ' ] = df .col. apply (math.sqrt)
df 2 [ ' new3 ' ] = df 2 . 总分.  apply (numpy.sqrt)
```

**特殊运算的实现方式**

例：取变量的第一个字符

```
def m_head（ temp_str ）:
    return tmpstr[:1]
```

**对所有单元格进行相同的函数运算**

df_new = df .applymap （函数表达式）#是以 cell 为单位在进行操作

```
df [[‘col1','col2']].applymap (math.sqrt)
```

**不修改原 df,而是生成新的 df**

df.assign(varname = expression)

```
df_new = df .assign (new = df .总分.apply (math, sqrt))
```

## 6.2 在指定位置插入新变量列

df.insert(

loc :插入位置的索引值，0 <= loc <= len (columns)

column :插入的新列名称

value : Series 或者类数组结构的变量值

allow_duplicates = False :是否允许新列重名

)#该方法会直接修改原 df

```
df.insert(1, 'new_var', ‘cons')
```

## 6.3 修改/替换变量值

本质上是如何直接指定单元格的问题，只要能准确定位单元地址，就能够做到准确替换

```
df .所在城市.isin ([，上海，])

df .所在城市[70]=，上海市，*这种引用方式会给出警告

df[69:75]

df [，所在城市，][70]=，上海市，#这种引用方式会给出警告

df.loc[70,，所在城市，]=，上海市，精确引用,消停了
```

### 6.3.1 对应数值的替换

df.replace(

to_replace = None :将被替换的原数值，所有严格匹配的数值将被用 value 替换，可以 str/regex/list/dict/Series/numeric/None

value = None :希望填充的新数值

inplace = False

)

```
df2 .所在城市.replace ('北京市七，帝都，)#单个值替换
```

```
df2.所在城市.replace ( [      京市 L，上海市[，帝都七，魔都，])却列表值批量替换
#字典批量映射替换
df2.所在城市.replace ({      京市，：，帝都七，上海市，:，魔都，})
```

### 6.3.2 指定数值范围的替换

**方法一**：使用正则表达式完成替换

df.replace(regex, newvalue)

**方法二**：使用行筛选方式完成替换

用行筛选方式得到行索引，然后用 loc 命令定位替换

目前也支持直接筛选出单元格进行数值替换

**注意**：query 命令的类 SQL 语句可以进行检索，但不直接支持数值替换

```
df .总分.iloc[0:2] = 10

df .loc [3:5,'总分']=20
df.head(10)

#用 loc 命令完成替换
df.1oc[df2.名次 < 10,，总分]=20  #用 index 引用出相应的
df.head(10)
```

## 6.4 虚拟变量变换

pd.get_dummies(

data :希望转换的数据框/变量列

prefix = None :哑变量名称前缀

prefix_sep = 11 :前缀和序号之间的连接字符，设定有

prefix 或列名时生效

dummy_na = False :是否为 NaNs 专门设定一个哑变量列

columns = None :希望转换的原始列名，如果不设定，则转换所有符合条件的列

drop_first = False :是否返回 k-l 个哑变量，而不是 k 个哑变量

）#返回值为数据框

```
df2.head()
pd.get_dummies(df2.类型，prefix = npren )
pd. get_d.ummies (df 2 r columns= [ '类型' ])
```

## 6.5 数值变量分段

pd.cut(

X :希望进行分段的变量列名称

bins :具体的分段设定

int :被等距等分的段数

sequence of scalars :具体的每一个分段起点，必须包括最值，可不等距

right = True :每段是否包括右侧界值

labels = None :为每个分段提供自定义标签

include_lowest = False :第一段是否包括最左侧界值，需要和 right 参数配合

)#分段结果是数值类型为 Categories 的序列

pd.qcut #按而不賤照取值范围进行等分

# 7 整体数据管理

## 7.1 数据拆分

### 7.1.1 数据分组

df.groupby(

by :用于分组的变量名/函数

level = None :相应的轴存在多重索引时，指定用于分组的级别

as_index = True :在结果中将组标签作为索引

sort = True :结果是否按照分组关键字进行排序

)#生成的是分组索引标记，而不是新的 df

```
dfg = df. groupby ('开设')
df.groups
df.describe( )
dfg2 = df .groupby (['性别','开设'])
dfg2.mean ()
```

### 7.1.2 基于拆分进行筛选

筛选出其中一组

dfgroup.get_group()

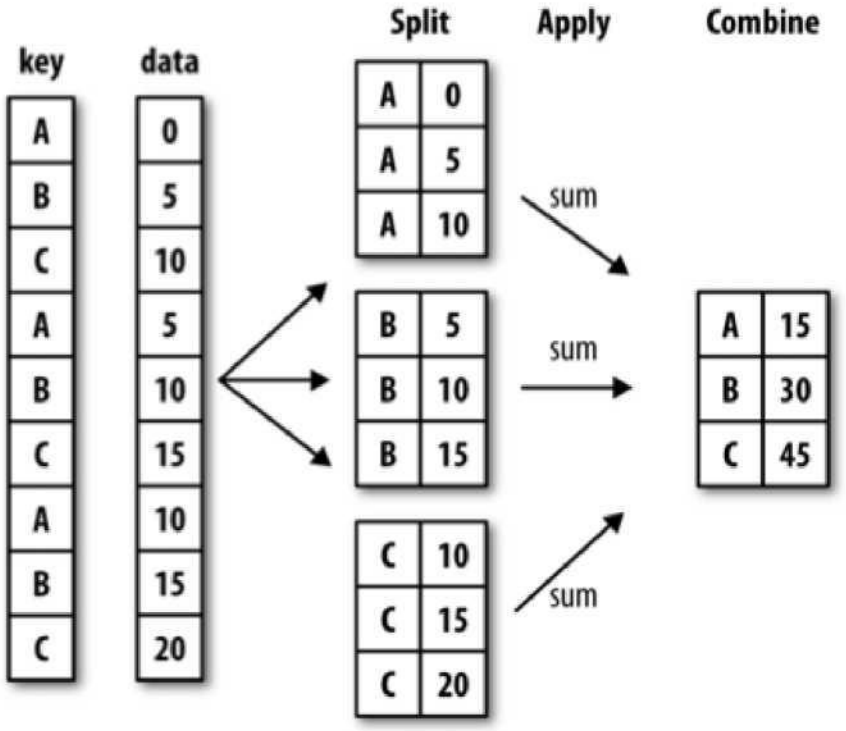
```
dfg.get_group ('课程').mean ()
```

筛选出所需的列

该操作也适用于希望对不同的变量列进行不同操作时

```
dfg['身高'].max()
```

## 7.2 分组汇总




在使用 groupby 完成数据分组后，就可以按照需求进行分组信息汇总，此时可以使用其它专门的汇总命令，如 agg 来完成汇总操作

### 7.2.1 使用 agg 函数进行汇总

df.aggregate( )

名称可以直接简写为 agg

可以用 axis 指定汇总维度

可以直接使用的汇总函数

| 名称 | 含义 |
| --- | --- |
| count () | Nuniber of non-null observations size () group sizes |
| sum() | Sum of values |

| | |
|---|---|
| mean() | Mean of values |
| median() | Arithmetic median of values |
| min () | Minimum |
| max() | Maximum |
| std() | Unbiased standard deviation |
| var () | Unbiased variance |
| skew() | Unbiased skewness(3rd moment) |
| kurt() | Unbiased kurtosis (4th moment) |
| quantile () | Sample quantile (value at %) apply() Generic apply |
| cov() | Unbiased covariance (binary) |
| corr() | Correlation (binary) |

```
df2g.agg( ‘count ‘)
df2g.agg('median')
df2g.agg(['mean', 'median'])
df2g.agg(['mean', 'median'])
#引用非内置函数
import  numpy as np
df2 .身高. agg (np. sum)
df2g.身高. agg (np. sum)
```

**引用自定义函数**

```
def mymean(x):
    return x.mean()
df2 .身高.agg (mymean)
df2g.agg(mymean)
```

### 7.2.2 其他分组汇总方法

在生成交叉表的同时对单元格指定具体的汇总指标和汇总函数

df.pivot_table()

pd.crosstab()

pd.crosstab (df .性别,df.身高)

## 7.3 长宽格式转换

基于多重索引，Pandas 可以很容易地完成长型、宽型数据格式的相互转换。

### 7.3.1 转换为最简格式

df.stack(

level = -1 :需要处理的索引级别，默认为全部，int/string/list

dropna = True :是否删除为缺失值的行

)#转换后的结果可能为 Series

```
df =pd.read_excel('person.xlsx')
dfs = df.stack()
```

### 7.3.2 长宽型格式的自由互转

df.unstack(

level = -1 :需要处理的索引级别，默认为全部，int/string/list

fill_value :用于填充缺失值的数值

)

```
df3s.unstack (1)
df3s.unstack([1,2])
```

**数据转置:** df.T

### 7.3.3 其他可用于格式转换的命令（扩展）

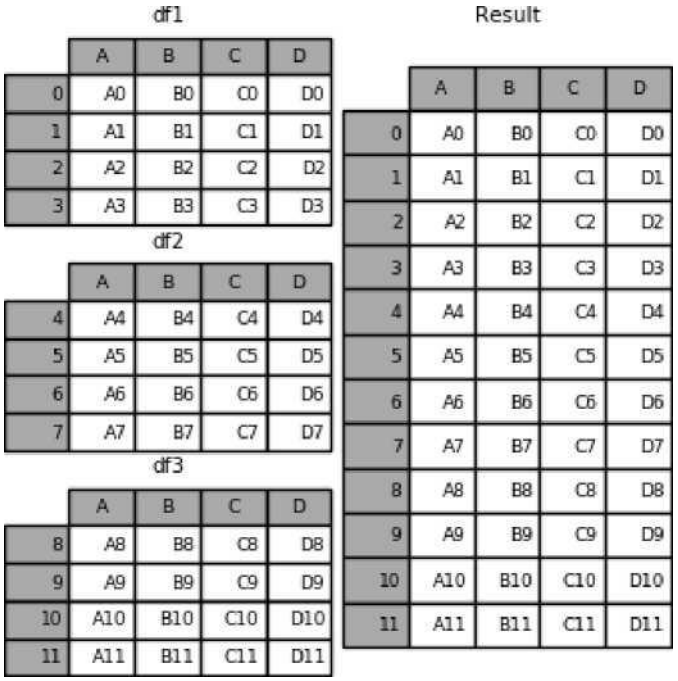
df.melt()

df.pivot()

df.pivot_table()

## 7.4 多个数据源的合并

### 7.4.1 数据的纵向合并




| | A | B | C | D |
|---|---|---|---|---|
| 0 | A0 | B0 | C0 | D0 |
| 1 | A1 | B1 | C1 | D1 |
| 2 | A2 | B2 | C2 | D2 |
| 3 | A3 | B3 | C3 | D3 |

| | A | B | C | D |
|---|---|---|---|---|
| 4 | A4 | B4 | C4 | D4 |
| 5 | A5 | B5 | C5 | D5 |
| 6 | A6 | B6 | C6 | D6 |
| 7 | A7 | B7 | C7 | D7 |

| | A | B | C | D |
|---|---|---|---|---|
| 8 | A8 | B8 | C8 | D8 |
| 9 | A9 | B9 | C9 | D9 |
| 10 | A10 | B10 | C10 | D10 |
| 11 | A11 | B11 | C11 | D11 |

| | A | B | C | D |
|---|---|---|---|---|
| 0 | A0 | B0 | C0 | D0 |
| 1 | A1 | B1 | C1 | D1 |
| 2 | A2 | B2 | C2 | D2 |
| 3 | A3 | B3 | C3 | D3 |
| 4 | A4 | B4 | C4 | D4 |
| 5 | A5 | B5 | C5 | D5 |
| 6 | A6 | B6 | C6 | D6 |
| 7 | A7 | B7 | C7 | D7 |
| 8 | A8 | B8 | C8 | D8 |
| 9 | A9 | B9 | C9 | D9 |
| 10 | A10 | B10 | C10 | D10 |
| 11 | A11 | B11 | C11 | D11 |

df.append(

other :希望添加的 DF/Series/字典/上述对象的列表 使用列表方式，就可以实现一次合并多个新对象

ignore_index = False :添加时是否忽略索引

verify_integrity = False :是否检查索引值的唯一性，有重复时报错

)

```
df = df.append( [df2, df3, df4])
```

### 7.4.2 数据的横向合并

df1

| | A | B | C | D |
|---|---|---|---|---|
| 0 | A0 | B0 | C0 | D0 |
| 1 | A1 | B1 | C1 | D1 |
| 2 | A2 | B2 | C2 | D2 |
| 3 | A3 | B3 | C3 | D3 |

df4

| | B | D | F |
|---|---|---|---|
| 2 | B2 | D2 | F2 |
| 3 | B3 | D3 | F3 |
| 6 | B6 | D6 | F6 |
| 7 | B7 | D7 | F7 |

Result

| | A | B | C | D | B | D | F |
|---|---|---|---|---|---|---|---|
| 0 | A0 | B0 | C0 | D0 | NaN | NaN | NaN |
| 1 | A1 | B1 | C1 | D1 | NaN | NaN | NaN |
| 2 | A2 | B2 | C2 | D2 | B2 | D2 | F2 |
| 3 | A3 | B3 | C3 | D3 | B3 | D3 | F3 |
| 6 | NaN | NaN | NaN | NaN | B6 | D6 | F6 |
| 7 | NaN | NaN | NaN | NaN | B7 | D7 | F7 |

merge 命令使用像 SQL 的连接方式

pd.merge（

需要合并的 DF

left :需要合并的左侧 DF

right :需要合并的右侧 DF

how = ' inner':具体的连接类型

{left、right 、outer 、 inner、 )

两个 DF 的连接方式

on :用于连接两个 DF 的关键变量（多个时为列表），必须在两侧都出现

left_on :左侧 DF 用于连接的关键变量（多个时为列表）

right_on :右侧 DF 用于连接的关键变量（多个时为列表）

left_index = False :是否将左侧 DF 的索引用于连接

right_index = False :是否将右侧 DF 的索引用于连接

其他附加设定

sort = False :是否在合并前按照关键变量排序（会影响合并后的案例顺序）

suffixes :重名变量的处理方式，提供长度为 2 的列表元素，分别作为后缀

suffixes=（'_x'、 '_y'）

copy = True

indicator = False :在结果 DF 中增加'_merge ' 列,用于记录数据来源 也可以直接提供相应的变量列名

Categorical 类型,取值： left_only 、 right_only 、 both

validate = None :核查合并类型是否为所指定的情况

one_to_one or 1:1

one_to_many or 1:m

many_to_one or m:1

many_to_many or m:m

```
df1 = pd.read_excel（"stu_data.xlsx", sheet_name = 'new4'）
df2 = pd.read_excel（"stu_data.xlsx", sheet_name = 'new5'）
pd.merge（df1，df2）
pd.merge（df1, df2[:20]）
df1 = df1.set_index（'学号'）
df2 = df2.set_index（'学号'）
pd.merge（df1，df2, left_index = True, right_index = True）
```

```
# df2 没建立索引时
pd.merge（ df1 ， df2, left_index = True, right_on = '学号' ）
```

### 7.4.3 Concat 命令简介

同时支持横向合并与纵向合并

```
pd.concat(
        objs :需要合并的对象，列表形式提供
        axis = 0 :对行还是对列方向进行合并
            (0 index 、 1 columns )
        join = outer :对另一个轴向的索引值如何进行处理
            (inner 、 outer )
        ignore_index = False
        keys = None :为不同数据源的提供合并后的索引值
        verify_integrity = False
        copy = True
)
```

```
#纵向合并
df1 = pd.read_excel（ "stu_data.xlsx", sheet_name = 'new1' )
df2 = pd.read_excel（ "stu_data.xlsx", sheet_name = 'new2' )
pd.concat（ [df1,df2] ）
pd.concat（ [df1,df2], key=['a','b'] ）
```

```
#横向合并
```

```
df1 = pd.read_excel ( "stu_data.xlsx", sheet_name = 'new4' )

df2 = pd.read_excel ( "stu_data.xlsx", sheet_name = 'new5' )

pd.concat ( [df1,df2 ], axis = 1 )
```

# 8 数据清洗


## 8.1 处理缺失值

### 8.1.1 系统默认的缺失值

系统默认的缺失值

None 和 np. nan

确定数值是否是缺失值

df. isna () # 别名为 isnull， 反函数为 notna

```
df.身高.iloc[:3] = None

df.身高.isna( )

import numpy as np

df.身高.iloc[:5] = np.nan
```

```
df.身高.isna( )
```

None 和 np.nan 的区别：能否进行比较

None == None

np.nan == np.nan

**设定 in 和 inf 是否被认定为缺失值**

pandas.options.mode.use_inf_as_na

### 8.1.2 处理自定义缺失值

目前 Pandas 不支持设定自定义缺失值，因此只能考虑将其替换为系统缺失值

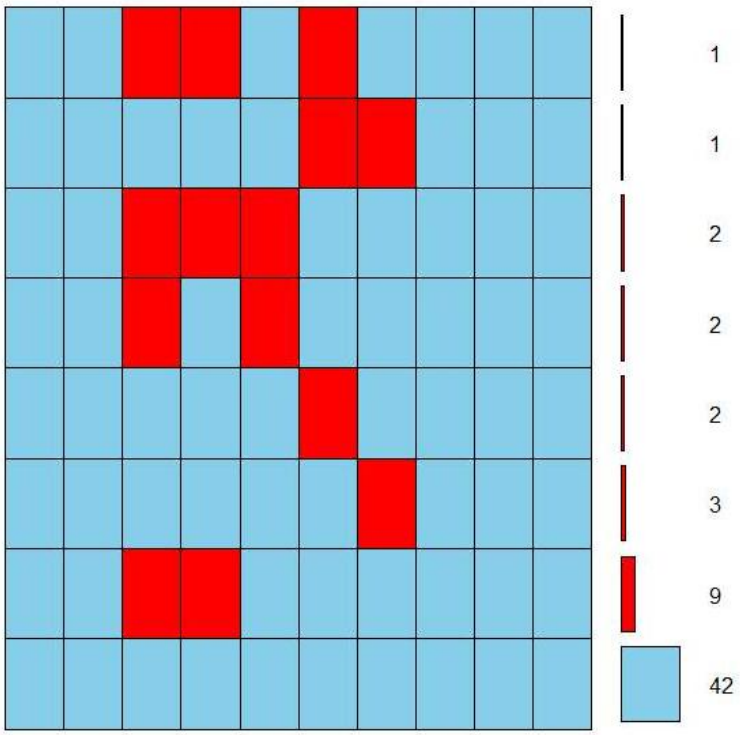
df.replace('自定义值'，np.nan)

```
df .开设.replace ("不清楚",np.nan)
```

**设定为 None 后的效果完全不同**

```
df .开设,replace ("不清楚",None)
df2 = df .replace ( [“不清楚”,171] ,[np.nan, np.nan] ) # 后面的中括号可以简写 np.nan
```

### 8.1.3 标识缺失值




df.isna(): 检查相应的数据是否为缺失值 同 df.isnull()

df2 = df .replace ( [“不清楚”,171] ,np.nan )

**检查多个单元格的取值是否为指定数值**

df.any(

axis : index (0), columns (1)

skipna = True :检查时是否忽略缺失值

level = None :多重索引时指定具体的级别

df.all(

axis : index (0), columns (1)

skipna = True :检查时是否忽略缺失值

level = None :多重索引时指定具体的级别

)

```
df.isna () .any(1)
df[df.isna().any(1)]
```

### 8.1.4 填充缺失值

df.fillna(

value :用于填充缺失值的数值，也可以提供 dict/Series/DataFrame 以进一步指明哪些索引/列会被替换 不能使用 list

method = None :有索引时具体的填充方法，向前填充，向后填充等

limit = None :指定了 method 后设定具体的最大填充步长，此步长不能填充

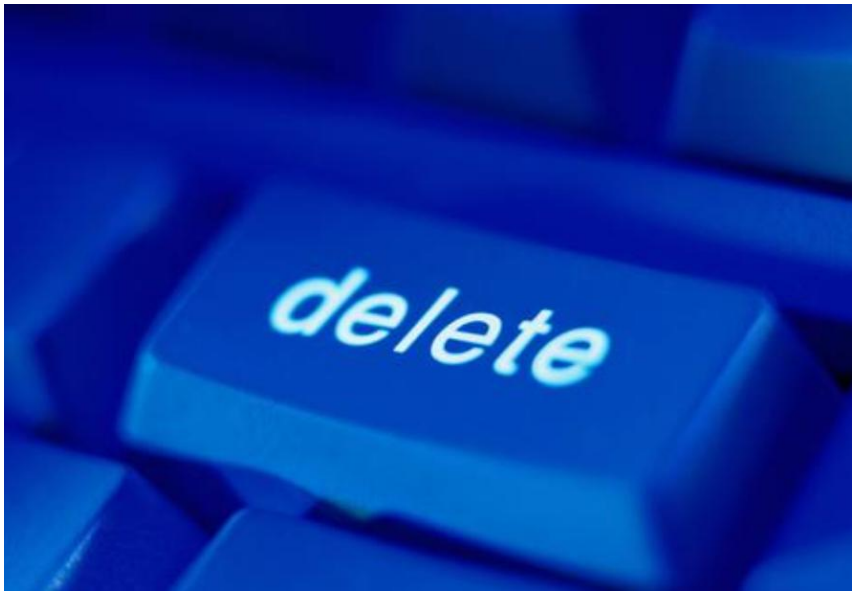
axis : index (0), columns (1)

inplace = False

)

```
df .replace ( ["不清楚",171] ,np.nan ). fillna ('未知')
df .replace ( ["不清楚",171] ,np.nan ) .fillna(df2.mean( ))
```

### 8.1.5 删除缺失值



df.dropna(

axis = 0 : index (0), columns (1)

how = any : any、all

any :任何一个为 NA 就删除

all :所有的都是 NA 删除

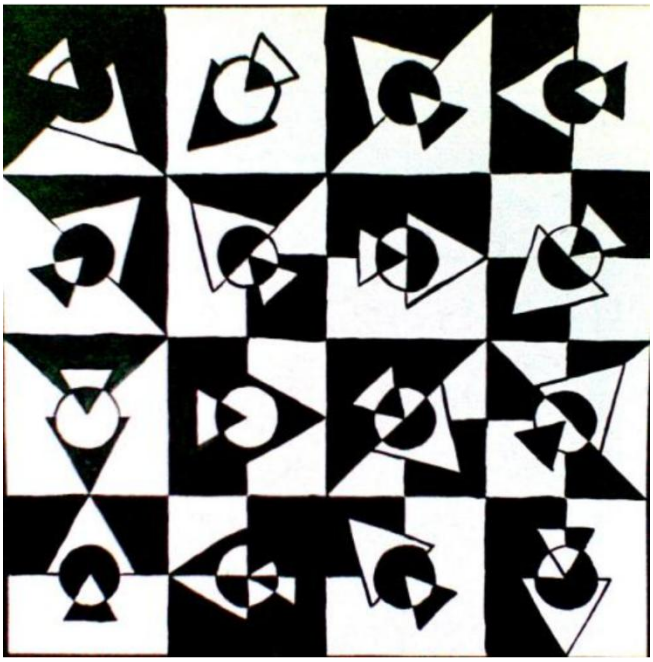
thresh = None :删除的数量阈值，int

subset :希望在处理中包括的行/列子集

inplace = False :

)

## 8.2 数据查重



### 8.2.1 标识出重复的行

标识出重复行的意义在于进一步检查重复原因，以便将可能的错误数据加以修改

Duplicated

```
df['dup' ] = df.duplicated( ['课程','开设'])
```

利用索引进行重复行标识

df.index.duplicated()

```
df2 = df.set_index ( ['课程','开设'] )
df2.index. duplicated ()
```

### 8.2.2 直接删除重复的行

drop_duplicates（

subset= “ ” 按照指定的行进行去重

keep='first' 、 'last' 、 False 是否直接删除有重复的所有记录

）

```
df. drop_duplicates ( ['课程', '开设' ] )
df. drop_duplicates ( ['课程', '开设' ] , keep = False )
```

利用查重标识结果直接删除

df[~df.duplicated( )

```
df[~df . duplicated  ( ['课程', '开设' ] )
```

# 9 日期时间变量

## 9.1 Timestamp 与 Period

### 9.1.1 Timestamp

```
from datetime import datetime
pd.Timestamp(datetime(2022,1,1))
pd.Timestamp(datetime(2022,1,2,3,4,5))
pd.Timestamp(2022,1,2)
pd.Timestamp('2022-01-02 3:4:5')
```

### 9.1.2 Peroid

**可以被看作是简化之后的 Timestamp 对象**

由于详细数据的不完整，而表示的是一段时间，而不是一个时点

但是实际使用中很可能是按照时点在使用

**很多使用方法和 Timestamp 相同，因此不再详细介绍**

```
pd.Period('2022-01')
pd.Period('2022-01',freq='D')
```

# 9.2 数据转换 Timestamp 类

## 9.2.1 使用 pd.Timestamp()直接转换

```
pd.Timestamp(df['date'] [0])
df['date'].apply(pd.Timestamp)
```

## 9.2.2 用 to_datetime 进同比量转换

pd.to_datetime(

arg :需要转换为 Timestamp 类的数值

Integer, float, string, datetime, list, tuple, 1-d array, Series,

errors = 'raise' : (' ignore' , 'raise' , 'coerce' )

'raise ' 抛出错误

'coerce' 设定为 NaT

'ignore' 返回原值

短日期的解释方式：类似 '10/11/12"这样的数据如何解释

dayfirst = 'False' : 数值是否 day 在前

yearfirst = 'False':数值是否 year 在前，该设定优先

box = True :是否返回为 DatetimeIndex, False 时返回 ndarray 数组

format = None :需要转换的字符串格式设定 )

代码如下：

```
pd.to_datetime(datetime(2022, 1, 2, 3, 4, 5))
pd.to_datetime('2022-01-02 3:4:5')
pd.to_datetime(['2022/01/02' , '2022.01.03'])
pd.to_datetime(df[ 'date' ], format = '%Y-%m-%d %H:%M')
```

### 9 2.3 基于所需的变量列合成 Timestamp 类

```
pd.to_datetime(df[ ['Year' , 'Month' , 'Day' , 'Hour']])
```

## 9.3 使用 DatetimeIndex 类

Datetimeindex 类对象除了拥有 Index 类对象的所有功能外，还针对日期时间的特点有如下增强:

基于日期时间的各个层级做快速索引操作

快速提取所需的时间层级

按照所指定的时间范围做快速切片

### 9.3.1 建立 Datetimeindex 对象

#### 1. 建立素引时自动转换

使用 Timestamp 对象建立索引，将会自动转换为 DatetimeIndex 对象

```
df_index = df.set_index(pd.to_datatime(df['data1']))
```

#### 2. 通过 date_range 建立

这种建立方式主要是和 reindx 命令配合使用，以快速完成对时间序列中缺失值的填充工作

pd.date_range(

start /end = None :日期时间范围的起点/终点，均为类日期时间格式的字符串/数据

periods = None :准备生成的总记录数

freq = 'D':生成记录时的时间周期，可以使用字母和数值倍数的组合，如'5H'

name = None :生成的 DatetimeIndex 对象的名称

)

pd.bdate_range(

主要参数和 pd.date_range 几乎完全相同，但默认 freq = 'B' (business daily) 另外附加了几个针对工作日/休息日筛选的参数

)

```
pd.date_range('2/1/2022', periods=5, freq='M')
```

# 9.4 时间序列做基本使用

## 9.4.1 序列的分组汇总

#### 9.4.1.1 直接取出素引的相应层级

DatetimeIndex 对象可直接引用的属性

| | |
|---|---|
| year | 日期时间的年份。 |
| month | 月份为1月= 1，12月= 12。 |
| day | 日期时间的日期。 |
| hour | 日期时间。 |
| minute | 日期时间的分钟数。 |
| second | 日期时间的秒数。 |
| microsecond | 日期时间的微秒。 |
| nanosecond | 日期时间的纳秒。 |
| date | 返回python datetime.date对象的numpy数组（即没有时区信息的Timestamps的日期部分）。 |
| time | 返回datetime.time的numpy数组。 |
| timetz | 返回datetime.time的numpy数组，其中还包含时区信息。 |
| dayofyear | 一年中的第几天。 |
| weekofyear | （不推荐）一年中的第几周。 |
| week | （不推荐）一年中的第几周。 |
| dayofweek | 星期一= 0，星期日= 6的星期几。 |
| weekday | 星期一= 0，星期日= 6的星期几。 |

| | |
|---|---|
| dayofweek | 星期一= 0，星期日= 6的星期几。 |
| weekday | 星期一= 0，星期日= 6的星期几。 |
| quarter | 日期的四分之一。 |
| tz | 返回时区（如果有）。 |
| freq | 返回频率对象（如果已设置），否则返回无。 |
| freqstr | 如果设置了频率对象，则将其作为字符串返回，否则返回None。 |
| is_month_start | 指示日期是否为每月的第一天。 |
| is_month_end | 指示日期是否为每月的最后一天。 |
| is_quarter_start | 日期是否为一个季度的第一天的指示器。 |
| is_quarter_end | 日期是否为一个季度的最后一天的指标。 |
| is_year_start | 指明日期是否为一年的第一天。 |
| is_year_end | 指示日期是否为一年的最后一天。 |
| is_leap_year | 布尔指标，如果日期属于the年。 |
| inferred_freq | 尝试返回由infer_freq生成的代表频率猜测的字符串。 |

```
df.index.year
```

#### 9.4.1.2 直接使用 groupby 方法进行汇总

```
df.groupby(df2.index.year).max( )
```

#### 9.4.1.3 resample( )

使用上比 groupby 更简单(输入更简洁)

可以将数值和汇总单位进行组合,实现更复杂的汇总计算

| 别名 | 偏移量类型 | 说明 |
|---|---|---|
| D | Day | 每日日历 |
| B | BusinessDay | 每工作日 |
| H | Hour | 每小时 |
| T或min | Minute | 每分 |
| S | Second | 每秒 |
| L或ms | Milli | 每毫秒 |
| U | Micro | 每微妙 |
| M | MonthEnd | 每月最后一个日历日 |
| BM | BusinessMonthEnd | 每月最后一个工作日 |
| MS | MonthBegin | 每月第一个日历日 |
| BMS | BusinessMonthBegin | 每月第一个工作日 |
| W-MON, W-TUE, ... | Week | 从指定的星期几(MON, TUE, WED, THU, FRI, SAT, SUN)开始算起, 每周 |
| WOM-1MON, WOM-2MON, ... | WeekOfMonth | 产生每月第一, 第二, 第三或第四周的星期几. 例如: WOM-3FRI表示每月第3个星期五 |
| Q-JAN, Q-FEB, ... | QuarterEnd | 对于指定月份(JAN, FEB, MAR, APR, MAY, JUN, JUL, AUG, SEP, OCT, NOV, DEC)结束的年度, 每季度最后一月的最后一个日历日 |
| BQ-JAN, BQ-FEB, ... | BusinessQuarterEnd | 对于指定月份(JAN, FEB, MAR, APR, MAY, JUN, JUL, AUG, SEP, OCT, NOV, DEC)结束的年度, 每季度最后一月的最后一个工作日 |
| QS-JAN, QS-FEB, ... | QuarterBegin | 对于指定月份(JAN, FEB, MAR, APR, MAY, JUN, JUL, AUG, SEP, OCT, NOV, DEC)结束的年度, 每季度最后一月的第一个日历日 |
| BQS-JAN, BQS-FEB, ... | BusinessQuarterBegin | 对于指定月份(JAN, FEB, MAR, APR, MAY, JUN, JUL, AUG, SEP, OCT, NOV, DEC)结束的年度, 每季度最后一月的第一个工作日 |
| A-JAN, A-FEB, ... | YearEnd | 每年指定月份(JAN, FEB, MAR, APR, MAY, JUN, JUL, AUG, SEP, OCT, NOV, DEC)的最后一个日历日 |
| BA-JAN, BA-FEB, ... | BusinessYearEnd | 每年指定月份(JAN, FEB, MAR, APR, MAY, JUN, JUL, AUG, SEP, OCT, NOV, DEC)的最后一个工作日 |
| AS-JAN, AS-FEB, ... | YearBegin | 每年指定月份(JAN, FEB, MAR, APR, MAY, JUN, JUL, AUG, SEP, OCT, NOV, DEC)的第一个日历日 |

```
df.resample('3M').mean( )
```

### 9.4.2 时间缺失值的处理

时间序列要求记录的时间点连贯无缺失，因此需要：

首先建立针对整个时间范围的完整序列框架

随后针对无数据的时间点进行缺失值处理

```
df = pd.read_excel('stu_data.xlsx', index_col = 2)
Index2 = pd.date_range('2005-01-01', '2018-01-01' )
df.reindex(index2)
```

### 9.4.3 序列数值平移

df.shift(

periods = 1: 希望移动的周期数

freq :时间频度字符串

axis =0

)

```
df.shift( 3 )
```

# 10 数据的图形展示

## 10.1 配置绘图系统环境

```
df ['体重'].plot .box (title='第一个图',ylim= (60,80))


#中文字符兼容问题
```

```
import matplotlib
matplotlib.reParams[‘font.sans-serif'] = [‘SimHei']
#绘图功能的进一步美化和功能增强包,参考 http: //seaborn .pydata. org/
import seaborn
seaborn.set_style(‘whitegrid')
#注意有中文兼容问题,需要重新导入中文设定!


进一步在一些细节上的美化和优化
import matplotlib.pyplot as plt
pit.figure ()
```

df.plot(

绘图用数据

data :数据框

x = None:行变量的名称/顺序号

y = None :列变量的名称/顺序号

kind = 'line':需要绘制的图形种类

line : line plot (default)

bar : vertical bar plot

barh : horizontal bar plot

hist : histogram

box : boxplot

kde : Kernel Density Estimation plot

density : same as kde

area : area plot

pie : pie plot

scatter : scatter plot

hexbin : hexbin plot

各种辅助命令

figsize : a tuple (width, height) in inches

xlim / ylim : X/Y 轴的取值范围，2-tuple/list 格式

logx / logy / loglog = False :对 X/Y/双轴同时使用对数尺度

title : string or list

Alpha :图形透明度，0-1

图组命令

subplots = False :是否分图组绘制图形

sharex :是否使用相同的 X 坐标

ax = None 时，取值为 True,否则取值为 False

sharey = False :是否使用相同的 Y 坐标

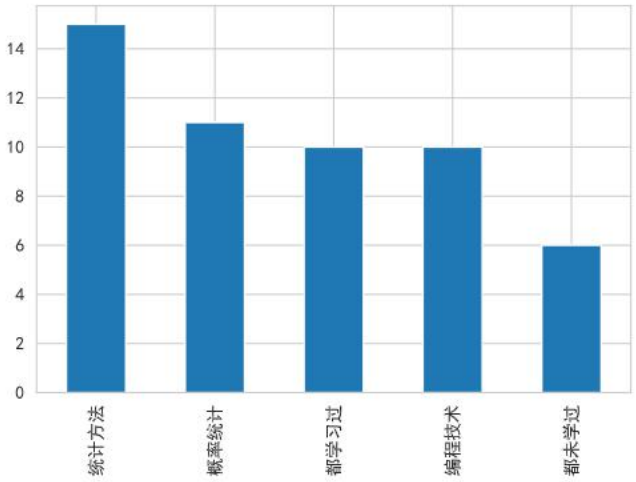
ax = None :需要叠加的 matplotlib 绘图对象

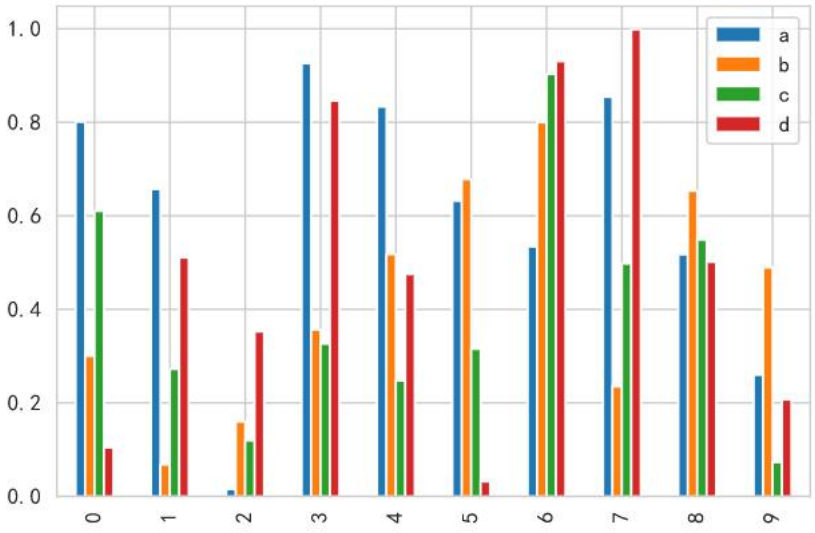
图形种类的等价写法

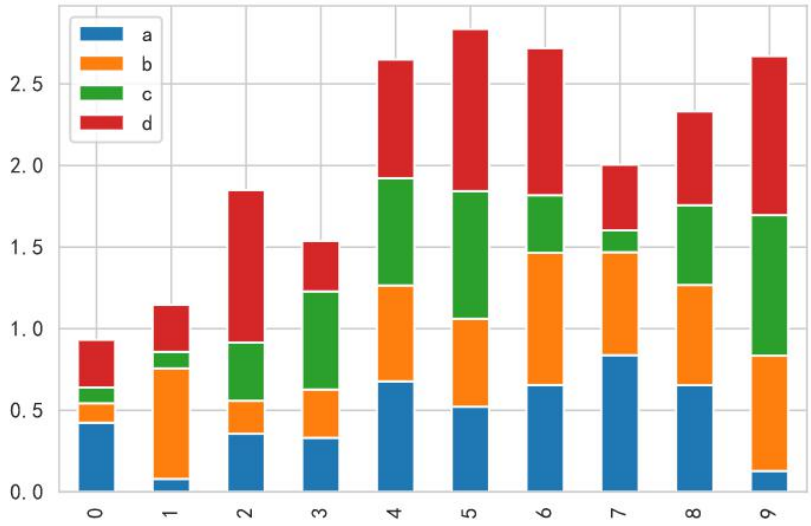
df.plot.kind()

```
df ['身高'].plot .box (title='第一个图',ylim= (60,80))
df.体重.plot(figsize = (12,8 ))
df.groupby(df.性别).体重.plot( )
```

## 10.2 条图


14
12
10
8
6
4
2
0
统计方法
概率统计
都学习过
编程技术
都未学过


1.0
0.8
0.6
0.4
0.2
0.0
a
b
c
d
0
1
2
3
4
5
6
7
8
9

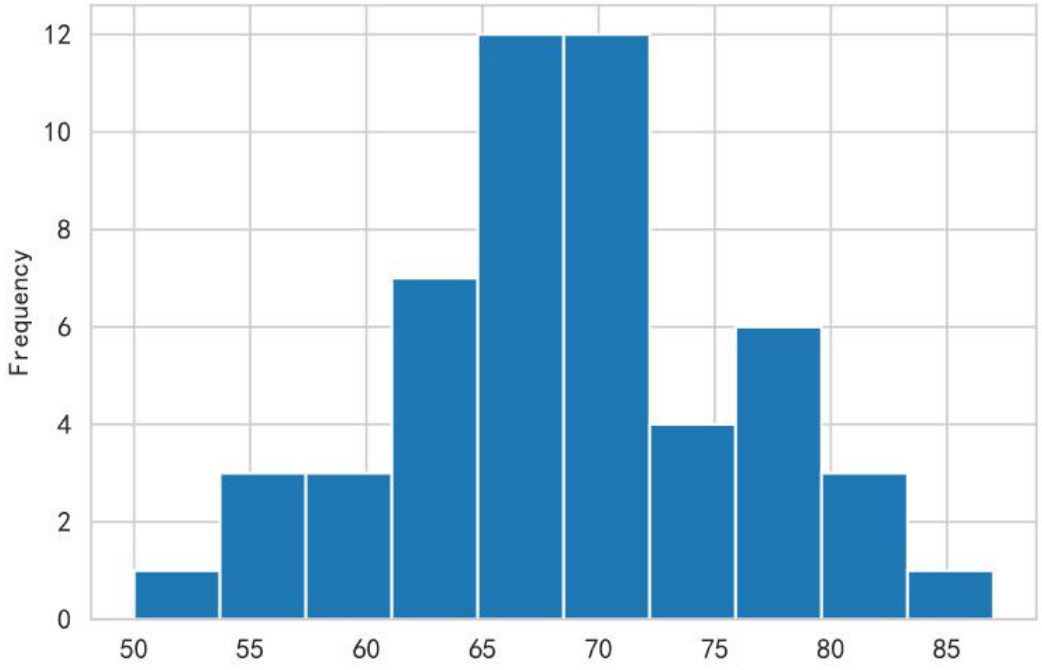



```
pd.value_counts(df.课程).plot.bar()

pd.value_counts(df.课程).plot.barh()


import numpy as np
df2 = pd.DataFrame(np.random.rand(10,4),columns=['a','b','c','d'])
df2.plot.bar( )
df2.plot.bar( stacked= True )
```

## 10.3 直方图




plot.hist(

by :在 df 中用于分组的变量列(绘制为图组)

bins = 10 :需要拆分的组数

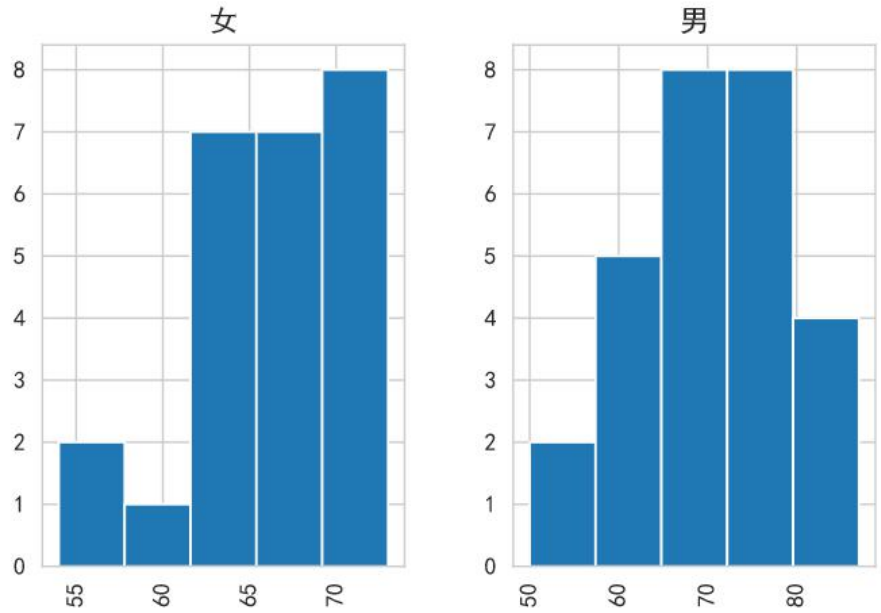
)

```
df.体重.plot.hist(bins=10)
```

hist(

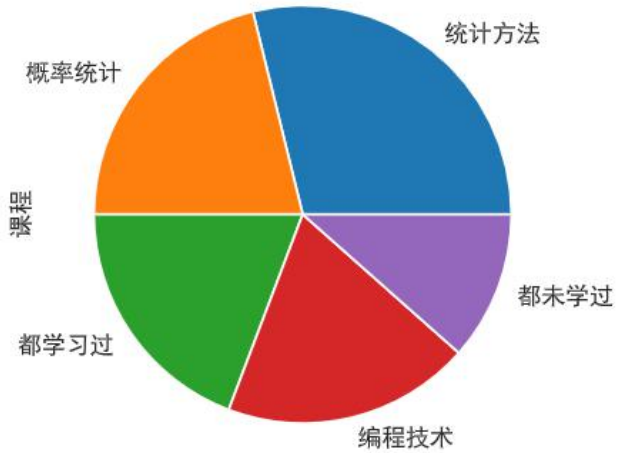
by :在 df 中用于分组的变量列(绘制为图组)

)




```
df.体重.hist(by=df.性别, bins=5)
```

## 10.4 饼图




```
plot.pie(
```

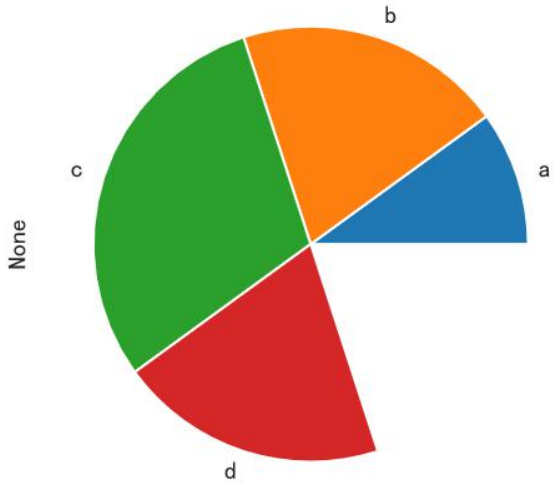
y :指定需要绘制的变量列名称

subplots = False :多个变量列时要求分组绘图

)

```
df.课程.value_counts( ).plot.pie( )

pd.value_counts(df.课程).plot.pie( )
```




```
pd.Series([0.1,0.2,0.3,0.2],index=['a','b','c','d']).plot.pie( )
```

## 10.5 箱图

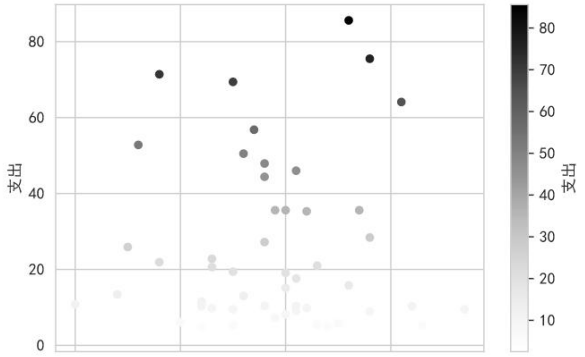
plot.box(

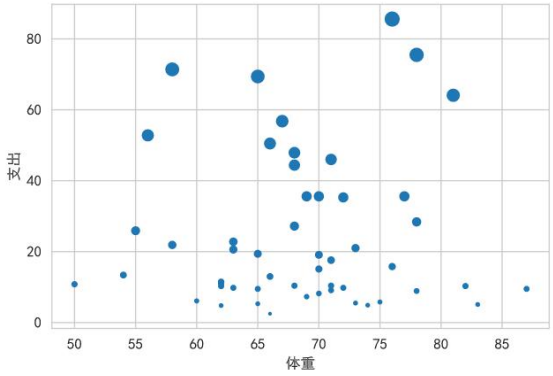
vert = True :是否纵向绘图

)

```
df.plot.box( vert = False )
```

## 10.6 散点图




plot.scatter（

S :控制散点大小的变量列，不能使用 df 中的简写方式指定

C :控制散点颜色的变量列

）

```
df.plot.scatter('体重','支出')
```

```
df.plot.scatter( x='体重', y='支出', c='支出')

df.plot.scatter( x='体重', y='支出', s=df.支出)
```

# 11 数据特征的分析探索

## 11.1 数值变量的基本描述

df.describe（

percentiles :需要输出的百分位数，列表格式提供，如[.25, .5, .75]

include = 'None ':要求纳入分析的变量类型白名单

None （ default ）:只纳入数值变量列

A list-like of dtypes :列表格式提供希望纳入的类型

all :全部纳入

exclude :要求剔除出分析的变量类型黑名单，选项同上

）

```
df.describe( )
df.describe(include = ‘all’ )
```

## 11.2 分类变量的频数统计

Series.value_counts（

normalize = False :是否返回构成比而不是原始频数

sort = True :是否按照频数排序（否则按照原始顺序排列）

ascending = False : 是否升序排列

bins :对数值变量直接进行分段，可看作是 pd.cut 的简便用法

dropna = True :结果中是否包括 NaN

)

```
pd. value_counts （ df2 .体重 ）
pd. value_counts （ df2 .体重,normalize=True ）
pd. value_counts （ df2 .体重，sort = False ）
df2.体重.value_counts ( bins =10)
```

## 11.3 交叉表/数据透视表

**df.pivot_table**（

行列设定

index / columns :行变量/列变量，多个时以 list 形式提供

单元格设定

values :在单元格中需要汇总的变量列，可不写

aggfunc = numpy.mean : 相应的汇总函数

汇总设定

margins = False :是否加入行列汇总

margins_name = 'All':汇总行/列的名称

缺失值处理

fill_value = None :用于替换缺失值的数值

dropna = True :

)

**pd.crosstab**（

选项和 pivot_table 几乎相同

相对而言需要打更多字母，因此使用更麻烦

但是计算频数最方便

输出格式为数据框 ）

```
df.pivot_table(index =['课程','性别'],columns='软件',aggfunc = sum)

pd.crosstab(index =[df.课程],columns=df.软件,values=df.体重,aggfunc=sum)
```

# 12 如何优化 Pandas

除非对相应的优化手段已经非常熟悉，否则代码的可读性应当被放在首位。过早优化是万恶之源！

pandas 为了易用性，确实牺牲了一些效率，但同时也预留了相应的优化路径。因此如果要进行优化，熟悉并优先使用 pandas 自身提供的优化套路至关重要。

尽量使用 pandas（或者 numpy）内置的函数进行运算，一般效率都会更高。

在可以用几种内部函数实现相同需求时，最好进行计算效率的比较，差距可能很大。

pandas 官方提供的讨论如何进行优化的文档：

https://pandas.pydata.org/pandas-docs/stable/user_guide/enhancingperf.html

## 12.1 学会使用各种计时工具

**%time 和%timeit，以及%%timeit**（需要在 IPython 下才可以使用）

```
%time df['new_col'] = df.体重-df.支出
```

```
%timeit df['new_col'] = df.体重-df.支出


%%timeit
df['new_col'] = df.体重-df.支出
df['new_col'] = df.体重-df.支出
df['new_col'] = df.体重-df.支出
df['new_col'] = df.体重-df.支出
```

**python 时间方式**

```
from datetime import datetime
import time
t1 = datetime.now()
t2 = time.time()

time.sleep(1)

print(datetime.now()- t1)
print(time.time()-t2)
```

**用 line_profiler 做深入分析**

```
from line_profiler import LineProfiler
import random

def m_test(numbers):
    s = sum(numbers)
    l = [numbers[i]/50 for i in range(len(numbers))]
```

```
    m = ['hello' + str(numbers[i]) for i in range(len(numbers))]
lp = LineProfiler() # 定义一个 LineProfiler 对象
lp_wrapper = lp(m_test) # 用 LineProfiler 对象监控需要分析的函数
```

## 12.2 超大数据文件的处理

超大数据文件在使用 pandas 进行处理时可能需要考虑两个问题：读取速度，内存用量。

一般情况会考虑读入部分数据进行代码编写和调试，然后再对完整数据进行处理。

在数据读入和处理时需要加快处理速度，减少资源占用。

### 12.2.1 一些基本原则

当明确知道数据列的取值范围时，读取数据时可以使用 dtype 参数来手动指定类型，

如 np.uint8 或者 np.int16，否则默认的 np.int64 类型等内存开销明显非常大。

尽量少用类型模糊的 object，改用更明确的 category 等（用 astype()转换）。

对类别取值较少，但案例数极多的变量，读入后尽量转换为数值代码进行处理。

```
data = pd.DataFrame({"a" : [0,1, 2, 3, 4, 5, 6, 7, 8, 9],
"b" : ["祖安狂人","祖安狂人","冰晶凤凰","冰晶凤凰",
"祖安狂人","祖安狂人","祖安狂人","冰晶凤凰", "冰晶凤凰","祖安狂人"]})
print(data)
data.info()

data['a'] = pd.to_numeric(data['a'], downcast = 'integer')
data['b'] = data['b'].astype('category')
print(data)
```

```
data.b.head() # category 格式明显更节省内存
data.info()


data.b.astype('str') # 必要时 category 也可以转换回 str
```

### 12.2.2 对文件进行分段读取

使用 chunksize 参数

```
dftmp = pd.read_csv('stu_date.csv', usecols = [0,2,3,4,5,6,7], chunksize = 5)


type(dftmp) # 注意得到的并不是一个数据框，而是 TextFileReader


n = 0
for item in dftmp: # 注意重复运行之后的效果
    print(item)
    n += 1
    if n > 2:
            break
```

使用 iterator 参数和 get_chunk()组合

```
dftmp = pd.read_csv('stu_date.csv', usecols = [0,2,3,4,5,6,7], iterator = True)


type(dftmp) # 注意得到的并不是一个数据框，而是 TextFileReader
```

```
dfiter.get_chunk(10) # 注意重复运行之后的效果
```

# 12.3 如何优化 pandas 的代码

## 12.3.1 简单优化

尽量不要在 pandas 中使用循环

如果循环很难避免，尽量在循环体中使用 numpy 做计算。

```
%%timeit

# 标准的行/列遍历循环方式效率最差，

df['new'] = 0
for i in range(df.shape[0]):
    df.iloc[i, 8] = df.iloc[i, 4] + 10



# apply 自定义函数/外部函数效率稍差
def m_add(x):
    return x + 10

%timeit df["new"] = df.支出.apply(lambda x : m_add(x))

#%%
# 在 apply 中应用内置函数方式多数情况下速度更快
```

```
%timeit df["new"] = df.支出.apply(lambda x: x + 10)


#%%
# 直接应用原生内置函数方式速度最快
%timeit df["new"] = df.支出 + 10
```

### 12.3.2 如何进行多列数据的计算

同时涉及多列数据的计算不仅需要考虑速度优化问题，还需要考虑代码简洁性的问题。

```
%%timeit
def m_add(a, b, c):
    return a + b + c
df['new'] = df.apply(lambda x :m_add(x['支出'], x['体重'], x['身高']), axis = 1)


%%timeit
# lambda 多参方式可能效率反而更低
df['new'] = df[['支出', '体重', '身高']].apply(lambda x :x[0] + x[1] + x[2], axis = 1)


# 最香
%timeit df['new'] = df['支出'] + df['体重'] + df['身高']
```

pd.eval()命令

pd.eval()的功能仍是给表达式估值，但是基于 pandas 时，就可以直接利用列名等信息用于计算。

```
#  eval()默认会使用 numexpr 而不是 python 计算引擎，复杂计算时效率很高
# engine 参数在调用 numexpr 失败时会切换为 python 引擎，一般不用干涉
%timeit df.eval('new = 支出 + 体重 + 身高', inplace = True)

# eval()默认会使用 numexpr 而不是 python 计算引擎，复杂计算时效率很高
%timeit df.eval('new = 支出 + 体重 + 身高', engine = 'python', inplace = True)

%%timeit
df['n1'] = df.支出 + 1 df['n2'] = df.支出 + 2 df['n3'] = df.体重 + 3
df.head()
```

pd.在 eval()表达式中进一步使用局部变量

@varname：在表达式中使用名称为 varname 的 Python 局部变量。该@符号在 query()和 eval()中均可使用。

```
%%timeit
arg = 3
bj.eval('new = 支出 + 体重 + 身高 + @arg').head()
```

## 12.4 利用各种 pandas 加速外挂

大部分外挂包都是基于 linux 环境，windows 下能用的不多。大部分外挂包还处于测试阶段，功能上并未完善。

### 12.4.1 numba

对编写的自定义函数进行编译，以提高运行效率。

A Just-In-Time Compiler for Numerical Functions in Python 具体使用的是 C++编写的 LLVM(Low Level Virtual Machine) compiler 实际上主要是针对 numpy 库进行优化编译，并不仅限于 pandas

```
import random


def monte_carlo_pi(nsamples): acc = 0
for i in range(nsamples):
    x =  random.random()
    y = random.random()
    if (x ** 2 + y ** 2) < 1.0:
            acc += 1
    return 4.0 * acc / nsamples
%timeit monte_carlo_pi(20) # 运行被监控的函数，获取各部分占用的运行时间数据



from numba import jit
@jit(nopython = True) # nopython 模式下性能最好
def monte_carlo_pi(nsamples): acc = 0
for i in range(nsamples):
    x =  random.random()
    y = random.random()
    if (x ** 2 + y ** 2) < 1.0:
            acc += 1
    return 4.0 * acc / nsamples
%timeit monte_carlo_pi(20)
```

```
from numba import jit
@jit
def monte_carlo_pi(nsamples): acc = 0
for i in range(nsamples):
    x =  random.random()
    y = random.random()
    if (x ** 2 + y ** 2) < 1.0:
            acc += 1
    return 4.0 * acc / nsamples
%timeit monte_carlo_pi(20)
```

### 12.4.2 swifter

对 apply 函数进行并行操作

是专门针对 pandas 进行优化的工具包

```
df = pd.DataFrame({'x': [1, 2, 3, 4], 'y': [5, 6, 7, 8]})


# runs on single core
%timeit df['x2'] = df['x'].apply(lambda x: x**2)


import swifter
# runs on multiple cores
%timeit df['x2'] = df['x'].swifter.apply(lambda x: x**2)



%timeit df['agg'] = df.apply(lambda x: x.sum() - x.min())
# use swifter apply on whole dataframe
%timeit df['agg'] = df.swifter.apply(lambda x: x.sum() - x.min())
```