Problem Formulation

$$
\begin{aligned}
max \quad & \sum_{i=1}^{m}\sum_{j=1}^{n} p_j x_{ij} \\
s.t. \quad & \sum_{j=1}^{n} w_j x_{ij} \leq c_i \\
& \sum_{i=1}^{m} x_{ij} \leq 1 \\
& x_{ij} \in \{0,1\}
\end{aligned}
$$

$$
where \quad x_{ij} = \begin{cases} \text{1 if item j is assigned to knapsack i} \\ \text{0 otherwise} \end{cases}
$$

Solution Encoding

The solution is represented with an n by b matrix where n is the number of items and b is the number of containers. Entry (i,j) represents whether item i is assigned to container j with binary encoding.

Solving using Pyomo

In [20]:

```
import random
import pyomo.environ as pyo
from pyomo.environ import *
```

In [21]:

```
random.seed(1)
n=100 #number of objects
b=5 #number of bins
c=50

#Generate random locations
vj=random.choices(range(10, 100),k=n)
wj=random.choices(range(5, 20),k=n)
```

In [22]:

```
model=pyo.ConcreteModel()

model.i=RangeSet(0,b-1)
model.j=RangeSet(0,n-1)
model.p=Param(model.j,initialize=vj)
model.w=Param(model.j,initialize=wj)
model.x=Var(model.j,model.i,within=Binary)

def objective(model):
    result=0
    for j in model.j:
        for i in model.i:
            result+=model.p[j]*model.x[j,i]
    return result
model.cost=Objective(rule=objective,sense=maximize)

def constraint1(model,i):
    total=0
    for j in model.j:
        total+=model.w[j]*model.x[j,i]
    return total<=c
model.cons1=Constraint(model.i,rule=constraint1)

def constraint2(model,j):
    total=0
    for i in model.i:
        total+=model.x[j,i]
```

```
    return total<=1
model.cons2=Constraint(model.j,rule=constraint2)

instance=model.create_instance()
opt=pyo.SolverFactory('gurobi')
opt.solve(instance,options={'TimeLimit': 10000},tee=True)
```

```
Set parameter Username
Academic license - for non-commercial use only - expires 2023-01-29
Read LP format model from file C:\Users\dell\AppData\Local\Temp\tmpb18oq4ov.pyomo.lp
Reading time = 0.01 seconds
x501: 106 rows, 501 columns, 1001 nonzeros
Set parameter TimeLimit to value 10000
Gurobi Optimizer version 9.5.0 build v9.5.0rc5 (win64)
Thread count: 4 physical cores, 8 logical processors, using up to 8 threads
Optimize a model with 106 rows, 501 columns and 1001 nonzeros
Model fingerprint: 0xfaca7c7f
Variable types: 1 continuous, 500 integer (500 binary)
Coefficient statistics:
  Matrix range     [1e+00, 2e+01]
  Objective range  [1e+01, 1e+02]
  Bounds range     [1e+00, 1e+00]
  RHS range        [1e+00, 5e+01]
Found heuristic solution: objective 1124.0000000
Presolve removed 1 rows and 1 columns
Presolve time: 0.00s
Presolved: 105 rows, 500 columns, 1000 nonzeros
Variable types: 0 continuous, 500 integer (500 binary)
Found heuristic solution: objective 2053.0000000

Root relaxation: objective 2.356357e+03, 91 iterations, 0.00 seconds (0.00 work units)

    Nodes    |    Current Node    |     Objective Bounds      |     Work
 Expl Unexpl |  Obj  Depth IntInf | Incumbent    BestBd   Gap | It/Node Time

     0     0 2356.35714    0    8 2053.00000 2356.35714  14.8%     -    0s
H    0     0                    2288.0000000 2356.35714  2.99%     -    0s
H    0     0                    2356.0000000 2356.35714  0.02%     -    0s
     0     0 2356.35714    0    8 2356.00000 2356.35714  0.02%     -    0s

Explored 1 nodes (91 simplex iterations) in 0.02 seconds (0.00 work units)
Thread count was 8 (of 8 available processors)

Solution count 4: 2356 2288 2053 1124

Optimal solution found (tolerance 1.00e-04)
Best objective 2.356000000000e+03, best bound 2.356000000000e+03, gap 0.0000%
```

Out[22]:

{'Problem': [{'Name': 'x501', 'Lower bound': 2356.0, 'Upper bound': 2356.0, 'Number of objectives': 1, 'Number of constraints': 106, 'Number of variables': 501, 'Number of binary variables': 500, 'Number of integer variables': 500, 'Number of continuous variables': 1, 'Number of nonzeros': 1001, 'Sense': 'maximize'}], 'Solver': [{'Status': 'ok', 'Return code': '0', 'Message': 'Model was solved to optimality (subject to tolerances), and an optimal solution is available.', 'Termination condition': 'optimal', 'Termination message': 'Model was solved to optimality (subject to tolerances), and an optimal solution is available.', 'Wall time': '0.022939682006835938', 'Error rc': 0, 'Time': 0.23836326599121094}], 'Solution': [OrderedDict([('number of solutions', 0), ('number of solutions displayed', 0)])]}

Random Sampling

In [23]:

```
import random
import numpy as np
import copy
```

```
import time
import matplotlib.pyplot as plt
import math
```

In [24]:

```
#convert the binary result to a table.
import pandas as pd

def final_table(x):
        container=list(range(1,b+1))
        container.append('Total')
        package=[]
        for i in range(b):
            package.append([])
            for j in range(n):
                if x[j,i]==1:
                    package[i].append(j+1)
        package.append('')

        profits=[]
        weights=[]
        for i in range(b):
            profit=0
            weight=0
            for j in package[i]:
                profit+=vj[j-1]
                weight+=wj[j-1]
            profits.append(profit)
            weights.append(weight)
        profits.append(sum(profits))
        weights.append('')
        data={'Container':container,'Packages':package,'Profit':profits,'Weight':weigh
        return pd.DataFrame(data)
```

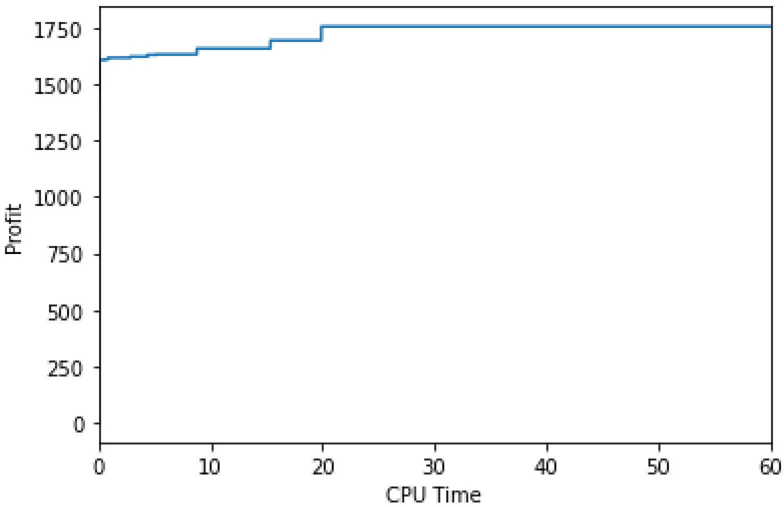
In [39]:

```
result=np.zeros((n,b))
iteration=0
best_value=0
start=time.time()
end=time.time()
time_list=[0]
value_list=[0]

while end-start<60:
    #randomly generate a result
    for i in range(n):
        #whether to put the item in a knapsack
        a=random.randint(0,3)
        if a==1:
            #decide to put in which knapsack
            j=random.randint(0,b-1)
            result[i,j]=1

    #check whether the result is valid
    valid=True
    overweight={}
    for i in range(b):
        weight=0
        for j in range(n):
            weight+=wj[j]*result[j,i]
        if weight>c:
            valid=False
            overweight[i]=weight
```

```
    #randomly remove items until it meets the constraint
    if valid==False:
        for i,w in overweight.items():
            in_bag=[]
            for j in range(n):
                if result[j,i]==1:
                    in_bag.append(j)
            while w>c:
                a=random.randint(0,len(in_bag)-1)
                result[in_bag[a],i]=0
                w-=wj[in_bag[a]]
                in_bag.pop(a)

    #calculate the new value
    cur_value=0
    for i in range(n):
        for j in range(b):
            cur_value+=vj[i]*result[i,j]
    if cur_value>best_value:
        best_value=cur_value
    end=time.time()
    time_list.append(end-start)
    value_list.append(best_value)
print(final_table(result))
plt.xlabel('CPU Time')
plt.ylabel('Profit')
plt.xlim((0,60))
plt.plot(time_list,value_list)
```

```
  Container          Packages  Profit  Weight
0         1      [54, 74, 83]   246.0      49
1         2  [65, 75, 82, 87]   261.0      42
2         3  [18, 29, 30, 59]   279.0      44
3         4  [13, 16, 22, 80]   308.0      40
4         5  [11, 47, 55, 89]   268.0      48
5     Total                    1758.0
```

Out[39]: [<matplotlib.lines.Line2D at 0x26d40445940>]



Local Search

The local search algorithm, as well as other algorithms for the rest of the exercises, uses first descent and bit flipping due to the large solution space and binary encoding.It seems that there is no improvement in the result, but there are some improvements at the beginning, which are not visible on the plot and then quickly converges to the local optimum.

In [25]:

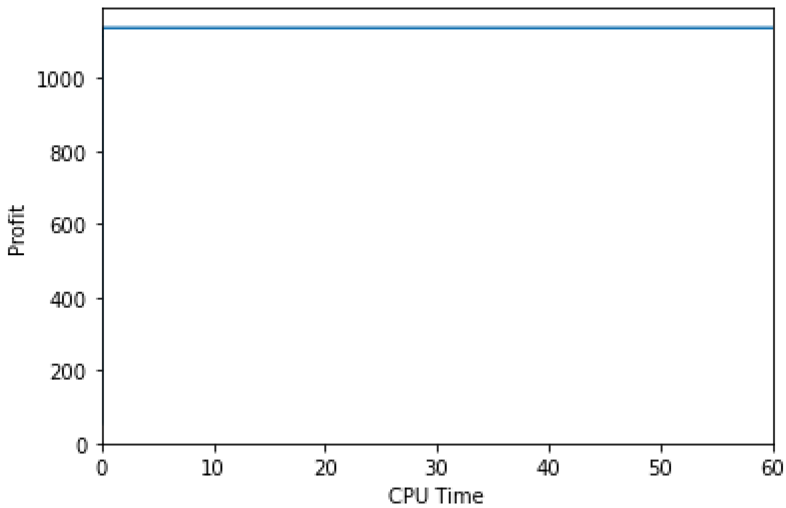
```
result=np.zeros((n,b))
iteration=0
best_value=0
start=time.time()
time_list=[0]
value_list=[0]

#generate a random solution
for i in range(n):
    #whether to put the item in a knapsack
    a=random.randint(0,50)
    if a==1:
        #decide to put in which knapsack
        j=random.randint(0,b-1)
        result[i,j]=1

#check whether the result is valid
valid=True
overweight={}
for i in range(b):
    weight=0
    for j in range(n):
        weight+=wj[j]*result[j,i]
    if weight>c:
        valid=False
        overweight[i]=weight

#randomly remove items until it meets the constraint
if valid==False:
    for i,w in overweight.items():
        in_bag=[]
        for j in range(n):
            if result[j,i]==1:
                in_bag.append(j)
        while w>c:
            a=random.randint(0,len(in_bag)-1)
            result[in_bag[a],i]=0
            w-=wj[in_bag[a]]
            in_bag.pop(a)

end=time.time()
for i in range(n):
    for j in range(b):
        best_value+=vj[i]*result[i,j]
value_list=[best_value]

while end-start<60:
    pre_result=copy.deepcopy(result)

    #choose which item to change
    i=random.randint(0,n-1)

    #check whether the item is already put in a bag, if so, remove it.
    for j in range(b):
        if result[i,j]==1:
            result[i,j]=0

    #choose a random bag to put it in
    j=random.randint(0,b-1)
    result[i,j]=1

    #check whether the result is valid
    valid=True
    for i in range(b):
```

```
        weight=0
        for j in range(n):
            weight+=wj[j]*result[j,i]
        if weight>c:
            valid=False
            result=pre_result

    #calculate the new value
    if valid==True:
        cur_value=0
        for i in range(n):
            for j in range(b):
                cur_value+=vj[i]*result[i,j]
        if cur_value>best_value:
            best_value=cur_value
        else:
            result=pre_result
    end=time.time()
    time_list.append(end-start)
    value_list.append(best_value)

print(final_table(result))
plt.xlabel('CPU Time')
plt.ylabel('Profit')
plt.xlim((0,60))
plt.plot(time_list,value_list)
```

	Container	Packages	Profit	Weight
0	1	[6, 21, 29, 40, 41, 52]	298	48
1	2	[1, 5, 22, 54]	220	49
2	3	[27, 51, 70, 71]	148	49
3	4	[15, 56, 77, 92]	168	48
4	5	[12, 28, 75, 93, 100]	303	48
5	Total		1137	

Out[25]:

```
[&lt;matplotlib.lines.Line2D at 0x2494e68a850&gt;]
```



Simulated Annealing

The initial temperature is 1000 and it cools down to 90% for each iteration, which is Tn=(0.9^n)*T.

In [26]:

```
result=np.zeros((n,b))
iteration=0
best_value=0
start=time.time()
time_list=[0]
```

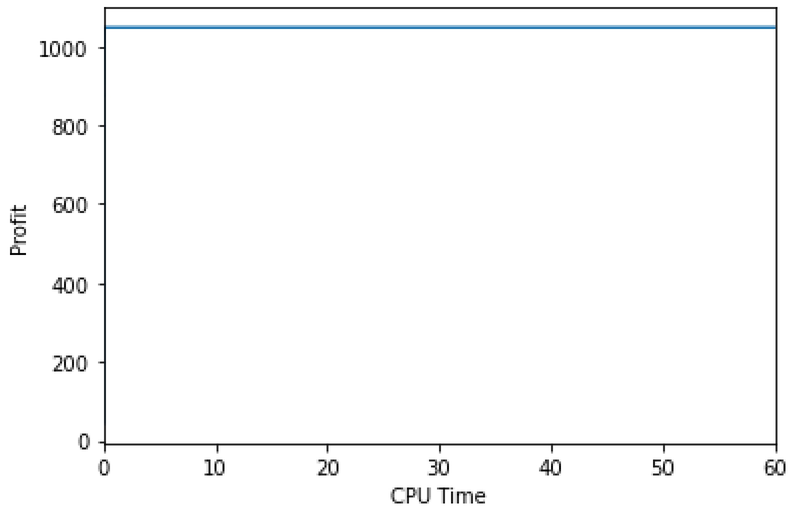
```
value_list=[0]
T=1000

#generate a random solution
for i in range(n):
    #whether to put the item in a knapsack
    a=random.randint(0,50)
    if a==1:
        #decide to put in which knapsack
        j=random.randint(0,b-1)
        result[i,j]=1

#check whether the result is valid
valid=True
overweight={}
for i in range(b):
    weight=0
    for j in range(n):
        weight+=wj[j]*result[j,i]
    if weight>c:
        valid=False
        overweight[i]=weight

#randomly remove items until it meets the constraint
if valid==False:
    for i,w in overweight.items():
        in_bag=[]
        for j in range(n):
            if result[j,i]==1:
                in_bag.append(j)
        while w>c:
            a=random.randint(0,len(in_bag)-1)
            result[in_bag[a],i]=0
            w-=wj[in_bag[a]]
            in_bag.pop(a)
for i in range(n):
    for j in range(b):
        best_value+=vj[i]*result[i,j]
value_list=[best_value]
end=time.time()
while end-start<60:
    pre_result=copy.deepcopy(result)

    #choose which item to change
    i=random.randint(0,n-1)

    #check whether the item is already put in a bag, if so, remove it.
    for j in range(b):
        if result[i,j]==1:
            result[i,j]=0

    #choose a random bag to put it in
    j=random.randint(0,b-1)
    result[i,j]=1

    #check whether the result is valid
    valid=True
    for i in range(b):
        weight=0
        for j in range(n):
            weight+=wj[j]*result[j,i]
        if weight>c:
            valid=False
            result=pre_result
```

```
    #calculate the new value
    if valid==True:
        cur_value=0
        for i in range(n):
            for j in range(b):
                cur_value+=vj[i]*result[i,j]
        p=random.random()
        if cur_value>best_value or p>math.exp((cur_value-best_value)/T):
            best_value=cur_value
        else:
            result=pre_result
    end=time.time()
    time_list.append(end-start)
    value_list.append(best_value)
    T=0.9*T

print(final_table(result))
plt.xlabel('CPU Time')
plt.ylabel('Profit')
plt.xlim((0,60))
plt.plot(time_list,value_list)
```

```
  Container                Packages  Profit  Weight
0         1       [14, 49, 87, 97]     238      46
1         2   [1, 25, 81, 92, 100]     188      50
2         3        [4, 20, 38, 98]     160      49
3         4  [24, 34, 45, 61, 99]     254      49
4         5  [10, 28, 48, 51, 94]     208      50
5     Total                           1048
```

Out[26]:

```
[<matplotlib.lines.Line2D at 0x2494e35bf70>]
```



Greedy Algorithm

The greedy algorithm first calculates the value-weight ratio of every item and the arrange them in a descending order. Then, the algorithm tries to put the items in the container base on the order previously obtained until nothing can fit inside the containers. This generates an initial solution used for the simulated annealing.

In [27]:

```
start=time.time()
#generate an initial solution using greedy algorithm
z=[] #value/weight ratio
for i in range(len(wj)):
    z.append(wj[i]/vj[i])
data={'value':vj,'weight':wj,'ratio':z}
```

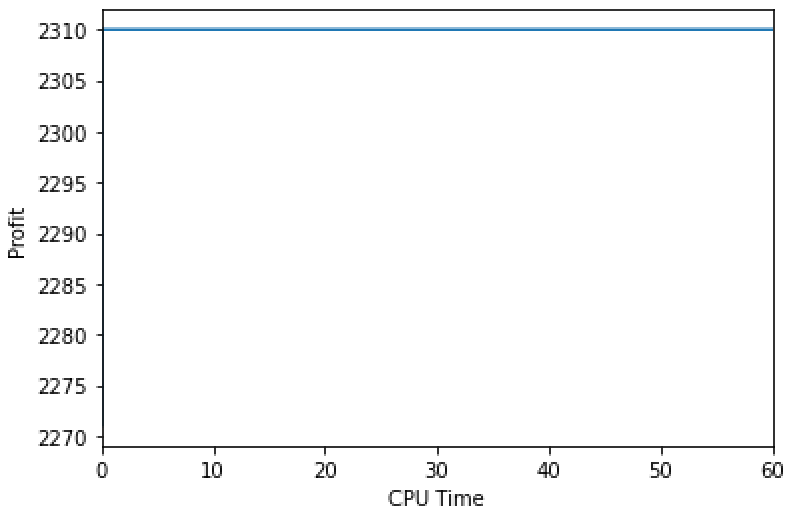
```
table=pd.DataFrame(data).sort_values('ratio')
order=list(table.index)

result=np.zeros((n,b))
bag=0
total_w=0
for i in order:
    result[i,bag]=1
    total_w+=wj[i]
    if total_w>c:
        if bag<b-1:
            result[i,bag]=0
            result[i,bag+1]=1
            bag+=1
            total_w=wj[i]
        else:
            result[i,bag]=0
            bag+=1
    if bag==b:
        break

#using simulated annealing to solve the problem
iteration=0
best_value=0
time_list=[0]
value_list=[0]
T=1000
end=time.time()
for i in range(n):
    for j in range(b):
        best_value+=vj[i]*result[i,j]
value_list=[best_value]
while end-start<60:
    pre_result=copy.deepcopy(result)

    #choose which item to change
    i=random.randint(0,n-1)

    #check whether the item is already put in a bag, if so, remove it.
    for j in range(b):
        if result[i,j]==1:
            result[i,j]=0

    #choose a random bag to put it in
    j=random.randint(0,b-1)
    result[i,j]=1

    #check whether the result is valid
    valid=True
    for i in range(b):
        weight=0
        for j in range(n):
            weight+=wj[j]*result[j,i]
        if weight>c:
            valid=False
            result=pre_result

    #calculate the new value
    if valid==True:
        cur_value=0
        for i in range(n):
            for j in range(b):
                cur_value+=vj[i]*result[i,j]
        p=random.random()
```

```
        if cur_value>best_value or p>math.exp((cur_value-best_value)/T):
            best_value=cur_value
        else:
            result=pre_result
    end=time.time()
    time_list.append(end-start)
    value_list.append(best_value)
    T=0.9*T

print(final_table(result))
plt.xlabel('CPU Time')
plt.ylabel('Profit')
plt.xlim((0,60))
plt.plot(time_list,value_list)
```

```
  Container                            Packages  Profit Weight
0         1  [13, 41, 45, 52, 53, 80, 86, 97]     677     46
1         2      [44, 59, 69, 75, 82, 93, 96]     488     49
2         3          [26, 29, 42, 63, 91, 95]     441     50
3         4           [8, 24, 39, 47, 51, 55]     377     49
4         5               [7, 15, 23, 79, 99]     327     49
5     Total                                      2310
```

Out[27]:

```
[<matplotlib.lines.Line2D at 0x2494e4a8970>]
```



Greedy Algorithm with Simulated Annealing

This is done in the previous section. Comparing to the result of optimization obtained in Section 2, the results are already quite close to the ones obtained using optimization, and there isn't much improvement during the simulated annealing process base on the graph. Therefore, I decided to use a better way to find the initial solution. I used recursion to solve the problem. For the first container, I optimize the result using every item, then I remove the items in container put in the first container and used the rest of the items for the second container. The process is repeated for every container sequentially. It is not the way to obtain the global optimum but should be better than the algorithm in the previous section. The rest is just the same simulated annealing process as Section 6. Luckily, it turns out that the initial solution I found is already the optimal solution obtained from Section 2 and it takes a shorter time for instance 2, which is about 6 minutes.

In [28]:

```
start=time.time()

#use optimisation to obtain a better initial solution
arrangement=[]
best_val=0
```

```python
    v=copy.deepcopy(vj)
    w=copy.deepcopy(wj)
    for a in range(b):
        table=np.zeros([len(v)+1,c+1])
        plan=[]
        for i in range(len(v)+1):
            plan.append([])
        for i in range(len(v)+1):
            for j in range(c+1):
                plan[i].append([])

        for i in range(1,len(v)+1):
            for j in range(1,c+1):
                if w[i-1]>j:
                    table[i,j]=table[i-1][j]
                    plan[i][j]=plan[i-1][j]
                else:
                    table[i,j]=max(table[i-1][j],table[i-1][j-w[i-1]]+v[i-1])
                    if table[i,j]==table[i-1][j]:
                        plan[i][j]=plan[i-1][j]
                    else:
                        plan[i][j]=plan[i-1][j-w[i-1]]+[i]
        arrangement.append(plan[-1][-1])
        best_val+=table[-1][-1]
        x=plan[-1][-1]
        for i in x:
            v[i-1]=0
            w[i-1]=10000000

    result=np.zeros((n,b))
    for i in range(len(arrangement)):
        for j in arrangement[i]:
            result[j-1][i]=1

    iteration=0
    best_value=best_val
    time_list=[0]
    value_list=[best_value]
    T=1000
    end=time.time()

    while end-start<60:
        pre_result=copy.deepcopy(result)

        #choose which item to change
        i=random.randint(0,n-1)

        #check whether the item is already put in a bag, if so, remove it.
        for j in range(b):
            if result[i,j]==1:
                result[i,j]=0

        #choose a random bag to put it in
        j=random.randint(0,b-1)
        result[i,j]=1

        #check whether the result is valid
        valid=True
        for i in range(b):
            weight=0
            for j in range(n):
                weight+=wj[j]*result[j,i]
            if weight>c:
                valid=False
```

```
            result=pre_result

    #calculate the new value
    if valid==True:
        cur_value=0
        for i in range(n):
            for j in range(b):
                cur_value+=vj[i]*result[i,j]
        p=random.random()
        if cur_value>best_value or p>math.exp((cur_value-best_value)/T):
            best_value=cur_value
        else:
            result=pre_result
    end=time.time()
    time_list.append(end-start)
    value_list.append(best_value)
    T=0.9*T

print(final_table(result))
plt.xlabel('CPU Time')
plt.ylabel('Profit')
plt.xlim((0,60))
plt.plot(time_list,value_list)
```

```
  Container                                Packages  Profit Weight
0         1  [13, 41, 52, 53, 69, 80, 82, 86, 97]     717     50
1         2                  [42, 45, 59, 75, 91, 96]     512     50
2         3                  [26, 29, 47, 63, 93, 95]     432     50
3         4                    [7, 8, 24, 39, 55, 79]     369     50
4         5                      [11, 15, 23, 51, 99]     326     50
5     Total                                              2356
```

Out[28]:

```
[<matplotlib.lines.Line2D at 0x249523de160>]
```