数据科学 1：Numpy 库

QuantEcon, 张帆

2019 年 12 月 16 日

1 科学计算

本部分主要介绍与科学计算相关的知识，这些内容是现代经济学、数据科学和统计学的核心。具体来讲，有以下部分：1. NumPy 库简介，该库是 Python 语言的一个扩展程序库，支持大量的维度数组与矩阵运算，此外也针对数组运算提供大量的数学函数库。2. 基于 matplotlib 库的数据可视化，是 Python 的绘图库，它可与 NumPy 一起使用 3. 温习一些线性代数关键概念 4. 回顾和经济模拟相关的概率知识 4. 最优化问题的处理

相关资料：

NumPy 教程

NumPy 官网

1.1 1.Numpy 简介

Numpy 库为我们提供了一种新数据类型：数组 (array)。它和列表类似，但是其内部的数据组织方式被做了调整。这使得 Numpy 可以：- 更有效地执行科学计算 - 通过函数运行机器学习和统计所需的线代运算

开始之前，请务必导入 Numpy，为了后续调用简便可将其起个别名 “np”

```
In [16]: import numpy as np
```

我们可以使用列表或元组来创建数组，以下采用列表来创建数组。

1.1.1 （1）一维数组

数组 (array) 是一个多维的数值网格

```
In [15]: # 创建一维数组
         x_1d = np.array([1,2,3])
         print(x_1d)
```

```
[1 2 3]
```

```
In [4]: # 数组索引
        print(x_1d[0])
        print(x_1d[0:2])
```

```
1
[1 2]
```

```
In [6]: # 全部值
        print(x_1d[:])
```

```
[1 2 3]
```

1.1.2 （2）二维数组

```
In [9]: # 创建二维数组
        x_2d = np.array([[1,2,3], [4,5,6], [7,8,9]])
        print(x_2d)
```

```
[[1 2 3]
 [4 5 6]
 [7 8 9]]
```

此时，数据是列和行两个维度，具体是三行三列。此时若想访问特定数值，则先键入行数，再键入列数。

```
In [11]: # 若想获得数字 6
         print(x_2d[1,2])
```

```
6
```

```
In [13]: # 若想获得第一行
         print(x_2d[0,:])
```

```
[1 2 3]
```

In [14]:
```
# 若想获得第二列
print(x_2d[:,1])
```

```
[2 5 8]
```

1.1.3 （3）三维数组

In [17]:
```
x_3d_list = [[[1,2,3], [4,5,6]], [[10,20,30], [40,50,60]]]
x_3d = np.array(x_3d_list)
print(x_3d)
```

```
[[[ 1  2  3]
  [ 4  5  6]]

 [[10 20 30]
  [40 50 60]]]
```

In [18]:
```
# 输出第一个矩阵
print(x_3d[0])
```

```
[[1 2 3]
 [4 5 6]]
```

In [21]:
```
print(x_3d[0, :, :])
```

```
[[1 2 3]
 [4 5 6]]
```

In [22]:
```
# 输出第一个矩阵第二行
print(x_3d[0,1,:])
```

```
[4 5 6]
```

In [23]:
```
# 输出 60
print(x_3d[1,1,2])
```

```
60
```

```
In [24]: # 输出矩阵 [[5, 6], [50, 60]]
         print(x_3d[:,1,1:3])
```

```
[[ 5  6]
 [50 60]]
```

1.1.4 （4）数组功能

Numpy 数组有很多有用的属性 (properties)，可以通过 . (dot notation) 来使用，但它们不是函数，因此不需要使用括号。

```
In [26]: #shape 每个维有多少个元素
         x = np.array([[1,2,3],[4,5,6]])
         print(x.shape)
```

```
(2, 3)
```

```
In [27]: #dtype 数组的元素类型
         print(x.dtype)
```

```
int32
```

```
In [28]: # 三维数组
         x = np.array([
             [[1.0,2.0],[3.0,4.0],[5.6,6.0]],
             [[7.0,8.0],[9.0,10.0],[11.0,12.0]]
         ])
         print(x.shape)
         print(x.dtype)
```

```
(2, 3, 2)
float64
```

我们可以使用 np.zeros np.ones 来创建数组

```
In [31]: #np.zeros np.ones 零数组和一数组
         sizes = (2,3,4)
         x = np.zeros(sizes)
         print(x)
```

```
[[[0. 0. 0. 0.]
  [0. 0. 0. 0.]
  [0. 0. 0. 0.]]

 [[0. 0. 0. 0.]
  [0. 0. 0. 0.]
  [0. 0. 0. 0.]]]
```

```
In [32]: y = np.ones((4))
         y
```

```
Out[32]: array([1., 1., 1., 1.])
```

广播 (Broadcast) 是 numpy 对数组间进行数值计算的方式，有两种操作类型：- 数组和单个数字 - 相同形状的两个数组

```
In [33]: x = np.ones((2,2))
         print(x)
```

```
[[1. 1.]
 [1. 1.]]
```

```
In [34]: print(2+x)
```

```
[[3. 3.]
 [3. 3.]]
```

```
In [35]: print(2*x)
```

```
[[2. 2.]
 [2. 2.]]
```

```
In [36]: print(x/2)
```

```
[[0.5 0.5]
 [0.5 0.5]]
```

broadcasting 可以这样理解：如果你有一个 mn 的矩阵，让它加减乘除一个 1n 的矩阵，它会被复制 m 次，成为一个 mn 的矩阵，然后再逐元素地进行加减乘除操作。同样地对 m1 的矩阵成立。

比如 2+x 2 会先变成 [[2,2],[2,2]]，然后再和 [[1,1],[1,1]]

```
In [37]: x = np.array([[1.0, 2.0], [3.0, 4.0]])
         y = np.ones((2, 2))
         print(x)
         print(y)
```

```
[[1. 2.]
 [3. 4.]]
[[1. 1.]
 [1. 1.]]
```

```
In [38]: print(x+y)
```

```
[[2. 3.]
 [4. 5.]]
```

```
In [39]: print(x-y)
```

```
[[0. 1.]
 [2. 3.]]
```

```
In [40]: print(x*y)
```

```
[[1. 2.]
 [3. 4.]]
```

```
In [41]: print(x/y)
```

```
[[1. 2.]
 [3. 4.]]
```

对于两个形状相同的数值，它们之间运算是针对对应的元素。

我们经常需要转换数据以满足需求，此时 Numpy 为我们提供了通用函数 (ufuncs)。Numpy 对此提供了一个帮助文档：通用函数

```
In [43]: #np.linspace 在指定的间隔内返回均匀间隔的数字
         x = np.linspace(0.5,25,10)
         print(x)
```

```
[ 0.5         3.22222222  5.94444444  8.66666667 11.38888889 14.11111111
 16.83333333 19.55555556 22.27777778 25.        ]
```

```
In [44]: np.sin(x)

Out[44]: array([ 0.47942554, -0.08054223, -0.33229977,  0.68755122, -0.92364381,
                 0.99966057, -0.9024271 ,  0.64879484, -0.28272056, -0.13235175])

In [45]: np.log(x)

Out[45]: array([-0.69314718,  1.17007125,  1.78245708,  2.15948425,  2.43263822,
                 2.64696251,  2.82336105,  2.97325942,  3.10358967,  3.21887582])
```

以上只是对 Numpy 的简单介绍，除此之外，Numpy 还能做很多。下面再演示一些有用的数组操作。

```
In [46]: x = np.linspace(0,25,10)

In [47]: # 均值
         np.mean(x)

Out[47]: 12.5

In [48]: # 标准差
         np.std(x)

Out[48]: 7.9785592313028175

In [49]: # 最大值
         np.max(x)
```

Out[49]: 25.0

In [51]:
```
y = np.array([1,3,5,9])
```

In [52]:
```
# 间隔差
np.diff(y)
```

Out[52]: array([2, 2, 4])

In [54]:
```
# 变换形状
np.reshape(y,(2,2))
```

Out[54]:
```
array([[1, 3],
       [5, 9]])
```

1.2 2. 绘图

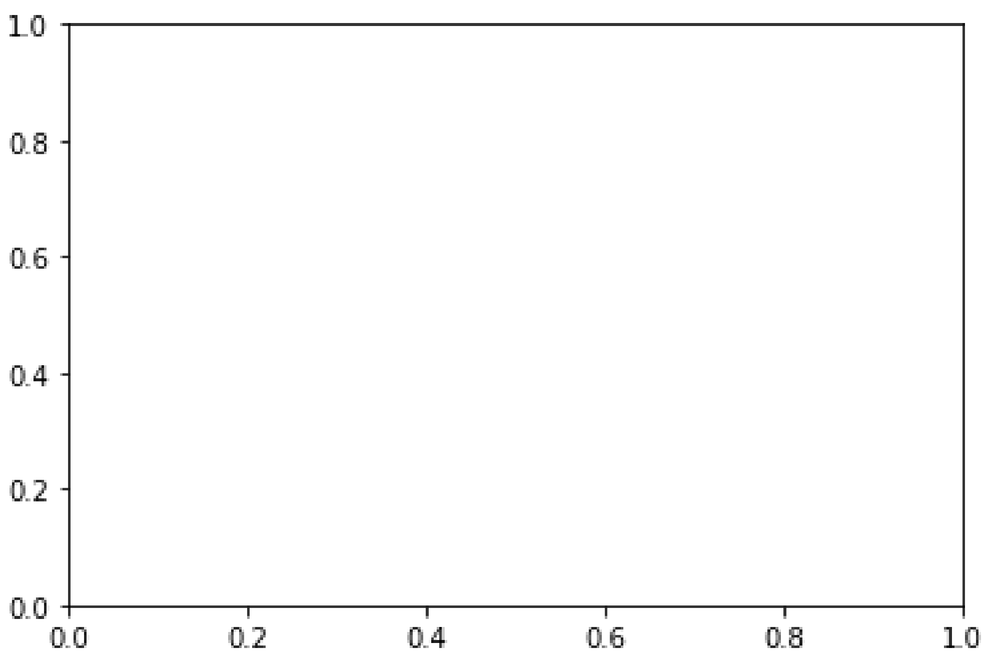
推荐一份资料：Introduction to Data Visualization
matplotlib 是 Python 中使用最广泛的绘图包，首先要导入相关包

In [17]:
```
import matplotlib.pyplot as plt
import numpy as np
```

1.2.1 （1）基础操作

第一步，创建一个画布和轴来存储未来的图形

In [18]:
```
fig, ax = plt.subplots()
```

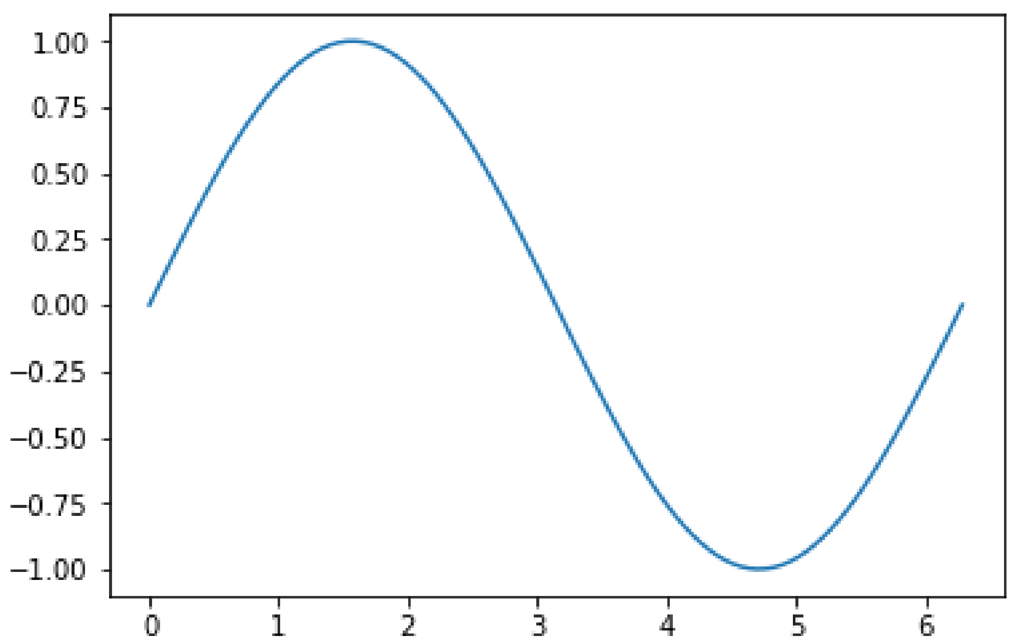


第二步，生成绘图用的数据

```
In [19]: x = np.linspace(0, 2*np.pi, 100)
         y = np.sin(x)
```

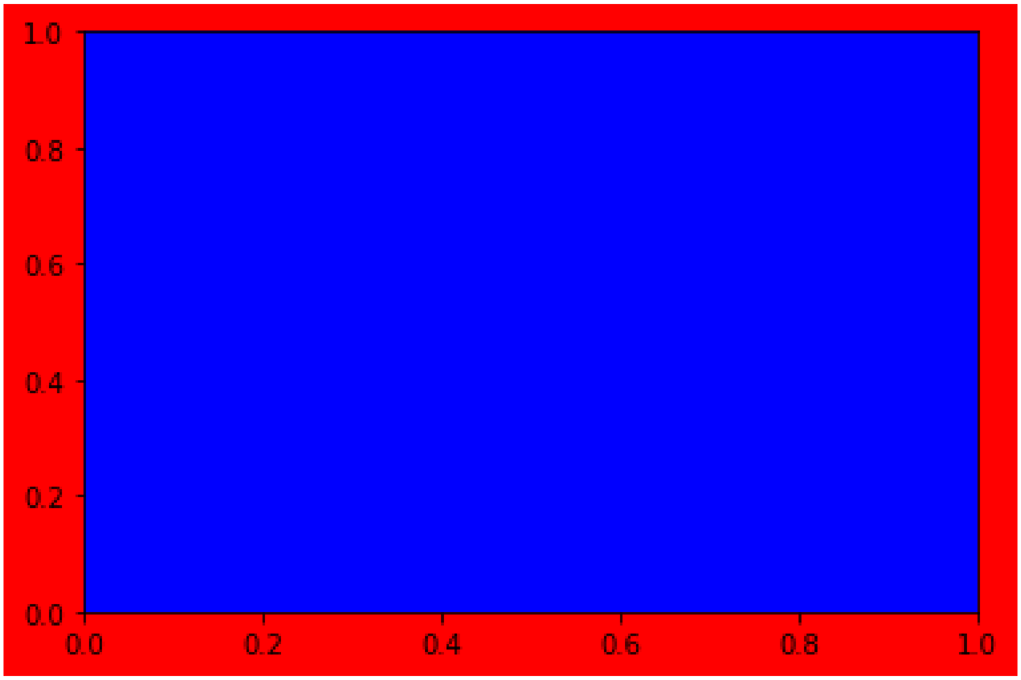
第三步，将数据对应的图画在画布上

```
In [20]: ax.plot(x, y)

Out[20]: [<matplotlib.lines.Line2D at 0x1aac797ac88>]

In [21]: fig, ax = plt.subplots()


         x = np.linspace(0, 2*np.pi, 100)
         y = np.sin(x)


         ax.plot(x, y)

Out[21]: [<matplotlib.lines.Line2D at 0x1aac7a4ea90>]
```



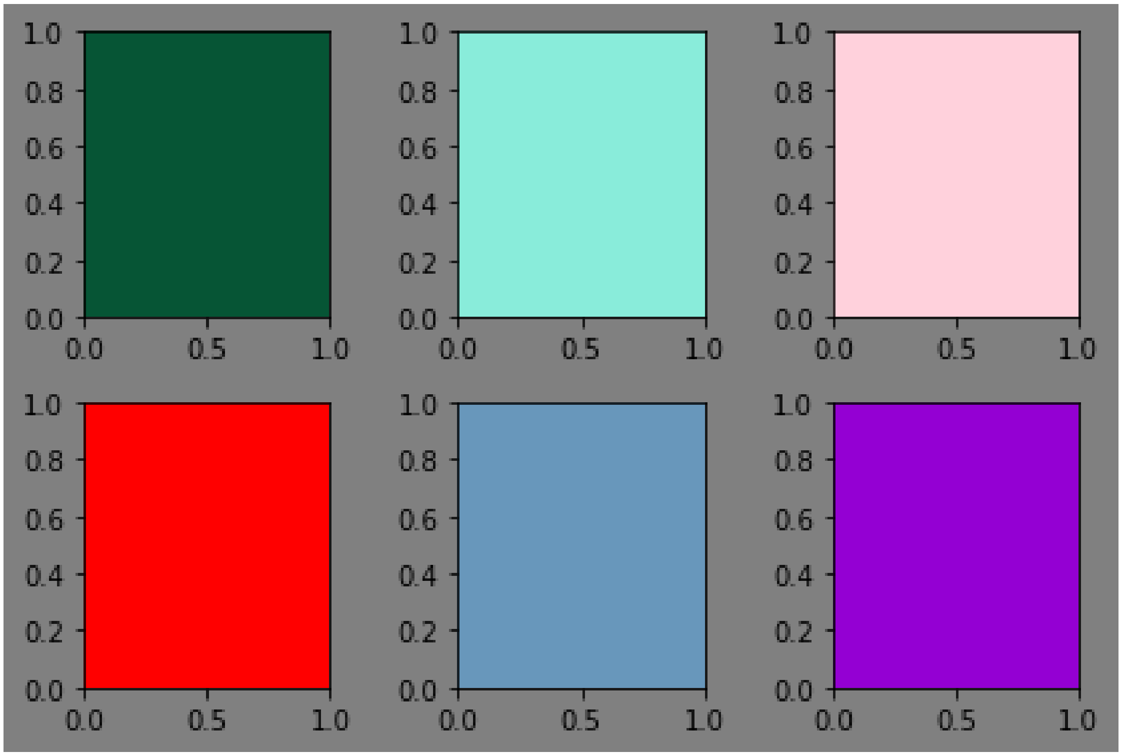
为不同区域添加不同颜色

```
In [14]: fig, ax = plt.subplots()

         fig.set_facecolor("red")
         ax.set_facecolor("blue")
```



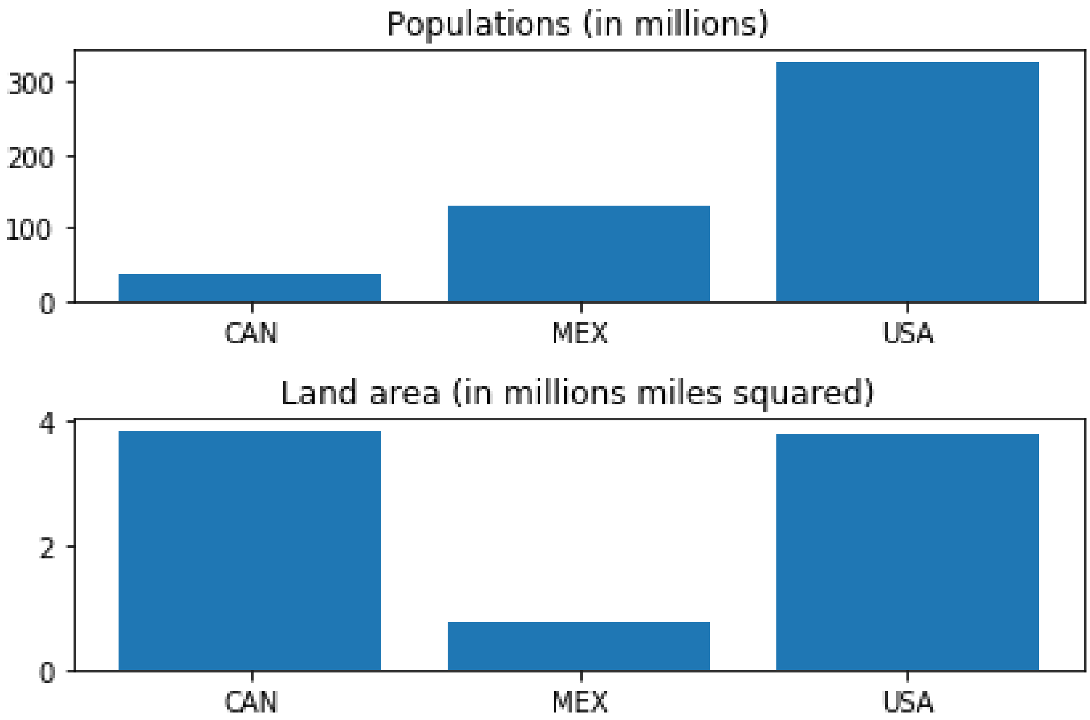
可以在一张图里绘制多张图

```
In [16]: fig, axes = plt.subplots(2, 3)

         fig.set_facecolor("gray")

         colors = ["#065535", "#89ecda", "#ffd1dc", "#ff0000", "#6897bb", "#9400d3"]

         for (ax, c) in zip(axes.flat, colors):
             ax.set_facecolor(c)

         fig.tight_layout()
```



1.2.2 （2）条形图

In [22]:
```
countries = ["CAN", "MEX", "USA"]
populations = [36.7, 129.2, 325.700]
land_area = [3.850, 0.761, 3.790]

fig, ax = plt.subplots(2)

ax[0].bar(countries, populations, align="center")
ax[0].set_title("Populations (in millions)")

ax[1].bar(countries, land_area, align="center")
ax[1].set_title("Land area (in millions miles squared)")

fig.tight_layout()
```

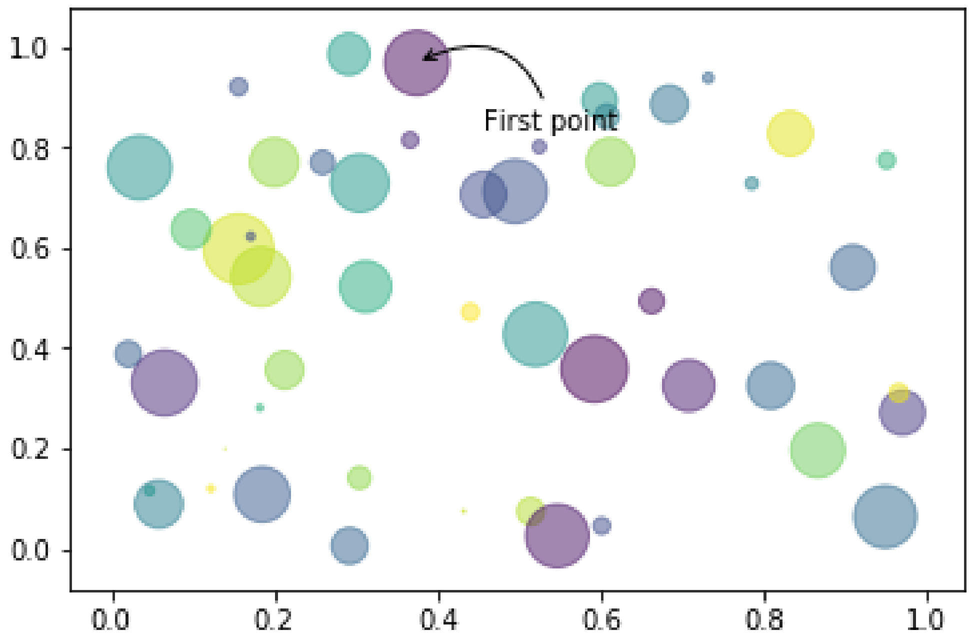


1.2.3 （3）散点图

In [23]:
```
N = 50

np.random.seed(42)

x = np.random.rand(N)
y = np.random.rand(N)
colors = np.random.rand(N)
area = np.pi * (15 * np.random.rand(N))**2  # 0 to 15 point radii

fig, ax = plt.subplots()

ax.scatter(x, y, s=area, c=colors, alpha=0.5)

ax.annotate(
    "First point", xy=(x[0], y[0]), xycoords="data",
    xytext=(25, -25), textcoords="offset points",
```

```
            arrowprops=dict(arrowstyle="->", connectionstyle="arc3,rad=0.6")
        )
```

Out[23]: Text(25,-25,'First point')



1.3 3. 线性代数

本部分主要介绍一些线性代数概念，然后做些线性代数运算。

In [27]:
```
# 导入相关库
import numpy as np
```

1.3.1 （1）向量

向量是一维数组

In [4]:
```
x = np.array([1, 2, 3])
y = np.array([4, 5, 6])
print(x)
print(y)
```

```
[1 2 3]
[4 5 6]
```

元素操作

```
In [5]: print("相加: ", x + y)
        print("相减: ", x - y)
        print("相乘: ", x * y)
        print("相除: ", x / y)
```

```
相加:  [5 7 9]
相减:  [-3 -3 -3]
相乘:  [ 4 10 18]
相除:  [0.25 0.4  0.5 ]
```

标量操作

```
In [6]: print("相加: ", 2 + y)
        print("相减: ", 2 - y)
        print("相乘: ", 2 * y)
        print("相除: ", 2 / y)
```

```
相加:  [6 7 8]
相减:  [-2 -3 -4]
相乘:  [ 8 10 12]
相除:  [0.5        0.4        0.33333333]
```

点积

```
In [7]: print("点积: ", np.dot(x,y))
```

```
点积:  32
```

1.3.2 （2）矩阵

矩阵是二维数组

```
In [8]: x = np.array([[1, 2, 3], [4, 5, 6]])
        y = np.ones((2,3))
        z = np.array([[1, 2], [3, 4], [5, 6]])
        print(x)
        print(y)
        print(z)
```

```
[[1 2 3]
 [4 5 6]]
[[1. 1. 1.]
 [1. 1. 1.]]
[[1 2]
 [3 4]
 [5 6]]
```

元素操作

```
In [9]: print("相加: ", x + y)
        print("相减: ", x - y)
        print("相乘: ", x * y)
        print("相除: ", x / y)
```

```
相加:  [[2. 3. 4.]
 [5. 6. 7.]]
相减:  [[0. 1. 2.]
 [3. 4. 5.]]
相乘:  [[1. 2. 3.]
 [4. 5. 6.]]
相除:  [[1. 2. 3.]
 [4. 5. 6.]]
```

标量操作

```
In [10]: print("相加: ", 2 + y)
         print("相减: ", 2 - y)
         print("相乘: ", 2 * y)
         print("相除: ", 2 / y)
```

```
相加： [[3. 3. 3.]
 [3. 3. 3.]]
相减： [[1. 1. 1.]
 [1. 1. 1.]]
相乘： [[2. 2. 2.]
 [2. 2. 2.]]
相除： [[2. 2. 2.]
 [2. 2. 2.]]
```

矩阵乘法

```
In [11]: x = np.reshape(np.arange(6), (3, 2))
         y = np.ones((2,3))
         print(x)
         print(y)
```

```
[[0 1]
 [2 3]
 [4 5]]
[[1. 1. 1.]
 [1. 1. 1.]]
```

```
In [12]: print(np.matmul(x, y))
```

```
[[1. 1. 1.]
 [5. 5. 5.]
 [9. 9. 9.]]
```

```
In [13]: print(np.dot(x, y))
```

```
[[1. 1. 1.]
 [5. 5. 5.]
 [9. 9. 9.]]
```

```
In [14]: print(x @ y)
```

```
[[1. 1. 1.]
 [5. 5. 5.]
 [9. 9. 9.]]
```

以上三种计算矩阵乘法的方法，建议使用 @，因为比较简洁。

矩阵转置

```
In [16]: x = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
         print(x)
```

```
[[1 2 3]
 [4 5 6]
 [7 8 9]]
```

```
In [17]: print(x.transpose())
```

```
[[1 4 7]
 [2 5 8]
 [3 6 9]]
```

单位矩阵

```
In [19]: I = np.eye(3)
         x = np.reshape(np.arange(9), (3, 3))
         y = np.array([1, 2, 3])

         print(I)
         print(x)
         print(y)
```

```
[[1. 0. 0.]
 [0. 1. 0.]
 [0. 0. 1.]]
[[0 1 2]
 [3 4 5]
 [6 7 8]]
[1 2 3]
```

```
In [20]: print(I @ x)
```

```
[[0. 1. 2.]
 [3. 4. 5.]
 [6. 7. 8.]]
```

```
In [21]: print(x @ I)
```

```
[[0. 1. 2.]
 [3. 4. 5.]
 [6. 7. 8.]]
```

```
In [22]: print(I @ y)
```

```
[1. 2. 3.]
```

```
In [23]: print(y @ I)
```

```
[1. 2. 3.]
```

矩阵的逆

```
In [24]: x = np.array([[1, 2, 0], [3, 1, 0], [0, 1, 2]])
         print(x)
```

```
[[1 2 0]
 [3 1 0]
 [0 1 2]]
```

```
In [25]: print(np.linalg.inv(x))
```

```
[[-0.2  0.4  0. ]
 [ 0.6 -0.2  0. ]
 [-0.3  0.1  0.5]]
```

```
In [26]: print(np.linalg.inv(x) @ x)
```

```
[[1. 0. 0.]
 [0. 1. 0.]
 [0. 0. 1.]]
```

1.4 4. 概率论和数理统计

```
In [3]: # 导入相关的库和模块
        import numpy as np
        import matplotlib.pyplot as plt
```

1.4.1 （1）随机性模拟

0 到 1 间的随机数

```
In [4]: np.random.rand()

Out[4]: 0.38773163344000316
```

随机数数组

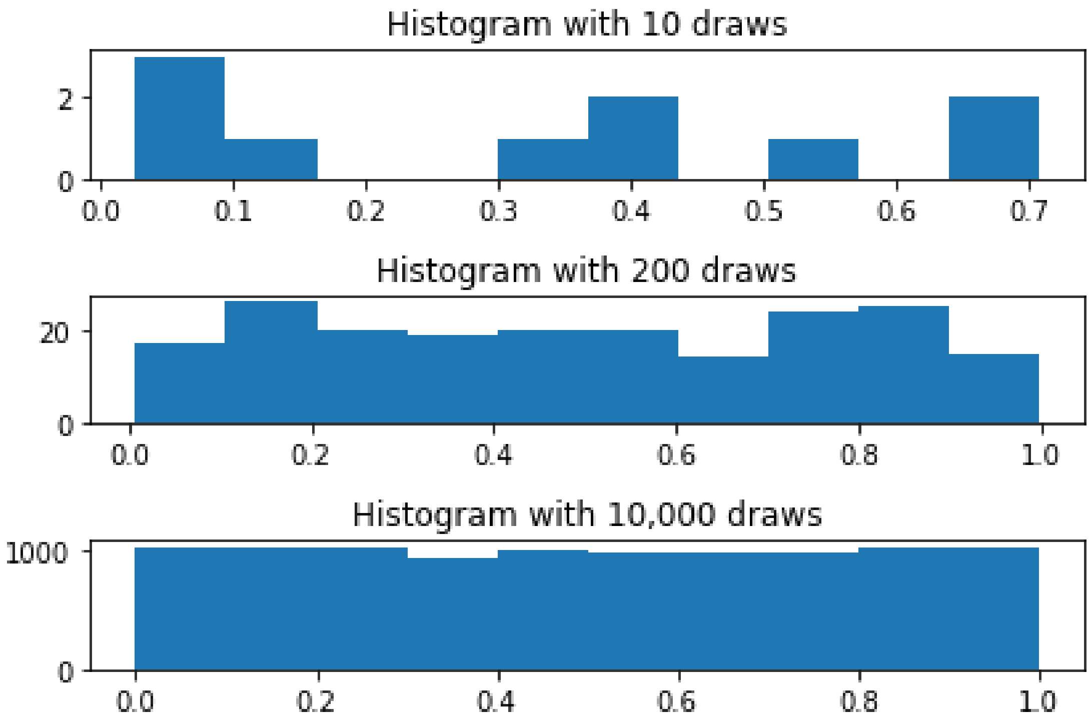
```
In [5]: np.random.rand(25)

Out[5]: array([0.29786873, 0.47830638, 0.07471268, 0.38289308, 0.16241408,
               0.0573538 , 0.60278932, 0.6813652 , 0.88305073, 0.5647692 ,
               0.65257099, 0.01344738, 0.95970231, 0.24117547, 0.62898494,
               0.39286254, 0.08617456, 0.18640692, 0.00179064, 0.93793884,
               0.98993892, 0.69706526, 0.94783645, 0.4707406 , 0.22139253])

In [6]: np.random.rand(5,5)

Out[6]: array([[0.09464577, 0.67021953, 0.42110162, 0.3288174 , 0.56441456],
               [0.91931402, 0.7260361 , 0.61271104, 0.96335401, 0.96812609],
               [0.035404  , 0.18535482, 0.4067497 , 0.95860237, 0.57185088],
               [0.7191573 , 0.34949615, 0.11002355, 0.95719805, 0.67494793],
               [0.73523762, 0.64726862, 0.10790041, 0.68194076, 0.44054488]])

In [7]: np.random.rand(2,3,4)

Out[7]: array([[[0.17559278, 0.38485834, 0.79731753, 0.05820101],
                [0.55750372, 0.63154436, 0.48124212, 0.97397713],
```

```
        [0.40113841, 0.92234602, 0.91526491, 0.41983515]],

       [[0.57200116, 0.38032162, 0.45243376, 0.84680362],
        [0.90225697, 0.48998352, 0.02386963, 0.04530768],
        [0.34033252, 0.71731386, 0.0800349 , 0.39651597]]])
```

1.4.2 （2）大数定律

随着模拟事件的数量趋于无穷大，模拟结果的分布将趋向真实的分布。

```
In [8]: # 生成 0 到 1 随机变量
        draws_10 = np.random.rand(10)
        draws_200 = np.random.rand(200)
        draws_10000 = np.random.rand(10_000)

        # 绘制区域
        fig, ax = plt.subplots(3)

        # 绘图
        ax[0].set_title("Histogram with 10 draws")
        ax[0].hist(draws_10)

        ax[1].set_title("Histogram with 200 draws")
        ax[1].hist(draws_200)

        ax[2].set_title("Histogram with 10,000 draws")
        ax[2].hist(draws_10000)

        fig.tight_layout()
```



1.4.3 （3）离散分布

For example, consider a small business loan company.Imagine that the company's loan requires a repayment of 25,000 and must be repaid 1 year after the loan was made.The company discounts the future at 5%. Additionally, the loans made are repaid in full with 75% probability, while 12,500 of loans is repaid with probability 20%, and no repayment with 5% probability.How much would the small business loan company be willing to loan if they'd like to – on average – break even?

慢版本

```
In [24]: def simulate_loan_repayments_slow(N, r=0.05, repayment_full=25000.0,
                                           repayment_part=12500.0):
             repayment_sims = np.zeros(N)
             for i in range(N):
                 x = np.random.rand()  # Draw a random number

                 # Full repayment 75% of time
                 if x < 0.75:
```

```
                repaid = repayment_full
            elif x < 0.95:
                repaid = repayment_part
            else:
                repaid = 0.0

            repayment_sims[i] = (1 / (1 + r)) * repaid

        return repayment_sims

    print(np.mean(simulate_loan_repayments_slow(25000)))
```

```
20242.85714285714
```

快版本

```
In [29]: def simulate_loan_repayments(N, r=0.05, repayment_full=25_000.0,
                                      repayment_part=12_500.0):
             """
             Simulate present value of N loans given values for discount rate and
             repayment values
             """
             random_numbers = np.random.rand(N)

             # start as 0 -- no repayment
             repayment_sims = np.zeros(N)

             # adjust for full and partial repayment
             partial = random_numbers <= 0.20
             repayment_sims[partial] = repayment_part

             full = ~partial & (random_numbers <= 0.95)
             repayment_sims[full] = repayment_full

             repayment_sims = (1 / (1 + r)) * repayment_sims

             return repayment_sims
```

```
np.mean(simulate_loan_repayments(25_000))
```

Out[29]: 20223.33333333333

```
In [30]: %timeit simulate_loan_repayments_slow(250_000)
```

323 ms ś 33.3 ms per loop (mean ś std. dev. of 7 runs, 1 loop each)

```
In [31]: %timeit simulate_loan_repayments(250_000)
```

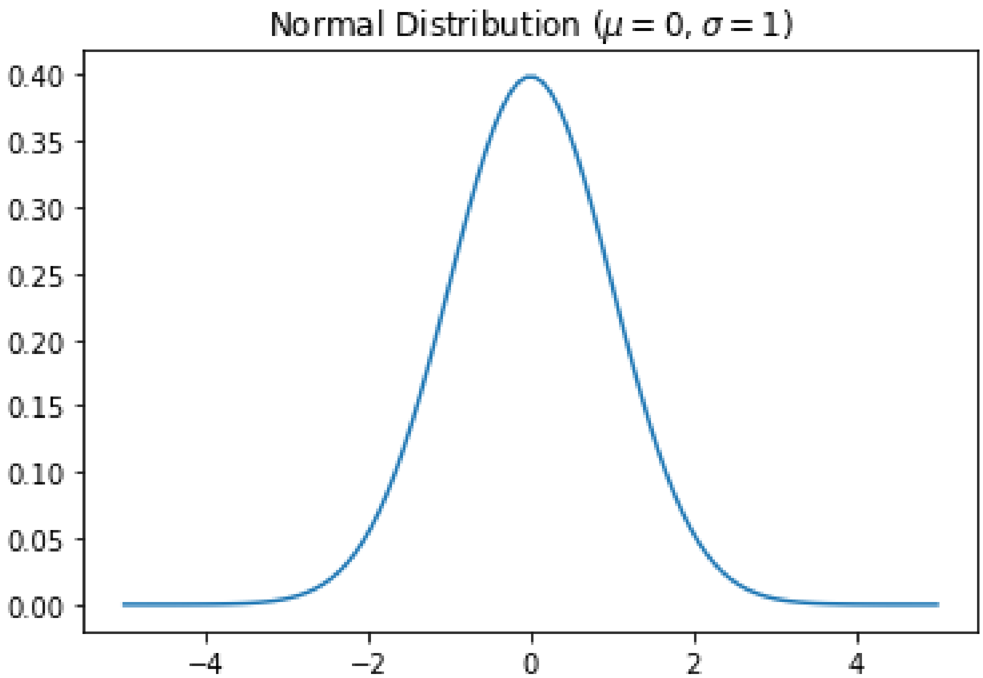
13.6 ms ś 777 ţs per loop (mean ś std. dev. of 7 runs, 100 loops each)

两个版本的时间相差 25 倍，这对于一些更为复杂的操作会很重要。快版本背后是向量化操作，即每次计算针对的是整个数组，这会比对数组元素一个个操作要速度快。

1.4.4 （4）连续分布

```
In [32]: # scipy is an extension of numpy, and the stats
         # subpackage has tools for working with various probability distributions
         import scipy.stats as st

         x = np.linspace(-5, 5, 100)

         # NOTE: first argument to st.norm is mean, second is standard deviation sigma (not si
         pdf_x = st.norm(0.0, 1.0).pdf(x)

         fig, ax = plt.subplots()

         ax.set_title(r"Normal Distribution ($\mu = 0, \sigma = 1$)")
         ax.plot(x, pdf_x)
```

Out[32]: [<matplotlib.lines.Line2D at 0x1a515f1a278>]



1.5 5. 最优化问题

```
In [1]: # 导入相关库和模块
        import numpy as np
        import matplotlib.pyplot as plt
```

1.5.1 （1）导数和最优化

考虑函数

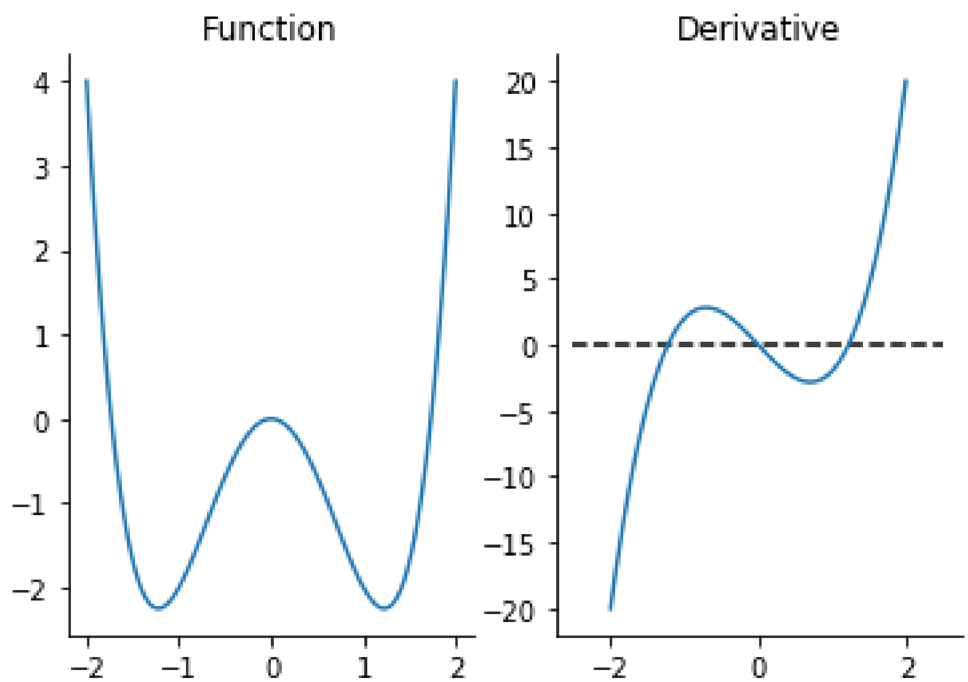
$$f(x) = x^4 - x^3 - 2x^2 + x$$

其导数为

$$\frac{\partial f}{\partial x} = 4x^3 - 3x^2 - 4x + 1$$

```
In [6]: def f(x):
            return x**4 - 3*x**2


        def fp(x):
            return 4*x**3 - 6*x
```

```python
# 随机生成值
x = np.linspace(-2., 2., 100)

# 估计函数值
fx = f(x)
fpx = fp(x)

# 绘制图形
fig, ax = plt.subplots(1, 2)

ax[0].plot(x, fx)
ax[0].set_title("Function")

ax[1].plot(x, fpx)
ax[1].hlines(0.0, -2.5, 2.5, color="k", linestyle="--")
ax[1].set_title("Derivative")

# 去掉边框
for _ax in ax:
    _ax.spines["right"].set_visible(False)
    _ax.spines["top"].set_visible(False)
```



通过手工计算有

$$f'(x) = 4x^3 - 6x = 0$$
$$\rightarrow x = \left\{0, \frac{\sqrt{(6)}}{2}, \frac{-\sqrt{(6)}}{2}\right\}$$

In [8]:
```python
# 使用 python 找到函数最值
import scipy.optimize as opt

neg_min = opt.minimize_scalar(f, [-2, -0.5])
pos_min = opt.minimize_scalar(f, [0.5, 2.0])
print("The negative minimum is: \n", neg_min)
print("The positive minimum is: \n", pos_min)
```

```
The negative minimum is:
      fun: -2.2499999999999996
    nfev: 16
     nit: 12
 success: True
       x: -1.2247448697638397
The positive minimum is:
      fun: -2.2499999999999996
```

```
    nfev: 16
     nit: 12
 success: True
       x: 1.2247448697638397
```

Scipy 优化包只有查找最小值的功能。关于最大值，可以利用 “求最大值等价于求负函数的最小值” 来解决。

```
In [9]: def neg_f(x):
            return -f(x)

        max_out = opt.minimize_scalar(neg_f, [-0.35, 0.35])
        print("The maximum is: \n", max_out)
```

```
The maximum is:
      fun: 1.1519919564363613e-23
    nfev: 13
     nit: 9
 success: True
       x: 1.9595849530247992e-12
```

1.5.2 （2）消费者理论

偏好和效应函数

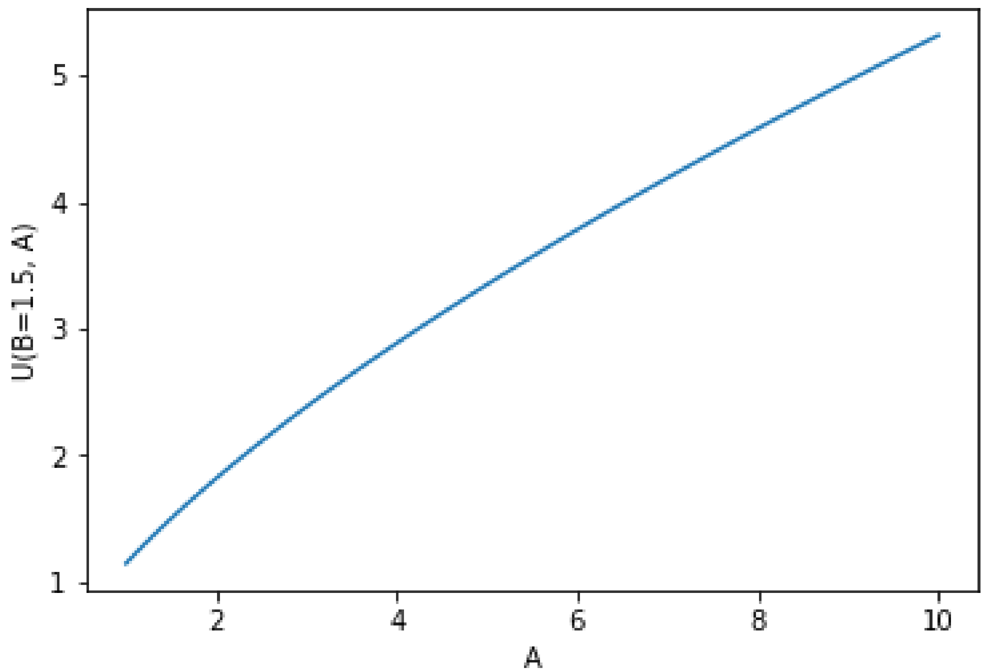
考虑苹果 (a) 和香蕉 (b) 的效应函数

$$U(B, A) = B^{\alpha} A^{1-\alpha}$$

假定 B=1.5

```
In [14]: def U(A, B, alpha=1/3):
             return B**alpha * A**(1-alpha)

         fig, ax = plt.subplots()
         B = 1.5
         A = np.linspace(1, 10, 100)
         ax.plot(A, U(A, B))
         ax.set_xlabel("A")
         ax.set_ylabel("U(B=1.5, A)")
```

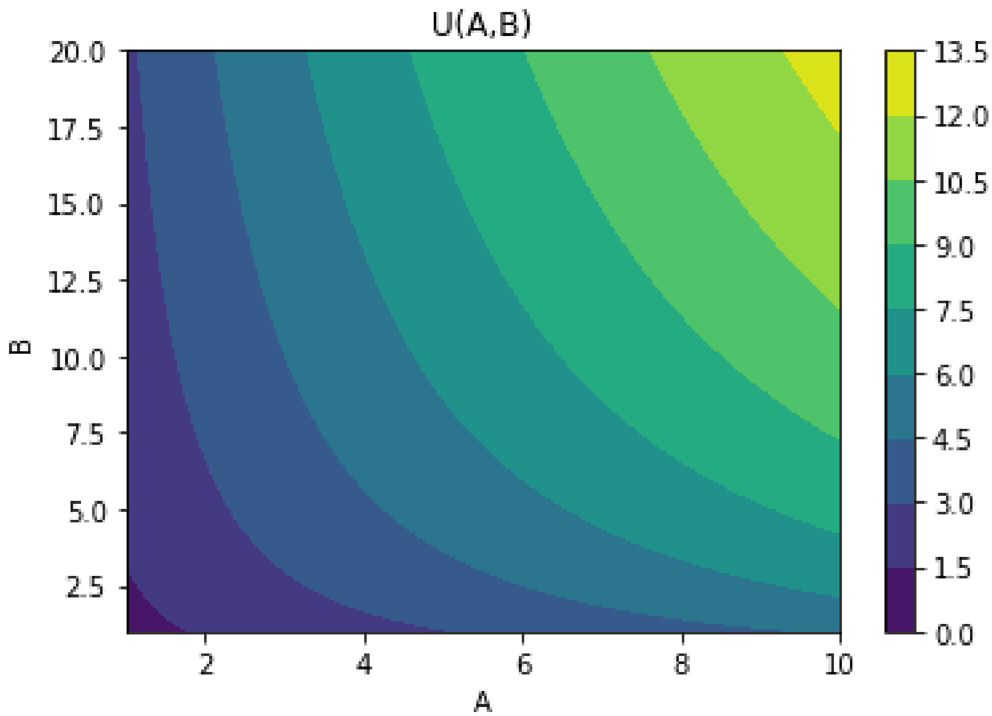
Out[14]: Text(0,0.5,'U(B=1.5, A)')



如果我们把 A 和 B 都画出来

In [15]:
```
fig, ax = plt.subplots()
B = np.linspace(1, 20, 100).reshape((100, 1))
contours = ax.contourf(A, B.flatten(), U(A, B))
fig.colorbar(contours)
ax.set_xlabel("A")
ax.set_ylabel("B")
ax.set_title("U(A,B)")
```

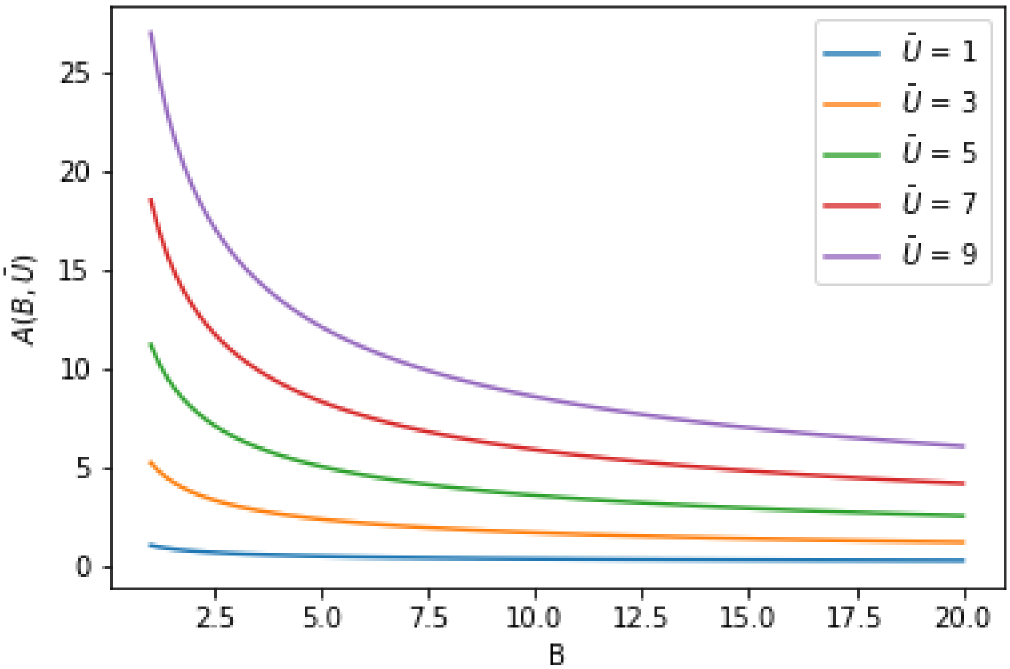
Out[15]: Text(0.5,1,'U(A,B)')



为了求得最优的 A 和 B 组合，我们可以先固定 U，用 U 和 B 来表示 A、

$$A(B,\bar{U}) = U^{\frac{1}{1-\alpha}} B^{\frac{-\alpha}{1-\alpha}}$$

```
In [16]: def A_indifference(B, ubar, alpha=1/3):
             return ubar**(1/(1-alpha)) * B**(-alpha/(1-alpha))

         def plot_indifference_curves(ax, alpha=1/3):
             ubar = np.arange(1, 11, 2)
             ax.plot(B, A_indifference(B, ubar, alpha))
             ax.legend([r"$\bar{U}$" + " = {}".format(i) for i in ubar])
             ax.set_xlabel("B")
             ax.set_ylabel(r"$A(B, \bar{U}$)")

         fig, ax = plt.subplots()
         plot_indifference_curves(ax)
```

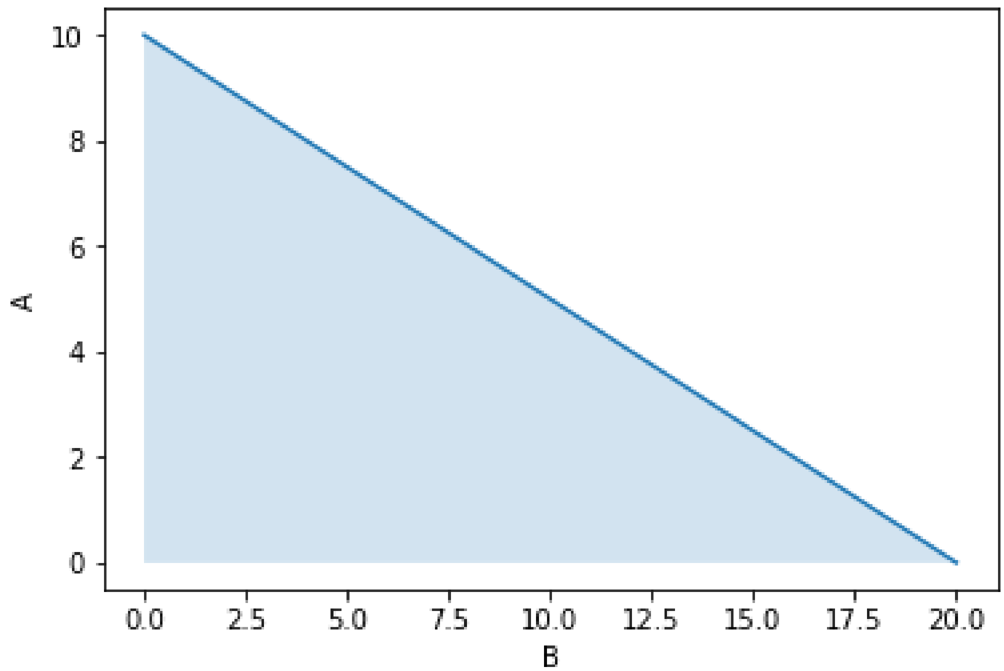


然后添加预算约束线

$$2A + B \leq W$$

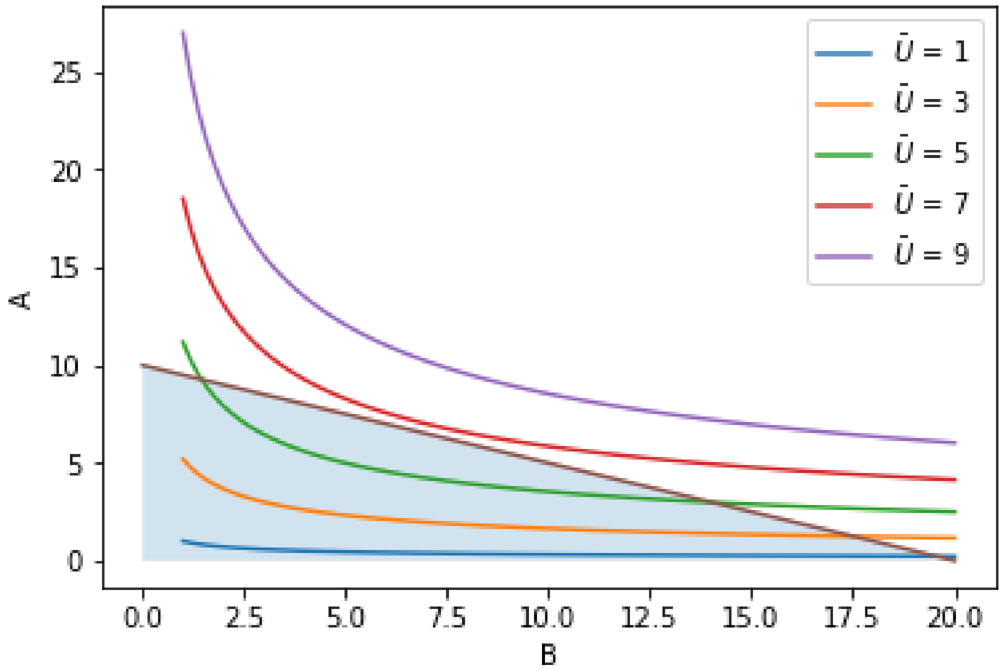
In [17]:
```
def A_bc(B, W=20, pa=2):
    return (W - B) / pa

def plot_budget_constraint(ax, W=20, pa=2):
    B_bc = np.array([0, W])
    A = A_bc(B_bc, W, pa)
    ax.plot(B_bc, A)
    ax.fill_between(B_bc, 0, A, alpha=0.2)
    ax.set_xlabel("B")
    ax.set_ylabel("A")
    return ax

fig, ax = plt.subplots()
plot_budget_constraint(ax, 20, 2)
```

Out[17]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x2a868469320>
```



我们把以上预算约束和效应函数即可获得最优商品组合

```
In [18]: fig, ax = plt.subplots()
         plot_indifference_curves(ax)
         plot_budget_constraint(ax)

Out[18]: <matplotlib.axes._subplots.AxesSubplot at 0x2a867b69c18>
```



利用 Scipy 可求得具体数值

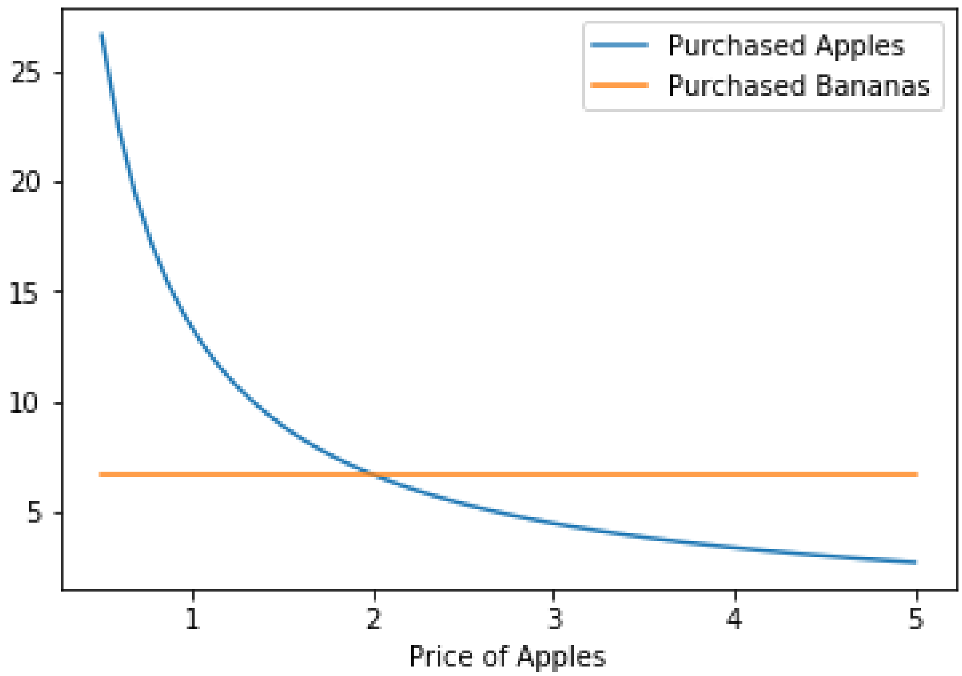
```
In [19]: from scipy.optimize import minimize_scalar

         def objective(B, W=20, pa=2):
             A = A_bc(B, W, pa)
             return -U(A, B)

         result = minimize_scalar(objective)
         optimal_B = result.x
         optimal_A = A_bc(optimal_B, 20, 2)
         optimal_U = U(optimal_A, optimal_B)

         print("The optimal U is ", optimal_U)
         print("and was found at (A,B) =", (optimal_A, optimal_B))
```

```
The optimal U is  6.666666666666667
and was found at (A,B) = (6.666666630651958, 6.666666738696083)
```

如果价格发生变化

In [20]:
```python
# Create various prices
n_pa = 50
prices_A = np.linspace(0.5, 5.0, n_pa)
W = 20

# Create lists to store the results of the optimal A and B calculation
optimal_As = []
optimal_Bs = []
for pa in prices_A:
    result = minimize_scalar(objective, args=(W, pa))
    opt_B_val = result.x

    optimal_Bs.append(opt_B_val)
    optimal_As.append(A_bc(opt_B_val, W, pa))

fig, ax = plt.subplots()

ax.plot(prices_A, optimal_As, label="Purchased Apples")
ax.plot(prices_A, optimal_Bs, label="Purchased Bananas")
ax.set_xlabel("Price of Apples")
ax.legend()
```

Out[20]:
```
<matplotlib.legend.Legend at 0x2a86871deb8>
```


Purchased Apples
Purchased Bananas
25
20
15
10
5
1
2
3
4
5
Price of Apples