A Brief Introduction to Python Part I

Wei Tianwen

2017

Table of contents

Integers and floats

Integer `int` and float `float` are elementary numeric types in Python.

- integer

```
>>> a=1
>>> a
1
>>> type(a)
<class 'int'>
```

Integers and floats

Integer `int` and float `float` are elementary numeric types in Python.

- integer

```
>>> a=1
>>> a
1
>>> type(a)
<class 'int'>
```

- float

```
>>> b=2.5
>>> b
2.5
>>> type(b)
<class 'float'>
```

Integers and floats

Integer `int` and float `float` are elementary numeric types in Python.

- integer

```
>>> a=1
>>> a
1
>>> type(a)
<class 'int'>
```

- float

```
>>> b=2.5
>>> b
2.5
>>> type(b)
<class 'float'>
```

- Note that to create a float type of number "1", we need to type `b=1.0` or `b=1.` instead of `b=1`. Integer "1" and float "1.0" are different in Python.

Arithmetic operations

Operations	Maths expression	Python expression
addition	$x + y$	`x + y`
subtraction	$x - y$	`x - y`
multiplication	$x \times y$	`x * y`
division	$x \div y$	`x / y`
power	x^y	`x ** y`

Those operations introduced above accept integer and/or

float as input.

```
>>> 2+4      # the output is an integer
6
>>> 2.0+4    # the output is a float
6.0
```

String

String `str`, i.e. a sequence of characters, is also an elementary type in Python.

```
>>> c='This is a string'
>>> c
'This is a string'
>>> type(c)
<class 'str'>
```

String

String `str`, i.e. a sequence of characters, is also an elementary type in Python.

```
>>> c='This is a string'
>>> c
'This is a string'
>>> type(c)
<class 'str'>
```

In Python, it is very convenient to manipulate strings. We can use function `len()` to obtain the number of characters contained in a string:

```
>>> x="hello, world!"
>>> x
'hello, world!'
>>> len(x)
13
```

String

String `str`, i.e. a sequence of characters, is also an elementary type in Python.

```
>>> c='This is a string'
>>> c
'This is a string'
>>> type(c)
<class 'str'>
```

In Python, it is very convenient to manipulate strings. We can use function `len()` to obtain the number of characters contained in a string:

```
>>> x="hello, world!"
>>> x
'hello, world!'
>>> len(x)
13
```

We can also do string arithmetics:

```
>>> x="My name is "
>>> y="Wei Tianwen."
>>> x+y
'My name is Wei Tianwen.'
>>> z="Python is very popular. "
>>> z*3
'Python is very popular. Python is very popular. Python is very popular. '
```

Exercise

Exercise 1.1

1. Evaluate the following expressions using python:

$$\frac{(3.14+5)^3}{128}$$

2. Create a string that consists of your favorite English phrase repeated 10 times

List

List `list` is a type that can be used to bundle stuff together. We can create a list using bracket `[...]`:

```
>>> a=[1,2,3,4,5]   # we can use list to create an array
>>> a
[1, 2, 3, 4, 5]
>>> type(a)
<class 'list'>
>>>
>>> b=[2, 7.139, 'hello']   # the elements may be of different types
>>> b
[2, 7.139, 'hello']
```

List

List `list` is a type that can be used to bundle stuff together. We can create a list using bracket `[...]`:

```
>>> a=[1,2,3,4,5]   # we can use list to create an array
>>> a
[1, 2, 3, 4, 5]
>>> type(a)
<class 'list'>
>>>
>>> b=[2, 7.139, 'hello']   # the elements may be of different types
>>> b
[2, 7.139, 'hello']
```

Function `len()` also returns the length of a list, i.e. the number of elements contained in a list:

```
>>> len(a)
5
>>> len(b)
3
>>> c=a+b   # the "sum" of two lists
>>> c
[1, 2, 3, 4, 5, 2, 7.139, 'hello']
>>> len(c)
8
>>> c*2     # list "multiplied" by an integer
[1, 2, 3, 4, 5, 2, 7.139, 'hello', 1, 2, 3, 4, 5, 2, 7.139, 'hello']
```

Slicing

To access the elements of a list, we also use brackets `[ ]`. Basically, to access the *n*-th element of a list `mylist`, we use `mylist[n-1]`:

```
>>> b=[2, 7.139, 'hello']   # b is a list that contains 3 elements
>>> b[0]    # the first element in b
2
>>> b[1]    # the second element
7.139
>>> b[2]    # the third element
'hello'
>>> b[3]    # trying to access the fourth element results in an error
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
IndexError: list index out of range
```

Slicing

To access the elements of a list, we also use brackets `[ ]`. Basically, to access the n-th element of a list `mylist`, we use `mylist[n-1]`:

```
>>> b=[2, 7.139, 'hello']   # b is a list that contains 3 elements
>>> b[0]    # the first element in b
2
>>> b[1]    # the second element
7.139
>>> b[2]    # the third element
'hello'
>>> b[3]    # trying to access the fourth element results in an error
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
IndexError: list index out of range
```

As is shown above, the index must be smaller than the length of the list. What about negative indices?

```
>>> a=[1,2,3,4,5]
>>> a[-1]   # the last element of the list
5
>>> a[-2]   # the element before the last element of the list
4
>>> a[-5]   # the first element of a
1
>>> a[-6]   # error
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
IndexError: list index out of range
```

Nested list

The elements contained in a list can be anything.

```
>>> a1=[1,2,3,4]
>>> a2=[5,6,7,8]
>>> a3=[9,10,11,12]
>>> A=[a1,a2,a3]
>>> A
[[1, 2, 3, 4], [5, 6, 7, 8], [9, 10, 11, 12]]
>>> type(A)
<class 'list'>
>>> len(A)  # A is a list containing 3 elements
3
>>> A[2]
[9, 10, 11, 12]
>>> len(A[2]) # A[2] is a list containing 4 elements
4
>>> A[2][1] # Accessing element of A[2]
10
```

The list of lists can be used to represent a multi-dimensional array. We will look into that matter later.

Exercise

Exercise 1.2

1. Assume that `z` is a list consisting of even integers between 0 and 100, i.e. $z = [2, 4, \ldots, 98, 100]$. Find indices n_1 and n_2 so that $z[n_1] = z[n_2] = 66$.

Exercise

Exercise 1.2

1. Assume that `z` is a list consisting of even integers between 0 and 100, i.e. $z = [2, 4, \ldots, 98, 100]$. Find indices n_1 and n_2 so that $z[n_1] = z[n_2] = 66$.

```
>>> z=list(range(2,101,2))  # generate list
>>> z
[2, 4, 6, 8, 10, 12, 14, 16, 18, 20, 22, 24, 26, 28, 30, 32, 34, 36, 38, 40,
  42, 44, 46, 48, 50, 52, 54, 56, 58, 60, 62, 64, 66, 68, 70, 72, 74, 76,
   78, 80, 82, 84, 86, 88, 90, 92, 94, 96, 98, 100]
>>> z=list(range(2,101,2))
>>> z[32]
66
>>> z[-18]
66
```

Exercise

Exercise 1.3

Create the following matrices using Python list:

$$\mathbf{A} = \begin{pmatrix} 1 & 1 & 1 & 1 & 1 & 1 \\ 2 & 2 & 2 & 2 & 2 & 2 \\ 3 & 3 & 3 & 3 & 3 & 3 \end{pmatrix}, \quad \mathbf{B} = \begin{pmatrix} 1 & 2 & 3 \\ 1 & 2 & 3 \\ 1 & 2 & 3 \\ 1 & 2 & 3 \\ 1 & 2 & 3 \\ 1 & 2 & 3 \end{pmatrix}$$

Exercise

Exercise 1.3

Create the following matrices using Python list:

$$\mathbf{A} = \begin{pmatrix} 1 & 1 & 1 & 1 & 1 & 1 \\ 2 & 2 & 2 & 2 & 2 & 2 \\ 3 & 3 & 3 & 3 & 3 & 3 \end{pmatrix}, \quad \mathbf{B} = \begin{pmatrix} 1 & 2 & 3 \\ 1 & 2 & 3 \\ 1 & 2 & 3 \\ 1 & 2 & 3 \\ 1 & 2 & 3 \\ 1 & 2 & 3 \end{pmatrix}$$

```
>>> A=[[1]*6, [2]*6, [3]*6]
>>> A
[[1, 1, 1, 1, 1, 1], [2, 2, 2, 2, 2, 2], [3, 3, 3, 3, 3, 3]]
>>> B=[[1,2,3]]*6
>>> B
[[1, 2, 3], [1, 2, 3], [1, 2, 3], [1, 2, 3], [1, 2, 3], [1, 2, 3]]
```

For loop

Assume now we want to calculate the sum

$$\sum_{n=1}^{100} \frac{1}{n^2}$$

using in Python. The following snippet using a "for" loop can achieve this:

```
s = 0   # initialize a variable that stores the partial sum
for n in range(100):    # do not forget the colon
    v = 1/(n+1)**2
    s = s + v   # update the partial sum
print(s)    # print result
```

From the snippet above, we remark that in Python it is the *indentation* of expressions that determines which part of the code is in the for loop. This is a unique features of Python compared to other popular programming languages.

Indentation

✗ *No indentation*

```
for n in range(100):
v = 1/(n+1)**2
s = s + v
```

✗ *Inconsistent indentations*

```
for n in range(100):
        v = 1/(n+1)**2
    s = s + v
```

✓ Consistent indentations

```
for n in range(100):
  v = 1/(n+1)**2
  s = s + v
```

In Python, the size of the indentation does not matter, so long as the indentations are consistent.

For loop

What if we want to calculate

$$\sum_{i=1}^{n} \frac{1}{z_i^2}$$

for $z_1, \ldots, z_n$ stored in a Python list ? One solution is the following:

```
# i iterates from i=0 to i=len(z)-1 (inclusive)
for i in range(len(z)):
    v = 1/z[i]**2
    s = s + v
```

There is also a more elegant solution:

```
# i iterates from i=z[0] to i=z[-1] (inclusive)
for i in z:
    v = 1/i**2
    s = s + v
```

See the difference?

Nested loop

Assume we have a matrix represented by a list:

```
>>> A=[[1,4,5,6], [-3,5,2,1], [8,9,0,2]]
>>> A
[[1, 4, 5, 6], [-3, 5, 2, 1], [8, 9, 0, 2]]
```

and we want to calculate the sum of squares of the elements of A. This can be achieved by using a nested for loop:

```
s = 0
for i in range(len(A)): # outer loop
    for j in range(len(A[0])):  # inner loop
        v = A[i][j]**2
        s = s + v
```

Indentation continued

Indentation in Python is something to be taken care of. The following codes throw an error due to incorrect use of indentation:

✗ *First line should not be indented*

```
    for i in range(len(A)):
for j in range(len(A[0])):
        v = A[i][j]**2
        s = s + v
```

✗ *The second loop contains no block*

```
for i in range(len(A)):
    for j in range(len(A[0])):
    v = A[i][j]**2
    s = s + v
```

✗ *Inconsistent indentation*

```
for i in range(len(A)):
    for j in range(len(A[0])):
        v = A[i][j]**2
            s = s + v
```

Function `range()`

In the snippet above we used function `range()`. This function returns a Python *iterator* that we shall speak of later. For know it suffices to knows that:

- `range(n)` behaves like list `[0,1,...,n-1]`;
- `range(n, m)` behaves like list `[n,n+1,...,m-1]`
- `range(n, m, s)` behaves like list `[n,n+s,...,n+rs]` for the maximum r such that $n + rs \leqslant m$.

Boolean value

In Python boolean values are `True` and `False`. Check this:

```
>>> 2 > 3
False
>>> 2 > 1
True
>>> x = True
>>> type(x)
<class 'bool'>
>>> y = False
>>> type(y)
<class 'bool'>
```

Note that only `True` and `False` are boolean values. They are reserved **keywords**. Python does not recognize `TRUE`, `FALSE`, `true` or `false`.

```
>>> z = true
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'true' is not defined
```

Boolean algebra

The boolean algebra in Python is straightforward:

	Maths expression	Python expression
A and B	$A \cap B$	A `and` B
A or B	$A \cup B$	A `or` B
complement of A	A^c	`not` A

Example:

```
>>> x = 2
>>> x > 1 and x < 5
True
>>> not x > 1
False
>>> x > 1 or x < -1
True
```

Functions

In Python functions are defined via the keyword `def` and `return`.

- The following function accept a list as input and returns the sum of squares of the elements of the list.

```
>>> def myfun1(x):
...     s = 0
...     for i in x:
...             s += i**2
...     return s
...
>>> myfun1([1,2,3])
14
```

Functions

In Python functions are defined via the keyword `def` and `return`.

- The following function accept a list as input and returns the sum of squares of the elements of the list.

```
>>> def myfun1(x):
...     s = 0
...     for i in x:
...             s += i**2
...     return s
...
>>> myfun1([1,2,3])
14
```

- The following function accept two numbers as input and returns the larger of the two.

```
>>> def myfun2(x, y):
...     if x >= y:
...         return x;
...     else:
...         return y
...
>>> myfun2(3, 5)
5
```

Function with default arguments

In the definition of a function, we may specify the **default value** for one or more input arguments.

```
>>> def myfun3(x, y=0): # the default value of y is set to be zero
...     if x >= y:
...         return x;
...     else:
...         return y
...
>>> myfun3(2) # max{2, 0}=2
2
>>> myfun3(-5) # max{-5, 0}=0
0
```

When only one argument is provided, i.e. `myfun3(2)`, Python assumes that the input `2` is for `x` not for `y`.

Function variable

Function `function` is a type in Python, just like `int` or `list`.

```
>>> fun = myfun3    # function myfun3 is attributed to variable 'fun'
>>> type(fun)
<class 'function'>
>>> fun(2, 5)   # 'fun' is the same as 'mufun3'
>>> 5
```

Function variable

Function `function` is a type in Python, just like `int` or `list`.

```
>>> fun = myfun3    # function myfun3 is attributed to variable 'fun'
>>> type(fun)
<class 'function'>
>>> fun(2, 5)   # 'fun' is the same as 'mufun3'
>>> 5
```

Attention: **naming conflict** may occur when the name of your function coincides with some existing name.

```
>>> max(1,4,-2,9)   # max is a buit-in function of Python
9
>>> max = myfun3    # now the name 'max' points to myfun3
>>> max(1,4)   # myfun3 only accepts 2 input arguments
4
>>> max(1,4,-2,9) # when feed 4 arguments it throws an error
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: max() takes 2 positional arguments but 4 were given
```

Module: `math`

Python provides many predefined maths functions. Those functions are contained in `math` module. To use them, you need to enter explicitly `import math`.

```
>>> log(1)  # Python does not recognize the name 'log'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'log' is not defined
>>> import math # declare that you will use math module
>>> math.log(1)
0.0
```

Module: `math`

Python provides many predefined maths functions. Those functions are contained in `math` module. To use them, you need to enter explicitly `import math`.

```
>>> log(1)  # Python does not recognize the name 'log'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'log' is not defined
>>> import math # declare that you will use math module
>>> math.log(1)
0.0
```

You may find that using `math.log()` is cumbersome. Can't we just type `log()`?

```
>>> from math import log    # this way we can use log() directly
>>> log(1)
0.0
```

Functions and Constants in `math`

In `math` module, there are not only basic functions such as `log`, `exp`, `sqrt`, `sin`, `tan`, etc, there are also mathematical constant `pi` and `e`. For instance:

```
>>> math.pi
3.141592653589793
>>> math.e
2.718281828459045
```

Functions and Constants in `math`

In `math` module, there are not only basic functions such as `log`, `exp`, `sqrt`, `sin`, `tan`, etc, there are also mathematical constant `pi` and `e`. For instance:

```
>>> math.pi
3.141592653589793
>>> math.e
2.718281828459045
```

If you are too lazy to type `math.` each time you use a function or constant in math module, there is a way to make your life easier.

```
>>> from math import *  # no need to type 'math.' anymore
>>> pi
3.141592653589793
>>> e
2.718281828459045
>>> sin(pi)
1.2246467991473532e-16
>>> log(e)
1.0
```

This convenience comes at a price: higher risk of naming conflict. Therefore it is not advocated by expert programmers.

Exercise 1.4

Define the following function:

$$f(x, \mu, \sigma) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{\frac{-(x-\mu)^2}{2\sigma^2}},$$

where $\mu = 0$ and $\sigma = 1$ are default values for μ and σ.

Exercise 1.4

Define the following function:

$$f(x, \mu, \sigma) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{\frac{-(x-\mu)^2}{2\sigma^2}},$$

where $\mu = 0$ and $\sigma = 1$ are default values for μ and σ.

Solution:

```
from math import pi, exp, sqrt
def gaussian(x, mu=0, sigma=1):
    c = 1/sqrt(2*pi*sigma**2)
    p = -(x-mu)**2 / (2*sigma**2)
    return c * exp(p)
```

To validate the result, try:

```
>>> gaussian(0)
0.3989422804014327
>>> gaussian(1, 0.5, 2)
0.19333405840142462
```

Object Oriented Programming

- Thus far we have learned Python's built-in **types**, such as `int`, `float`, `str` and `list`, and we will encounter several more in near future.
- Although Python provides a number of different predefined types that serve various purposes, those types cannot cope with all needs in applications. For instance, there is no predefined "matrix" type in Python, while matrix is a fundamental concept in scientific computing.
- Fortunately Python allows us to define our own "types", or "classes". This provides great flexibility and is a huge advantage in practice. We say Python is a **Object Oriented Programming** (OOP) language because of this feature.
- As a matter of fact, everything in Python is an object of a certain class.

Example: Student

Let us begin with the simplest example.

```
class Student:
    def __init__(self, n, g):
        self.name = n
        self.grade = g

    def info(self):
        print("Student name: ", self.name)
```

The code above defines a "Student" class. Now try:

```
>>> x = Student('Han Meimei', 2013)
>>> x.name
'Han Meimei'
>>> x.grade
2013
>>> x.info()
Student name:  Han Meimei
```

Example: Student

Let us begin with the simplest example.

```
class Student:
    def __init__(self, n, g):
        self.name = n
        self.grade = g

    def info(self):
        print("Student name: ", self.name)
```

The code above defines a "Student" class. Now try:

```
>>> x = Student('Han Meimei', 2013)
>>> x.name
'Han Meimei'
>>> x.grade
2013
>>> x.info()
Student name:  Han Meimei
```

Using the OOP jargon, we say

- `x` is an **instance** or **object** of class "Student".
- `name` and `grade` are **attributes** of class "Student".
- `info()` and `__init__()` are **methods** of class "Student".
- `__init__()` is a special method called the **constructor**.

Exercise 2.1

- Create an instance of `Student` that describes yourself.
- Add another attribute called `year` to the class `Student` that stores the year of birth of the student.
- Add another method called `age()` that print and then return the age of the student.

Solution:

```
class Student:
    def __init__(self, n, g, y):
        self.name = n
        self.grade = g
        self.year = y   # attribute 'year' added

    def info(self):
        print("Student name: ", self.name)

    def age(self, this_year):  # method 'age()' added
        age = this_year - self.year
        print("Current age: ", age)
        return age
```

```
>>> x = Student('Han Meimei', 2013, 1995)
>>> x.year
1995
>>> x.age(2017)
Current age:  22
```

Exercise 2.2

Define a class named `Point`, such that:

- Its constructor accepts two inputs `px` and `py`, which are the coordinates of the point to be constructed.
- It has two attributes `x` and `y`, storing respectively the x-axis and y-axis coordinate.
- It has a method named `length()`, which returns the distance from the point to the origin.

Exercise 2.2

Define a class named `Point`, such that:

- Its constructor accepts two inputs `px` and `py`, which are the coordinates of the point to be constructed.
- It has two attributes `x` and `y`, storing respectively the x-axis and y-axis coordinate.
- It has a method named `length()`, which returns the distance from the point to the origin.

```
import math
class Point:
    def __init__(self, px, py):
        self.x = px
        self.y = py

    def length(self):
        return math.sqrt(self.x**2 + self.y**2)
```

Now try:

```
>>> a = Point(2,5)
>>> a.x
2
>>> a.y
5
>>> a.length()
5.385164807134504
```

Define addition for `Point`

Assume now we have two `Point` objects:

```
>>> a = Point(2, 5)
>>> b = Point(1,-3)
```

Let us check out what `a + b` gives:

```
>>> a + b
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for +: 'Point' and 'Point'
```

Define addition for `Point`

Assume now we have two `Point` objects:

```
>>> a = Point(2, 5)
>>> b = Point(1,-3)
```

Let us check out what `a + b` gives:

```
>>> a + b
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for +: 'Point' and 'Point'
```

It turns out that addition "+" is not defined for `Point` objects. But we can tell Python what to do with "Point + Point" arithmetic.

```
# add the following method to Point class
def __add__(self, other):   # define '+' for points
    s = Point(self.x + other.x, self.y + other.y)
    return s
```

Now we have

```
>>> c = a + b
>>> type(c)
<class '__main__.Point'>
>>> c.x
3
>>> c.y
2
```

Operator overloading

Operator	Function	Description
$+$	__add__(self, other)	addition
$-$	__sub__(self, other)	subtraction
$*$	__mul__(self, other)	multiplication
$/$	__truediv__(self, other)	division

Example: Matrix

Exercise 2.3

Let us try to define a `Matrix` class. In this exercise, we consider only 2×2 square matrix.

1. Write an appropriate constructor. During construction, store the matrix elements in some class attributes.
2. Write a method named `show()` that print the elements of the matrix.
3. Write a method named `det()` that return the determinant of the matrix.
4. Write a method named `inv()` that return the inverse of the matrix. If the matrix is singular, print a phrase indicating the singularity of the matrix and then return `None`.
5. Define multiplication `*` for matrices.

```
class Matrix:
  def __init__(self, x): # we assume x looks like [[1,2],[3,4]]
    self.data = x
```

```
class Matrix:
  def __init__(self, x): # we assume x looks like [[1,2],[3,4]]
    self.data = x
```

```
  def show(self):   # print the first row, then the second row
    print(self.data[0][0], self.data[0][1])
    print(self.data[1][0], self.data[1][1])
```

```
class Matrix:
  def __init__(self, x): # we assume x looks like [[1,2],[3,4]]
    self.data = x

  def show(self):   # print the first row, then the second row
    print(self.data[0][0], self.data[0][1])
    print(self.data[1][0], self.data[1][1])

  def det(self):
    return self.data[0][0]*self.data[1][1] - self.data[0][1]*self.data[1][0]
```

```
class Matrix:
  def __init__(self, x): # we assume x looks like [[1,2],[3,4]]
    self.data = x

  def show(self):   # print the first row, then the second row
    print(self.data[0][0], self.data[0][1])
    print(self.data[1][0], self.data[1][1])

  def det(self):
    return self.data[0][0]*self.data[1][1] - self.data[0][1]*self.data[1][0]

  def inv(self):
    det = self.det()
    if det == 0:
      print("This matrix is singular. ")
      return None
    else:
      a00 =  self.data[1][1] / det
      a01 = -self.data[0][1] / det
      a10 = -self.data[1][0] / det
      a11 =  self.data[0][0] / det
      return Matrix([[a00, a01], [a10, a11]])
```

```
class Matrix:
  def __init__(self, x): # we assume x looks like [[1,2],[3,4]]
    self.data = x

  def show(self):   # print the first row, then the second row
    print(self.data[0][0], self.data[0][1])
    print(self.data[1][0], self.data[1][1])

  def det(self):
    return self.data[0][0]*self.data[1][1] - self.data[0][1]*self.data[1][0]

  def inv(self):
    det = self.det()
    if det == 0:
      print("This matrix is singular. ")
      return None
    else:
      a00 =  self.data[1][1] / det
      a01 = -self.data[0][1] / det
      a10 = -self.data[1][0] / det
      a11 =  self.data[0][0] / det
      return Matrix([[a00, a01], [a10, a11]])

  def __mul__(self, other):
    a00=self.data[0][0]*other.data[0][0] + self.data[0][1]*other.data[1][0]
    a01=self.data[0][0]*other.data[0][1] + self.data[0][1]*other.data[1][1]
    a10=self.data[1][0]*other.data[0][0] + self.data[1][1]*other.data[1][0]
    a11=self.data[1][0]*other.data[0][1] + self.data[1][1]*other.data[1][1]
    return Matrix([[a00, a01], [a10, a11]])
```

Let us verify our code:

```
>>> a=Matrix([[1,2],[3,4]])
>>> a.show()
1 2
3 4
>>> a.det()
-2
>>> b=a.inv()
>>> b.show()
-2.0 1.0
1.5 -0.5
>>> c=a*b
>>> c.show()
1.0 0.0
0.0 1.0
>>>
```

and

```
>>> p = Matrix([[1,2],[1,2]])
>>> p.det()
0
>>> p.inv()
This matrix is singular.
```

Class: list

- `list` is a built-in class in Python.
- Instance of `list` can be created by using brackets `[...]`.
- Class `list` contains many methods. Below are some of them:

Class: list

- `list` is a built-in class in Python.
- Instance of `list` can be created by using brackets `[...]`.
- Class `list` contains many methods. Below are some of them:

Name	Use
`append(x)`	Add an item `x` to the end of the list.
`pop()`	Removes and returns the last item in the list.
`insert(i, x)`	Insert an item `x` at a position `i`.
`reverse()`	Reverse the elements of the list in place.
`sort()`	Sort the list in place.

Example

```
>>> a = [2,4,6,8]
>>> a.apppend(10)   # equivalent to a + [10]
>>> a
[2, 4, 6, 8, 10]
>>> a.reverse() # reverse the list in place.
>>> a
[10, 8, 6, 4, 2]
>>> a.pop() # remove the last element 2 from list
2
>>> a
[10, 8, 6, 4]
>>> a.insert(3, 7) # insert element 7 before the element at index 3
>>> a
[10, 8, 6, 7, 4]
>>> a.sort()    # sort the list in place
>>> a
[4, 6, 7, 8, 10]
```

Example

```
>>> a = [2,4,6,8]
>>> a.apppend(10)   # equivalent to a + [10]
>>> a
[2, 4, 6, 8, 10]
>>> a.reverse() # reverse the list in place.
>>> a
[10, 8, 6, 4, 2]
>>> a.pop() # remove the last element 2 from list
2
>>> a
[10, 8, 6, 4]
>>> a.insert(3, 7) # insert element 7 before the element at index 3
>>> a
[10, 8, 6, 7, 4]
>>> a.sort()    # sort the list in place
>>> a
[4, 6, 7, 8, 10]
```

From the snippet above we see that a `list` object is **mutable**, which means that the state of the object can be changed.

Shallow Copy

To understand **shallow copy**, let us go over an example of the **deep copy**:

```
>>> a = 1
>>> b = a
>>> b
1
>>> a = 2
>>> a
2
>>> b    # the value of b is not changed
1
```

However, for `list` object:

```
>>> a = [1,2,3]
>>> b = a
>>> b
[1, 2, 3]
>>> a.append(4)
>>> b   # we see that b is changed
[1, 2, 3, 4]
```

The copy of a `list` object turns out to be a shallow copy.

Class: tuple

- `tuple` is a built-in Python class. A `tuple` object can be created by using parentheses `(...)`:

```
>>> x=(1, 4)
>>> x
(1, 4)
>>> type(x)
<class 'tuple'>
>>> len(x)
2
>>> x[1]
4
```

Class: tuple

- `tuple` is a built-in Python class. A `tuple` object can be created by using parentheses `(...)`:

```
>>> x=(1, 4)
>>> x
(1, 4)
>>> type(x)
<class 'tuple'>
>>> len(x)
2
>>> x[1]
4
```

- We saw that a `tuple` object behaves pretty much like a `list`. The "only" difference between the two is that a `tuple` object is **immutable**.

```
>>> x[0]=5
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
>>> x + (1, 8) # this yields another tuple
(2, 4, 1, 8)
>>> x # x remains unchanged
(2, 4)
```

Class: dict

- Dictionary `dict` is another Python class that sees frequent use in programs.
- To understand its usage, let us consider a simple application: we want to store the result of a exam. The data consists of many (*name*, *score*) pairs, something like

```
('Li Lei', 88)
('Han Meimei', 91)
('Gao Hui', 97)
...
```

- We may create a `list` to store these data, but such solution does not allow for fast lookup of someone's score (one need to iterate over the entire list to find a given *name*).

- Class `dict` is suitable for this kind of task. An instance of `dict` can be created by using curly braces `{...}`. Let us see a concrete example:

```
>>> data = {'Li Lei': 88, 'Han Meimei': 91, 'Gao Hui': 97}
{'Li Lei': 88, 'Gao Hui': 97, 'Han Meimei': 91}
>>> type(data)
<class 'dict'>
>>> len(data)
3
>>> data['Li Lei']  # this allow for fast lookup
88
```

- Class `dict` is suitable for this kind of task. An instance of `dict` can be created by using curly braces `{...}`. Let us see a concrete example:

```
>>> data = {'Li Lei': 88, 'Han Meimei': 91, 'Gao Hui': 97}
{'Li Lei': 88, 'Gao Hui': 97, 'Han Meimei': 91}
>>> type(data)
<class 'dict'>
>>> len(data)
3
>>> data['Li Lei']  # this allow for fast lookup
88
```

- The basic syntax for creating a `dict` object is as follows:

```
d = {key1:value1, key2:value2, key3:value3, ...}
```

where `key` and `value` can be anything having type `int`, `float` or `str`. The `update()` method in `dict` class allows for "extending" a given `dict` object.

```
>>> z = {'Jim Green': 82}
>>> data.update(z)
>>> data['Jim Green']
82
```