第2课　Python基础知识2

函数

- 函数：**def**

```
def function_name(args):
    statements
```

- 注意statements前面的**缩进**！
- return：函数返回值（可以多个）

```
return val1, val2, val3
```

- 函数的类型
  - 自建的函数，例如print_greeting()
  - 内置Python函数， 例如print()
  - 标准库中的函数 ， 例如math.sqrt()

```
# 定义函数
def print_greeting():
    print "Hello!"
    print "How are you today?"

print_greeting() # 调用函数
```

输出：
Hello!
How are you today?

函数

- Python里面使用引用传递(pass by reference)：多个变量，指向同一个内存地址，修改任意一个，全部改动。
- 只有**可变类型**的对象可以在函数里被修改

```
def hello_func(name, somelist):
    print("Hello,", name, "!\n")
    name = "Caitlin"
    somelist[0] = 3


myname = "Ben"
mylist = [1,2]
a,b = hello_func(myname, mylist)

print(myname, mylist)
print(a, b)
```

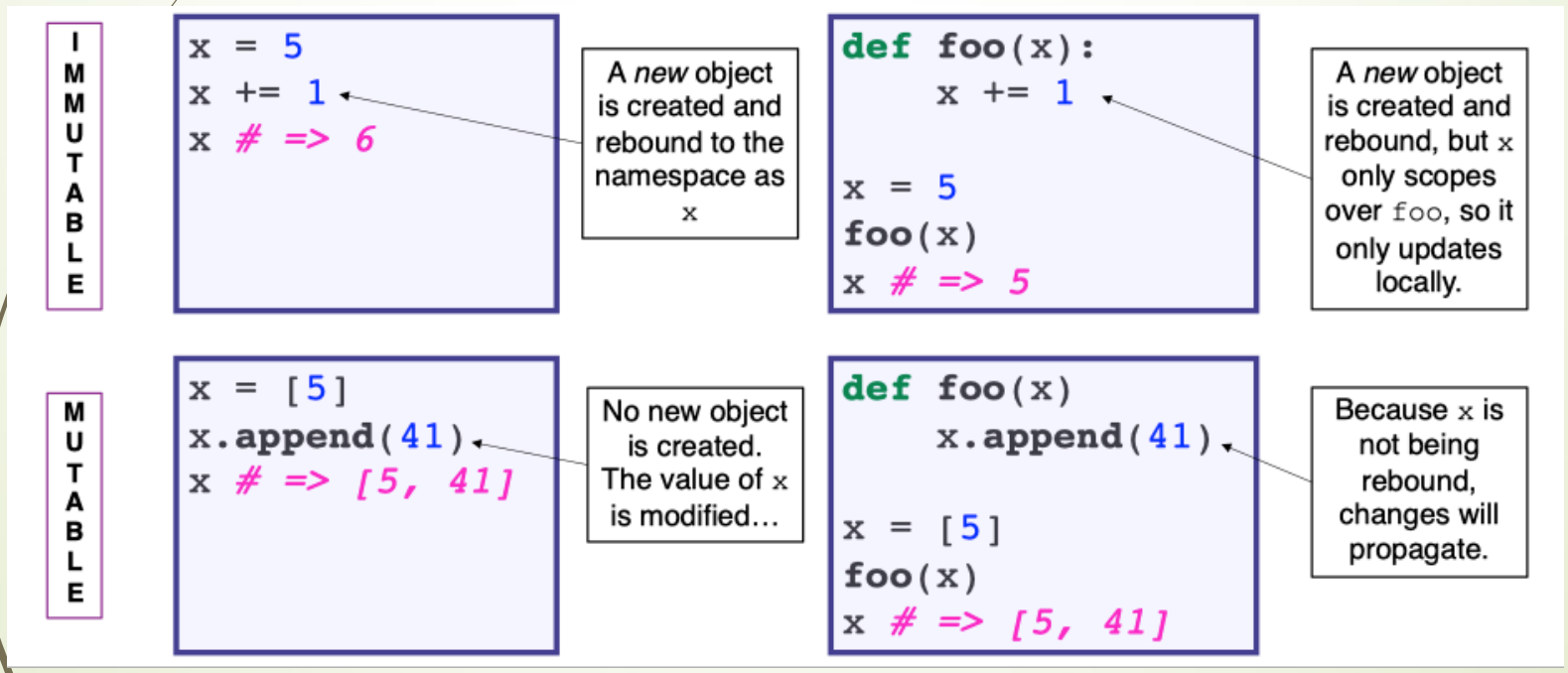
输出：

Hello, Ben !

Ben [3, 2]

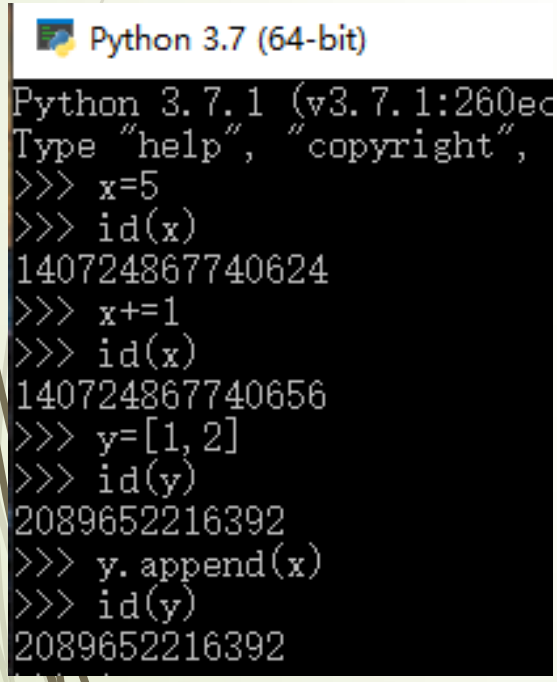
可变 和 不可变 类型

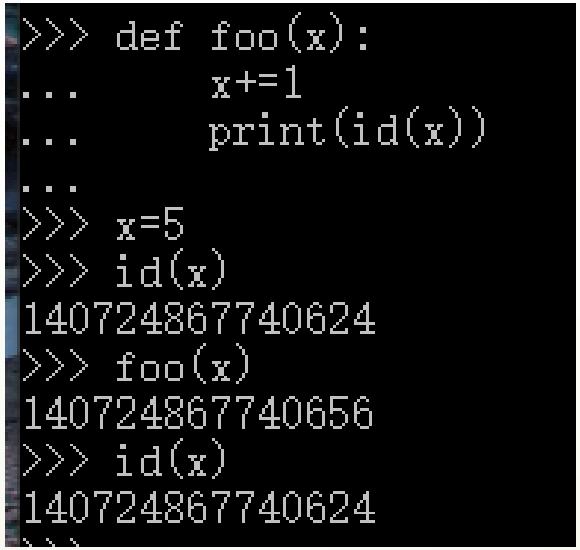
- Python的数据类型分为mutable（可变） 和 immutable （不可变）
- mutable : list ,dict
- immutable : int , string , float ,tuple...

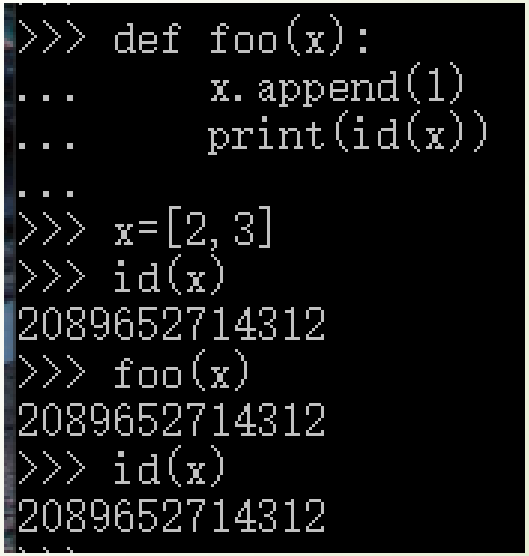


可变 和 不可变 类型

- Python的数据类型分为mutable（可变） 和 immutable （不可变）
- mutable : list ,dict
- immutable : int , string , float ,tuple...







mutable和 immutable

- **字符串（string）和元组（tuple）属于不可变序列，**不能通过下标的方式来修改其中的元素值，试图修改元组中元素的值时会抛出异常。

```
>>> x = (1,2,3)
>>> print(x)
(1, 2, 3)

>>> x[1] = 5
Traceback (most recent call last):
  File "<pyshell#7>", line 1, in <module>
    x[1] = 5
TypeError: 'tuple' object does not support item
  assignment
```

函数：可变类型

下面程序的输出结果是？（提示：列表对象是可变类型）

```
def hello_func(names):
    for n in names:
        print("Hello,", n, "!")
    names[0] = 'Susie’
    names[1] = 'Pete’
    names[2] = 'Will’

names = ['Susan', 'Peter', 'William’]

hello_func(names)

print("The names are now", names, ".")
```

函数：可变类型

下面程序的输出结果是？（提示：list对象是可变类型）

```
def hello_func(names):
    for n in names:
        print("Hello,", n, "!")
    names[0] = 'Susie’
    names[1] = 'Pete’
    names[2] = 'Will’

names = ['Susan', 'Peter', 'William’]

hello_func(names)

print("The names are now", names, ".")
```

输出：

Hello, Susan !
Hello, Peter !
Hello, William !
The names are now [‘Susie’, ‘Pete’, ‘Will’] .

练习：使用函数

- Fibonacci 序列：从第三项开始，每一项等于前两项的和
- 假设从1，2开始，前面10项是 1，2，3，5，8，13，21，34，55，89，...

- 练习1：输出第11项到第20项
- 练习2：输出前20项的偶数
- 练习3：计算Fibonacci 序列中不超过4000000的所有偶数的和

- **函数：（1） 指定Fibonacci 序列的范围；（2）检查偶数；（3）求和**

练习：使用函数

```
def print_fib_numbers(min_index,large_index):
    f1, f2 = 1, 2
    for i in xrange(1,large_index+1):
        if i>=min_index:
            print f1
        f1, f2 = f2, f1 + f2
    print "\n"

def iseven(number):
    if number%2==0:
        return True
    else:
        return False

def sum_even_fib(largerst_num):
    total = 0
    f1, f2 = 1, 2
    while f1 < largerst_num:
        if iseven(f1):
            total = total + f1
        f1, f2 = f2, f1 + f2
    return total


if __name__ == "__main__":
    print_fib_numbers(11,20)


    largerst_num = 4000000
    print(sum_even_fib(largerst_num))
```

Python的object类

- 在Python中处理的一切都是对象（object）。
- isinstance() 函数来判断一个对象是否是一个已知的类型
- 语法： isinstance(object, classinfo)

```
isinstance(4,object)
isinstance(4,int)

isinstance("tom",object)
isinstance("tom",str)
isinstance(str,object)

isinstance(None,object)
```

- 以上全部返回True

Python的object类

- **Python语言是面向对象的编程语言，可以说在Python中一切皆对象。**对象是某类型具体实例中的某一个，每个对象都有身份、类型和值。
  - 身份（Identity）与对象都是唯一对应关系，每一个对象的身份产生后就都是独一无二的，并无法改变。对象的ID是对象在内存中获取的一段地址的标识。
  - 类型（Type）是决定对象将以哪种数据类型进行存储。
  - 值（Value）存储对象的数据，某些情况下可以修改值，某些对象声明值过后就不可以修改了（可变和不可变）。

Python的object类

- object具有身份（identity）、类型（type）和值（value）
- id() 函数用于获取对象的内存地址： id(object)

```
id(10) # => 140335174937008 (例子)
```

- type() 函数用于获取第参数对象的类型： type(object)

```
type("shenzhen") # => str
type(10) # => int
type(2.5) # => float
```

- Object 包含指向其基础数据块的指针（pointer）。

```
(10).__sizeof__() # => ? (bytes)
```

变量

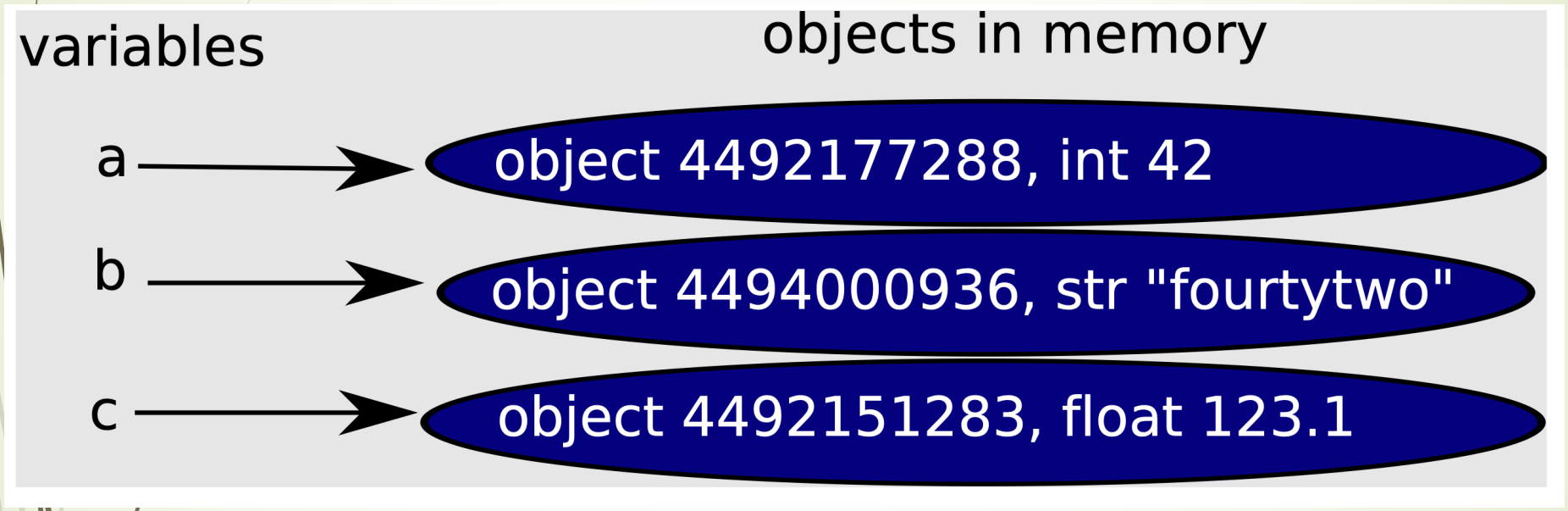
- 变量的赋值

```
x = 4654
y = "Hello!"
```

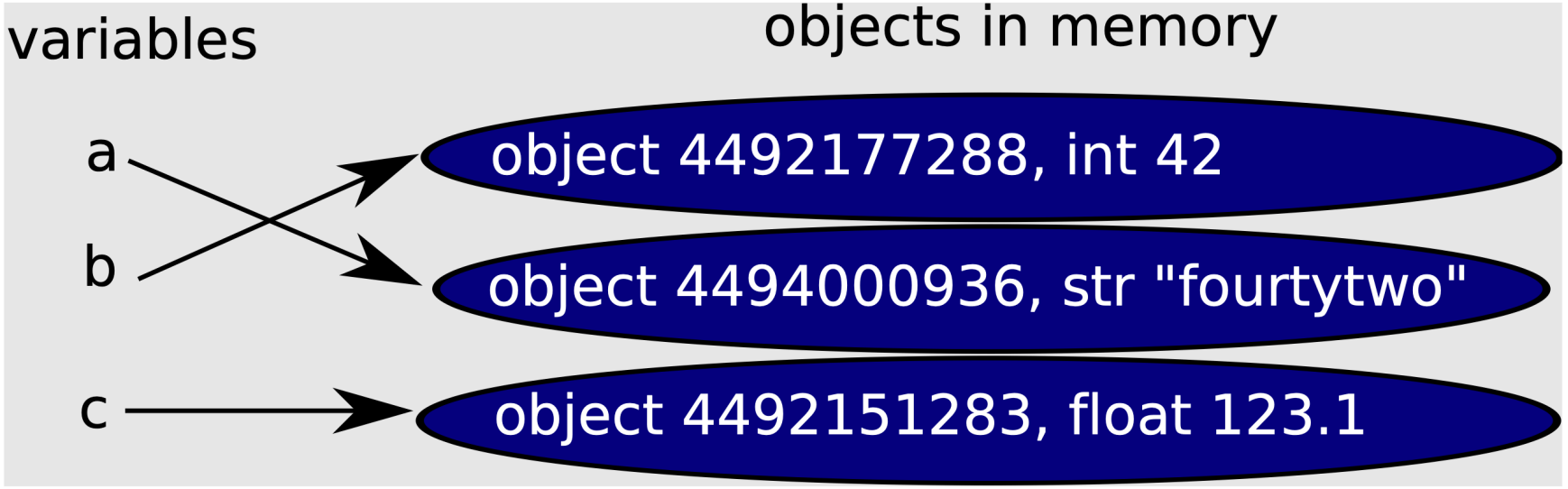
- 如果变量已经被定义，当给一个变量赋值的时候，本质上是 **修改了数据的引用**
- 变量 **不再** 对之前的数据引用
- 变量 **改为** 对新赋值的数据引用

```
x = 4654
y = x
```

变量

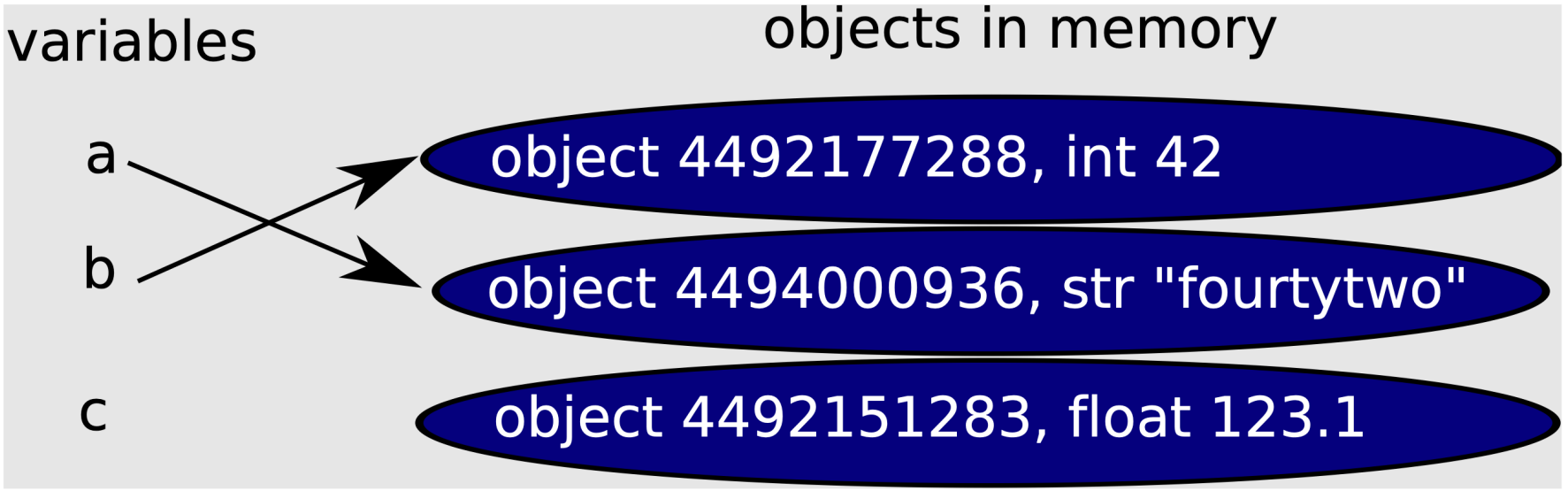

variables
objects in memory
a
object 4492177288, int 42
b
object 4494000936, str "fourtytwo"
c
object 4492151283, float 123.1

变量


variables
objects in memory
a
b
c
object 4492177288, int 42
object 4494000936, str "fourtytwo"
object 4492151283, float 123.1

```
>>> a,b = b,a
```

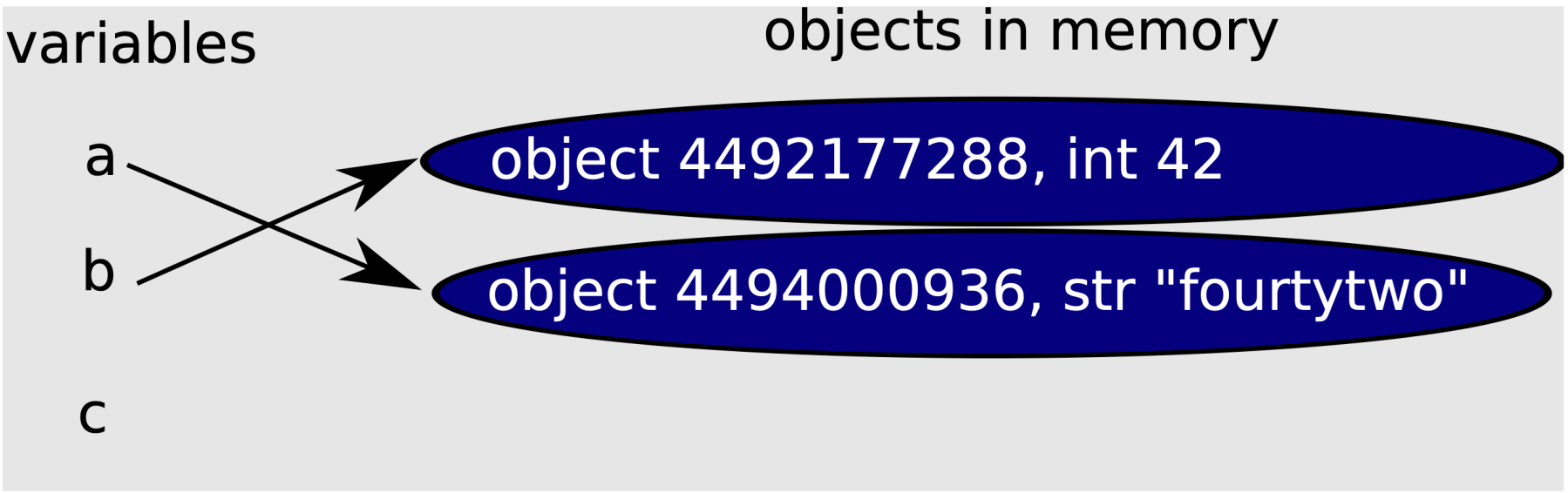
变量




```
>>> a,b = b,a
```

```
>>> c=None
```

变量




```
>>> a,b = b,a
```

```
>>> c=None
```

Garbage collector removes unreferenced object.

是否相等？

```
1 == 1.0 # True; why ???
type(1) == type(1.0) # False; why ???
```

- is运算符：检查标识，而不是相等性。

```
1 is not 1.0 # True; why ???

x = "shenzhen"
y = "shenzhen"

x == y # => True
x is y # => True

id(x) # => 1740267419248
id(y) # => 1740267419248
```

是否相等？

- 同一性测试运算符（identity comparison）is用来测试两个对象是否是同一个对象，如果是则返回True，否则返回False。如果两个对象是同一个，二者具有*相同的内存地址*。

```
>>> x = [300, 300, 300]
>>> x[0] is x[1]                #基于值的内存管理，同一个值在内存中只有一份
True
>>> x=[1,2,3]
>>> id(x)
140198338361472
>>> y=x                         #这样创建的x和y是同一个列表对象
>>> id(y)
140198338361472
>>> z=[1,2,3]                   #这样创建的x和z不是同一个列表对象
>>> id(z)
140198336986688
```

函数也是对象

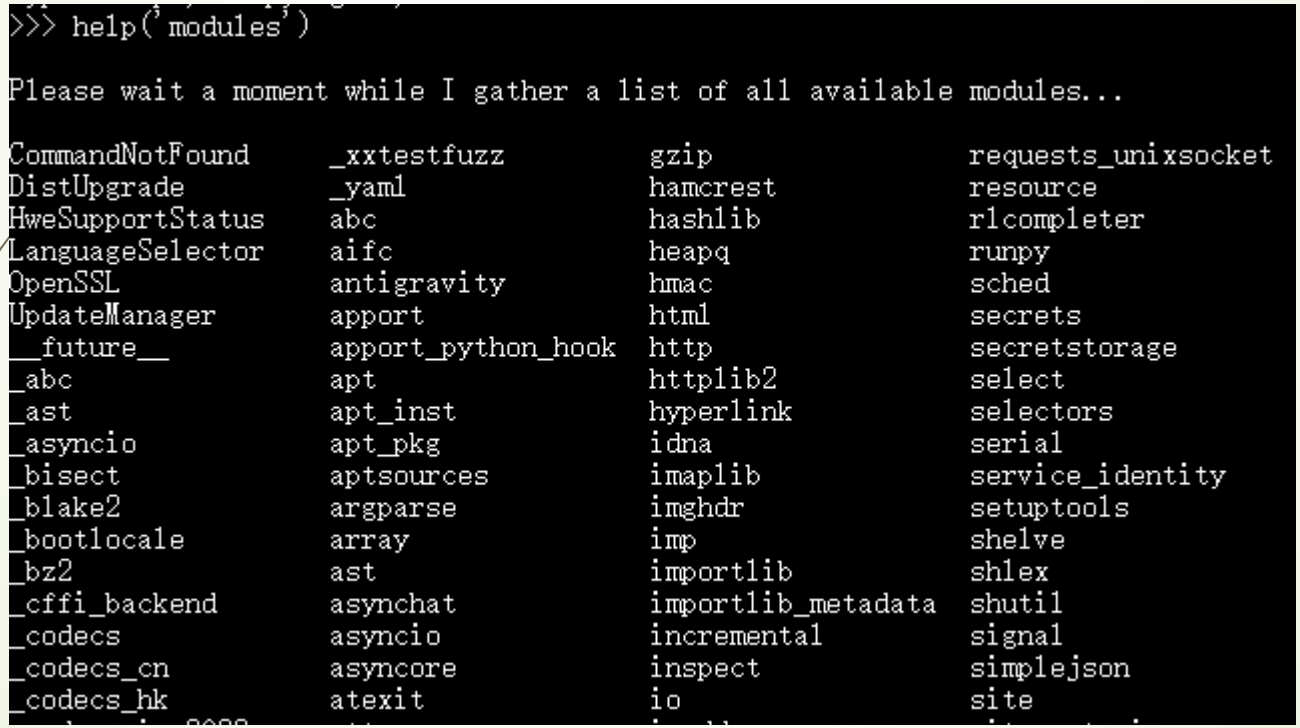
```
def echo(arg):
    return arg

type(echo) # => <class 'function'>
id(echo) # => 4393732128
print(echo) # => <function echo at 0x105e30820>

foo = echo
type(foo) # => <class 'function'>
id(foo) # => 4393732128
print(foo) # => <function echo at 0x105e30820>

isinstance(echo, object) # => True
```

Python 模块

- 随着程序代码越写越多，在一个文件里代码就会越来越长，越来越不容易维护…
- 一个Python模块包括定义，语句，函数，…
- 文件名 = 模块名 + .py
- 内置标准模块（又称标准库）执行help（’modules’）查看所有Python所有自带模块列表。
- 第三方开源块，可通过 *pip install 模块名* 联网安装
- 模块的好处：提高可维护性，可重复利用

Python 模块

- 内置标准模块（又称标准库）执行help（’modules’）查看所有Python所有自带模块列表。



Python 模块

- from 模块名 import 对象名[as 别名]

#可以减少查询次数，提高执行速度

- from math import *

```
>>> from math import sin
>>> sin(3)
0.1411200080598672

>>> from math import sin as f #别名
>>> f(3)
0.141120008059867
```

Python 模块

- 易错点！

```
>>> import math
>>> sin(1)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'sin' is not defined
>>> math.sin(1)
0.8414709848078965
>>>
>>>
>>> sin(1)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'sin' is not defined
>>>
>>> from math import sin
>>> sin(1)
0.8414709848078965
>>>
```

Python 模块

- 导入内置标准模块
- 一般情况下，import语句写在Python文件的最开始的地方

```
# Import a module.
import math
math.sqrt(16) # => 4.0

# Import specific symbols from a module
from math import ceil, floor
ceil(3.7) # => 4.0
floor(3.7) # => 3.0
```

❖如果需要导入多个模块，一般建议按如下顺序进行导入：

✓标准库

✓成熟的第三方扩展库

✓自己开发的库

Python模块

- 当一个.py文件作为模块被导入时，我们可能不希望一部分代码被运行。
- 当.py文件被直接运行时，
  if __name__ == '__main__'之下的代码块将被运行；
- 当.py文件以模块形式被导入时，
  if __name__ == '__main__'之下的代码块不被运行。
- 方便模块的调用与测试

```
def print_fib_numbers(min_index,large_index):
    f1, f2 = 1, 2
    for i in xrange(1,large_index+1):
        if i>=min_index:
            print f1
        f1, f2 = f2, f1 + f2
    print "\n"

def iseven(number):
    if number%2==0:
        return True
    else:
        return False

def sum_even_fib(largerst_num):
    total = 0
    f1, f2 = 1, 2
    while f1 < largerst_num:
        if iseven(f1):
            total = total + f1
        f1, f2 = f2, f1 + f2
    return total


if __name__ == "__main__":
    print_fib_numbers(11,20)


    largerst_num = 4000000
    print(sum_even_fib(largerst_num))
```

Python 模块：小测

- foo.py 和 bar.py.

```
''' Module bar.py '''

print "Hi from bar's top
level!"

def print_hello():
    print "Hello from bar!"

if __name__ == "__main__":
    print "bar's __name__ is
__main__"
```

```
''' Module foo.py'''

import bar

print "Hi from foo's top
level!"

if __name__ == "__main__":
    print "foo's __name__ is
__main__"
    bar.print_hello()
```

Python 模块：小测

- foo.py 和 bar.py.

```
''' Module bar.py '''

print "Hi from bar's top
level!"

def print_hello():
    print "Hello from bar!"

if __name__ == "__main__":
    print "bar's __name__ is
__main__"
```

```
''' Module foo.py'''

import bar

print "Hi from foo's top
level!"

if __name__ == "__main__":
    print "foo's __name__ is
__main__"
    bar.print_hello()
```

$ python bar.py　　**直接运行模块 bar，输出什么？**

Python 模块：小测

- foo.py 和 bar.py.

```
''' Module bar.py '''

print "Hi from bar's top
level!"

def print_hello():
    print "Hello from bar!"

if __name__ == "__main__":
    print "bar's __name__ is
__main__"
```

```
''' Module foo.py'''

import bar

print "Hi from foo's top
level!"

if __name__ == "__main__":
    print "foo's __name__ is
__main__"
    bar.print_hello()
```

```
$ python bar.py
Hi from bar's top level!
bar's __name__ is __main__
```

Python 模块：小测

- foo.py 和 bar.py.

```
''' Module bar.py '''

print "Hi from bar's top
level!"

def print_hello():
    print "Hello from bar!"

if __name__ == "__main__":
    print "bar's __name__ is
__main__"
```

```
''' Module foo.py'''
import bar

print "Hi from foo's top
level!"

if __name__ == "__main__":
    print "foo's __name__ is
__main__"
    bar.print_hello()
```

如果运行模块foo，输出什么？
（注意 import bar）

```
$ python foo.py
```

Python 模块：小测

- foo.py 和 bar.py.

```
''' Module bar.py '''

print "Hi from bar's top level!"

def print_hello():
    print "Hello from bar!"

if __name__ == "__main__":
    print "bar's __name__ is __main__"
```

```
''' Module foo.py'''

import bar

print "Hi from foo's top level!"

if __name__ == "__main__":
    print "foo's __name__ is __main__"
    bar.print_hello()
```

```
$ python foo.py
Hi from bar's top level!
Hi from foo's top level!
foo's __name__ is __main__
Hello from bar!
```

Python 模块：小测

- foo.py 和 bar.py.

```
''' Module bar.py '''

print "Hi from bar's top
level!"

def print_hello():
    print "Hello from bar!"

if __name__ == "__main__":
    print "bar's __name__ is
__main__"
```

```
''' Module foo.py'''

import bar

print "Hi from foo's top
level!"

if __name__ == "__main__":
    print "foo's __name__ is
__main__"
    bar.print_hello()
```

import foo 到 Python解析器？

```
$ python
>>> import foo
```

Python 模块：小测

- foo.py 和 bar.py.

```
''' Module bar.py '''

print "Hi from bar's top
level!"

def print_hello():
    print "Hello from bar!"

if __name__ == "__main__":
    print "bar's __name__ is
__main__"
```

```
''' Module foo.py'''

import bar

print "Hi from foo's top
level!"

if __name__ == "__main__":
    print "foo's __name__ is
__main__"
    bar.print_hello()
```

```
$ python
>>> import foo
Hi from bar's top level!
Hi from foo's top level!
```

Python 模块：小测

- foo.py 和 bar.py.

```
''' Module bar.py '''

print "Hi from bar's top
level!"

def print_hello():
    print "Hello from bar!"

if __name__ == "__main__":
    print "bar's __name__ is
__main__"
```

```
''' Module foo.py'''

import bar

print "Hi from foo's top
level!"

if __name__ == "__main__":
    print "foo's __name__ is
__main__"
    bar.print_hello()
```

```
$ python
>>> import bar
Hi from bar's top level!
>>>
```

Import bar 到 Python解析器？

Python 模块：小测

- foo.py 和 bar.py.

```
''' Module bar.py '''

print "Hi from bar's top level!"

def print_hello():
    print "Hello from bar!"

if __name__ == "__main__":
    print "bar's __name__ is __main__"
```

```
''' Module foo.py'''

import bar

print "Hi from foo's top level!"

if __name__ == "__main__":
    print "foo's __name__ is __main__"
    bar.print_hello()
```

如果先import foo，再import bar, 到Python解析器？
为什么import bar之后没有输出？

不管执行了多少次import，一个模块只会被导入一次。

```
$ python
>>> import foo
Hi from bar's top level!
Hi from foo's top level!
>>> import bar
>>>
```

Python 如何寻找模块？

- Python按照以下顺序寻找模块:
- 解释器首先寻找具有该名称的**内置模块**
- 如果没有找到，然后解释器从 sys.path 变量给出的目录列表里寻找
- sys.path 初始有这些目录地址:
  - 包含输入脚本的目录（或者未指定文件时的当前目录）。
  - PYTHONPATH （一个包含目录名称的列表，它和shell变量 PATH 有一样的语法）。
  - 取决于安装的默认设置

Python 如何寻找模块?

- sys.path

```
>>> import sys
>>> print(sys.path)
```

['', '/usr/lib/python38.zip', '/usr/lib/python3.8', '/usr/lib/python3.8/lib-dynload', '/usr/local/lib/python3.8/dist-packages', '/usr/lib/python3/dist-packages']

文件

- Python有文件对象可以用来文件操作。有两种方法可以创建文件对象。
  - 使用file() 构造函数：第二个参数接受几个特殊字符：“r”表示读取（默认值），“w”表示写入，“a”表示追加，“r+”表示读取和写入，“b”表示二进制模式。

```
>>> f = file("filename.txt", 'r')
```

  - 使用open()
    - 第一个参数是文件名，第二个参数是模式

```
>>> f = open("filename.txt", 'rb')
```

注意：当文件操作失败时，会引发IOError异常。

- 通常使用open()。在Python3.x中删除file() 构造函数。

读取文件

- 有几种方法可以读取文件
- f.read(size)
  - read():以字符串形式返回文件的**全部内容**。
  - 提供size参数来限制所选的字符数。
- f.readline()
  - 逐个返回文件的**每一行**作为字符串（以换行符结尾）。
  - 返回字符串为空时，到达文件结尾。
- **在文件对象上循环**
  - 最常见，就是使用for循环！

```
>>> f = open("somefile.txt",'r')
>>> f.read()
"Here's a line.\nHere's another line.\n"
>>> f.close()

>>> f = open("somefile.txt",'r')
>>> f.readline()
"Here's a line.\n"
>>> f.readline()
"Here's another line.\n"
>>> f.readline()
''
>>> f.close()

>>> f = open("somefile.txt",'r')
>>> for line in f:
...     print(line)
...
Here's a line.

Here's another line.
```

读取文件

- 用f.close()关闭文件，并释放资源。

```
>>> f = open("somefile.txt", 'r')
>>> f.readline()
"Here's line in the file! \n"
>>> f.close()
```

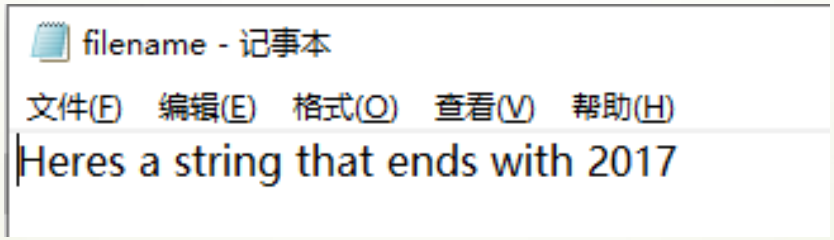
- 另一种打开和读取文件的方式
  - ✓ 不需要关闭文件，文件对象读取超出范围时会自动关闭。

```
with open("somefile.txt", "r") as txt:
    for line in txt:
        print(line)
```

写入文件

- f.write(*str*)
  - ✓ 将字符串参数str写入file对象，并返回None
  - ✓ 为了确保传递字符串，必要时使用str()构造函数。

```
>>> f = open("filename.txt", 'w')
>>> f.write("Heres a string that ends with " + str(2017))
>>> f.close()
```

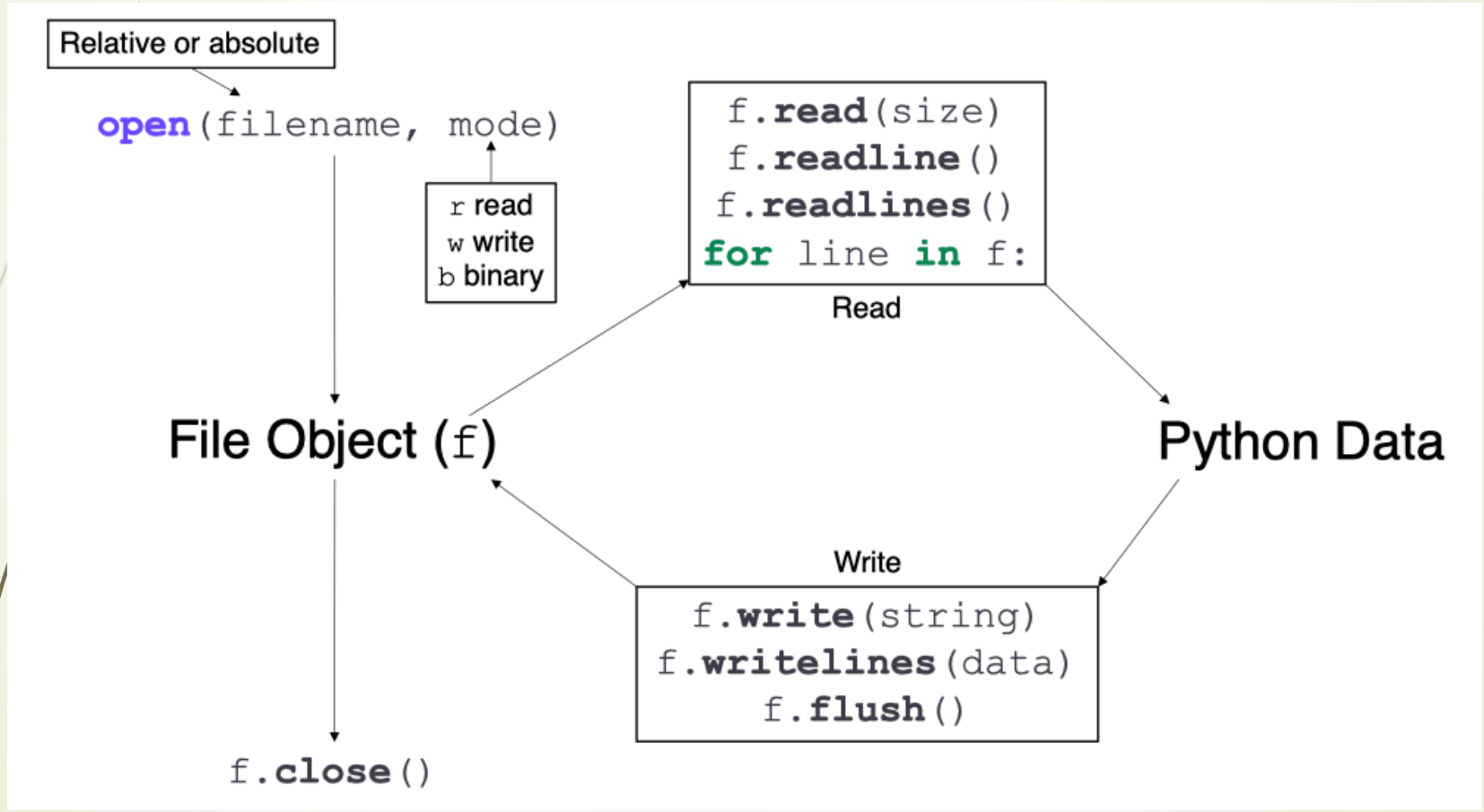


文件操作

- 默认情况下，Python在当前目录中查找文件。也可以提供文件的绝对路径，或者使用os.chdir() 函数更改当前工作目录

  - os.chdir(*newcwd*)改变当前工作目录到指定的路径
  - os.rename(*current_name*, *new_name*) 重命名
  - os.remove(*filename*) 删除文件
  - os.mkdir(*newdirname*)创建目录
  - os.getcwd()返回当前工作目录
  - os.rmdir(*dirname*)删除空目录

文件操作总结


Relative or absolute
open(filename, mode)
r read
w write
b binary
f.read(size)
f.readline()
f.readlines()
for line in f:
Read
File Object (f)
Python Data
Write
f.write(string)
f.writelines(data)
f.flush()
f.close()

异常

- 异常就是一个事件，该事件会在程序执行过程中有语法等错误的时候发生，异常会影响程序的正常执行。
- 通常在Python无法正常处理程序时就会发生一个异常，程序会终止执行。

```
>>> print(spam)
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
NameError: name 'spam' is not defined
```

```
>>> '2' + 2
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
TypeError: cannot concatenate 'str' and 'int' objects
```

处理异常

- 当Python程序发生异常时，我们需要检测捕获处理它
- 显式处理异常允许我们控制程序中其他未定义的行为，并提醒用户出现错误。使用try/except块捕获异常并从异常中恢复。

```
>>> while True:
...     try:
...         x = int(raw_input("Enter a number: "))
...     except ValueError:
...         print("Ooops !! That was not a valid number. Try again.") ...
```

注：在Python 2.x中，raw_input()函数返回结果的类型一律为字符串

处理异常

- 首先，执行try块。如果没有错误，则跳过except。
- 如果有错误，则跳过try块的其余部分。继续执行具有匹配异常类型的except块。

```
>>> while True:
...     try:
...         x = int(raw_input("Enter a number: "))
...     except ValueError:
...         print("Ooops !! That was not a valid number. Try again.") ...

Enter a number: two
Ooops !! That was not a valid number. Try again.
Enter a number: 100
```

处理异常

- 如果没有与异常类型匹配的except块，则异常未经处理，执行将停止。

```
>>> while True:
...     try:
...         x = int(input("Enter a number: "))
...     except ValueError:
...         print("Ooops !! That was not a valid number. Try again.")
...
Enter a number: 3/0
Traceback (most recent call last):
  File "<stdin>", line 3, in <module>
  File "<string>", line 1, in <module>
ZeroDivisionError: integer division or modulo by zero
```

注：我们把raw_input() 改为 input()

处理异常

- try/except 选项如下：

语句	解释
`except:`	捕获所有（或所有其他）异常类型
`except name:`	仅捕获特定异常
`except name as value:`	捕获列出的异常及其实例
`except (name1, name2):`	捕获列出的任何异常
`except (name1, name2) as value:`	捕获列出的任何异常及其实例
`else:`	如果未引发异常，则运行
`finally:`	始终执行语句

- 未来的课程将详细学习异常与程序的调试

Python 常用内置函数

- max()、min()、sum()这三个内置函数分别用于计算列表、元组或其他可迭代对象中所有元素最大值、最小值以及所有元素之和，sum()要求元素支持加法运算，max()和min()则要求序列或可迭代对象中的元素之间可比较大小。

```
>>> a = [72, 26, 80, 65, 34, 86, 19, 74, 52, 40]
>>> print(max(a), min(a), sum(a))
86 19 548
```

- 如果需要计算该列表中的所有元素的平均值，可以直接这样用：

```
>>> sum(a)*1.0/len(a)                    #Python 2.x
54.8
>>> sum(a)/len(a)                        #Python 3.x
54.8
```

Python 常用内置函数

- 内置函数max()和min()的key参数可以用来指定比较规则

```
>>> x = ['21', '1234', '9']
>>> max(x)              #字符串的比较是比较ASCII码值。
'9'
>>> max(x, key=len)
'1234'
>>> max(x, key=int)
'1234'
```

Python 常用内置函数

- enumerate()函数用来枚举可迭代对象中的元素，返回可迭代的enumerate对象，其中每个元素都是包含索引和值的元组。

```
>>> list(enumerate('abcd'))                          #枚举字符串中的元素
[(0, 'a'), (1, 'b'), (2, 'c'), (3, 'd')]

>>> list(enumerate(['Python', 'Greate']))            #枚举列表中的元素
[(0, 'Python'), (1, 'Greate')]

>>> for index, value in enumerate(range(10, 15)):  #枚举range对象中的元素
        print((index, value), end=' ')
(0, 10) (1, 11) (2, 12) (3, 13) (4, 14)
```

Python 常用内置函数

- zip()函数用来把多个可迭代对象中的元素压缩到一起，返回一个可迭代的zip对象，其中每个元素都是包含原来的多个可迭代对象对应位置上元素的元组（如同拉拉链一样）。

```
>>> list(zip('abcd', [1, 2, 3]))                #压缩字符串和列表
[('a', 1), ('b', 2), ('c', 3)]

>>> list(zip('123', 'abc', ',.!'))              #压缩3个序列
[('1', 'a', ','), ('2', 'b', '.'), ('3', 'c', '!')]

>>> x = zip('abcd', '1234')
>>> list(x)
[('a', '1'), ('b', '2'), ('c', '3'), ('d', '4')]
```

Python 常用内置函数

- sorted()对列表、元组、字典、集合或其他可迭代对象进行排序并返回新列表;
- reversed()对可迭代对象进行翻转（首尾交换）并返回可迭代的reversed对象。

```
>>> x=['aaaa','bc','d']
>>> sorted(x,key=len)                    #按长度排序
['d', 'bc', 'aaaa']

>>> reversed(x)                          #逆序，返回reversed对象
<list_reverseiterator object at 0x101e4d518>

>>> list(reversed(x))                    #reversed对象是可迭代的
['d', 'bc', 'aaaa']
```