返回目录

目录

返回目录

返回目录

1. 创建类：

```
1. class Student(object):
2. def __init__(self, name, score):
3.     self.name = name
4.     self.score = score
5.
6. bart = Student('Bart Simpson', 59)
```

类名首字母大写，基本类为 object, __init__方法的第一个参数永远是 self，表示创建的实例本身，在__init__方法内部，就可以把各种属性绑定到 self 有了__init__ 方法，在创建实例的时候，就不能传入空的参数了，必须传入与__init__ 方法匹配的参数，但 self 不需要传

2. 定义方法

在类的定义内：

```
1. def print_score(self):
2.     print '%s: %s' % (self.name, self.score)
```

要定义一个方法，除了第一个参数是 self 外，其它和普通函数一样

3. 内部变量与 private 变量(此处变量即属性)

私有变量/属性/方法：属性的名称前加上两个下划线_，就变成了一个私有变量（private）

特殊变量/属性/方法：变量名类似__xxx__的，以双下划线开头，以双下划线结尾的，是特殊变量，特殊变量是可以直接访问的，不是 private 变量

一个下划线开头的实例变量名，比如_name ，这样的实例变量外部可以访问，但按照约定俗成“请视为私有变量，不要随意访问”

4. 程序入口

if __name__ == '__main__'的意思是：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

5. super()函数/方法

super() 函数是用于调用父类(超类)的一个方法。单继承：super 与直接用类名调用父类方法无异；多继承，会涉及到查找顺序（MRO）、重复调用（钻石继承）等问题，此时应区分 super 与直接类名调用父类。

MRO 就是类的方法解析顺序表，其实也就是继承父类方法时的顺序表(见附录 1)

6. assert 断言

```
1. assert condition
```

用来让程序测试这个 condition，如果 condition 为 false，那么 raise 一个 AssertionError 出来。逻辑上等同于：

```
1. if not condition:
2.     raise AssertionError()
```

assert 断言语句添加异常参数，其实就是在断言表达式后添加字符串信息，用来解释断言并更好的知道是哪里出了问题。格式如下：

assert expression [, arguments]

assert 表达式 [, 参数]

```
1. >>> assert len(lists) >=5,'列表元素个数小于 5'
2.
3. Traceback (most recent call last):
4. File "D:/Data/Python/helloworld/helloworld.py", line 1, in <module>
5. assert 2>=5,'列表元素个数小于 5'
6. AssertionError: 列表元素个数小于 5
```

7. try...except…finally

```
1. a=10
2. b=0
3. try:
4.     print(a/b)
5. except:
6.     print("error")
7. finally:
8.     print("always excute")
```

先执行 try，如果出错执行 except，无论如何都执行 finally.

```
1. a=10
2. b=0
3. try:
4.     print(a/b)
5. except:
6.     print("error")
7. else:
8.     print("always excute")
```

先执行 try，如果出错执行 except，如果不出错执行 else

8. __str__与__repr__方法

__repr__：repr() 函数将对象转化为供解释器读取的形式，返回一个对象的 string 格式

__str__：str() 函数将对象转化为适于人阅读的形式，返回一个对象的 string 格式，

两个方法均用于返回对象供人阅读，__str__()用于显示给用户，而__repr__()用于显示给开发人员。print 调用的是__str__方法，直接输出实例调用的是__repr__方法

```
1. >>> class Student(object):
2. ... def __init__(self, name):
3. ... self.name = name
4. ...
5. >>> print(Student('Michael'))
6. <__main__.Student object at 0x109afb190>
7. >>> s = Student('Michael')
8. >>> s
9. <__main__.Student object at 0x109afb310>
```

9. __call__方法

当实例以函数形式被调用，其调用的即为__calll__方法

```
1. In [35]: class A:
2.    ....:     def __init__(self):
3.    ....:         print "init"
4.    ....:     def __call__(self):
5.    ....:         print "call"
6.
7. In [36]: a = A()
8. init
9.
10.   In [37]: a()
11.   call
```

10.@property

提供如下功能：有没有既能检查参数，又可以用类似属性这样简单的方式来访问类的变量(即把方法变成属性)

```
1. class Student(object):
```

```
2.
3.     def get_score(self):
4.         return self._score
5.
6.     def set_score(self, value):
7.         if not isinstance(value, int):
8.             raise ValueError('score must be an integer!')
9.         if value < 0 or value > 100:
10.                raise ValueError('score must between 0 ~ 100!')
11.            self._score = value
12.
13.    >>> s = Student()
14.    >>> s.set_score(60) # ok!
15.    >>> s.get_score()
16.    60
17.    >>> s.set_score(9999)
18.    Traceback (most recent call last):
19.      ...
20.    ValueError: score must between 0 ~ 100!
21.    #  通过如下方式解决上述问题
22.    class Student(object):
23.
24.        @property
25.        def score(self):
26.            return self._score
27.
28.        @score.setter
29.        def score(self, value):
30.            if not isinstance(value, int):
31.                raise ValueError('score must be an integer!')
32.            if value < 0 or value > 100:
33.                raise ValueError('score must between 0 ~ 100!')
34.            self._score = value
35.    >>> s = Student()
36.    >>> s.score = 60 # OK，实际转化为 s.set_score(60)
37.    >>> s.score # OK，实际转化为 s.get_score()
```

```
60
>>> s.score = 9999
Traceback (most recent call last):
  ...
ValueError: score must between 0 ~ 100!
```

11.@staticmethod

参考： https://stackoverflow.com/questions/40834145/whats-the-point-of-staticmethod-in-python

- instance methods: require the instance as the first argument
- class methods: require the class as the first argument
- static methods: require neither as the first argument

```
class TestClass:

    weight = 200                        # class attr

    def __init__(self, size):
        self.size = size                 # instance attr

    def instance_mthd(self, val):
        print("Instance method, with 'self':", self.size*val)

    @classmethod
    def class_mthd(cls, val):
        print("Class method, with `cls`:", cls.weight*val)

    @staticmethod
    def static_mthd(val):
        print("Static method, with neither args:", val)

a = TestClass(1000)

a.instance_mthd(2)
# Instance method, with 'self': 2000

TestClass.class_mthd(2)
```

```
25.    # Class method, with `cls`: 400
26.
27.    a.static_mthd(2)
28.    # Static method, with neither args: 2
```

12.查找模块下的方法(函数)、属性

均在 python 解释器下查询，详见附录 2.查询模块的帮助文档

13.Import 与 from…import…的区别

import datetime 是引入整个 datetime 包，如果使用 datetime 包中的 datetime 类,需要加上模块名的限定；from datetime import datetime 是只引入 datetime 包里的 datetime 类,在使用时无需添加模块名的限定

14.Tuple 与()的使用区别

tuple(seq)，seq -- 要转换为元组的序列。用法如下：

```
1. aList = [123, 'xyz', 'zara', 'abc'];
2. aTuple = tuple(aList)
3. print "Tuple elements : ", aTuple
4.
5. >>>Tuple elements :  (123, 'xyz', 'zara', 'abc')
```

()则直接使用：

```
1. aTuple = (123, 'xyz', 'zara', 'abc')
```

15.浅拷贝与深拷贝

16.Python 时间格式化输出

strftime()函数接收以时间元组(struct_time 对象)，并返回以可读字符串表示的当地时间，格式由参数 format 决定

```
1. t = time.time()//获得以秒为单位的时间
2. print(time.strftime("%b %d %Y %H:%M:%S", time.gmtime(t)))//gmtime 获得
   struct_time 对象
```

格式参数 format，详见附录 3.python 中时间日期格式化符号

17.格式化输出

格式输出详见参考文档 《python 的格式化输出》

18.Windows 环境下文件路径表示

Python 代码里面，反斜杠“\”是转义符，例如“\n”表示回车，采用以下三种方式表示路径：

- 斜杠 “/”，如“c:/test.txt”
- 两个反斜杠 “\\”，如“c:\\test.txt”
- 字符串前面加上字母 r，表示后面是一个原始字符串 raw string，如“r“c:\\test.txt””

19.Python 接收命令行参数

导入 argv，结果即为参数列表

```
1. from sys import argv
2. print(argv)
3. >>>python xx.py xxx
4. >>>['xx.py','xxx']
```

20.'\u'前缀字符串

\u4f60 十六进制代表对应汉字的 utf-16 编码

21.定义 1 个元素的 tuple

定义 1 个元素的 tuple: t = (1,)，加上一个逗号，避免成为数学意义上的括号

22.创建生成器(generator)的两种方式

方式 1: g = (x * x for x in range(10))，列表生成式的[]更改为()

方式 2: 如果一个函数定义中包含 yield 关键字，那么函数就不再是一个普通函数，而是一个 generator，函数是顺序执行，遇到 return 语句或者最后一行函数语句返回。而 generator 在每次调用 next()的时候执行，遇到 yield 语句返回，再次执行时从上次返回的 yield 语句处继续执行

```
1. def fib(max):
2.     n, a, b = 0, 0, 1
3.     while n < max:
4.         yield b
5.         a, b = b, a + b
6.         n = n + 1
7.     return 'done'
```

23.python 代码规范

请参考《python 代码规范》

24.迭代器总结

- 凡是可作用于 for 循环的对象都是 Iterable 类型；
- 凡是可作用于 next()函数的对象都是 Iterator 类型，它们表示一个惰性计算的序列；
- 集合数据类型如 list、dict、str 等是 Iterable 但不是 Iterator，不过可以通过 iter()函数获得一个 Iterator 对象。
- Python 的 for 循环本质上就是通过不断调用 next()函数实现的

25.Map 函数总结

map()函数接收两个参数，一个是函数，一个是 Iterable，map 将传入的函数依次作用到序列的每个元素，返回新的 Iterator。

```
map(function, iterable1,iterable2 ...)
```

例如三个列表相乘：

```
1. list1 = [1,2,3,4,5,6,7,8,9]
2. list2 = [1,2,3,4,5,6,7,8,9]
3. list3 = [9,8,7,6,5,4,3,2,1]
4. def foo(l1,l2,l3):
5.     return l1*l2*l3
6.
7. print(list(map(foo,list1,list2,list3)))
8. >>>[9,32,63,96,125,144,147,128,81]
```

26.Reduce 函数总结

reduce 把一个函数作用在一个序列[x1, x2, x3, ...]上，这个函数必须接收两个参数，reduce 把结果继续和序列的下一个元素做累积计算。

例如序列求和：

```
1. >>> from functools import reduce
2. >>> def add(x, y):
3. ...     return x + y
4. ...
5. >>> reduce(add, [1, 3, 5, 7, 9])
6. 25
```

27.匿名函数(lambda)总结

lambda parameters: expression

parameters：可选，如果提供，通常是逗号分隔的变量表达式形式，即位置参数。

expression：**不能包含分支或循环（但允许条件表达式）**，也不能包含 return（或 yield）函数。如果为元组，则应用圆括号将其包含起来。调用 lambda 函数，**返回的结果是对表达式计算产生的结果**

```
1. #根据参数是否为 1 决定 s 为 yes 还是 no
2. >>> s = lambda x:"yes" if x==1 else "no"
```

28.全局变量的使用

使用到的全局变量只是作为引用，不在函数中修改它的值的话，不需要加 global 关键字

```
1. a = 1
2.
3. def func():
4.     if a == 1:
5.         print("a: %d" %a)
```

使用到的全局变量，需要在函数中修改的话，就涉及到歧义问题，因此，需要修改全局变量 a，可以在"a = 2"之前加入 global a 声明

```
1. a = 1
2.
3. def func():
4.     global a
5.     a = 2
6.     print("in func a:", a)
```

29.python 定义类时，内部方法的互相调用

每次调用内部的方法时，方法前面加 self

```
1. class MyClass:
2.     def __init__(self):
3.         pass
4.     def func1(self):
5.         print('a')
6.         self.common_func()
7.      def func2(self):
8.         self.common_func()
```

```
9.
10.     def common_func(self):
11.         pass
```

30.filter 函数总结

filter()也接收一个函数和一个序列。filter()把传入的函数依次作用于每个元素，然后根据返回值是 True 还是 False 决定保留还是丢弃该元素

31.sorted 函数总结

sorted()函数也是一个高阶函数，它还可以接收一个 key 函数来实现自定义的排序，key 指定的函数将作用于 list 的每一个元素上，并根据 key 函数返回的结果进行排序

```
1. >>> sorted([36, 5, -12, 9, -21], key=abs)
2. [5, 9, -12, -21, 36]
```

32.返回函数(闭包)

33.装饰器

这种在代码运行期间动态增加功能的方式，称之为“装饰器”（Decorator）。本质上，decorator 就是一个返回函数的高阶函数。

```
1. def log(func):
2.     def wrapper(*args, **kw):
3.         print('call %s():' % func.__name__)
4.         return func(*args, **kw)
5.     return wrapper
```

如果将上述的 log 函数作为装饰器，则在其装饰的函数前添加@log

```
1. @log
2. def now():
3.     print('2015-3-25')
4. >>> now()
5. call now():
6. 2015-3-25
```

装饰器原理及引入机制较为复杂，详情可参考《python 装饰器解释》

34.函数参数

- 必选参数(位置参数):
- 默认参数：如 def power(x, n=2)，x 即为位置参数，n 为默认参数，必选参数在前，默认参数在后
- 可变参数：传入的参数个数是可变的，不必自行将所有参数组装成一个 list 或 tuple，如 def calc(*numbers)。定义可变参数和定义一个 list 或 tuple 参数相比，仅仅在参数前面加了一个*号。在函数内部，参数 numbers 接收到的是一个 tuple

```
1. def calc(*numbers):
2.     sum = 0
3.     for n in numbers:
4.         sum = sum + n * n
5.     return sum
```

- 关键字参数:可变参数在函数调用时自动组装为一个 tuple。而关键字参数允许你传入 0 个或任意个含参数名的参数，这些关键字参数在函数内部自动组装为一个 dict:

```
1. def person(name, age, **kw):
2.     print('name:', name, 'age:', age, 'other:', kw)
3.
4. >>> person('Michael', 30)
5. name: Michael age: 30 other: {}
6. >>> person('Bob', 35, city='Beijing')
7. name: Bob age: 35 other: {'city': 'Beijing'}
```

35.字符串里面的引号

单引号''定义字符串的时候，它就会认为你字符串里面的双引号" "是普通字符，从而不需要转义。反之当你用双引号定义字符串的时候，就会认为你字符串里面的单引号是普通字符无需转义

```
1. Str1 = "We all know that 'A' and 'B' are two capital letters."
2. Str2 = 'The teacher said: "Practice makes perfect" is a very famous
   proverb.'
```

36.偏函数

functools.partial 的作用就是，把一个函数的某些参数给固定住（也就是设置默认值），返回一个新的函数，调用这个新函数会更简单

```
1. >>> import functools
2. >>> int2 = functools.partial(int, base=2)
3. >>> int2('1000000')
```

```
4.  64
```

37.算法速度的定义

- 大 O：T(N) = O(f(N))，T 增长率小于等于 f
- Ω：T(N) = Ω (f(N))，T 增长率大于 f
- ⊖：T(N) = ⊖ (f(N))，T 增长率等于 f
- 小 o：T(N) =o (f(N))，T 增长率小于 f

38.计算运行时间的一般法则

- 法则 1：for 循环：一个 for 循环的运行时间至多是该循环内语句的运行时间乘以迭代次数
- 法则 2：嵌套 for 循环：嵌套循环内部的一条语句总运行时间为该语句运行时间乘以所有 for 循环的大小，如下程序片段运行时间为 $O(N^2)$

```
1. for i in range(N):
2.     for j in range(N):
3.         j += 1
```

- 法则 3：顺序语句：各个语句运行时间求和
- 法则 4：if/else 语句：运行时间至多是判断时间+S1 或 S2 中运行时间长者

```
1. if (condition):
2.     S1
3. else:
4.     S2
```

39.导入自定义模块

40.子类继承父类的属性问题

更加细致的继承中的属性和方法问题，请参考《python 类的继承、属性总结和方法总结》

如果子类自己定义了__init__方法，那么父类的属性是不能调用的，如下：

```
1. class Animal:
2.     def __init__(self):
3.         self.a = 'aaa'
4.
```

```
5. class Cat(Animal):
6.     def __init__(self):
7.         pass
8.
9. cat = Cat()
10.    print(cat.a)
11.
12.    >>>AttributeError: 'Cat' object has no attribute 'a'
```

可以在子类的 __init__中调用一下父类的 __init__ 方法,这样就可以调用父类的属性

```
1. class Animal:
2.     def __init__(self):
3.         self.a = 'aaa'
4.
5. class Cat(Animal):
6.     def __init__(self):
7.         super().__init__()
8.
9. cat = Cat()
10.    print(cat.a)
11.
12.    >>>aaa
```

41.子类扩展父类属性

42.list 逆序

- list.reverse()会直接在原来的列表里面将元素进行逆序排列，不需要创建新的副本用于存储结果，调用 list.reverse() 的返回值是 None
- 使用切片 [::-1]，mylist[start:end:step]：上面的操作表示取 mylist 的第 start 个（列表索引从 0 开始）到第 end 个元素（不包括第 end 个），其中每隔 step 个（默认 1 ）取一个。

```
>>> mylist
[1, 2, 3, 4, 5]
>>> mylist[-1:-3:1]
[]
>>> mylist[-3:-1:1]
[3, 4]
>>> mylist[-1:-3:-1]
[5, 4]
>>> mylist[-1:-3:-2]
[5]
```

- 使用 reversed() 方法，reversed 方法会将列表逆序的结果存储到迭代器里面，这种方式不会改变原来的列表，也不会创建原来列表的完整副本

43.i = false 与 i == 0

在 python3 中，如果将 i 赋值 False，则 i == 0 为 True

44.多进程

1. Linux/类 UNIX 下创建多进程(fork()系统调用)

```
1. import os
2.
3. print('Process (%s) start...' % os.getpid())
4. pid = os.fork()
5. if pid == 0:
6.     print('I am child process (%s) and my parent is %s.' %
   (os.getpid(), os.getppid()))
7. else:
8.     print('I (%s) just created a child process (%s).' % (os.getpid(),
   pid))
9. >>>Process (876) start...
10.    >>>I (876) just created a child process (877).
11.    >>>I am child process (877) and my parent is 876.
```

普通的函数调用，调用一次，返回一次，但是 fork()调用一次，返回两次，因为操作系统自动把当前进程（称为父进程）复制了一份（称为子进程），然后分别在父进程和子进程内返回。子进程永远返回 0，而父进程返回子进程的 ID.

2. 跨平台的多进程模块(multiprocessing)创建多进程

返回目录

```
1. from multiprocessing import Process
2. import os
3.
4. def run_proc(name):
5.     print('Run child process %s (%s)...' % (name, os.getpid()))
6.
7. if __name__=='__main__':
8.     print('Parent process %s.' % os.getpid())
9.     p = Process(target=run_proc, args=('test',))
10.       print('Child process will start.')
11.       p.start()
12.       p.join()
13.       print('Child process end.')
```

multiprocessing 模块提供了一个 Process 类来代表一个进程对象，创建子进程时，只需要传入一个执行函数和函数的参数，创建一个 Process 实例，用 start()方法启动，join()方法可以等待子进程结束后再继续往下运行，通常用于进程间的同步。

3. 创建子进程(此时子进程为外部进程，并非父进程的复制)暂未理解

4. 进程间通信

- Queue 方式(相当于创建了一个临时交换区)

Queue 有两个方法：A.Put 方法：以插入数据到队列中，两个可选参数：blocked 和 timeout. B.Get 方法：从队列读取并且删除一个元素。两个可选参数：blocked 和 timeout

```
1. #- * -coding: utf - 8 - * -
2. from multiprocessing import Process, Queue
3. import os, time, random
4.
5. # 写数据进程执行的代码:
6. def write(q):
7.     print('Process to write: %s' % os.getpid())
8.     for value in ['A', 'B', 'C']:
```

```
        print('Put %s to queue...' % value)
        q.put(value)
        time.sleep(random.random())

    # 读数据进程执行的代码:
    def read(q):
        print('Process to read: %s' % os.getpid())
        while True:
            value = q.get()
            print('Get %s from queue.' % value)

    if __name__ == '__main__': #父进程创建 Queue， 并传给各个子进程:
        q = Queue()
        pw = Process(target = write, args = (q, ))
        pr = Process(target = read, args = (q, ))# 启动子进程 pw， 写入:
        pw.start()# 启动子进程 pr， 读取:
        pr.start()# 等待 pw 结束:
        pw.join()# pr 进程里是死循环， 无法等待其结束， 只能强行终止:
        pr.terminate()
```

输出如下：

```
Process to read: 5836
Process to write: 6472
Put A to queue...
Put B to queue...
Get A from queue.
Put C to queue...
Get B from queue.
Get C from queue.
Process finished with exit code 0
```

- pipe 方式

```
#- * -coding: utf - 8 - * -
from multiprocessing import Process, Pipe

def f(conn):
  conn.send([42, None, 'hello'])
```

```
6.   while True:
7.       print(conn.recv())
8.
9. if __name__ == '__main__':
10.   parent_conn, child_conn = Pipe()
11.   p = Process(target = f, args = (child_conn, ))
12.   p.start()
13.   print parent_conn.recv()# prints "[42, None, 'hello']"
14.   parent_conn.send('666')
15.   p.terminate()
```

输出如下：

```
1. [42, None, 'hello']
2. 666
3.
4. Process finished with exit code 0
```

Pipe（）函数返回一对由管道连接的连接对象（Connection（conn1，conn2）对象），默认情况下是双工（双向）。Pipe（）返回的两个连接对象代表管道的两端。每个连接对象都有 send（）和 recv（）方法（等等）。conn1 发出(send)的消息传递到了 conn2 的接收端(recv)，反之亦然。

请注意，如果两个进程（或线程）尝试同时读取或写入管道的同一端，管道中的数据可能会损坏。当然，同时使用管道不同端的过程也不会有风险。如果 duplex 为 True（默认），则管道是双向的。如果 duplex 是 False，那么管道是单向的：conn1 只能用于接收消息，conn2 只能用于发送消息.

45.多线程

创建一个线程就是把一个函数传入并创建 Thread 实例，然后调用 start()开始执行：

```
1. import time, threading
2.
3. # 新线程执行的代码:
4. def loop():
5.     print('thread %s is running...' % threading.current_thread().name)
6.     n = 0
7.     while n < 5:
8.         n = n + 1
9.         print('thread %s >>> %s' % (threading.current_thread().name,
   n))
10.          time.sleep(1)
```

返回目录

```
        print('thread %s ended.' % threading.current_thread().name)

print('thread %s is running...' % threading.current_thread().name)
t = threading.Thread(target=loop, name='LoopThread')
t.start()
t.join()
print('thread %s ended.' % threading.current_thread().name)
```

Threadlocal

多进程模式的缺点是创建进程的代价大，另外，操作系统能同时运行的进程数也是有限的(受内存和 CPU 的限制)

多线程模式通常比多进程快一点，致命的缺点就是任何一个线程挂掉都可能直接造成整个进程崩溃，因为所有线程共享进程的内存.

返回目录

46.附录

5. 方法解析顺序（Method Resolution Order, MRO）列表

MRO 列表的顺序遵循以下三条原则：

- 子类永远在父类前面
- 如果有多个父类，会根据它们在列表中的顺序被检查
- 如果对下一个类存在两个合法的选择，选择第一个父类

6. 查询模块的帮助文档

- 先导入模块，再查询普通模块的使用方法：help(module_name)，例：help(math)
- 先导入 sys，再查询系统内置模块的使用方法：sys.bultin_modulenames
- 查看模块下所有函数：dir(module_name)，例：dir(math)
- 查看模块下特定函数：help(module_name.func_name)，例：help(math.sin)
- 查看函数信息的另一种方法：print(func_name.__doc__)，例：print(sin.__doc__)
- 查看 python 关键字：help('keywords')

7. python 中时间日期格式化符号

- %y 两位数的年份表示（00-99）
- %Y 四位数的年份表示（000-9999）
- %m 月份（01-12）
- %d 月内中的一天（0-31）
- %H 24 小时制小时数（0-23）

返回目录

- %I 12 小时制小时数（01-12）
- %M 分钟数（00=59）
- %S 秒（00-59）
- %a 本地简化星期名称
- %A 本地完整星期名称
- %b 本地简化的月份名称
- %B 本地完整的月份名称
- %c 本地相应的日期表示和时间表示
- %j 年内的一天（001-366）
- %p 本地 A.M.或 P.M.的等价符
- %U 一年中的星期数（00-53）星期天为星期的开始
- %w 星期（0-6），星期天为星期的开始
- %W 一年中的星期数（00-53）星期一为星期的开始
- %x 本地相应的日期表示
- %X 本地相应的时间表示
- %Z 当前时区的名称
- %% %号本身