

目录

Python教程②

正则表达式

限定符

*
+
?
{}

或运算符

字符类

元字符

贪婪、懒惰匹配

函数

函数的定义

函数的调用

默认/关键字参数
不定长参数

File方法

数据结构

栈 (Stack)

队列 (Queue)

面向对象

类属性与方法

构造函数

类的继承

方法重写
运算符重载

正则表达式

正则表达式（Regular Expression，简称regex或regexp）

是一种文本模式描述的工具，它使用单个字符串来描述、匹配一系列符合某个句法规则的字符串。

正则表达式在文本搜索、文本替换、字符串验证等多种场景中都有广泛的应用。

```
var str = "abc123def";  
var patt1 = /[0-9]+/;  
document.write(str.match(patt1));
```

以下标记的文本是获得的匹配的表达式：

123

从大量数据中获取我们想要的部分

限定符 (Quantifier)

a* a出现0次或多次

a+ a出现1次或多次

a? a出现0次或1次

a{6} a出现6次

a{2,6} a出现2-6次

a{2,} a出现两次以上

或运算符 (OR Operator)

(a|b) 匹配a或者b

(ab)|(cd) 匹配ab或者cd

字符类 (Character Classes)

[abc] 匹配a或者b或者c

[a-c] 同上

[a-zA-Z0-9] 匹配小写+大写英文字符以及数字

[^0-9] 匹配非数字字符

元字符 (Meta-characters)

\d 匹配数字字符

\D 匹配非数字字符

\w 匹配单词字符(英文、数字、下划线)

\W 匹配非单词字符

\s 匹配空白符(包含换行符、Tab)

\S 匹配非空白字符

. 匹配任意字符(换行符除外)

\bword\b \b标注字符的边界 (全字匹配)

^ 匹配行首

\$ 匹配行尾

贪婪/懒惰匹配 (Greedy / Lazy Match)

<.+> 默认贪婪匹配 “任意字符”

<.+?> 懒惰匹配 “任意字符”

在下面情况下

https://www.LIN.com

限定符 (Quantifier)

a* a出现0次或多次

a+ a出现1次或多次

a? a出现0次或1次

a{6} a出现6次

a{2,6} a出现2-6次

a{2,} a出现两次以上

REGULAR EXPRESSION

:/ a*bcd

TEST STRING

abcd

aaaabcd

aabcd

bcd

REGULAR EXPRESSION

:/ a?bcd

TEST STRING

abcd

aaaabcd

aabcd

bcd

REGULAR EXPRESSION

:/ a{4}bcd

TEST STRING

abcd

aaaabcd

aabcd

bcd

REGULAR EXPRESSION

:/ a+bcd

TEST STRING

abcd

aaaabcd

aabcd

bcd

或运算符 (OR Operator)

(a|b) 匹配a或者b

(ab)|(cd) 匹配ab或者cd

括号是表示捕获组,括号内会被作为一个整体捕获

REGULAR EXPRESSION

/ (a|b)cd

TEST STRING

acd

bcd

fcd

abef

cdef

acef

|

REGULAR EXPRESSION

/ ((ab)|(cd))ef

TEST STRING

acd

bcd

fcd

abef

cdef

acef

字符类 (Character Classes)

[abc] 匹配a或者b或者c

[a-c] 同上

[a-zA-F0-9] 匹配小写+大写英文字符以及数字

[^0-9] 匹配非数字字符

REGULAR EXPRESSION	MATCH INFORMATION															
<code>/ [a-z]</code>	<table><tr><td>Match 1</td><td>0-1</td><td>a</td></tr><tr><td>Match 2</td><td>1-2</td><td>b</td></tr><tr><td>Match 3</td><td>2-3</td><td>c</td></tr><tr><td>Match 4</td><td>12-13</td><td>b</td></tr><tr><td>Match 5</td><td>13-14</td><td>c</td></tr></table>	Match 1	0-1	a	Match 2	1-2	b	Match 3	2-3	c	Match 4	12-13	b	Match 5	13-14	c
Match 1	0-1	a														
Match 2	1-2	b														
Match 3	2-3	c														
Match 4	12-13	b														
Match 5	13-14	c														
TEST STRING																
abc234 123 Abc																

REGULAR EXPRESSION	MATCH INFORMATION																		
<code>/ [1-9]</code>	<table><tr><td>Match 1</td><td>3-4</td><td>2</td></tr><tr><td>Match 2</td><td>4-5</td><td>3</td></tr><tr><td>Match 3</td><td>5-6</td><td>4</td></tr><tr><td>Match 4</td><td>7-8</td><td>1</td></tr><tr><td>Match 5</td><td>8-9</td><td>2</td></tr><tr><td>Match 6</td><td>9-10</td><td>3</td></tr></table>	Match 1	3-4	2	Match 2	4-5	3	Match 3	5-6	4	Match 4	7-8	1	Match 5	8-9	2	Match 6	9-10	3
Match 1	3-4	2																	
Match 2	4-5	3																	
Match 3	5-6	4																	
Match 4	7-8	1																	
Match 5	8-9	2																	
Match 6	9-10	3																	
TEST STRING																			
abc234 123 Abc																			

元字符 (Meta-characters)

- \d 匹配数字字符
- \D 匹配非数字字符
- \w 匹配单词字符(英文、数字、下划线)
- \W 匹配非单词字符
- \s 匹配空白符(包含换行符、Tab)
- \S 匹配非空白字符
- . 匹配任意字符(换行符除外)
- \bword\b \b标注字符的边界 (全字匹配)
- ^ 匹配行首
- \$ 匹配行尾

REGULAR EXPRESSION

:: / \D{3}\d{3}

TEST STRING

abc234

123

Abc

REGULAR EXPRESSION

:: / .{3}\d{3}

TEST STRING

abc234

123234

123

Abc

REGULAR EXPRESSION

:: / ^abc

TEST STRING

abc234

234abc

123

Abc

REGULAR EXPRESSION

:: / abc\$

TEST STRING

abc234

234abc

123

Abc

贪婪/懒惰匹配 (Greedy / Lazy Match)

<.+> 默认贪婪匹配“任意字符”
<.+?> 懒惰匹配“任意字符” 在下面情况下

https://www.LIN.com

默认情况下，正则表达式中的量词是贪婪的，这意味着它们会尽可能多地匹配字符。贪婪匹配会从左到右尽可能多地“吞噬”字符。

对于字符串 <a>，懒惰匹配会匹配 <a> 和 ，因为每个 *? 都会尽可能少地匹配字符，直到遇到下一个 >。

REGULAR EXPRESSION

:/ <.+>

TEST STRING

http...

REGULAR EXPRESSION

:/ <.+?>

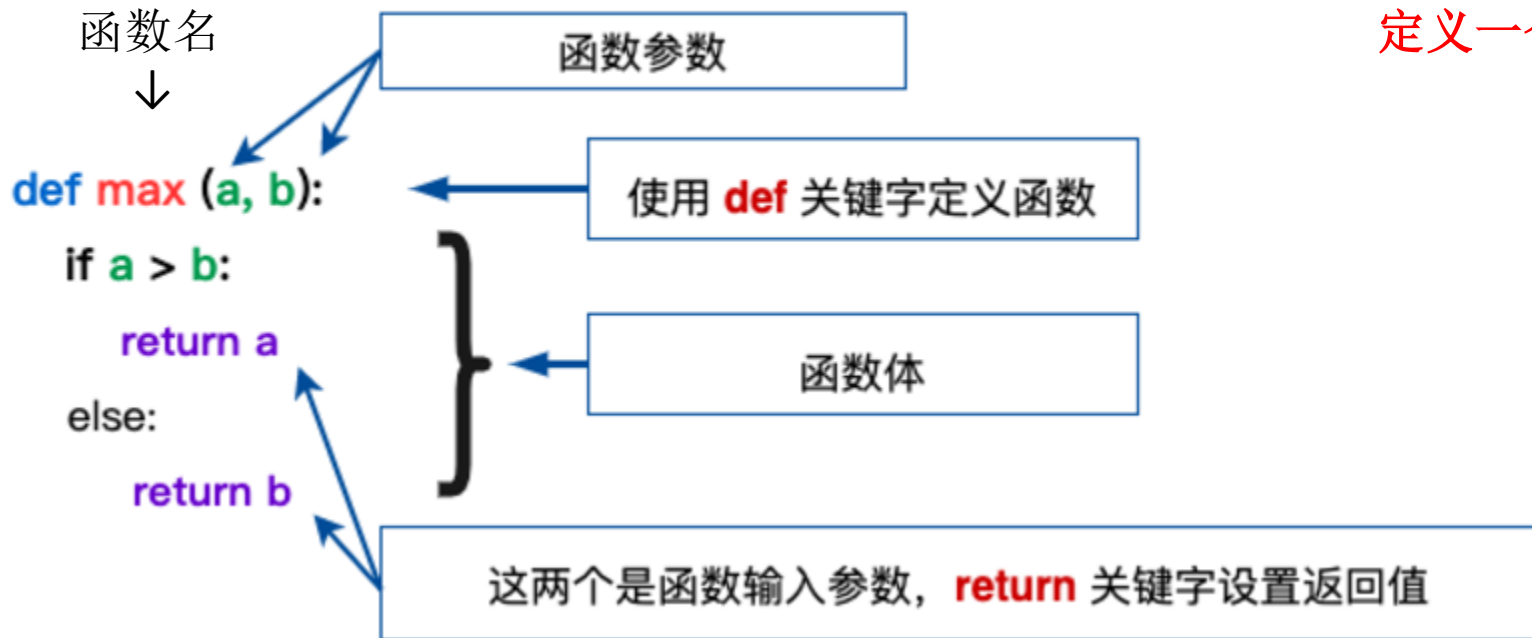
TEST STRING

http...

函数

函数可以输入参数，并设置返回值

定义一个函数



函数调用

```
#!/usr/bin/python3

# 定义函数
def printme( str ):
    # 打印任何传入的字符串
    print (str)
    return

# 调用函数
printme("我要调用用户自定义函数!")
printme("再次调用同一函数")
```

参数传递

```
1 #!/usr/bin/python3
2
3 # 可写函数说明
4 def printinfo(name, age=20): 4 用法
5     "打印任何传入的字符串"
6     print("名字: ", name)
7     print("年龄: ", age)
8     return
9
10
11 # 调用printinfo函数
12 printinfo( name: "YJL", age: 20)
13 print("-----")
14
15 printinfo( name: 20, age: "YJL")
16 print("-----")
17
18 printinfo(age=20, name="YJL")
19 print("-----")
20
21 printinfo(name="YJL")
```

E:\Anaconda\envs\py38\python.exe C:\Users\1\Desktop\人工智能备课\习题\test.py

名字: YJL

年龄: 20

名字: 20

年龄: YJL

名字: YJL

年龄: 20

名字: YJL

年龄: 20

进程已结束, 退出代码为 0

默认/关键字参数传递

加了星号 * 的参数会以元组(tuple)的形式导入，存放所有未命名的变量参数。

不定长参数

```
#!/usr/bin/python3

# 可写函数说明
def printinfo( arg1, *vartuple ):
    "打印任何传入的参数"
    print ("输出: ")
    print (arg1)
    print (vartuple)

# 调用printinfo 函数
printinfo( 70, 60, 50 )
```

输出:

```
70
(60, 50)
```

加了两个星号 ****** 的参数会以字典的形式导入

```
#!/usr/bin/python3

# 可写函数说明
def printinfo( arg1, **vardict ):
    "打印任何传入的参数"
    print ("输出: ")
    print (arg1)
    print (vardict)

# 调用printinfo 函数
printinfo(1, a=2,b=3)
```

输出:

1

{'a': 2, 'b': 3}

File方法

```
open(file, mode='r', buffering=-1, encoding=None, errors=None, newline=None, closefd=True, opener=None)
```

file: 必需，文件路径（相对或者绝对路径）。

mode: 可选，文件打开模式

buffering: 设置缓冲

encoding: 一般使用utf8

errors: 报错级别

newline: 区分换行符

closefd: 传入的file参数类型

opener: 设置自定义开启器，开启器的返回值必须是一个打开的文件描述符。

模式	描述
t	文本模式 (默认)。
x	写模式, 新建一个文件, 如果该文件已存在则会报错。
b	二进制模式。
+	打开一个文件进行更新(可读可写)。
U	通用换行模式 (Python 3 不支持) 。
r	以只读方式打开文件。文件的指针将会放在文件的开头。这是默认模式。
rb	以二进制格式打开一个文件用于只读。文件指针将会放在文件的开头。这是默认模式。一般用于非文本文件如图片等。
r+	打开一个文件用于读写。文件指针将会放在文件的开头。
rb+	以二进制格式打开一个文件用于读写。文件指针将会放在文件的开头。一般用于非文本文件如图片等。
w	打开一个文件只用于写入。如果该文件已存在则打开文件, 并从开头开始编辑, 即原有内容会被删除。如果该文件不存在, 创建新文件。

序号	方法及描述		
1	<code>file.close()</code> 关闭文件。关闭后文件不能再进行读写操作。		
2	<code>file.flush()</code> 刷新文件内部缓冲，直接把内部缓冲区的数据立刻写入文件, 而不是被动的等待输出缓冲区写入。		
3	<code>file.fileno()</code> 返回一个整型的文件描述符(file descriptor FD 整型), 可以用在如os模块的read方法等一些底层操作上。		
4	<code>file.isatty()</code> 如果文件连接到一个终端设备返回 True，否则返回 False。	7	<code>file.readline([size])</code> 读取整行，包括 "\n" 字符。
5	<code>file.next()</code> Python 3 中的 File 对象不支持 next() 方法。 返回文件下一行。	8	<code>file.readlines([sizeint])</code> 读取所有行并返回列表，若给定sizeint>0，返回总和大约为sizeint字节的行, 实际读取值可能比 sizeint 较大, 因为需要填充缓冲区。
6	<code>file.read([size])</code> 从文件读取指定的字节数，如果未给定或为负则读取所有。	9	<code>file.seek(offset[, whence])</code> 移动文件读取指针到指定位置
		10	<code>file.tell()</code> 返回文件当前位置。
		11	<code>file.truncate([size])</code> 从文件的首行首字符开始截断，截断文件为 size 个字符，无 size 表示从当前位置截断；截断之后后面的所有字符被删除，其中 windows 系统下的换行代表2个字符大小。
		12	<code>file.write(str)</code> 将字符串写入文件，返回的是写入的字符长度。
		13	<code>file.writelines(sequence)</code> 向文件写入一个序列字符串列表，如果需要换行则要自己加入每行的换行符。

数据结构

列表 (List)

元组 (Tuple)

字典 (Dictionary)

集合 (Set)

数组 (Array)

栈 (Stack)

队列 (Queue)

NumPy数组

树 (Tree)

栈（Stack）

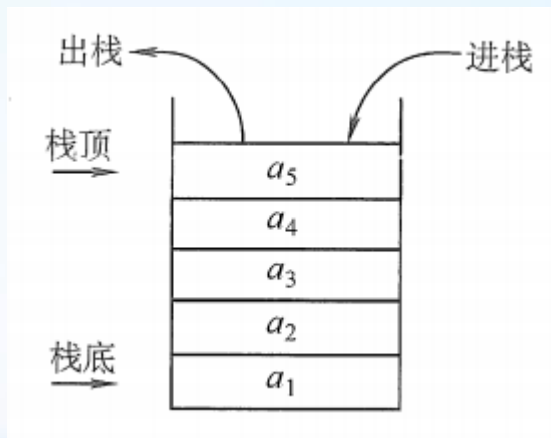
栈是一种遵循**后进先出**（LIFO, Last In First Out）原则的数据结构。

Push: 将一个元素添加到栈顶。

Pop: 移除栈顶元素，并返回它。

Peek/Top: 查看栈顶元素，但不移除它。

IsEmpty: 检查栈是否为空。



```
1 #创建一个空栈
2 stack = []
3
4 #压入 (Push) 操作
5 stack.append(1)
6 stack.append(2)
7 stack.append(3)
8 print(stack) # 输出: [1, 2, 3]
9
10 #弹出 (Pop) 操作
11 top_element = stack.pop()
12 print(top_element) # 输出: 3
13 print(stack)      # 输出: [1, 2]
14
15 #检查是否为空 (IsEmpty) |
16 is_empty = len(stack) == 0
17 print(is_empty) # 输出: False
18
19 #获取栈的大小 (Size)
20 size = len(stack)
21 print(size) # 输出: 2
```

使用列表 (list) 来实现栈的功能

```
E:\Anaconda\envs\py38\python
[1, 2, 3]
3
[1, 2]
False
2
```

进程已结束，退出代码为 0

队列（Queue）

队列是一种遵循**先进先出**（FIFO，First In First Out）原则的数据结构。

Enqueue: 在队列的一端（通常是尾部）添加一个元素。

Dequeue: 从队列的另一端（通常是头部）移除一个元素，并返回它。

Front: 查看队列头部的元素，但不移除它。

Rear: 查看队列尾部的元素，但不移除它。

IsEmpty: 检查队列是否为空。



使用列表时，如果频繁地在列表的开头插入或删除元素，性能会受到影响，因为这些操作的时间复杂度是 $O(n)$ 。为了解决这个问题，Python 提供了 **collections.deque**，它是双端队列，可以在两端高效地添加和删除元素。

```
1 from collections import deque
```

```
2  
3 # 创建一个空队列
```

```
4 queue = deque()
```

```
5  
6 # Enqueue: 在队列的尾部添加一个元素
```

```
7 queue.append('a')
```

```
8 queue.append('b')
```

```
9  
10 # Dequeue: 从队列的头部移除一个元素，并返回它
```

```
11 dequeued_item = queue.popleft()
```

```
12 print(dequeued_item)
```

```
13  
14 # Front: 查看队列头部的元素，但不移除它
```

```
15 front_item = queue[0]
```

```
16 print(front_item) # 输出: 'b'
```

```
    # Rear: 查看队列尾部的元素，但不移除它
```

```
    rear_item = queue[-1]
```

```
    print(rear_item) # 输出: 'b'
```

```
    # IsEmpty: 检查队列是否为空
```

```
    is_empty = len(queue) == 0
```

```
    print(is_empty) # 输出: False
```

```
E:\Anaconda\envs\py38\py
```

```
a
```

```
b
```

```
b
```

```
False
```

```
进程已结束，退出代码为 0
```

OOP面向对象

```
class Employee: 1 个用法
```

```
    '所有员工的基类'
```

```
    empCount = 0
```

```
    def __init__(self, name, salary):
```

```
        self.name = name
```

```
        self.salary = salary
```

```
emp1 = Employee(name="YIN", salary=111111111111111111);
```

```
#emp1是Employee的实例（对象）
```

类(Class): 用来描述具有相同的属性和方法的对象的集合。它定义了该集合中每个对象所共有的属性和方法。**对象是类的实例。**

```
#!/usr/bin/python
# -*- coding: UTF-8 -*-
```

```
class Employee:
    '所有员工的基类'
    empCount = 0

    def __init__(self, name, salary):
        self.name = name
        self.salary = salary
        Employee.empCount += 1

    def displayCount(self):
        print "Total Employee %d" % Employee.empCount

    def displayEmployee(self):
        print "Name : ", self.name, ", Salary: ", self.salary
```

创建类

使用 **class** 语句来创建一个新类，**class** 之后为类的名称并以冒号结尾：

```
#!/usr/bin/python
# -*- coding: UTF-8 -*-
```

```
class Employee:
    '所有员工的基类'
    empCount = 0
```

```
def __init__(self, name, salary):
    self.name = name
    self.salary = salary
    Employee.empCount += 1
```

```
def displayCount(self):
    print "Total Employee %d" % Employee.empCount
```

```
def displayEmployee(self):
    print "Name : ", self.name, ", Salary: ", self.salary
```

类属性与方法

empCount 变量是一个类变量，它的值将在这个类的所有实例之间共享。你可以在内部类或外部类使用 **Employee.empCount** 访问。

类的方法，也就是类中定义的函数

```
class JustCounter:
    __secretCount = 0 # 私有变量
    publicCount = 0   # 公开变量

    def count(self):
        self.__secretCount += 1
        self.publicCount += 1
        print self.__secretCount

counter = JustCounter()
counter.count()
counter.count()
print counter.publicCount
print counter.__secretCount # 报错，实例不能访问私有变量
```

类的私有方法

两个下划线开头，声明该方法为私有方法，不能在类的外部调用。

```
#!/usr/bin/python
# -*- coding: UTF-8 -*-
```

```
class Employee:
    '所有员工的基类'
    empCount = 0

    def __init__(self, name, salary):
        self.name = name
        self.salary = salary
        Employee.empCount += 1

    def displayCount(self):
        print "Total Employee %d" % Employee.empCount

    def displayEmployee(self):
        print "Name : ", self.name, ", Salary: ", self.salary
```

构造函数

`__init__()`方法被称为类的构造函数或初始化方法

类的**构造函数**是类的一种特殊的成员函数，它会在每次创建类的新对象时执行。

self代表类的实例

在实例方法中，**self** 用于访问实例变量和调用其他实例方法。它代表当前对象，使得我们可以在方法内部访问和修改对象的状态。

创建实例对象

"创建 Employee 类的第一个对象"

```
emp1 = Employee("Zara", 2000)
```

"创建 Employee 类的第二个对象"

```
emp2 = Employee("Manni", 5000)
```

访问属性/方法

```
emp1.displayEmployee()
```

```
emp2.displayEmployee()
```

```
print "Total Employee %d" % Employee.empCount
```

Python内置类属性

`__dict__`: 类的属性（包含一个字典，由类的数据属性组成）

`__doc__`: 类的文档字符串

`__name__`: 类名

`__module__`: 类定义所在的模块（类的全名是'`__main__.className`'，如果类位于一个导入模块`mymod`中，那么`className.__module__` 等于 `mymod`）

`__bases__`: 类的所有父类构成元素（包含了一个由所有父类组成的元组）

类的继承

通过继承创建的新类称为子类或派生类，被继承的类称为基类、父类或超类。

```
class 派生类名(基类名)
```

```
.....
```

```
class A:          # 定义类 A  
.....
```

```
class B:          # 定义类 B  
.....
```

```
class C(A, B):    # 继承类 A 和 B  
.....
```

```
1 class Parent: # 定义父类 1个用法
2     parentAttr = 100
3     def __init__(self):
4         print("调用父类构造函数")
5
6     def parentMethod(self): 1个用法
7         print('调用父类方法')
8
9 class Child(Parent): # 定义子类 1个用法
10     def __init__(self):
11         print("调用子类构造方法")
12
13     def childMethod(self):
14         print('调用子类方法')
15
16 c = Child()
17 c.parentMethod()
```

E:\Anaconda\envs\py38\python.exe C:\Users

调用子类构造方法

调用父类方法

进程已结束，退出代码为 0

方法重写

如果父类方法的功能不能满足需求，可以在子类重写父类的方法

```
1 class Parent: # 定义父类 1个用法
2     def myMethod(self):
3         print ('调用父类方法')
4
5
6 class Child(Parent): # 定义子类 1个用法
7     def myMethod(self): 1个用法
8         print ('调用子类方法')
9
10
11 c = Child() # 子类实例
12 c.myMethod() # 子类调用重写方法
13
```

E:\Anaconda\envs\py38\python

调用子类方法

进程已结束，退出代码为 0

运算符重载

```
class Vector: 3 用法
    def __init__(self, a, b):
        self.a = a
        self.b = b

    def __str__(self):
        return 'Vector (%d, %d)' % (self.a, self.b)

    def __add__(self, other):
        return Vector(self.a + other.a, self.b + other.b)
```

```
v1 = Vector(a: 2, b: 10)
v2 = Vector(a: 5, -2)
print (v1 + v2)
```

E:\Anaconda\envs\py38\python.exe
Vector (7, 8)

进程已结束，退出代码为 0