**Python学习笔记**

官网：https://www.python.org

# 1安装

## 1.1 window下安装python

|  |  |
| --- | --- |
| 下载和安装python | https://www.python.org/ftp/python/3.4.4/python-3.4.4.amd64.msi |
| 添加环境变量 | 把安装目录C:\Python34添加到环境变量，支持命令行 |
| 安装setuptools | 打开终端，输入命令安装setuptools  python -m pip install -U pip setuptools  安装完后使用命令easy\_install 包名 |
| 添加环境变量 | 把C:\Python34\Scripts添加到环境变量 |

安装第三方包的两种方法

1. 使用命令方式

例如安装django

打开终端，输入命令easy\_install django即可完成安装

1. 下载第三方包源码安装

下载django源码文件，然后解压，打开终端进入解压的目录

先测试一下是否能够安装

python setup.py build

注：如第三方包底层依赖c或c++时，window必须先安装编译器

安装第三方包到python环境中

python setup.py install

安装完第三方包存放在c:\Python34\lib\site-packages目录下

## 1.2 linux下安装python

|  |  |
| --- | --- |
| 下载python源码 | wget https://www.python.org/ftp/python/3.6.0/Python-3.6.0.tgz |
| 解压 | tar zxvf Python-3.6.0.tgz |
| 进入目录 | cd Python-3.6.0 |
| 配置目录 | ./configure --prefix=/usr/local/python |
| 编译和安装 | make && make install |
| 创建软链接 | ln -s /usr/local/python/bin/python /bin/python |
| 下载setup-tools | wget https://pypi.python.org/packages/e3/6b/bb793ca610f364d1888b9bd7b83dccd0c27d779e4fd91a953ce8bae74996/setuptools-34.0.2.zip#md5=8e33d59407b6714e4e97274da538d08f |
| 解压 | unzip setuptools-34.0.2.zip |
| 进入目录 | cd setuptools-34.0.2 |
| 编译 | python setup.py build |
| 安装 | python setup.py install |
| 创建软链接 | ln -s /usr/local/python/bin/easy\_install /bin/easy\_install |

# 2四个重要自醒函数

**type 类型**

|  |
| --- |
| *import* sys print(type(sys)) # 结果：<class 'module'> |

**dir 列出所有方法**

|  |
| --- |
| print(dir(str)) # 列出字符串所有方法 |

**help 帮助信息**

|  |
| --- |
| *import* sys help(sys) |

**str 以语言的方式输出字符串信息**

|  |
| --- |
| a=1 print(str(a)) #结果：'1' |

# 3 基本类型

python变量是在赋值的时候才决定

变量类型可以强制转换，例如整形和浮点型相互强制转换

|  |
| --- |
| a=1  b=float(a)  c=int(b) |

## 3.1 常量

在运行过程中一直是一个固定的值，定义时和其他语言有些区别

例如数字1就是表示常量，C语言常量表示const a=1

## 3.2 整形和长整形

|  |
| --- |
| #整形  a=1  type(a)  #长整形  a=100L #python2使用，在python3中，当数字大到一定程度自动转为长整形  type(a) |

## 3.3 浮点型

|  |
| --- |
| a=3.14  type(a) |

## 3.4 字符串

字符串表示有单引号(‘’),双引号(“”)，三单引号(’’’’’’)，三双引号(””””””)

|  |
| --- |
| a=**'hello'** print(a) b=**"hello"** print(b) c=**'hello "world"'** print(c) d=**"hello 'world'"** print(d) e=**'''hello'''** print(e) f=**"""hello"""** print(f) |

其中三单引号和三双引号没有赋值给变量是作为块注释用

|  |
| --- |
| **'''abc''' """abc"""** |

转义字符和C语言转义类似

|  |
| --- |
| a=**"hello** \n **world"** # 输出时会换行 print(a)  #输出原生字符串(按原样输出)，转义符无效 b=**r"hello \n world"** print(b) |

有中文时，尽量在前面加u，表示unicode

|  |
| --- |
| a=**u"中文"** print(a) |

字符的一些方法，用dir(str)查看

|  |
| --- |
| [**'\_\_add\_\_'**, **'\_\_class\_\_'**, **'\_\_contains\_\_'**, **'\_\_delattr\_\_'**, **'\_\_dir\_\_'**, **'\_\_doc\_\_'**, **'\_\_eq\_\_'**, **'\_\_format\_\_'**, **'\_\_ge\_\_'**, **'\_\_getattribute\_\_'**, **'\_\_getitem\_\_'**, **'\_\_getnewargs\_\_'**, **'\_\_gt\_\_'**, **'\_\_hash\_\_'**, **'\_\_init\_\_'**, **'\_\_init\_subclass\_\_'**, **'\_\_iter\_\_'**, **'\_\_le\_\_'**, **'\_\_len\_\_'**, **'\_\_lt\_\_'**, **'\_\_mod\_\_'**, **'\_\_mul\_\_'**, **'\_\_ne\_\_'**, **'\_\_new\_\_'**, **'\_\_reduce\_\_'**, **'\_\_reduce\_ex\_\_'**, **'\_\_repr\_\_'**, **'\_\_rmod\_\_'**, **'\_\_rmul\_\_'**, **'\_\_setattr\_\_'**, **'\_\_sizeof\_\_'**, **'\_\_str\_\_'**, **'\_\_subclasshook\_\_'**, **'capitalize'**, **'casefold'**, **'center'**, **'count'**, **'encode'**, **'endswith'**, **'expandtabs'**, **'find'**, **'format'**, **'format\_map'**, **'index'**, **'isalnum'**, **'isalpha'**, **'isdecimal'**, **'isdigit'**, **'isidentifier'**, **'islower'**, **'isnumeric'**, **'isprintable'**, **'isspace'**, **'istitle'**, **'isupper'**, **'join'**, **'ljust'**, **'lower'**, **'lstrip'**, **'maketrans'**, **'partition'**, **'replace'**, **'rfind'**, **'rindex'**, **'rjust'**, **'rpartition'**, **'rsplit'**, **'rstrip'**, **'split'**, **'splitlines'**, **'startswith'**, **'strip'**, **'swapcase'**, **'title'**, **'translate'**, **'upper'**, **'zfill'**] |

常用的count，join等用法

|  |
| --- |
| #可以用help来查看如何使用  a=”112345678900”  help(a.count)  help(a.join)  a.count('1',0,-1) #计算有多少个字符串1  ‘,’.join(a) #a的所有字符都以逗号隔开 |

内置函数eval，把字符的表达式转为计算结果

|  |
| --- |
| x=1  y='x+1'  print(eval(y))  结果：2  a="'1'+'1'" # 字符串  print(type(eval(a))) # 字符串类型  print(type(eval(eval(a)))) # 整形 |

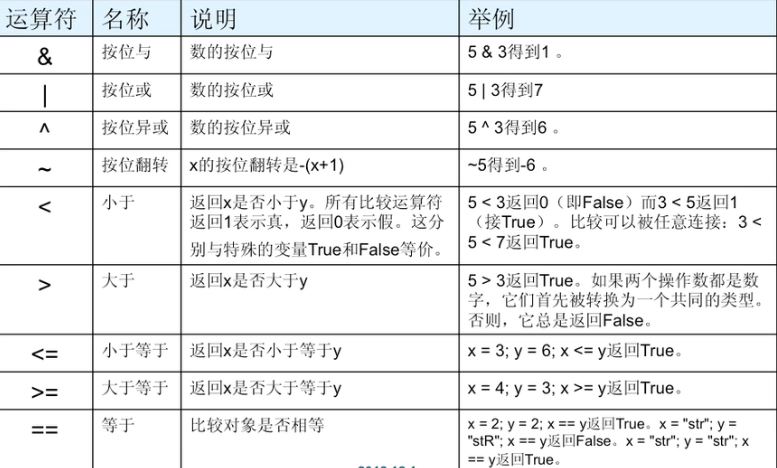
字符串复制

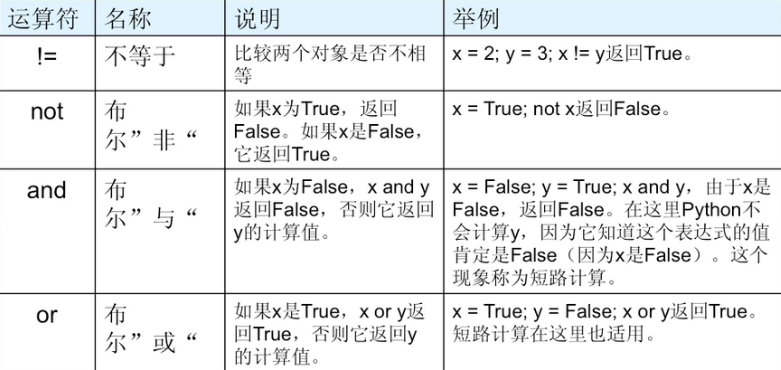
|  |
| --- |
| a=**'ABC'** print(a\*3) # 重复3次a的值 # 结果：ABCABCABC |

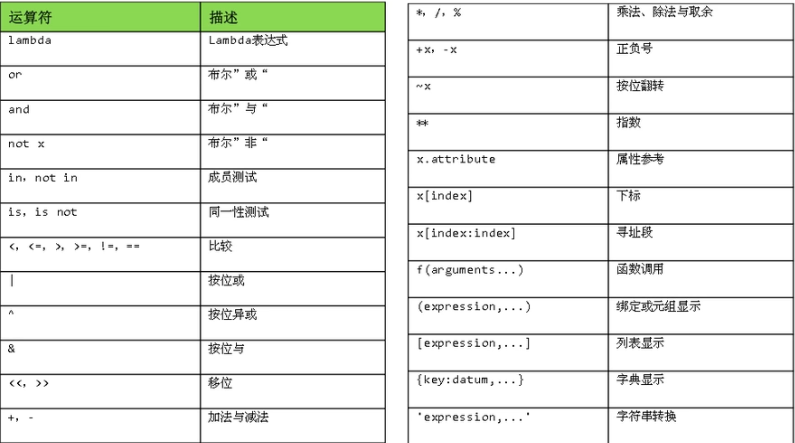
## 3.5 全局变量和局部变量

|  |
| --- |
| a = 1  *def* func1():  a = 5 # 局部变量  print(a)   *def* func2():  *global* a # 使用全局变量关键字  a = a + 10 # 全局变量  print(a)  *if* \_\_name\_\_ == **"\_\_main\_\_"**:  func1()  func2()  print(a) # 全局变量 |

# 4 运算符







# 5 控制结构

## 5.1 if 控制结构

|  |
| --- |
| a = 1 *if* a == 1:  print(a) *elif* a == 2:  print(a) *else*:  print(**'out of range'**)  a = *None if* a != *None*:  print(a) *else*:  print(**"a is none."**) |

## 5.2 for 循环控制结构

|  |
| --- |
| # 循环数字 *for* i *in* range(20): # 设置stop=20，默认start=0，step=1  print(i)  *for* j *in* range(5, 10): # start=5，stop=10，默认step=1  print(j)  *for* k *in* range(3, 30, 3): # start=3，stop=30，step=3  print(k)  # 遍历数组元素 people = (**"张三"**, **"male"**, 22) *for* v *in* (people):  print(v) |

## 5.3 while 循环控制结构

|  |
| --- |
| i = 0 *while* i < 20:  i += 1  *if* i % 2:  *continue* print(i) |

# 6 函数

## 6.1 常用函数

|  |
| --- |
| *def* f1(): # 不带参数函数  print(**"f1 function"**)  f1()  *def* f2(*num*): # 带参数函数  print(*num*)  f2(3)  *def* f3(*a=1*,*b=2*,*c=3*): # 带参数函数，当传参为空时，设置默认值  print(**"a ="**,*a*)  print(**"b ="**,*b*)  print(**"c ="**,*c*)  f3() f3(c=300)# 指定传哪个参数 f3(100) f3(100,200) f3(100,200,300)  # 带返回值 *def* f4(*x*):  *return* 3.14\**x*\**x* print(f4(5))  # 多个返回值 *def* f5(*length*,*width*):  *return* 2\*(*length*+*width*),*length*\**width* circle,area=f5(3,4) print(circle, area)  val=f5(3,4) # 多返回值函数返回给单个变量时，该变量为数组类型 print(type(val)) |

## 6.2 匿名函数lambda

|  |
| --- |
| fl = *lambda* x, y: x \* y # 参数x,y,返回结果x\*y print(fl(3, 4)) |

## 6.3 闭包

|  |
| --- |
| *def* func(*x*):  *def* f(*y*):  *return* x + *y  return* f  a = func(10) print(a(20)) |

# 7 面向对象编程

对象函数名字前有双下划线的函数属于该对象私有函数，不能被子类继承，例如def \_\_f1()

对象函数名字前后都有双下划线，表示函数为该对象专有函数

7.1 对象封装、实例化

|  |
| --- |
| *class* people: # 类定义  *def \_\_init\_\_*(self): # 析构函数，作用是在对象实例化的时候，对象的属性初始化  self.age=10 # 属性  self.gender=**"male"** self.weight=30  *def* sayHello(self): # 参数必须传参self  print(**"hello, this is people object."**)  p=people() # 对象实例化 print(p.age,p.gender,p.weight) p.sayHello() # 对象方法 |

7.2 对象继承

|  |
| --- |
| *class* Rectangle:# 定义长方形类  *def \_\_init\_\_*(self): # 初始化  self.length = 3  self.width = 4   *def* area(self): # 计算面积方法  print(self.length \* self.width)   *class* Square(Rectangle): # 定义正方形类，继承长方形类  *def \_\_init\_\_*(self):  self.length = 5  self.width = 5  *def* circumference(self): # 计算周长方法  print((self.length + self.width) \* 2)   s = Square() s.area() s.circumference() |

# 8 包导入

包文件路径./peopleClass/people.py,内容如下

|  |
| --- |
| *class* people: # 类定义  *def \_\_init\_\_*(self): # 析构函数，作用是在对象实例化的时候，对象的属性初始化  self.age=10 # 属性  self.gender=**"male"** self.weight=30  *def* sayHello(self): # 参数必须传参self  print(**"hello, this is people object."**) |

另一个文件需要调用people.py文件

|  |
| --- |
| *from* peopleClass *import* people  p = people.people() p.sayHello()  #或者只导入指定使用的对象  *from* peopleClass.people *import* people  p = people() p.sayHello() |

注：(1)在调用包下面新建一个\_\_init\_\_.py文件，当调用包时，会先执行\_\_init\_\_.py里面的代码后在再调用包的代码

(2) 在.py文件开始位置添加指定编码格式(#coding=utf-8)，避免中文时造成乱码情况

# 9 文件和文件夹操作

## 9.1 读或写文件

|  |
| --- |
| f=open(**"./a.txt"**,**"r"**) # 以只读方式打开文件 print(f.read()) # 读取所有内容 f.close()  f2=open(**"./b.txt"**,**"a"**) # 以读和写方式打开文件，不存在则创建 f2.write(**"hello world"**) # 写入内容 f2.close() |

## 9.2 按行读取内容

|  |
| --- |
| # 方式1 f = open(**"a.txt"**,**"r"**) lines = f.readlines() #读取全部内容 *for* line *in* lines:  print(line) f.close()  # 方式2 *for* line *in* open(**"a.txt"**):  print(line)  # 方式3 f = open(**"a.txt"**) # 返回一个文件对象 line = **" "** *while* line:  line = f.readline() # 调用文件的 readline()方法 f.close() |

## 9.3 遍历文件夹

|  |
| --- |
| # 只遍历一级目录 *import* os  # s = os.sep # 判断平台路径分割符号 # root = "d:" + s + "ll" + s root = **"."** *for* i *in* os.listdir(root):  *if* os.path.isfile(os.path.join(root, i)):  print(i) # 只输出文件名  print(os.path.join(root, i)) # 输出路径和文件名 |

|  |
| --- |
| # 遍历所有目录下的所有文件  *import* os *def* traverseFiles(*path*):  path = os.path.expanduser(*path*)  *for* (dirname, subdir, subfile) *in* os.walk(*path*):  # print('dirname is %s, subdir is %s, subfile is %s' % (dirname, subdir, subfile))  print(**'['** + dirname + **']'**) # 路径名称  *for* f *in* subfile: # 遍历该路径下所有文件  print(os.path.join(dirname, f))  traverseFiles(**"."**) |

# 10 内置容器类型

## 10.1 list

list包括go语言的slice大多用法，但比slice灵活和丰富

|  |
| --- |
| # 声明list a = [1, 2, 3] b = [4, **'hello'**, 3.14] # 可以不同类型 c = [5, **'world'**, [6, 7]] # 嵌入list print(a, b, c)  a = [i *for* i *in* range(0, 10)] # 通过for range声明整形的list print(a) b = (4, (**'hello'**, 3.14), 5) c = [j *for* j *in* b] # 也可以通过已知元祖声明list，然后增删改查list print(c)  # 遍历list a = [1,2,3] *for* v *in* a:  print(v)  # 切片 a = [1, 2, 3, 4] b = a[1:3] print(b)  # 查看list有哪些方法 print(dir(list))  # append追加 a.append(100) print(a) a.append(b) # 追加一个list print(a)  # extend把list元素打散追加到另一个list后面 a = [1, 2, 3] b = [4, **'hello'**, 3.14] # 可以不同类型 a.extend(b) print(a)  # insert插入一个元素 a = [1, 2, 3] a.insert(1, 100) # 在第一个位置插入一个元素 a.insert(2, [**'hello'**, 3.14]) # 在第二个位置插入一个list print(a)  # sort排序 a = [12, 42, 23, 8] a.sort() # 升序 print(a) a.sort(reverse=*True*) print(a) # 降序  # pop弹出一个元素，不指定直接弹出最后一个元素 a = [1, 2, 3, 4] a.pop(3) # 弹出指定位置元素 print(a)  # remove移除一个指定值的元素 a = [1, 2, 3, 4, 3.14] a.remove(3.14) # 删除指定的值的元素 print(a)  # reverse反转元素 a = [1, 2, 3, 4] a.reverse() print(a) |

list内置函数insert结合shell或其他应用导入路径

|  |
| --- |
| *import* sys print(sys.path) sys.path.insert(**"/usr/local/app"**) |

## 10.2 tuple元组

元组和list的区别是，元组没有内置的增删改查方法，元组一旦声明是不可改变，这是由元组底层决定的，所以遍历速度比list快很多，用途也不一样，当声明不需要改变，只用来查询时用元组，当需要编辑修改时用list。

|  |
| --- |
| a = (0, 1, 2, 3) # 声明元组 b = (4, (**'hello'**, 3.14), 5) # 声明嵌套元组 c = (x *for* x *in* a) # 生成器，通过type(c)查看类型，dir(c)查看方法 d = ([1, 2], 3, 4) # 元组嵌套list，这种方式可以对元祖可变元素操作 print(a, b, c, d)  # 遍历元组 *for* v *in* a:  print(v)  # 索引切片操作 print(a[1:3]) print(a[:2])  # 查看元组有哪些方法 print(dir(a))  # 通过元组格式化输出 print(**"get %d %s"** % (10, **"apple"**)) |

## 10.3 字典

|  |
| --- |
| # 字典声明 a={**"name"**:**"zhangsan"**,**"gender"**:**"male"**,**"age"**:22,}  # 字典遍历 *for* kv *in* a.items():  print(kv) # 获取字典的键值对(元组方式)  print(kv[0]) # 获取字典的键名  print(kv[1]) # 获取字典的值  # 判断key是否存在 *if* (**"age"** *in* a.keys()):  print(**"age is exist."**)  # 获取字典的所有key(list) print(a.keys())  # 获取字典的所有value(list) print(a.values())  # 查看字典的所有方法 print(dir(dict))  # 字典排序  a = {**'a'**: 6, **'d'**: 2, **'c'**: 11, **'b'**: 7} # 方式一，只返回键的list b = sorted(a, key=*lambda* x: x[0]) # 按键排序，默认升序，可以通过传参数reverse=True反转 print(b)  # 方式二，返回键值对list c = sorted(a.items(), key=*lambda* x: x[0]) # 按键排序 print(c) d = sorted(a.items(), key=*lambda* x: x[1]) # 按值排序 print(d) |

## 10.4 集合set

|  |
| --- |
| # 定义集合 a = set([1, 2, 2, 3]) # 通过list声明 b = set((1, 2, 2, 3)) # 通过tuple声明 print(a, b)  c = [1, 2, 2, 3, 4, 5, 5] d = set(c) # 集合的特性是唯一性，可以对list或tuple进行去重 print(d)  # 遍历集合 *for* v *in* a:  print(v)  # 查看集合其他方法，包括集合之间操作 print(dir(set)) |

## 10.5 生成器generator和迭代器iterator

**(1) 生成器generator**

生成器可以看作状态寄存器，各个状态可以保存的。

一个带有 yield 的函数就是一个 generator，它和普通函数不同，生成一个 generator 看起来像函数调用，但不会执行任何函数代码，直到对其调用 next()（在 for 循环中会自动调用 next()）才开始执行。虽然执行流程仍按函数的流程执行，但每执行到一个 yield 语句就会中断，并返回一个迭代值，下次执行时从 yield 的下一个语句继续执行。看起来就好像一个函数在正常执行的过程中被 yield 中断了数次，每次中断都会通过 yield 返回当前的迭代值。

yield 的好处是显而易见的，把一个函数改写为一个 generator 就获得了迭代能力，比起用类的实例保存状态来计算下一个 next() 的值，不仅代码简洁，而且执行流程异常清晰。

|  |
| --- |
| # 生成斐波那契（Fibonacci）數列函数，特性：除第一个和第二个数外，任意一个数都可由前两个数相加得到 *def* fab(*max*):  n, a, b = 0, 0, 1  *while* n < *max*:  *yield* b  a, b = b, a + b  n = n + 1  f=fab(6) print(type(f))  # 遍历生成器 # for v in f: # print(v)  # 一个一个获取值(通过next操作可以记录到当前允许位置) *try*:  print(f.\_\_next\_\_()) *except* StopIteration: # StopIteration表示生成器或迭代器的异常处理标记  *pass* # 判断函数是否返回generator类型函数 *from* inspect *import* isgeneratorfunction print(isgeneratorfunction(fab)) |

|  |
| --- |
| *import* os  a = os.walk(**"."**) # 返回的类型是生成器  # 查看生成器方法 print(dir(a))  # 生成器遍历遍历 # for v in a: # print(v) # 格式(当前文件夹, 子文件夹, 当前文件夹下的文件名) #  # 一个一个获取值(通过next操作可以记录到当前允许位置) *try*:  print(a.\_\_next\_\_()) *except* StopIteration: # StopIteration表示生成器或迭代器的异常处理标记  *pass* |

**(2) 迭代器iterator**

|  |
| --- |
| # 定义 a=[1,2,3,4] b=a.\_\_iter\_\_() # 内置函数定义迭代器 c=iter(a) # 使用内置函数生成迭代器 print(type(b),type(c))  # 查看迭代器方法 print(dir(b))  # 遍历迭代器 # for v in b: # print(v)  # 一个一个获取值(通过next操作可以记录到当前允许位置) *try*:  print(b.\_\_next\_\_()) *except* StopIteration: # StopIteration表示生成器或迭代器的异常处理标记  *pass* |

## 10.6 使用内置容器实现笛卡尔方法

解决现实复杂的问题时

1. 尽量分解为多个比较简单的小问题，一个一个地分开解决。
2. 小问题从简单到复杂排列，先从容易解决的问题着手。
3. 问题解决后，再综合起来检验，看是否完全，是否将问题彻底解决了。

|  |
| --- |
| # 拆分独立并处理map a = [1, 3, 11, 4, 6] # 有时候对某些元素不做处理，可以用filter函数过滤该元素 b = filter(*lambda* x: x < 5, a) # 过滤a元素大于等于5的元素,返回类型为迭代器类型 print(b) c = list(map(*lambda* x: x \* x \* 3.14, b)) # list的每个成员计算面积(处理方法，参数为匿名函数),注：map返回的类型iterators(迭代器)，需要强制转换为list print(c)  # 归并reduced *from* functools *import* reduce  d = reduce(*lambda* x, y: x + y, c) print(d) |

# 11 并发和进程间通信

## 11.1 创建子进程

fork进程

|  |
| --- |
| *import* os *import* time  *def* myFork():  a = 0  pid = os.fork()  *if* pid == 0: # 子进程  print(**"this is child"**, os.getpid())  time.sleep(1) # 模拟子进程执行时间比父进程长  print(a + 1)  *else*: # 父进程  os.waitpid(pid, 0) # 第一个参数是进程pid，第二个参数0表示等待，1表示不等待  print(**"this is parent"**, os.getpid())  print(a + 10)  *if* \_\_name\_\_ == **"\_\_main\_\_"**: # 程序入口  myFork() |

创建一个守护进程

|  |
| --- |
| *import* os,sys,time  # 创建一个守护进程，一直写入文件数据 *def* daemon():  pid=os.fork()  *if* pid>0:  sys.exit(0) # 退出父进程   os.setsid() # 脱离终端  os.umask(0) # 设置当前文件的操作权限   pid=os.fork() # 再次fork  *if* pid>0:  sys.exit(0) # 退出父进程   f=open(**"test.txt"**, **'w'**)  *for* i *in* range(1000000):  f.write(str(i)+**' '**)  time.sleep(0.001)  f.close()  *if* \_\_name\_\_ == **"\_\_main\_\_"**:  daemon() |

## 11.2 进程间通信

**(1) 管道方式**

使用一个进程发送数据到管道，另一个进程从管道读取数据

|  |
| --- |
| *import* os *import* time  *def* myFork():  r, w = os.pipe() # 声明管道  pid = os.fork()  *if* pid == 0: # 子进程  os.close(r) # 关闭读文件符  wr = os.fdopen(w, **'w'**)  *for* i *in* range(100):  wr.write(str(i)+**' '**) # 从管道发送数据  time.sleep(0.02)  wr.close() # 关闭写文件符后，另一个进程才能读取管道的数据  *else*: # 父进程  os.close(w) # 关闭写文件符  re = os.fdopen(r, **'r'**)  print(re.read()+**' '**) # 从管道读取数据  re.close() # 关闭读文件符   *if* \_\_name\_\_ == **"\_\_main\_\_"**:  myFork() |

(2) 信号

|  |
| --- |
| *import* os *import* signal *import* time  **''' 常用信号 SIGINT 终止进程 中断进程 (control+c) SIGTERM 终止进程 软件终止信号 SIGKILL 终止进程 杀死进程 (不能被捕捉，强制停止) SIGALRM 闹钟信号 '''** # 定义信号处理函数(这里通知父进程退出) *def* onSignalTerm(*a*,*b*):  print(**"exit"**)  exit()  # 把信号处理函数注册到系统信号里 signal.signal(signal.SIGTERM,onSignalTerm)  *def* myFork():  pid = os.fork()  *if* pid == 0: # 子进程  *for* i *in* range(100):  *if* i==50:  os.kill(os.getppid(), signal.SIGTERM) # 通知全组  time.sleep(0.02)  *else*: # 父进程  time.sleep(100)   *if* \_\_name\_\_ == **"\_\_main\_\_"**:  myFork() |

## 11.3 多线程

**函数式编程方式**

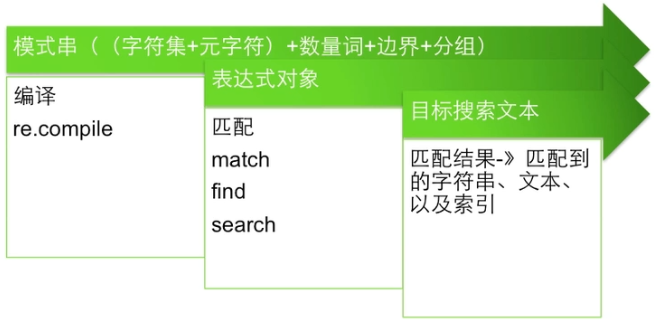
|  |
| --- |
| *import* \_thread,time  *def* func(*tid*,*v*):  *for* i *in* range(10):  print(**'thread id=%d, value=%d'**%(*tid*,i))  time.sleep(0.3)  *def* threadTest():  \_thread.start\_new\_thread(func,(1,1)) # 创建线程  \_thread.start\_new\_thread(func,(2,2))  \_thread.start\_new\_thread(func,(3,3))  time.sleep(10)  *if* \_\_name\_\_ == **"\_\_main\_\_"**:  threadTest() |

**对象编程方式**

|  |
| --- |
| *import* threading, time  count = 0 myLock = threading.\_RLock() # 定义线程锁  *class* threadObject(threading.Thread):  *def \_\_init\_\_*(self, *tid*):  threading.Thread.\_\_init\_\_(self)  self.tid = *tid   def* run(self):  *global* count  *for* i *in* range(10):  myLock.acquire()  count += 1  myLock.release()  print(**'thread id=%d, count=%d'** % (self.tid, count))  time.sleep(0.3)  *def* func():  t1 = threadObject(1) # 实例化对象  t2 = threadObject(2)  t3 = threadObject(3)  t1.start() # 启动一个线程  t2.start()  t3.start()   t1.join()  t2.join()  t3.join() # 主线程等待子线程结束  *if* \_\_name\_\_ == **"\_\_main\_\_"**:  func() |

# 12 正则表达式

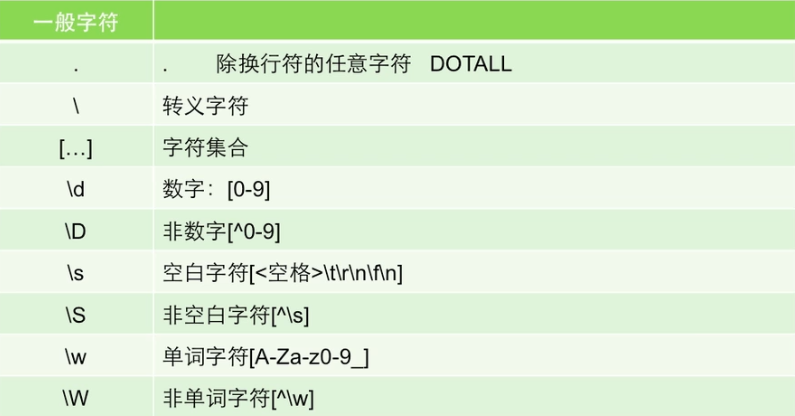
正则表达式过程



一个简单的正则

|  |
| --- |
| *import* re  # 声明匹配模式 p = re.compile(**r"abc"**) # 待查找字符串abc  # 查看方法 print(dir(p))  # 匹配一个字符串 result = p.match(**"abc123\nabc456\nabc789"**) *if* result != *None*: # 不为None说明已经匹配到字符串  print(dir(result))  print(result.group())  # 查找所有匹配的字符串 result = p.findall(**"abc123\nabc456\nabc789"**) print(result) |

## 12.1 元字符



匹配字符集合

|  |
| --- |
| *import* re  p = re.compile(**'[abc]'**) result = p.findall(**'1a2b3c4d5e'**) print(result) |

## 12.2 数量词



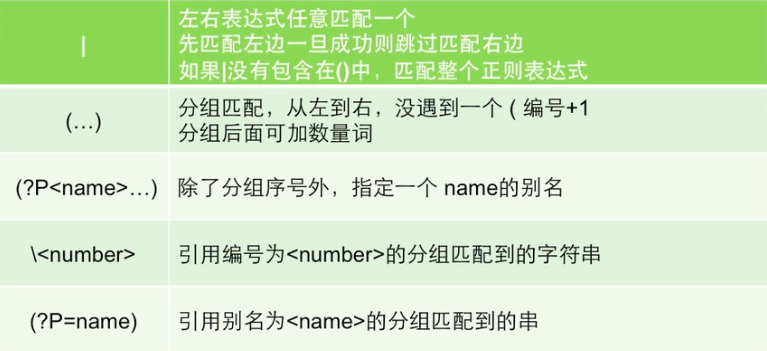
|  |
| --- |
| *import* re  # 默认是贪婪模式，后面加问号为非贪婪模式 p = re.compile(**'[abc]\*'**) print(p.findall(**'abc\_abc'**)) # 结果：['abc', '', 'abc', '']  p = re.compile(**'[abc]\*?'**) print(p.findall(**'abc\_abc'**)) # 结果：['', '', '', '', '', '', '', '']  p = re.compile(**'[abc]+'**) print(p.findall(**'abc\_abc'**)) # 结果：['abc', 'abc']  p = re.compile(**'[abc]+?'**) print(p.findall(**'abc\_abc'**)) # 结果：['a', 'b', 'c', 'a', 'b', 'c']  p = re.compile(**'[abc]{2}'**) print(p.findall(**'abc\_abc'**)) # 结果：['ab', 'ab']  p = re.compile(**'[abc]{1,4}'**) print(p.findall(**'abc\_abc'**)) # 结果：['abc', 'abc']  p = re.compile(**'[abc]{1,4}?'**) print(p.findall(**'abc\_abc'**)) # 结果：['a', 'b', 'c', 'a', 'b', 'c'] |

## 12.3 边界



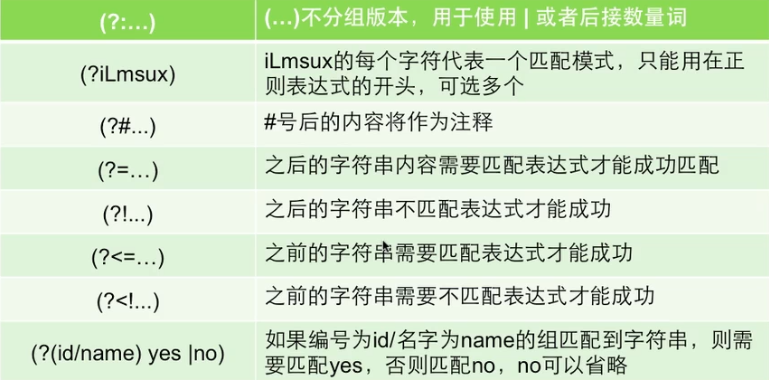
|  |
| --- |
| *import* re  # 匹配以xxx开头的字符串 p = re.compile(**'^[abc]\*'**) print(p.findall(**'abc** \n **1abc** \n **2abc** \n **bc** \n **ab** \n **ca'**)) # 结果：['abc']  p = re.compile(**'[^abc]\*'**) # 取反 print(p.findall(**'abc1abc2abc'**)) # 结果：['', '', '', '1', '', '', '', '2', '', '', '', '']  # 以xxx开头并且以xxx结尾的字符串 p = re.compile(**'^[abc]\*e$'**) print(p.findall(**'abce'**)) # 结果：['abce'] print(p.findall(**'abe'**)) # 结果：['abe'] print(p.findall(**'bce'**)) # 结果：['bce'] print(p.findall(**'ce'**)) # 结果：['ce'] print(p.findall(**'abcabce'**)) # 结果：['abcabce'] print(p.findall(**'abc----e'**)) # 结果：[] |

## 12.4 逻辑和分组



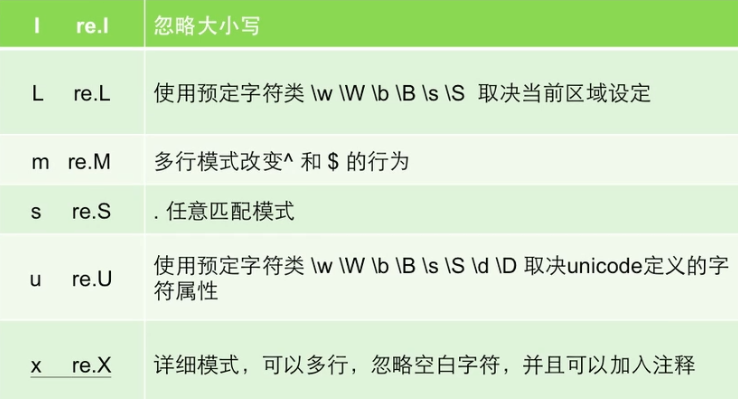
|  |
| --- |
| *import* re  p = re.compile(**'(a)(b)(c)'**) result=p.match(**'abcdef'**) print(result.groups()) # 结果：('a', 'b', 'c')  p = re.compile(**'(a)b(c)'**) result=p.match(**'abcdef'**) print(result.groups()) # 结果：('a', 'c')  p = re.compile(**'(?P<aKey>a)(b)(c)'**) result=p.match(**'abcdef'**) print(result.groups()) # 结果：('a', 'b', 'c') print(result.groupdict()) # 结果：{'aKey': 'a'} |

## 12.5 特殊构造



|  |
| --- |
| *import* re  # 前缀约束 p = re.compile(**r'(?:abc){2}'**) # 匹配连续两个abc字符串 print(p.findall(**'abcabcabcabc'**)) # 结果：['abcabc', 'abcabc']  p = re.compile(**r'a(?=\d)'**) # 匹配固定字符串+数字 print(p.findall(**'a1a2a3'**)) # 结果：['a', 'a', 'a']  p = re.compile(**r'a(?!\d)'**) # 匹配固定字符串+非数字 print(p.findall(**'a1ab2abc3'**)) # 结果：['a', 'a']  p = re.compile(**r'\w+(?=\d)'**) # 匹配字符串+数字 print(p.findall(**'apple1 egg2 banana3'**)) # 结果：['apple', 'egg', 'banana']  # 后缀约束 p = re.compile(**r'(?<=\d)a'**) # 匹配数字+固定字符串 print(p.findall(**'1a2a3a'**)) # 结果：['a', 'a', 'a']  p = re.compile(**r'(?<!\d)a'**) # 匹配非数字+固定字符串 print(p.findall(**'1a2ba3cba'**)) # 结果：['a', 'a']  # 逻辑匹配 p = re.compile(**r'(\d)?abc(?(1)\d|abc)'**) # 匹配1abc或abc result = p.match(**'1abc2abc'**) print(result.group()) # 结果：1abc2  result = p.match(**'abcabc'**) print(result.group()) # 结果：abcabc |

## 12.6 选项开关iLmsux



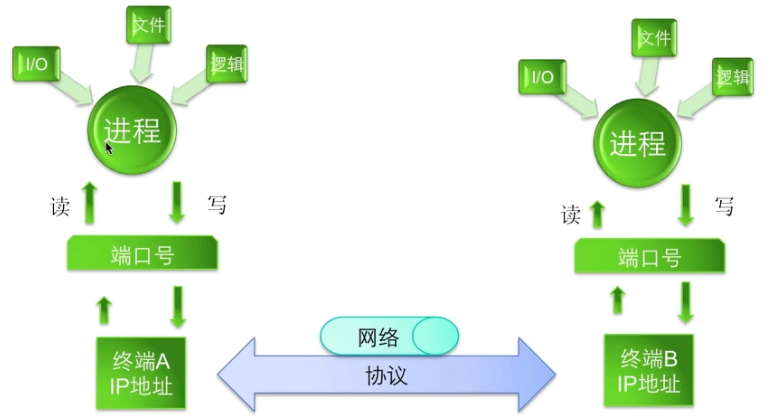
|  |
| --- |
| *import* re  p = re.compile(**r'abc'**, re.I) # 忽略大小写 print(p.findall(**'Abc'**)) # 结果：['Abc'] print(p.findall(**'ABCabc'**)) # 结果：['ABC', 'abc'] |

## 12.7 正则方法

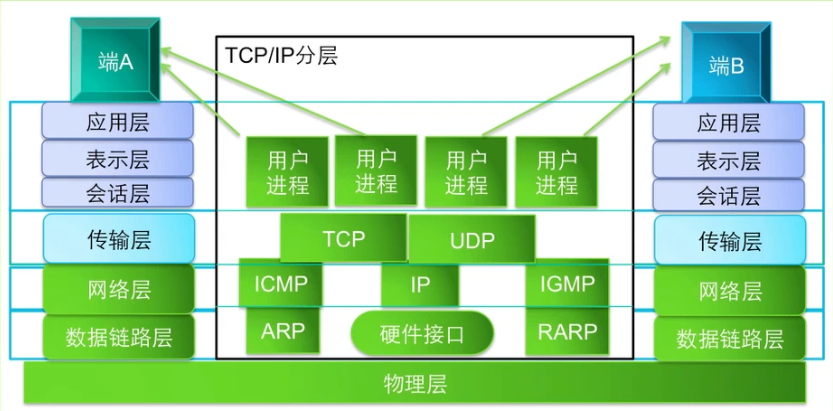
|  |
| --- |
| *import* re  # match和search的区别 p = re.compile(**r'abc'**) result = p.match(**'zzabczzaaa'**) # match是从开始位置匹配的 *if* result != *None*:  print(result.group()) *else*:  print(**'None'**) result = p.search(**'zzabczzaaa'**) # search是匹配任意位置 print(result.group()) # 结果：abc  # split p = re.compile(**r','**) print(p.split(**'apple,banana,egg'**))  # finditer p = re.compile(**'a'**) result = p.finditer(**'banana'**) # 返回迭代器类型 *try*:  print(result.\_\_next\_\_().group()) *except* StopIteration:  *pass* # sub p = re.compile(**r'(\w+) (\w+)'**) str = **'hello world, study python hi'** print(p.sub(**r'\2 \1'**, str)) # 两两调换顺序，结果：world hello, python study hi print(p.sub(**r'xxx \1'**, str)) # 替换字符串并掉顺序，结果：xxx hello, xxx study hi |

# 13 socket编程

网络模型



TCP/IP协议分层



C/S网络模型

