

*Linux, Programming, CLI

Basic Skills to CTF and Hacking

September 19, 2021

Now we ready have the topics about 'Introduction to CTF'

Review:

- * CTF introduction
- * Categories
- * Tools and resources to CTF
- * How to use search engines
- * Related books

Make sure to use <https://wiki.compass.college/> to check materials

Today we have contents as follow:

- * Linux Field Guide
- * Python Programming Basics
- * CLI Toolkit

Most of time we use Linux as operating system and server. Windows is a good PC desktop OS but not open source. Linux server has better open source software environment and better for professional users.

In Linux basics, we have the following chapters:

- * 1. 学会使用命令帮助
- * 2. 文件及目录管理
- * 3. 文本处理
- * 4. 磁盘管理
- * 5. 进程管理工具
- * 6. 性能监控
- * 7. 网络工具
- * 8. 用户管理工具
- * 9. 系统管理及IPC资源管理

1. 学会使用命令帮助

在linux终端，面对命令不知道怎么用，或不记得命令的拼写及参数时，我们要求助于系统的帮助文档；linux系统内置的帮助文档很详细，通常能解决我们的问题，我们需要掌握如何正确的去使用它们

- ✦ 在只记得部分命令关键字的场合，我们可通过`man -k`来搜索；
- ✦ 需要知道某个命令的简要说明，可以使用`whatis`；而更详细的介绍，则可用`info`命令；
- ✦ 查看命令在哪个位置，我们需要使用`which`；
- ✦ 而对于命令的具体参数及使用方法，我们需要用到强大的`man`；

1. 学会使用命令帮助

查看命令的简要说明

简要说明命令的作用（显示命令所处的man分类页面）：

Solution

```
$whatis command
```

正则匹配:

Solution

```
$whatis -w "loca*"
```

更加详细的说明文档:

Solution

```
$info command
```

1. 学会使用命令帮助

使用man

查询命令command的说明文档:

Solution

```
$man command
```

```
eg: man date
```

使用page up和page down来上下翻页

在man的帮助手册中，将帮助文档分为了9个类别，对于有的关键字可能存在多个类别中，我们就需要指定特定的类别来查看；（一般我们查询bash命令，归类在1类中）；

man页面所属的分类标识(常用的是分类1和分类3)

1. 学会使用命令帮助

- (1)、用户可以操作的命令或者是可执行文件
- (2)、系统核心可调用的函数与工具等
- (3)、一些常用的函数与数据库
- (4)、设备文件的说明
- (5)、设置文件或者某些文件的格式
- (6)、游戏
- (7)、惯例与协议等。例如Linux标准文件系统、网络协议、ASCII, 码等说明内容
- (8)、系统管理员可用的管理条令
- (9)、与内核有关的文件

1. 学会使用命令帮助

前面说到使用`whatis`会显示命令所在的具体的文档类别，我们学习如何使用它

Solution

eg:

\$whatis printf

printf (1) - format and print data

printf (1p) - write formatted output

printf (3) - formatted output conversion

printf (3p) - print formatted output

printf [builtins] (1) - bash built-in commands, see bash(1)

1. 学会使用命令帮助

我们看到printf在分类1和分类3中都有；分类1中的页面是命令操作及可执行文件的帮助；而3是常用函数库说明；如果我们想看的是C语言中printf的用法，可以指定查看分类3的帮助：

Solution

```
$man 3 printf
```

```
$man -k keyword
```

查询关键字 根据命令中部分关键字来查询命令，适用于只记住部分命令的场合；

eg: 查找GNOME的config配置工具命令：

Solution

```
$man -k GNOME config| grep 1
```

对于某个单词搜索，可直接使用/word来使用：/-a; 多关注下SEE ALSO 可看到更多精彩内容

1. 学会使用命令帮助

查看路径

查看程序的binary文件所在路径:

Solution

```
$which command
```

eg:查找make程序安装路径:

Solution

```
$which make
```

```
/opt/app/openav/soft/bin/make install
```

查看程序的搜索路径:

Solution

```
$whereis command
```

当系统中安装了同一软件的多个版本时，不确定使用的是哪个版本时，这个命令就能派上用场；

1. 学会使用命令帮助

总结

- * whatis
- * info
- * man
- * which
- * whereis

2. 文件及目录管理

文件管理不外乎文件或目录的创建、删除、查询、移动，有`mkdir/rm/mv`

文件查询是重点，用`find`来进行查询；`find`的参数丰富，也非常强大；

查看文件内容是个大的话题，文本的处理有太多的工具供我们使用，在本章中只是点到即止，后面会有专门的一章来介绍文本的处理工具；

有时候，需要给文件创建一个别名，我们需要用到`ln`，使用这个别名和使用原文件是相同的效果；

2. 文件及目录管理

2.1. 创建和删除

- * 创建: `mkdir`
- * 删除: `rm`
- * 删除非空目录: `rm -rf file目录`
- * 删除日志 `rm *log` (等价: `$find ./ -name "*log" -exec rm {} ;`)
- * 移动: `mv`
- * 复制: `cp` (复制目录: `cp -r`)

查看当前目录下文件个数:

Solution

```
$find ./ | wc -l
```

复制目录:

Solution

```
$cp -r source_dir dest_dir
```

2. 文件及目录管理

2.2. 目录切换

- * 找到文件/目录位置: `cd`
- * 切换到上一个工作目录: `cd -`
- * 切换到home目录: `cd` or `cd`
- * 显示当前路径: `pwd`
- * 更改当前工作路径为path: `$cd path`

2.3. 列出目录项

- ✦ 显示当前目录下的文件 `ls`
- ✦ 按时间排序，以列表的方式显示目录项 `ls -lrt`

以上这个命令用到的频率如此之高，以至于我们需要为它建立一个快捷命令方式：
在 `.bashrc` 中设置命令别名：

Solution

```
alias ls='ls -lrt'
```

```
alias lm='ls -al|more'
```

这样，使用 `ls`，就可以显示目录中的文件按照修改时间排序；以列表方式显示；

2. 文件及目录管理

给每项文件前面增加一个id编号(看上去更加整洁):

Solution

```
>ls | cat -n
```

```
1 a 2 a.out 3 app 4 b 5 bin 6 config
```

注: `.bashrc` 在 `/home/你的用户名/` 文件夹下, 以隐藏文件的方式存储; 可使用 `ls -a` 查看;

2. 文件及目录管理

2.4. 查找目录及文件 find/locate

搜寻文件或目录:

Solution

```
$find ./ -name "core*" | xargs file
```

查找目标文件夹中是否有obj文件:

Solution

```
$find ./ -name '*.o'
```

递归当前目录及子目录删除所有.o文件:

Solution

```
$find ./ -name "*.o" -exec rm \;
```

2. 文件及目录管理

`find`是实时查找，如果需要更快的查询，可试试`locate`；`locate`会为文件系统建立索引数据库，如果有文件更新，需要定期执行更新命令来更新索引库：

Solution

```
$locate string
```

寻找包含有`string`的路径：

Solution

```
$updatedb
```

与`find`不同，`locate`并不是实时查找。你需要更新数据库，以获得最新的文件索引信息。

2. 文件及目录管理

2.5. 查看文件内容

查看文件: `cat vi head tail more`

显示时同时显示行号:

Solution

```
$cat -n
```

按页显示列表内容:

Solution

```
$ls -al | more
```

只看前10行:

Solution

```
$head - 10 **
```

显示文件第一行:

Solution

```
$head -1 filename
```

2. 文件及目录管理

显示文件倒数第五行:

Solution

```
$tail -5 filename
```

查看两个文件间的差别:

Solution

```
$diff file1 file2
```

动态显示文本最新信息:

Solution

```
$tail -f crawler.log
```

2. 文件及目录管理

2.6. 查找文件内容

使用egrep查询文件内容:

Solution

```
egrep '03.1\|/CO\|/AE' TSF_STAT_111130.log.012
```

```
egrep 'A_LMCA777:C' TSF_STAT_111130.log.035 > co.out2
```

2. 文件及目录管理

2.7. 文件与目录权限修改

- * 改变文件的拥有者 `chown`
- * 改变文件读、写、执行等属性 `chmod`
- * 递归子目录修改: `chown -R tuxapp source/`
- * 增加脚本可执行权限: `chmod a+x myscript`

2. 文件及目录管理

2.8. 给文件增加别名

创建符号链接/硬链接:

Solution

`ln cc ccAgain` :硬连接; 删除一个, 将仍能找到;

`ln -s cc ccTo` :符号链接(软链接); 删除源, 另一个无法使用; (后面一个`ccTo` 为新建的文件)

2. 文件及目录管理

2.9. 管道和重定向

- * 批处理命令连接执行，使用 |
- * 串联: 使用分号 ;
- * 前面成功，则执行后面一条，否则，不执行:&&
- * 前面失败，则后一条执行: ||

Solution

```
ls /proc && echo suss! || echo failed.
```

能够提示命名是否执行成功or失败;

与上述相同效果的是:

Solution

```
if ls /proc; then echo suss; else echo fail; fi
```

2. 文件及目录管理

重定向:

Solution

```
ls proc/*.c > list 2> &| 将标准输出和标准错误重定向到同一文件;
```

等价的是:

Solution

```
ls proc/*.c &> list
```

清空文件:

Solution

```
:> a.txt
```

重定向:

Solution

```
echo aa » a.txt
```

2. 文件及目录管理

2.10. 设置环境变量

启动帐号后自动执行的是 文件为 `.profile`，然后通过这个文件可设置自己的环境变量；
安装的软件路径一般需要加入到`path`中：

Solution

```
PATH=$APPDIR:/opt/app/soft/bin:$PATH:/usr/local/bin:\
$TUXDIR/bin:$ORACLE_HOME/bin;export PATH
```

2. 文件及目录管理

2.11. Bash快捷输入或删除

快捷键:

- ✦ Ctl-U 删除光标到行首的所有字符,在某些设置下,删除全行
- ✦ Ctl-W 删除当前光标到前边的最近一个空格之间的字符
- ✦ Ctl-H backspace,删除光标前边的字符
- ✦ Ctl-R 匹配最相近的一个文件,然后输出

2. 文件及目录管理

2.12. 综合应用

查找record.log中包含AAA，但不包含BBB的记录总数:

Solution

```
cat -v record.log | grep AAA | grep -v BBB | wc -l
```

2. 文件及目录管理

2.13. 总结

文件管理，目录的创建、删除、查询、管理: `mkdir rm mv`

文件的查询和检索: `find locate`

查看文件内容: `cat vi tail more`

管道和重定向: `;` `|` `&&` `>`

3. 文本处理

3.1. find 文件查找

查找txt和pdf文件:

Solution

```
find . \( -name "*.txt" -o -name "*.pdf" \) -print
```

正则方式查找.txt和pdf:

Solution

```
find . -regex ".*\(\.txt|\.pdf\) $"
```

-iregex: 忽略大小写的正则

否定参数 ,查找所有非txt文本:

Solution

```
find . ! -name "*.txt" -print
```

指定搜索深度,打印出当前目录的文件(深度为1):

Solution

```
find . -maxdepth 1 -type f
```

3. 文本处理

定制搜索

按类型搜索

Solution

```
find . -type d -print //只列出所有目录
```

-type f 文件 / l 符号链接 / d 目录

find支持的文件检索类型可以区分普通文件和符号链接、目录等，但是二进制文件和文本文件无法直接通过find的类型区分出来；

file命令可以检查文件具体类型（二进制或文本）：

Solution

```
$file redis-cli # 二进制文件
```

```
redis-cli: ELF 64-bit LSB executable, x86-64, version 1 (SYSV), dynamically linked  
(uses shared libs), for GNU/Linux 2.6.9, not stripped
```

```
$file redis.pid # 文本文件
```

```
redis.pid: ASCII text
```


3. 文本处理

所以,可以用以下命令组合来实现查找本地目录下的所有二进制文件:

Solution

```
ls -lrt | awk '{print $9}' | xargs \
file | grep ELF | awk '{print $1}' | tr -d ':'
```

按时间搜索

- * `-atime` 访问时间 (单位是天, 分钟单位则是`-amin`, 以下类似)
- * `-mtime` 修改时间 (内容被修改)
- * `-ctime` 变化时间 (元数据或权限变化)

3. 文本处理

最近第7天被访问过的所有文件:

Solution

```
find . -atime 7 -type f -print
```

最近7天内被访问过的所有文件:

Solution

```
find . -atime -7 -type f -print
```

查询7天前被访问过的所有文件:

Solution

```
find . -atime +7 type f -print
```

3. 文本处理

按大小搜索:

w字 k M G 寻找大于2k的文件:

Solution

```
find . -type f -size +2k
```

按权限查找:

Solution

```
find . -type f -perm 644 -print //找具有可执行权限的所有文件
```

按用户查找:

Solution

```
find . -type f -user weber -print// 找用户weber所拥有的文件
```

3. 文本处理

找到后的后续动作

删除

删除当前目录下所有的swp文件:

Solution

```
find . -type f -name "*.swp" -delete
```

另一种语法:

Solution

```
find . type f -name "*.swp" | xargs rm
```

执行动作（强大的exec）

将当前目录下的所有权变更为weber:

Solution

```
find . -type f -user root -exec chown weber {} \;
```

注: {}是一个特殊的字符串, 对于每一个匹配的文件, {}会被替换成相应的文件名;

3. 文本处理

将找到的文件全都copy到另一个目录:

Solution

```
find . -type f -mtime +10 -name "*.txt" -exec cp {} OLD \;
```

结合多个命令

如果需要后续执行多个命令，可以将多个命令写成一个脚本。然后 `-exec` 调用时执行脚本即可:

Solution

```
-exec ./commands.sh {} \;
```

-print的定界符

默认使用'\n'作为文件的定界符;

-print0 使用'\0'作为文件的定界符，这样就可以搜索包含空格的文件;

3. 文本处理

3.2. grep 文本搜索

Solution

```
grep match_patten file // 默认访问匹配行
```

常用参数

- * -o 只输出匹配的文本行 VS -v 只输出没有匹配的文本行
- * -c 统计文件中包含文本的次数
 - * `grep -c "text" filename`
- * -n 打印匹配的行号
- * -i 搜索时忽略大小写
- * -l 只打印文件名

3. 文本处理

在多级目录中对文本递归搜索(程序员搜代码的最爱)：

Solution

```
grep "class" . -R -n
```

匹配多个模式:

Solution

```
grep -e "class" -e "vital" file
```

grep输出以0作为结尾符的文件名 (-z)：

Solution

```
grep "test" file* -lZ | xargs -0 rm
```

综合应用：将日志中的所有带where条件的sql查找查找出来：

Solution

```
cat LOG.* | tr a-z A-Z | grep "FROM " | grep "WHERE" > b
```

3. 文本处理

查找中文示例：工程目录中utf-8格式和gb2312格式两种文件，要查找字的是中文；

1. 查找到它的utf-8编码和gb2312编码分别是E4B8ADE69687和D6D0CEC4

2. 查询:

Solution

```
grep: grep -rnP "\xE4\xB8\xAD\xE6\x96\x87|\xD6\xD0\xCE\xC4" *即可
```

汉字编码查询: <http://bm.kdd.cc/>

3. 文本处理

3.3. xargs 命令行参数转换

xargs 能够将输入数据转化为特定命令的命令行参数；这样，可以配合很多命令来组合使用。比如grep，比如find； - 将多行输出转化为单行输出

Solution

```
cat file.txt | xargs
```

n 是多行文本间的定界符

将单行转化为多行输出

Solution

```
cat single.txt | xargs -n 3
```

-n: 指定每行显示的字段数

3. 文本处理

xargs参数说明

- * -d 定义定界符（默认为空格 多行的定界符为 n）
- * -n 指定输出为多行
- * -l {} 指定替换字符串，这个字符串在xargs扩展时会被替换掉,用于待执行的命令需要多个参数时
- * -0: 指定0为输入定界符

示例:

Solution

```
cat file.txt | xargs -l {} ./command.sh -p {} -l
```

#统计程序行数

```
find source_dir/ -type f -name "*.cpp" -print0 |xargs -0 wc -l
```

#redis通过string存储数据，通过set存储索引，需要通过索引来查询出所有的值：

```
./redis-cli smembers $1 | awk '{print $1}'|xargs -l {} ./redis-cli get {}
```

3. 文本处理

3.4. sort 排序

字段说明

- * `-n` 按数字进行排序 VS `-d` 按字典序进行排序
- * `-r` 逆序排序
- * `-k N` 指定按第N列排序

示例:

Solution

```
sort -nrk 1 data.txt
```

```
sort -bd data // 忽略像空格之类的前导空白字符
```

3. 文本处理

3.5. uniq 消除重复行

消除重复行

Solution

```
sort unsort.txt | uniq
```

统计各行在文件中出现的次数

Solution

```
sort unsort.txt | uniq -c
```

找出重复行

Solution

```
sort unsort.txt | uniq -d
```

可指定每行中需要比较的重复内容: `-s` 开始位置 `-w` 比较字符数

3. 文本处理

3.6. 用tr进行转换

通用用法

Solution

```
echo 12345 | tr '0-9' '9876543210' //加解密转换，替换对应字符  
cat text | tr '\t' ' ' //制表符转空格
```

tr删除字符

Solution

```
cat file | tr -d '0-9' // 删除所有数字
```

-c 求补集

Solution

```
cat file | tr -c '0-9' //获取文件中所有数字  
cat file | tr -d -c '0-9 \n' //删除非数字数据
```

3. 文本处理

`tr -s` 压缩文本中出现的重复字符；最常用于压缩多余的空格：

Solution

```
cat file | tr -s ' '
```

`tr`中可用各种字符类：

- * `alnum`: 字母和数字
- * `alpha`: 字母
- * `digit`: 数字
- * `space`: 空白字符
- * `lower`: 小写
- * `upper`: 大写
- * `cntrl`: 控制（非可打印）字符
- * `print`: 可打印字符

使用方法：`tr [:class:] [:class:]`

Solution

```
tr '[:lower:]' '[:upper:]'
```

3. 文本处理

3.7. cut 按列切分文本

截取文件的第2列和第4列

Solution

```
cut -f2,4 filename
```

去文件除第3列的所有列

Solution

```
cut -f3 -complement filename
```

-d 指定定界符

Solution

```
cat -f2 -d";" filename
```

3. 文本处理

cut 取的范围

- * N- 第N个字段到结尾
- * -M 第1个字段为M
- * N-M N到M个字段

cut 取的单位

- * -b 以字节为单位
- * -c 以字符为单位
- * -f 以字段为单位（使用定界符）

3. 文本处理

示例:

Solution

```
cut -c1-5 file //打印第一到5个字符
```

```
cut -c-2 file //打印前2个字符
```

截取文本的第5到第7列

Solution

```
$echo string | cut -c5-7
```

3. 文本处理

3.8. paste 按列拼接文本

将两个文本按列拼接到一起;

Solution

```
paste file1 file2
```

```
1 colin
```

```
2 book
```

默认的分界符是制表符，可以用-d指明分界符:

Solution

```
paste file1 file2 -d ","
```

```
1,colin
```

```
2,book
```

3.9. wc 统计行和字符的工具

Solution

```
$wc -l file // 统计行数
```

```
$wc -w file // 统计单词数
```

```
$wc -c file // 统计字符数
```

3. 文本处理

3.10. sed 文本替换利器

首处替换

Solution

```
sed 's/text/replace_text/' file //替换每一行的第一处匹配的text
```

全局替换

Solution

```
sed 's/text/replace_text/g' file
```

默认替换后，输出替换后的内容，如果需要直接替换原文件,使用-i:

Solution

```
sed -i 's/text/replace_text/g' file
```

3. 文本处理

移除空白行

Solution

```
sed '/^$/d' file
```

变量转换

已匹配的字符串通过标记&来引用.

Solution

```
echo this is en example | sed 's/\w+/[&]/g'  
$>[this] [is] [en] [example]
```

子串匹配标记

第一个匹配的括号内容使用标记 1 来引用

Solution

```
sed 's/hello\([0-9]\)/\1/'
```

3. 文本处理

双引号求值

sed通常用单引号来引用；也可使用双引号，使用双引号后，双引号会对表达式求值：

Solution

```
sed 's/$var/HLLOE/'
```

当使用双引号时，我们可以在sed样式和替换字符串中指定变量；

Solution

eg:

```
p=patten
```

```
r=replaced
```

```
echo "line con a patten" | sed "s/$p/$r/g"
```

```
$>line con a replaced
```

其它示例

字符串插入字符：将文本中每行内容（ABCDEF）转换为 ABC/DEF：

Solution

```
sed 's/^\{3\}/&\/g' file
```

3. 文本处理

3.11. awk 数据流处理工具

awk脚本结构

Solution

```
awk ' BEGIN{ statements } statements2 END{ statements } '
```

工作方式

1. 执行begin中语句块;
2. 从文件或stdin中读入一行, 然后执行statements2, 重复这个过程, 直到文件全部被读取完毕;
3. 执行end语句块;

3. 文本处理

print 打印当前行

使用不带参数的**print**时，会打印当前行

Solution

```
echo -e "line1\nline2" | awk 'BEGIN{print "start"} \
{print } END{ print "End" }'
```

print 以逗号分割时，参数以空格定界；

Solution

```
echo | awk ' {var1 = "v1" ; var2 = "V2"; var3="v3"; \
print var1, var2 , var3; }'
$>v1 V2 v3
```


3. 文本处理

使用-拼接符的方式（""作为拼接符）；

Solution

```
echo | awk ' {var1 = "v1" ; var2 = "V2"; var3="v3"; \  
print var1 "-"var2 "-"var3; }'  
$>v1-V2-v3
```

3. 文本处理

特殊变量: NR NF \$0 \$1 \$2

NR:表示记录数量, 在执行过程中对应当前行号;

NF:表示字段数量, 在执行过程总对应当前行的字段数;

\$0:这个变量包含执行过程中当前行的文本内容;

\$1:第一个字段的文本内容;

\$2:第二个字段的文本内容;

Solution

```
echo -e "line1 f2 f3\n line2 \n line 3" | awk '{print NR:"$0"- "$1"- "$2}'
```

3. 文本处理

打印每一行的第二和第三个字段

Solution

```
awk '{print $2, $3}' file
```

统计文件的行数

Solution

```
awk 'END {print NR}' file
```

累加每一行的第一个字段

Solution

```
echo -e "1\n 2\n 3\n 4\n" | awk 'BEGIN{num = 0 ;\nprint "begin";} {sum += $1;} END {print "==" ; print sum }'
```

3. 文本处理

传递外部变量

Solution

```
var=1000
```

```
echo | awk '{print vara}' vara=$var # 输入来自stdin
```

```
awk '{print vara}' vara=$var file # 输入来自文件
```

用样式对awk处理的行进行过滤

Solution

```
awk 'NR < 5' #行号小于5
```

```
awk 'NR==1,NR==4 {print}' file #行号等于1和4的打印出来
```

```
awk '/linux/' #包含linux文本的行（可以用正则表达式来指定，超级强大）
```

```
awk '!/linux/' #不包含linux文本的行
```

3. 文本处理

设置定界符

使用-F来设置定界符（默认为空格）：

Solution

```
awk -F: '{print $NF}' /etc/passwd
```

读取命令输出

使用getline，将外部shell命令的输出读入到变量cmdout中：

Solution

```
echo | awk '{"grep root /etc/passwd" | getline cmdout; print cmdout}'
```

3. 文本处理

在awk中使用循环

Solution

```
for(i=0;i<10;i++){print $i;}
```

```
for(i in array){print array[i];}
```

eg:以下字符串，打印出其中的时间串:

Solution

```
2015_04_02 20:20:08: mysql connect failed, please check connect info
```

```
$echo '2015_04_02 20:20:08: mysql connect failed, please check connect info'|awk  
-F ":" '{ for(i=1;i<=;i++) printf("%s:",$i)}'
```

```
>2015_04_02 20:20:08: # 这种方式会将最后一个冒号打印出来
```

```
$echo '2015_04_02 20:20:08: mysql connect failed, please check connect info'|awk  
-F:' '{print $1 ":" $2 ":" $3; }'
```

```
>2015_04_02 20:20:08 # 这种方式满足需求
```

3. 文本处理

而如果需要将后面的部分也打印出来(时间部分和后文分开打印):

Solution

```
$echo '2015_04_02 20:20:08: mysql connect failed, please check connect info'|awk  
-F:' '{print $1 ":" $2 ":" $3; print $4;}'  
  
>2015_04_02 20:20:08  
>mysql connect failed, please check connect info
```

以逆序的形式打印行: (tac命令的实现):

Solution

```
seq 9| \  
awk '{lifo[NR] = $0; lno=NR} \  
END{ for(;lno>-1;lno-){print lifo[lno];}  
}'
```

3. 文本处理

awk结合grep找到指定的服务，然后将其kill掉

Solution

```
ps -fe| grep msv8 | grep -v MFORWARD | awk '{print $2}' | xargs kill -9;
```

awk实现head、tail命令

head

Solution

```
awk 'NR<=10{print}' filename
```

tail

Solution

```
awk '{buffer[NR%10] = $0;} END{for(i=0;i<11;i++){ \
print buffer[i %10]}} ' filename
```


3. 文本处理

打印指定列

awk方式实现

Solution

```
ls -lrt | awk '{print $6}'
```

cut方式实现

Solution

```
ls -lrt | cut -f6
```

打印指定文本区域

确定行号

Solution

```
seq 100 | awk 'NR==4,NR==6{print}'
```

3. 文本处理

确定文本

打印处于start_pattern 和end_pattern之间的文本:

Solution

```
awk '/start_pattern/, /end_pattern/' filename
```

示例:

Solution

```
seq 100 | awk '/13/,/15/'
```

```
cat /etc/passwd| awk '/mai.*mail/,/news.*news/'
```

3. 文本处理

awk常用内建函数

`index(string,search_string)`:返回`search_string`在`string`中出现的位置

`sub(regex,replacement_str,string)`:将正则匹配到的第一处内容替换为`replacement_str`;

`match(regex,string)`:检查正则表达式是否能够匹配字符串;

`length(string)`: 返回字符串长度

Solution

```
echo | awk '{ "grep root /etc/passwd" | getline cmdout; print length(cmdout) }'
```

`printf` 类似c语言中的`printf`, 对输出进行格式化:

Solution

```
seq 10 | awk '{printf "->%4s\n", $1}'
```

3. 文本处理

3.12. 迭代文件中的行、单词和字符

1. 迭代文件中的每一行

while 循环法

Solution

```
while read line;
```

```
do
```

```
echo $line;
```

```
done < file.txt
```

改成子shell:

```
cat file.txt | (while read line;do echo $line;done)
```

awk法

Solution

```
cat file.txt| awk '{print}'
```

3. 文本处理

2. 迭代一行中的每一个单词

Solution

```
for word in $line;
```

```
do
```

```
echo $word;
```

```
done
```

3. 文本处理

3. 迭代每一个字符

`$string:start_pos:num_of_chars`: 从字符串中提取一个字符; (`bash`文本切片)

`$#word`:返回变量`word`的长度

Solution

```
for((i=0;i< $#word;i++))  
do  
echo ${word:i:1};  
done
```

以ASCII字符显示文件:

Solution

```
$od -c filename
```

4. 磁盘管理

4.1. 查看磁盘空间

查看磁盘空间利用大小:

Solution

```
df -h
```

-h: human缩写, 以易读的方式显示结果 (即带单位: 比如M/G, 如果不加这个参数, 显示的数字以B为单位)

Solution

```
$df -h
```

```
/opt/app/todeav/config#df -h
```

```
Filesystem Size Used Avail Use% Mounted on
```

```
/dev/mapper/VolGroup00-LogVol00
```

```
2.0G 711M 1.2G 38% /
```

```
/dev/mapper/vg1-lv2 20G 3.8G 15G 21% /opt/applog
```

```
/dev/mapper/vg1-lv1 20G 13G 5.6G 70% /opt/app
```

4. 磁盘管理

查看当前目录所占空间大小:

Solution

```
du -sh
```

-h 人性化显示

-s 递归整个目录的大小

Solution

```
$du -sh
```

```
653M
```

查看当前目录下所有子文件夹排序后的大小:

Solution

```
for i in `ls`; do du -sh $i; done | sort
```

或者:

```
du -sh `ls` | sort
```


4. 磁盘管理

4.2. 打包/ 压缩

在linux中打包和压缩是分两步来实现的;

打包

打包是将多个文件归并到一个文件:

Solution

```
tar -cvf etc.tar /etc <==仅打包，不压缩！
```

- * -c :打包选项
- * -v :显示打包进度
- * -f :使用档案文件

4. 磁盘管理

注：有的系统中指定参数时不需要在前面加上-，直接使用tar xvf

示例：用tar实现文件夹同步，排除部分文件不同步：

Solution

```
tar -exclude '*.svn' -cvf - /path/to/source | ( cd /path/to/target; tar -xf -)
```

压缩

Solution

```
$gzip demo.txt
```

生成 demo.txt.gz

4. 磁盘管理

4.3. 解包/解压缩

解包

Solution

```
tar -xvf demo.tar
```

-x 解包选项

解压后缀为 .tar.gz的文件 1. 先解压缩, 生成**.tar:

Solution

```
$gunzip demo.tar.gz
```

解包:

Solution

```
$tar -xvf demo.tar
```

```
$bzip2 -d demo.tar.bz2
```

4. 磁盘管理

bz2解压:

Solution

```
tar jxvf demo.tar.bz2
```

如果tar 不支持j, 则同样需要分两步来解包解压缩, 使用bzip2来解压, 再使用tar解包:

Solution

```
bzip2 -d demo.tar.bz2
```

```
tar -xvf demo.tar
```

-d decompose,解压缩

tar解压参数说明:

- * -z 解压gz文件
- * -j 解压bz2文件
- * -J 解压xz文件

4. 磁盘管理

4.4. 总结

查看磁盘空间 `df -h`

查看目录大小 `du -sh`

打包 `tar -cvf`

解包 `tar -xvf`

压缩 `gzip`

解压缩 `gunzip bzip`

5. 进程管理工具

这一节我们介绍进程管理工具；

使用进程管理工具，我们可以查询程序当前的运行状态，或终止一个进程；

任何进程都与文件关联；我们会用到lsdf工具（list opened files），作用是列举系统中已经被打开的文件。在linux环境中，任何事物都是文件，设备是文件，目录是文件，甚至sockets也是文件。用好lsdf命令，对日常的linux管理非常有帮助。

5.1. 查询进程

查询正在运行的进程信息

Solution

```
$ps -ef
```

5. 进程管理工具

eg:查询归属于用户colin115的进程

Solution

```
$ps -ef | grep colin115
```

```
$ps -lu colin115
```

查询进程ID（适合只记得部分进程字段）

Solution

```
$pgrep 查找进程
```

eg:查询进程名中含有re的进程

```
[/home/weber#]pgrep -l re
```

```
2 kthreadd
```

```
28 ecryptfs-kthrea
```

```
29515 redis-server
```

5. 进程管理工具

以完整的格式显示所有的进程

Solution

```
$ps -ajx
```

显示进程信息，并实时更新

Solution

```
$top
```

查看端口占用的进程状态:

Solution

```
lsof -i:3306
```

查看用户username的进程所打开的文件

Solution

```
$lsof -u username
```


5. 进程管理工具

查询init进程当前打开的文件

Solution

```
$lsOF -c init
```

查询指定的进程ID(23295)打开的文件:

Solution

```
$lsOF -p 23295
```

查询指定目录下被进程开启的文件（使用+D 递归目录）：

Solution

```
$lsOF +d mydir1/
```

5. 进程管理工具

5.2. 终止进程

杀死指定PID的进程 (PID为Process ID)

Solution

```
$kill PID
```

杀死相关进程

Solution

```
$kill -9 3434
```

杀死job工作 (job为job number)

Solution

```
$kill %job
```

5. 进程管理工具

5.3. 进程监控

查看系统中使用CPU、使用内存最多的进程；

Solution

\$top

(->)P

输入top命令后，进入到交互界面；接着输入字符命令后显示相应的进程状态：

对于进程，平时我们最常想知道的就是哪些进程占用CPU最多，占用内存最多。以下两个命令就可以满足要求：

Solution

P: 根据CPU使用百分比大小进行排序。

M: 根据驻留内存大小进行排序。

i: 使top不显示任何闲置或者僵死进程。

这里介绍最使用的几个选项,对于更详细的使用, 详见 **top linux**下的任务管理器；

5. 进程管理工具

5.4. 分析线程栈

使用命令 `pmap`，来输出进程内存的状况，可以用来分析线程堆栈；

Solution

```
$pmap PID
```

```
eg:
```

```
[/home/weber#]ps -fe| grep redis
```

```
weber 13508 13070 0 08:14 pts/0 00:00:00 grep --color=auto redis
```

```
weber 29515 1 0 2013 ? 02:55:59 ./redis-server redis.conf
```

```
[/home/weber#]pmap 29515
```

```
29515: ./redis-server redis.conf
```

```
08048000 768K r-x- /home/weber/soft/redis-2.6.16/src/redis-server
```

```
08108000 4K r— /home/weber/soft/redis-2.6.16/src/redis-server
```

```
08109000 12K rw— /home/weber/soft/redis-2.6.16/src/redis-server
```

5. 进程管理工具

5.5. 综合运用

将用户 colin115 下的所有进程名以 av_ 开头的进程终止:

Solution

```
ps -u colin115 | awk '/av_/ {print "kill -9 " $1}' | sh
```

将用户 colin115 下所有进程名中包含 HOST 的进程终止:

Solution

```
ps -fe| grep colin115|grep HOST |awk '{print $2}' | xargs kill -9;
```

5. 进程管理工具

5.6. 总结

- * ps
- * top
- * lsof
- * kill
- * pmap

6. 性能监控

在使用操作系统的过程中，我们经常需要查看当前的性能如何，需要了解CPU、内存和硬盘的使用情况；本节介绍的这几个工具能满足日常工作要求；

6.1. 监控CPU

查看CPU使用率

Solution

```
$sar -u
```

eg:

```
$sar -u 1 2
```

```
[/home/weber#]sar -u 1 2
```

```
Linux 2.6.35-22-generic-pae (MyVPS) 06/28/2014 _i686_ (1 CPU)
```

```
09:03:59 AM CPU %user %nice %system %iowait %steal %idle
```

```
09:04:00 AM all 0.00 0.00 0.50 0.00 0.00 99.50
```

```
09:04:01 AM all 0.00 0.00 0.00 0.00 0.00 100.00
```

后面的两个参数表示监控的频率，比如例子中的1和2，表示每秒采样一次，总共采样2次；

6. 性能监控

查看CPU平均负载

Solution

```
$sar -q 1 2
```

sar指定-q后，就能查看运行队列中的进程数、系统上的进程大小、平均负载等；

6. 性能监控

6.2. 查询内存

查看内存使用状况 `sar`指定`-r`之后，可查看内存使用状况；

Solution

```
$sar -r 1 2
```

```
09:08:48 AM kbmemfree kbmemused %memused kbbuffers kbcached kbcommit  
%commit kbactive kbinact
```

```
09:08:49 AM 17888 359784 95.26 37796 73272 507004 65.42 137400 150764
```

```
09:08:50 AM 17888 359784 95.26 37796 73272 507004 65.42 137400 150764
```

```
Average: 17888 359784 95.26 37796 73272 507004 65.42 137400 150764
```

查看内存使用量

Solution

```
$free -m
```

6. 性能监控

6.3. 查询页面交换

查看页面交换发生状况 页面发生交换时，服务器的吞吐量会大幅下降；服务器状况不良时，如果怀疑因为内存不足而导致了页面交换的发生，可以使用`sar -W`这个命令来确认是否发生了大量的交换；

Solution

```
$sar -W 1 3
```

6. 性能监控

6.4. 查询硬盘使用

查看磁盘空间利用情况

Solution

```
$df -h
```

查询当前目录下空间使用情况

Solution

```
$du -sh -h是人性化显示 s是递归整个目录的大小
```

查看该目录下所有文件夹的排序后的大小

Solution

```
for i in `ls`; do du -sh $i; done | sort
```

或者

```
du -sh `ls`
```

6. 性能监控

6.5. 综合应用

当系统中sar不可用时，可以使用以下工具替代：linux下有 vmstat、Unix系统有prstat

eg: 查看cpu、内存、使用情况: vmstat n m (n 为监控频率、m为监控次数)

Solution (/home/weber#)

```
vmstat 1 3
```

```
procs ————memory——— —swap— —io— -system— —cpu—
```

```
r b swpd free buff cache si so bi bo in cs us sy id wa
```

```
0 0 86560 42300 9752 63556 0 1 1 1 0 0 0 0 99 0
```

```
1 0 86560 39936 9764 63544 0 0 0 52 66 95 5 0 95 0
```

```
0 0 86560 42168 9772 63556 0 0 0 20 127 231 13 2 84 0
```

6. 性能监控

使用watch 工具监控变化 当需要持续的监控应用的某个数据变化时，watch工具能满足要求； 执行watch命令后，会进入到一个界面，输出当前被监控的数据，一旦数据变化，便会高亮显示变化情况；

eg: 操作redis时，监控内存变化:

Solution

```
$watch -d -n 1 './redis-cli info | grep memory'
```

(以下为watch工具中的界面内容，一旦内存变化，即实时高亮显示变化)

```
Every 1.0s: ./redis-cli info | grep memory Mon Apr 28 16:10:36 2014
```

```
used_memory:45157376
```

```
used_memory_human:43.07M
```

```
used_memory_rss:47628288
```

```
used_memory_peak:49686080
```

```
used_memory_peak_human:47.38M
```

6. 性能监控

6.6. 总结

- * top
- * sar
- * free
- * watch

7. 网络工具

7.1. 查询网络服务和端口

`netstat` 命令用于显示各种网络相关信息，如网络连接，路由表，接口状态 (Interface Statistics)，masquerade 连接，多播成员 (Multicast Memberships) 等等。

列出所有端口 (包括监听和未监听的):

Solution

```
$netstat -a
```

列出所有 tcp 端口:

Solution

```
$netstat -at
```

列出所有有监听的服务状态:

Solution

```
$netstat -l
```

7. 网络工具

使用netstat工具查询端口:

Solution

```
$netstat -antp | grep 6379
```

```
tcp 0 0 127.0.0.1:6379 0.0.0.0:* LISTEN 25501/redis-server
```

```
$ps 25501
```

```
PID TTY STAT TIME COMMAND
```

```
25501 ? Ssl 28:21 ./redis-server ./redis.conf
```

lsof (list open files) 是一个列出当前系统打开文件的工具。在linux环境下，任何事物都以文件的形式存在，通过文件不仅仅可以访问常规数据，还可以访问网络连接和硬件。所以如传输控制协议 (TCP) 和用户数据报协议 (UDP) 套接字等； 在查询网络端口时，经常会用到这个工具。

7. 网络工具

查询7902端口现在运行什么程序:

Solution

```
#分为两步
```

```
#第一步，查询使用该端口的进程的PID;
```

```
$lsof -i:7902
```

```
COMMAND PID USER FD TYPE DEVICE SIZE NODE NAME
```

```
WSL 30294 tuapp 4u IPv4 447684086 TCP 10.6.50.37:tnos-dp (LISTEN)
```

```
#查到30294
```

```
#使用ps工具查询进程详情:
```

```
$ps -fe | grep 30294
```

```
tdev5 30294 26160 0 Sep10 ? 01:10:50 tdesl -k 43476
```

```
root 22781 22698 0 00:54 pts/20 00:00:00 grep 11554
```

7. 网络工具

7.2. 网络路由

查看路由状态:

Solution

```
$route -n
```

发送ping包到地址IP:

Solution

```
$ping IP
```

探测前往地址IP的路由路径:

Solution

```
$tracert IP
```

7. 网络工具

DNS查询，寻找域名domain对应的IP:

Solution

```
$host domain
```

反向DNS查询:

Solution

```
$host IP
```

7.3. 镜像下载

直接下载文件或者网页:

Solution

```
$wget url
```

常用选项:

- * `-limit-rate` : 下载限速
- * `-o`: 指定日志文件; 输出都写入日志;
- * `-c`: 断点续传

7.4. ftp sftp lftp ssh

SSH登陆:

Solution

```
$ssh ID@host
```

ssh登陆远程服务器host, ID为用户名。

ftp/sftp文件传输:

Solution

```
$sftp ID@host
```

7. 网络工具

登陆服务器host，ID为用户名。sftp登陆后，可以使用下面的命令进一步操作：

- * `get filename #` 下载文件
- * `put filename #` 上传文件
- * `ls #` 列出host上当前路径的所有文件
- * `cd #` 在host上更改当前路径
- * `lls #` 列出本地主机上当前路径的所有文件
- * `lcd #` 在本地主机更改当前路径

lftp同步文件夹(类似rsync工具):

Solution

```
lftp -u user:pass host
```

```
lftp user@host: > mirror -n
```

7.5. 网络复制

将本地localpath指向的文件上传到远程主机的path路径:

Solution

```
$scp localpath ID@host:path
```

以ssh协议，遍历下载path路径下的整个文件系统，到本地的localpath:

Solution

```
$scp -r ID@site:path localpath
```

7.6. 总结

- * netstat
- * lsof
- * route
- * ping
- * host
- * wget
- * sftp
- * scp

8. 用户管理工具

8.1. 用户

添加用户

Solution

```
$useradd -m username
```

该命令为用户创建相应的帐号和用户目录/home/username;

用户添加之后，设置密码:

密码以交互方式创建:

Solution

```
$passwd username
```

8. 用户管理工具

删除用户

Solution

```
$userdel -r username
```

不带选项使用 `userdel`，只会删除用户。用户的家目录将仍会在 `/home` 目录下。要完全的删除用户信息，使用 `-r` 选项；

帐号切换 登录帐号为 `userA` 用户状态下，切换到 `userB` 用户帐号工作：

Solution

```
$su userB
```

进入交互模型，输入密码授权进入；

8. 用户管理工具

8.2. 用户的组

将用户加入到组

默认情况下，添加用户操作也会相应的增加一个同名的组，用户属于同名组； 查看当前用户所属的组：

Solution

```
$groups
```

一个用户可以属于多个组，将用户加入到组：

Solution

```
$usermod -G groupNname username
```

变更用户所属的根组(将用加入到新的组，并从原有的组中除去)：

Solution

```
$usermod -g groupName username
```

8. 用户管理工具

查看系统所有组

系统的所有用户及所有组信息分别记录在两个文件中：`/etc/passwd`，`/etc/group` 默认情况下这两个文件对所有用户可读：

查看所有用户及权限：

Solution

```
$more /etc/passwd
```

查看所有的用户组及权限：

Solution

```
$more /etc/group
```

8.3. 用户权限

使用`ls -l`可查看文件的属性字段，文件属性字段总共有10个字母组成，第一个字母表示文件类型，如果这个字母是一个减号“-”，则说明该文件是一个普通文件。字母“d”表示该文件是一个目录，字母“d”，是directory(目录)的缩写。后面的9个字母为该文件的权限标识，3个为一组，分别表示文件所属用户、用户所在组、其它用户的读写和执行权限；例如：

Solution (/home/weber#)

```
ls -l /etc/group
```

```
-rwxrw-r- colin king 725 2013-11-12 15:37 /home/colin/a
```

表示这个文件对文件拥有者colin这个用户可读写、可执行；对colin所在的组（king）可读可写；对其它用户只可读；

8. 用户管理工具

更改读写权限

使用chmod命令更改文件的读写权限，更改读写权限有两种方法，一种是字母方式，一种是数字方式

字母方式:

Solution

```
$chmod userMark(+|-)PermissionsMark
```

userMark取值:

- * u: 用户
- * g: 组
- * o: 其它用户
- * a: 所有用户

8. 用户管理工具

PermissionsMark取值:

- * r: 读
- * w: 写
- * x: 执行

例如:

Solution

`$chmod a+x main` 对所有用户给文件`main`增加可执行权限

`$chmod g+w blogs` 对组用户给文件`blogs`增加可写权限

8. 用户管理工具

数字方式:

数字方式直接设置所有权限，相比字母方式，更加简洁方便；

使用三位八进制数字的形式来表示权限，第一位指定属主的权限，第二位指定组权限，第三位指定其他用户的权限，每位通过4(读)、2(写)、1(执行)三种数值的和来确定权限。如6(4+2)代表有读写权，7(4+2+1)有读、写和执行的权限。

例如:

Solution

`$chmod 740 main` 将`main`的用户权限设置为`rwxr`——

8. 用户管理工具

更改文件或目录的拥有者

Solution

```
$chown username dirOrFile
```

使用-R选项递归更改该目下所有文件的拥有者:

Solution

```
$chown -R weber server/
```

8.4. 环境变量

`bashrc`与`profile`都用于保存用户的环境信息，`bashrc`用于交互式`non-login`shell，而`profile`用于交互式`login` shell。

`/etc/profile`，`/etc/bashrc` 是系统全局环境变量设定

`/.profile`，`/.bashrc`用户目录下的私有环境变量设定

当登入系统获得一个shell进程时，其读取环境设置脚本分为三步：

1. 首先读入的是全局环境变量设置文件`/etc/profile`，然后根据其内容读取额外的文档，如`/etc/profile.d`和`/etc/inputrc`
2. 读取当前登录用户Home目录下的文件 `/.bash_profile`，其次读取 `/.bash_login`，最后读取 `/.profile`，这三个文档设定基本上是一样的，读取有优先关系
3. 读取 `/.bashrc`

`/.profile`与 `/.bashrc`的区别:

- * 这两者都具有个性化定制功能
- * `/.profile`可以设定本用户专有的路径, 环境变量, 等, 它只能登入的时候执行一次
- * `/.bashrc`也是某用户专有设定文档, 可以设定路径, 命令别名, 每次shell script的执行都会使用它一次

8. 用户管理工具

例如，我们可以在这些环境变量中设置自己经常进入的文件路径，以及命令的快捷方式：

Solution

```
.bashrc  
  
alias m='more'  
  
alias cp='cp -i'  
  
alias lsl='ls -lrt'  
  
alias lm='ls -al|more'  
  
log=/opt/applog/common_dir  
unit=/opt/app/unittest/common  
  
.bash_profile  
  
. /opt/app/tuxapp/openav/config/setenv.prod.sh.linux  
export PS1='$PWD#'
```

通过上述设置，我们进入log目录就只需要输入cd \$log即可；

8.5. 总结

- * useradd
- * passwd
- * userdel
- * usermod
- * chmod
- * chown
- * .bashrc
- * .bash_profile

9. 系统管理及IPC资源管理

9.1. 系统管理

查询系统版本

查看Linux系统版本:

Solution

```
$uname -a
```

```
$lsb_release -a
```

查看Unix系统版本: 操作系统版本:

Solution

```
$more /etc/release
```

9. 系统管理及IPC资源管理

查询硬件信息

查看CPU使用情况:

Solution

```
$sar -u 5 10
```

查询CPU信息:

Solution

```
$cat /proc/cpuinfo
```

查看CPU的核的个数:

Solution

```
$cat /proc/cpuinfo | grep processor | wc -l
```

查看内存信息:

Solution

```
$cat /proc/meminfo
```

9. 系统管理及IPC资源管理

显示内存page大小（以KByte为单位）：

Solution

```
$pagesize
```

显示架构：

Solution

```
$arch
```


设置系统时间

显示当前系统时间:

Solution

```
$date
```

设置系统日期和时间(格式为2014-09-15 17:05:00):

Solution

```
$date -s 2014-09-15 17:05:00
```

```
$date -s 2014-09-15
```

```
$date -s 17:05:00
```

9. 系统管理及IPC资源管理

设置时区:

Solution

选择时区信息。命令为: `tzselect`

根据系统提示, 选择相应的时区信息。

强制把系统时间写入CMOS (这样, 重启后时间也正确了):

Solution

```
$clock -w
```

格式化输出当前日期时间:

Solution

```
$date +%Y%m%d.%H%M%S
```

```
>20150512.173821
```

9. 系统管理及IPC资源管理

9.2. IPC资源管理

IPC资源查询

查看系统使用的IPC资源:

Solution

```
$ipcs
```

```
—— Shared Memory Segments ——
```

```
key shmid owner perms bytes nattch status
```

```
—— Semaphore Arrays ——
```

```
key semid owner perms nsems
```

```
0x00000000 229376 weber 600 1
```

```
—— Message Queues ——
```

```
key msqid owner perms used-bytes messages
```

查看系统使用的IPC共享内存资源:

Solution

```
$ipcs -m
```

9. 系统管理及IPC资源管理

查看系统使用的IPC队列资源:

Solution

```
$ipcs -q
```

查看系统使用的IPC信号量资源:

Solution

```
$ipcs -s
```

应用示例: 查看IPC资源被谁占用

有个IPCKEY: 51036 , 需要查询其是否被占用;

9. 系统管理及IPC资源管理

1. 首先通过计算器将其转为十六进制:

Solution

```
51036 -> c75c
```

2. 如果知道是被共享内存占用:

Solution

```
$ipcs -m | grep c75c
```

```
0x0000c75c 40403197 tdea3 666 536870912 2
```

3. 如果不确定, 则直接查找:

Solution

```
$ipcs | grep c75c
```

```
0x0000c75c 40403197 tdea3 666 536870912 2
```

```
0x0000c75c 5079070 tdea3 666 4
```

检测和设置系统资源限制

显示当前所有的系统资源limit 信息:

Solution

```
$ulimit -a
```

对生成的 core 文件的大小不进行限制:

Solution

```
$ulimit -c unlimited
```

9.3. 总结

- * uname
- * sar
- * arch
- * date
- * ipcs
- * ulimit

1. 安装

Mac 安装

Mac用户安装 `python` 会比较方便,直接到官网下载安装包, 下载自己需要的版本, 默认路径安装即可。

之后的某节讲到如何给Mac中的`python`安装其他模块, 比如比较常用的`numpy` 或者`matplotlib`.

1. 安装

Windows 安装

请到官网下载需要的版本的安装包， 下载所需(注意自己的系统是32位还是64位)， 安装路径最好选择默认， 不然对于新手容易出现各种问题。

Windows 安装附加要点: 设置环境变量: 1.找到安装路径, 默认 C:\Users\你的用户名\AppData\Local\Programs\Python\Python35-32 粘贴路径 2.我的电脑 - 属性 - 高级 - 环境变量 - 系统变量中的PATH为 (复制路径): C:\Users\你的用户名\AppData\Local\Programs\Python\Python35-32;

pip3 设置环境变量: C:\Users\你的用户名\AppData\Local\Programs\Python\Python35-32\Scripts;

1. 安装

检查安装是否成功

打开idle, `print(1)` 如果系统输出1,则表明安装成功.

Solution

```
»> print(1)
```

```
1
```

```
»>
```

2. 基本使用

print 字符串

python 中 print 字符串 要加"或者"

Solution

```
»> print('hello world')
```

```
hello world
```

```
»> print("hello world 2")
```

```
hello world 2
```

2. 基本使用

print 字符串叠加

可以使用 + 将两个字符串链接起来, 如以下代码.

Solution

```
»> print('Hello world'+ ' Hello Hong Kong')
```

```
Hello world Hello Hong Kong
```

2. 基本使用

简单运算

可以直接print 加法+,减法-,乘法*,除法/. 注意: 字符串不可以直接和数字相加, 否则出现错误。

Solution

```
»> print(1+1)
```

```
2
```

```
»> print(12/4)
```

```
3.0
```

```
»> print('iphone'+4) #字符串不可以直接和数字相加
```

```
Traceback (most recent call last):
```

```
File "<pyshell#10>", line 1, in <module>
```

```
print('iphone'+4)
```

```
TypeError: Can't convert 'int' object to str implicitly
```

2. 基本使用

`int()` 和 `float()`；当 `int()` 一个浮点型数时，`int` 会保留整数部分，比如 `int(1.9)`，会输出 1，而不是四舍五入。

Solution

```
»> print(int('2')+3) #int为定义整数型
```

```
5
```

```
»> print(int(1.9)) #当int一个浮点型数时，int会保留整数部分
```

```
1
```

```
»> print(float('1.2')+3) #float()是浮点型，可以把字符串转换成小数
```

```
4.2
```

2. 基本使用

基本的加减乘除

python可以直接运算数字，也可以加print 进行运算.

Solution

```
»> 1+1
```

```
2
```

```
»> 2-1
```

```
1
```

```
»> 2*3
```

```
6
```

```
»> 4/3
```

```
1.3333333333333333
```

2. 基本使用

^ 与 **

python当中^符号，区别于Matlab，在python中，^用两个**表示，如3的平方为3**2，**3表示立方，**4表示4次方，依次类推

Solution

```
»> 3**2 # **2 表示2次方
```

```
9
```

```
»> 3**3 # **3 表示3次方
```

```
27
```

```
»> 3**4
```

```
81
```


2. 基本使用

取余数 %

余数符号为"%",见代码.

Solution

```
» > 8%3
```

```
2
```

2. 基本使用

变量 `variable`

自变量命名规则

可以将一个数值，或者字符串赋值给自变量，如`apple=1` 中，`apple`为自变量的名称，`1`为自变量的值。也可以将字符串赋值给自变量 `apple='iphone7 plus'`

Solution

```
apple=1 #赋值 数字
```

```
print(apple)
```

```
1
```

```
apple='iphone 7 plus' #赋值 字符串
```

```
print(apple)
```

```
iphone 7 plus
```

2. 基本使用

如果需要用多个单词来表示自变量，需要加下划线，如`apple_2016='iphone 7 plus'` 请看代码

Solution

```
apple_2016='iphone 7 plus and new macbook'  
print(apple_2016)  
iphone 7 plus and new macbook
```

一次定义多个自变量 `a,b,c=1,2,3`。

Solution

```
a,b,c=11,12,13  
print(a,b,c)  
11 12 13
```

3. while 和 for 循环

在 Python 语言中用来控制循环的主要有两个句法，`while` 和 `for` 语句，本讲将简单介绍 `while` 句法的使用。

基本使用

`while` 语句同其他编程语言中 `while` 的使用方式大同小异，主要结构如下

Solution

```
while condition:  
....expressions
```

其中 `condition` 为判断条件，在 Python 中就是 `True` 和 `False` 其中的一个，如果为 `True`，那么将执行 `exexpressions` 语句，否则将跳过该 `while` 语句块接着往下执行。

3. while 和 for 循环

实例

比如要打印出 0 - 9 的所有数据,

Solution

```
condition = 0
while condition < 10:
....print(condition)
....condition = condition + 1
```

输出的结果将是 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 第一行设置 `condition` 的初始值为 0, 在进行 `while` 判断的时候 `0 < 10` 为 `True`, 将会执行 `while` 内部的代码, 首先打印出该值, 然后将 `condition` 值加 1, 至此将完成一次循环; 再 `condition` 的值与 10 进行比较, 仍然为 `True`, 重复如上过程, 至到 `condition` 等于 10 后, 不满足 `condition < 10` 的条件 (`False`), 将不执行 `while` 内部的内容 所以 10 不会被打印。

3. while 和 for 循环

注意点

在使用 `while` 句法的时候一定要注意在循环内部一定要修改判断条件的值，否则程序的 `while` 部分 将永远执行下去。

Solution

```
while True:  
    ....print("I'm True")
```

如果这样做的话，程序将一直打印出 `I'm True`，要停止程序，使用 `ctrl + c` 终止程序。

高级主题

在 Python 中除了常规比较操作

- * 小于 (`<`)
- * 大于 (`>`)
- * 不大于 (`<=`)
- * 不小于 (`>=`)
- * 等于 (`==`)
- * 不等于 (`!=`)

会返回 `True` 和 `False`值，例如其他也会返回 `True` 和 `False`

3. while 和 for 循环

1 数字

整数和浮点数也能进行 Boolean 数据操作，具体规则，如果该值等于 0 或者 0.0 将会返回 False 其余的返回 True

Solution

```
condiiton = 10
while condiiton:
    ....print(condiiton)
    ....condiiton -= 1
```

输出的结果将会是 10, 9, 8, 7, 6, 5, 4, 3, 2, 1, 在这里 condition 在 while 语句中，如果该值 大于0，那么将会返回为 True,执行循环内部语句，直至 condition 等于0，返回 False。

2 None 类型

如果 while 后面接着的语句数据类型 None, 将会返回 False。

3. while 和 for 循环

3 集合类型

在 Python 中集合类型有 list、tuple、dict 和 set 等，如果该集合对象作为 while 判断语句，如果集合中的元素数量为 0，那么将会返回 False，否则返回 True。

Solution

```
a = range(10)
while a:
    ....print(a[-1])
    ....a = a[:-1]
```

上述程序将会返回 9, 8, 7, 6, 5, 4, 3, 2, 1, 0，程序首先判断列表是否空，如果不为空，则打印出最后一个内容，然后使用切片操作去掉最后一个元素，并更新列表；如此重复，直至列表为空。

3. while 和 for 循环

for 循环

基本使用

不同编程语言都有 for 语言，比如 C# 语言中的 foreach, Java 语言中的 for, 在 Python 中的基本使用方法如下。

Solution

```
for item in sequence:  
    ....expressions
```

sequence 为可迭代的对象，item 为序列中的每个对象。

实例

Solution

```
example_list = [1,2,3,4,5,6,7,12,543,876,12,3,2,5]  
for i in example_list:  
    ....print(i)
```

3. while 和 for 循环

输出的结果为 1,2,3,4,5,6,7,12,543,876,12,3,2,5, 内容依次为 `example_list` 中的每一个元素 注意 Python 是使用缩进表示程序的结构, 如果程序这样编写,

Solution

```
example_list = [1,2,3,4,5,6,7,12,543,876,12,3,2,5]
for i in example_list:
    ....print(i)
    ....print('inner of for')
print('outer of for')
```

那么每次循环都会输出 inner of for,在循环结束后, 输出 outer of for 一次。

3. while 和 for 循环

高级主题

4.1 内置集合

Python 共内置了 list、tuple、dict 和 set 四种基本集合，每个集合对象都能够迭代。

tuple 类型

Solution

```
tup = ('python', 2.7, 64)
```

```
for i in tup:
```

```
....print(i)
```

程序将以此按行输出 'python', 2.7 和 64。

3. while 和 for 循环

dictionary 类型

Solution

```
dic = {} dic['lan'] = 'python' dic['version'] = 2.7 dic['platform'] = 64 for key in dic:
....print(key, dic[key])
```

输出的结果为: platform 64, lan python, version 2.7, 字典在迭代的过程中将 key 作为可迭代的对象返回。注意字典中 key 是乱序的, 也就是说和插入的顺序是不一致的。如果想要使用顺序一致的字典, 请使用 collections 模块中的 OrderedDict 对象。

set 类型

Solution

```
s = set(['python', 'python2', 'python3', 'python']) for item in s: ....print(item)
```

将会输出 python, python3, python2 set 集合将会去除重复项, 注意输出的结果也不是按照输入的顺序。

3. while 和 for 循环

4.2 迭代器

Python 中的 for 句法实际上实现了设计模式中的迭代器模式，所以我们自己也可以按照迭代器的要求自己生成迭代器对象，以便在 for 语句中使用。只要类中实现了 `__iter__` 和 `next` 函数，那么对象就可以在 for 语句中使用。现在创建 Fibonacci 迭代器对象，

Solution

```
# define a Fib class  
class Fib(object):  
    ....def __init__(self, max):  
        .....self.max = max  
        .....self.n, self.a, self.b = 0, 0, 1  
    ....def __iter__(self):  
        .....return self
```

3. while 和 for 循环

Solution

```
....def __next__(self):  
.....if self.n < self.max:  
.....r = self.b  
.....self.a, self.b = self.b, self.a + self.b  
.....self.n = self.n + 1  
.....return r  
.....raise StopIteration()  
# using Fib object  
for i in Fib(5):  
....print(i)
```

将会输出前 5 个 Fibonacci 数据 1, 1, 2, 3, 5

3. while 和 for 循环

4.3 生成器

除了使用迭代器以外，Python 使用 `yield` 关键字也能实现类似迭代的效果，`yield` 语句每次 执行时，立即返回结果给上层调用者，而当前的状态仍然保留，以便迭代器下一次循环调用。

Solution

```
def fib(max):  
    ....a, b = 0, 1  
    ....while max:  
        .....r = b  
        .....a, b = b, a+b  
        .....max -= 1  
        .....yield r  
for i in fib(5):  
    ....print(i)
```

将会输出前 5 个 Fibonacci 数据 1, 1, 2, 3, 5

4. if 判断

if 判断

基本使用

与其他编程语言中的 if 语句一样，使用方法如下

Solution

```
if condition:
```

```
....expressions
```

如果 `condition` 的值为 `True`,将会执行 `expressions` 语句的内容，否则将跳过该语句往下执行。

4. if 判断

实例

Solution

```
x,y,z = 1,2,3
```

```
if x < y:
```

```
....print('x is less than y')
```

上述代码中，if 语句的条件为 $x < y$ 为 **True**，那么将执行条件内部语句，程序将输出 x is less than y 。当我们将代码修改为一下

Solution

```
if x < y < z:
```

```
....print('x is less than y, and y is less than z')
```

在这里的条件变成了 $x < y < z$ ，其相当于 $x < y$ and $y < z$ ，如果 and 两边的条件都为 **True** 那么才会返回 **True**。注意这个用法是 python 语言特有，不鼓励大家写出这样的代码，以便其他语言的程序员能够看懂你的代码。

4. if 判断

注意点

在 python 语言中等号的判断使用 `==` 而不是 `=`, 因为后一种是赋值语句。

Solution

```
x,y,z = 1,2,3 if x = y: ...print('x is equal to y')
```

如果这样写的话, 是有句法错误的, 程序将无法执行。当然如果是从 C/C++ 语言转过来的同学, 刚才那一句是非常熟悉的, 也是我们经常错误的来源。

修改如下

Solution

```
x,y,z = 2,2,0  
if x == y:  
...print('x is equal to y')
```

因为 `x` 和 `y` 都等于2, 所以将会输出 `x is equal to y`。

4. if 判断

if else 判断

基本使用

Solution

```
if condition:
```

```
...true_expressions
```

```
else:
```

```
...false_expressions
```

当 if 判断条件为 True, 执行 true_expressions 语句; 如果为 False, 将执行 else 的内部
的 false_expressions。

4. if 判断

实例

Solution

```
x,y,z = 1,2,3
```

```
if x > y:
```

```
....print('x is greater than y')
```

```
else:
```

```
....print('x is less or equal to y')
```

在这个例子中，因为 $x > y$ 将会返回 `False`，输出 `x is less or equal to y`

Solution

```
x,y,z = 4,2,3
```

```
if x > y:
```

```
....print('x is greater than y')
```

```
else:
```

```
....print('x is less or equal y')
```

在这里，因为 `condition` 条件为 `True`，那么将会输出 `x is greater than y`。

4. if 判断

高级主题

对于从其他编程语言转过来的同学一定非常想知道 python 语言中的三目操作符怎么使用，很遗憾的是 python 中并没有类似 `condition ? value1 : value2` 三目操作符。然后现实中很多情况下我们只需要简单的判断 来确定返回值，但是冗长的 if-else 语句似乎与简单的 python 哲学不一致。别担心，python 可以通过 if-else 的行内表达式完成类似的功能。

Solution

```
var = var1 if condition else var2
```

可以这么理解上面这段语句，如果 condition 的值为 True，那么将 var1 的值赋给 var；如果为 False 则将 var2 的值赋给 var。

Solution

```
worked = True  
result = 'done' if worked else 'not yet'  
print(result)
```

首先判断如果 work 为 True,那么将 done 字符串赋给 result，否则将 not yet 赋给 result。结果将输出 done。

4. if 判断

if elif else 判断

基本使用

Solution

```
if condition1:
```

```
....true1_expressions
```

```
elif condition2:
```

```
....true2_expressions
```

```
elif condtion3:
```

```
....true3_expressions
```

```
else:
```

```
....else_expressions
```

如果有多个判断条件，那可以通过 `elif` 语句添加多个判断条件，一旦某个条件为 `True`，那么将执行对应的 `expression`。并在之代码执行完毕后跳出该 `if-elif-else` 语句块，往下执行。

4. if 判断

实例

Solution

```
x,y,z = 4,2,3
```

```
if x > 1: ....print ('x > 1') elif x < 1: ....print('x < 1') else: ....print('x = 1')  
print('finish')
```

因为 $x = 4$ 那么满足 if 的条件，则将输出 $x > 1$ 并且跳出整个 if-elif-else 语句块，那么紧接着输出 finish。如果将 $x = -2$ 那么将满足 elif $x < 1$ 这个条件，将输出 $x < 1$, finish。

5. 定义功能

def 函数

如果我们用代码实现了一个小功能，但想要在程序代码中重复使用，不能在代码中到处粘贴这些代码，因为这样做违反了软件工程中 DRY原则。Python 提供了函数功能，可以将我们这部分功能抽象成一个函数以方便程序调用，或者提供给其他模块使用。

基本使用

Solution

```
def function_name(parameters):  
    ...expressions
```

Python 使用 `def` 开始函数定义，紧接着是函数名，括号内部为函数的参数，内部为函数的具体功能实现代码，如果想要函数有返回值，在 `expressions` 中的逻辑代码中用 `return` 返回。

5. 定义功能

实例

Solution

```
def function():  
....print('This is a function')  
....a = 1+2  
....print(a)
```

上面我们定义了一个名字为 `function` 的函数，函数没有接收参数，所以括号内部为空，紧接着就是函数的功能代码。如果执行该脚本，发现并没有输出任何输出，因为我们只定义了函数，而并没有执行函数。这时我们在 `Python` 命令提示符中输入函数调用 `function()`，注意这里调用函数的括号不能省略。那么函数内部的功能代码将会执行，输出结果：

Solution

```
This is a function  
3
```

5. 定义功能

如果我们想要在脚本中调用的脚本，只需要在脚本中最后添加函数调用语句

Solution

```
function()
```

那么在执行脚本的时候，将会执行函数。

5. 定义功能

函数参数

我们在使用的调用函数的时候，想要指定一些变量的值在函数中使用，那么这些变量就是函数的参数，函数调用的时候，传入即可。

基本使用

Solution

```
def function_name(parameters):  
    ....expressions
```

`parameters` 的位置就是函数的参数，在调用的时候传入即可。

5. 定义功能

实例

Solution

```
def func(a, b):  
    ...c = a+b  
    ...print('the c is ', c)
```

在这里定义的一个函数，其参数就是两个数值，函数的功能就是把两个参数加起来。运行脚本后，在 Python 提示符内调用函数 `func`，如果不指定参数 `func()`，那么将会出错；输出 `func(1, 2)`，将 `a=1`, `b=2` 传入函数，输出 `the c is 3`。所以在调用函数时候，参数个数和位置一定要按照函数定义。如果我们忘记了函数的参数的位置，只知道各个参数的名字，可以在函数调用的过程中给指明特定的参数 `func(a=1, b=2)`，这样的话，参数的位置将不受影响，所以 `func(b=2,a=1)`是同样的效果。

5. 定义功能

函数默认参数

我们在定义函数时有时候有些参数在大部分情况下是相同的，只不过为了提高函数的适用性，提供了一些备选的参数，为了方便函数调用，我们可以将这些参数设置为默认参数，那么该参数在函数调用过程中可以不需要明确给出。

基本使用

Solution

```
def function_name(para_1,...,para_n=defau_n,..., para_m=defau_m):  
    ....expressions
```

函数声明只需要在需要默认参数的地方用 = 号给定即可,但是要注意所有的默认参数都不能出现在非默认参数的前面。

5. 定义功能

实例

Solution

```
def sale_car(price, color='red', brand='carmy', is_second_hand=True):  
    ....print('price', price,  
    ..... 'color', color,  
    ..... 'brand', brand,  
    ..... 'is_second_hand', is_second_hand,)
```

在这里定义了一个 `sale_car` 函数，参数为车的属性，但除了 `price` 之外，像 `color`，`brand` 和 `is_second_hand` 都是有默认值的，如果我们调用函数 `sale_car(1000)`，那么与 `sale_car(1000, 'red', 'carmy', True)` 是一样的效果。当然也可以在函数调用过程中传入特定的参数用来修改默认参数。通过默认参数可以减轻我们函数调用的复杂度。

进阶

3.1 自调用

如果要在执行脚本的时候执行一些代码，比如单元测试，可以在脚本最后加上单元测试代码，但是该脚本作为一个模块对外提供功能的时候单元测试代码也会执行，这些往往我们不想要的，我们可以把这些代码放入脚本最后：

Solution

```
if __name__ == '__main__':  
    ...#code_here
```

如果执行该脚本的时候，该 `if` 判断语句将会是 `True`，那么内部的代码将会执行。如果外部调用该脚本，`if` 判断语句则为 `False`，内部代码将不会执行。

5. 定义功能

3.2 可变参数

顾名思义，函数的可变参数是传入的参数可以变化的，1个，2个到任意个。当然可以将这些参数封装成一个 list 或者 tuple 传入，但不够 pythonic。使用可变参数可以很好解决该问题，注意可变参数在函数定义不能出现在特定参数和默认参数前面，因为可变参数会吞噬掉这些参数。

Solution

```
def report(name, *grades):  
    ....total_grade = 0  
  
    ....for grade in grades:  
        .....total_grade += grade  
  
    ....print(name, 'total grade is ', total_grade)
```

定义了一个函数，传入一个参数为 name，后面的参数 *grades 使用了 * 修饰，表明该参数是一个可变参数，这是一个可迭代的对象。该函数输入姓名和各科的成绩，输出姓名和总成绩。所以可以这样调用函数 report('Mike', 8, 9)，输出的结果为 Mike total grade is 17，也可以这样调用 report('Mike', 8, 9, 10)，输出的结果为 Mike total grade is 27

5. 定义功能

3.3 关键字参数

关键字参数可以传入0个或者任意个含参数名的参数，这些参数名在函数定义中并没有出现，这些参数在函数内部自动封装成一个字典(dict).

Solution

```
def portrait(name, **kw):  
    ....print('name is', name)  
    ....for k,v in kw.items():  
    .....print(k, v)
```

定义了一个函数，传入一个参数 `name`，和关键字参数 `kw`，使用了 `**` 修饰。表明该参数是关键字参数，通常来讲关键字参数是放在函数参数列表的最后。如果调用参数 `portrait('Mike', age=24, country='China', education='bachelor')` 输出：

Solution

```
name is Mike  
age 24  
...
```

6. 变量形式

局部变量

在 `def` 中, 我们可以定义一个局部变量, 这个变量 `a` 只能在这个功能 `fun` 中有效, 出了这个功能, `a` 这个变量就不是那个局部的 `a`.

Solution

```
def fun():  
    ....a = 10  
    ....print(a)  
    ....return a+100  
  
print(fun())  
  
10  
  
110
```

6. 变量形式

下面这个例子就验证了如果在 `fun` 外面调用 `a`, 会报错, 这表明外面的这个 `print(a)` 不能找到那个局部的 `a`, 只有全局变量再能在外面被调用, 比如 `APPLE`.

Solution

```
APPLE = 100 # 全局变量
```

```
def fun():
```

```
....a = 10 # 局部变量
```

```
....return a+100
```

```
print(APPLE) # 100
```

```
print(a) # 报错, 不能拿到一个局部变量的值
```

6. 变量形式

全局变量

那如何在外部也能调用一个在局部里修改了的全局变量呢. 首先我们在外部定义一个全局变量 `a=None`, 然后再 `fun()` 中声明 这个 `a` 是来自外部的 `a`. 声明方式就是 `global a`. 然后对这个外部的 `a` 修改后, 修改的效果会被施加到外部的 `a` 上. 所以我们将能看到运行完 `fun()`, `a` 的值从 `None` 变成了 `20`.

Solution

```
APPLY = 100 # 全局变量  
  
a = None  
  
def fun():  
    ....global a # 使用之前在全局里定义的 a  
    ....a = 20 # 现在的 a 是全局变量了  
    ....return a+100
```

6. 变量形式

Solution

```
print(APPLE) # 100
```

```
print('a past:', a) # None
```

```
fun()
```

```
print('a now:', a) # 20
```

7. 模块安装

安装外部的模块有很多种方式, 不同的系统安装形式也不同. 比如在 Windows 上安装 Python 的一些包, 可能还会要了你的老命. 哈哈.

什么是外部模块

外部模块就是在你 `import` 什么东西去python 脚本的时候会用到的.

Solution

```
import numpy as np
```

```
import matplotlib.pyplot as plt
```

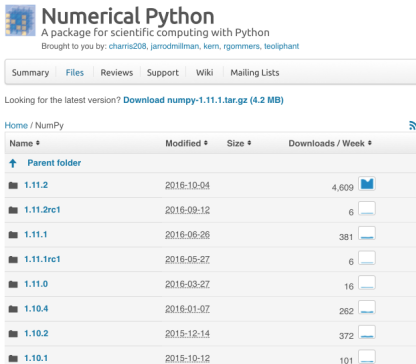
这里的 Numpy 和 matplotlib 都是外部模块, 需要安装以后才会有的. 他不属于 python 自带的模块.

7. 模块安装

安装 Numpy

这里我们举例说明, 对于一些科学运算的模块, 比如 `numpy`, 他的安装方式就有很多. 在 `Windows` 上, 最简单的方式就安装 `Anaconda` 这种平台, 他会自带很多必要的外部模块. 安装一个, 省去安装其它的烦恼.

不过我这里要讲的是下载安装包的方式在 `Windows` 安装. 比如 在 `Numpy` 安装包的网站中, 你能找到 `numpy` 的各种版本的安装包.





















The screenshot shows the PyPI page for the `Numpy` package. The title is "Numerical Python" and the subtitle is "A package for scientific computing with Python". Below the title, it says "Brought to you by: charris208, jarrodmillman, kern, rgommers, teoliphant". There are tabs for "Summary", "Files", "Reviews", "Support", "Wiki", and "Mailing Lists". A link to "Download numpy-1.11.1.tar.gz (4.2 MB)" is visible. Below the navigation, there is a table listing various versions of the package.

Name	Modified	Size	Downloads / Week
Parent folder			
1.11.2	2016-10-04		4,609
1.11.2rc1	2016-09-12		6
1.11.1	2016-06-26		381
1.11.1rc1	2016-05-27		6
1.11.0	2016-03-27		16
1.10.4	2016-01-07		262
1.10.2	2015-12-14		372
1.10.1	2015-10-12		101

7. 模块安装

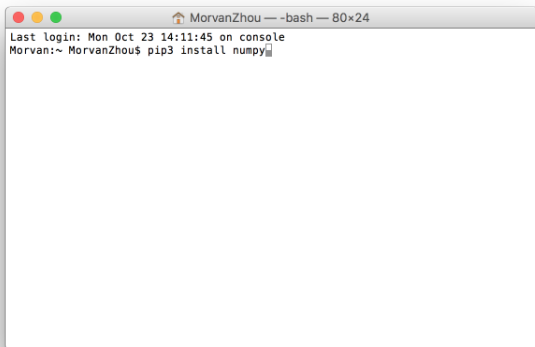
在 NumPy 1.10.2 这个版本中, 我们能找到适合 Windows 的安装包, 新版中目前还没添加 Windows 的安装包. 然后根据你的系统和 python 版本选择合适的“exe”安装包. 下载安装.

Home / NumPy / 1.10.2

Name *	Modified *	Size *	Downloads / Week *
↑ Parent folder			
numpy-1.10.2.tar.gz	2015-12-14	4.1 MB	10  
Changelog	2015-12-14	44.9 kB	0  
numpy-1.10.2.zip	2015-12-14	4.6 MB	2  
numpy-1.10.2-win32-superpack-pyth...	2015-12-14	8.9 MB	322  
numpy-1.10.2-win32-superpack-pyth...	2015-12-14	9.0 MB	7  
numpy-1.10.2-win32-superpack-pyth...	2015-12-14	8.9 MB	24  
README.txt	2015-12-14	9.3 kB	6  
reference.pdf	2015-12-14	8.2 MB	0  
userguide.pdf	2015-12-14	503.3 kB	1  
Totals: 9 Items		44.2 MB	372

7. 模块安装

如果你是 MacOS 或者 Linux, 这种外部模块, 安装起来就方便多了. 在你的电脑 Terminal 上输入一句话, 就能轻松安装. Windows 好像要经过特殊设置才能实现一样的功能, 具体忘记了... 你可能要查一下. 在我电脑中, Terminal 长这样.

A screenshot of a macOS Terminal window. The title bar shows the user 'MorvanZhou' and the shell '-bash' with a window size of '80x24'. The terminal content shows the last login time as 'Mon Oct 23 14:11:45 on console' and the current command prompt 'Morvan:~ MorvanZhou\$ pip3 install numpy' with a cursor at the end of the line.

```
MorvanZhou -- -bash -- 80x24
Last login: Mon Oct 23 14:11:45 on console
Morvan:~ MorvanZhou$ pip3 install numpy
```

7. 模块安装

然后只要你在里面输入这种形式就可以安装了.

Solution

```
$ pip install 你要的模块名
```

比如

Solution

```
$ pip install numpy # 这是 python2+ 版本的用法
```

```
$ pip3 install numpy # 这是 python3+ 版本的用法
```

7. 模块安装

更新外部模块

使用 `pip` 更新外部模块会十分简单. 主需要在 `Terminal` 中输入下面的指令就行. 这里的 `-U` 就是 `update` 意思.

Solution

```
$ pip install -U numpy # 这是 python2+ 版本的用法
```

```
$ pip3 install -U numpy # 这是 python3+ 版本的用法
```

8. 文件读取

\n 换行命令

定义 `text` 为字符串, 并查看使用 `\n` 和不适用 `\n` 的区别:

Solution

```
text='This is my first test. This is the second line. This the third '
```

```
print(text) # 无换行命令
```

```
This is my first test. This is the second line. This the third
```

```
text='This is my first test.\nThis is the second line.\nThis the third line'
```

```
print(text) # 输入换行命令 \n, 要注意斜杆的方向。注意换行的格式和c++一样
```

```
This is my first test.
```

```
This is the second line.
```

```
This the third line
```

8. 文件读取

open 读文件方式

使用 `open` 能够打开一个文件, `open` 的第一个参数为文件名和路径 `'my file.txt'`, 第二个参数为将要以什么方式打开它, 比如 `w` 为可写方式. 如果计算机没有找到 `'my file.txt'` 这个文件, `w` 方式能够创建一个新的文件, 并命名为 `my file.txt`

Solution

```
my_file=open('my file.txt','w') #用法: open('文件名','形式'), 其中形式有 'w':write; 'r':read.
```

```
my_file.write(text) #该语句会写入先前定义好的 text
```

```
my_file.close() #关闭文件
```

8. 文件读取

`\t` tab 对齐

使用 `\t` 能够达到 tab 对齐的效果:

Solution

```
text='\tThis is my first test.\n\tThis is the second line.\n\tThis is the third line'  
print(text) #延伸 使用 \t 对齐  
....This is my first test.  
....This is the second line.  
....This is the third line
```

8. 文件读取

给文件增加内容

我们先保存一个已经有3行文字的“my file.txt”文件，文件的内容如下：

Solution

This is my first test.

This is the second line.

This the third

8. 文件读取

然后使用添加文字的方式给这个文件添加一行 “This is appended file.”, 并将这行文字储存在 `append_file` 里, 注意 `\n` 的适用性:

Solution

```
append_text='\nThis is appended file.' # 为这行文字提前空行 "\n"  
my_file=open('my file.txt','a') # 'a'=append 以增加内容的形式打开  
my_file.write(append_text)  
my_file.close()  
  
This is my first test.  
  
This is the second line.  
  
This the third line.  
  
This is appended file.  
  
#运行后再去打开文件, 会发现会增加一行代码中定义的字符串
```


8. 文件读取

总结

掌握 `append` 的用法：`open('my file.txt','a')` 打开类型为 `a`，`a` 即表示 `append`
可以思考，如果用 `w` 形式打开，运行后会发生什么呢？

8. 文件读取

读取文件内容 `file.read()`

使用 `file.read()` 能够读取到文本的所有内容.

Solution

```
file= open('my file.txt','r')
```

```
content=file.read()
```

```
print(content)
```

```
This is my first test.
```

```
This is the second line.
```

```
This the third line.
```

```
This is appended file.
```

8. 文件读取

按行读取 `file.readline()`

如果想在文本中一行行的读取文本, 可以使用 `file.readline()`, `file.readline()` 读取的内容和你使用的次数有关, 使用第二次的时候, 读取到的是文本的第二行, 并可以以此类推:

Solution

```
file= open('my file.txt','r')
content=file.readline() # 读取第一行
print(content)
This is my first test.

second_read_time=file.readline() # 读取第二行
print(second_read_time)
This is the second line.
```

8. 文件读取

读取所有行 `file.readlines()`

如果想要读取所有行, 并可以使用像 `for` 一样的迭代器迭代这些行结果, 我们可以使用 `file.readlines()`, 将每一行的结果存储在 `list` 中, 方便以后迭代.

Solution

```
file= open('my file.txt','r')
```

```
content=file.readlines() # python_list 形式
```

```
print(content)
```

```
['This is my first test.\n', 'This is the second line.\n', 'This the third line.\n', 'This is appended file.']
```

之后如果使用 for 来迭代输出:

```
for item in content:
```

```
....print(item)
```

8. 文件读取

Solution

This is my first test.

This is the second line.

This the third line.

This is appended file.

9. class 类

class 定义一个类

class 定义一个类, 后面的类别首字母推荐以大写的形式定义, 比如Calculator. class可以先定义自己的属性, 比如该属性的名称可以写为 `name='Good Calculator'`. class后面还可以跟def, 定义一个函数. 比如`def add(self,x,y):` 加法, 输出`print(x+y)`. 其他的函数定义方法一样, 注意这里的self 是默认值.

Solution

```
class Calculator: #首字母要大写, 冒号不能缺
...name='Good Calculator' #该行为class的属性
...def add(self,x,y):
.....print(self.name)
.....result = x + y
.....print(result)
...def minus(self,x,y):
.....result=x-y
.....print(result)
```

9. class 类

Solution

»> *cal=Calculator()* #注意这里运行class的时候要加"()",否则调用下面函数的时候会出现错误,导致无法调用.

»> *cal.name*

'Good Calculator'

»> *cal.add(10,20)*

Good Calculator

30

»> *cal.minus(10,20)*

-10

»>

总结

注意定义自变量`cal`等于`Calculator`要加括号“`()`”,`cal=Calculator()`否则运行下面函数的时候会出现错误,导致无法调用.

9. class 类

class 类 init 功能

init

`__init__` 可以理解成初始化class的变量，取自英文中initial 最初的意思.可以在运行时，给初始值赋值，

运行`c=Calculator('bad calculator',18,17,16,15)`,然后调出每个初始值的值。看如下代码。

Solution

```
class Calculator:
    ....name='good calculator'
    ....price=18
    ....def __init__(self,price,height,width): # 注意，这里的下划线是双下划线
    .....self.name=name
    .....self.h=height
    .....self.wi=width
```

Solution

```
»> c=Calculator('bad calculator',18,17,16,15)
```

```
»> c.name
```

```
'bad calculator'
```

```
»> c.h
```

```
17
```

```
»> c.wi
```

```
16
```

```
»>
```

9. class 类

如何设置属性的默认值, 直接在def里输入即可, 如下:

```
def __init__(self,name,price,height=10,width=14,weight=16):
```

查看运行结果, 三个有默认值的属性, 可以直接输出默认值, 这些默认值可以在code中更改, 比如c.wi=17再输出c.wi就会把wi属性值更改为17.同理可推其他属性的更改方法。

总结

```
def __init__(self,name,price,height,width,weight):
```

注意, 这里的下划线是双下划线

10. input 输入

input

`variable=input()` 表示运行后，可以在屏幕中输入一个数字，该数字会赋值给自变量。
看代码：

Solution

```
a_input=input('please input a number:')
```

```
print('this number is:',a_input)
```

```
please input a number:12 #12 是我在硬盘中输入的数字
```

```
this number is: 12
```

10. input 输入

`input()`应用在if语句中.

在下面代码中, 需要将`input()` 定义成整型, 因为在if语句中自变量 `a_input` 对应的是1 and 2 整型。输入的内容和判断句中对应的内容格式应该一致。

也可以将if语句中的1and2 定义成字符串, 其中区别请读者自定写入代码查看。

Solution

```
a_input=int(input('please input a number:'))#注意这里要定义一个整型
if a_input==1:
    ....print('This is a good one')
elif a_input==2:
    ....print('See you next time')
else:
    ....print('Good luck')
```

10. input 输入

Solution

please input a number:1 #input 1

This is a good one

please input a number:2 #input 2

See you next time

please input a number:3 #input 3 or other number

Good luck

10. input 输入

input扩展

用input()来判断成绩

Solution

```
score=int(input('Please input your score: \n'))
```

```
if score>=90:
```

```
....print('Congradulation, you get an A')
```

```
elif score >=80:
```

```
....print('You get a B')
```

```
elif score >=70:
```

```
....print('You get a C')
```

10. input 输入

Solution

```
elif score >=60:
```

```
....print('You get a D')
```

```
else:
```

```
....print('Sorry, You are failed ')
```


11. 元组, 列表, 字典

Tuple

叫做 `tuple`, 用小括号、或者无括号来表述, 是一连串有顺序的数字。

Solution

```
a_tuple = (12, 3, 5, 15, 6)
```

```
another_tuple = 12, 3, 5, 15, 6
```

List

而`list`是以中括号来命名的:

Solution

```
a_list = [12, 3, 67, 7, 82]
```

11. 元组, 列表, 字典

两者对比

他们的元素可以一个一个地被迭代、输出、运用、定位取值:

Solution

```
for content in a_list:
```

```
....print(content)
```

```
for content_tuple in a_tuple:
```

```
....print(content_tuple)
```

下一个例子, 依次输出a_tuple和a_list中的各个元素:

Solution

```
for index in range(len(a_list)):
```

```
....print("index = ", index, ", number in list = ", a_list[index])
```

11. 元组, 列表, 字典

List 添加

列表是一系列有序的数列, 有一系列自带的功能, 例如:

Solution

```
a = [1,2,3,4,1,1,-1]
```

```
a.append(0) # 在a的最后面追加一个0
```

```
print(a)
```

```
# [1, 2, 3, 4, 1, 1, -1, 0]
```

11. 元组, 列表, 字典

在指定的地方添加项:

Solution

```
a = [1,2,3,4,1,1,-1]
```

```
a.insert(1,0) # 在位置1处添加0
```

```
print(a)
```

```
# [1, 0, 2, 3, 4, 1, 1, -1]
```

11. 元组, 列表, 字典

List 移除

删除项:

Solution

```
a = [1,2,3,4,1,1,-1]
a.remove(2) # 删除列表中第一个出现的值为2的项
print(a)
# [1, 3, 4, 1, 1, -1]
```

11. 元组, 列表, 字典

List 索引

显示特定位:

Solution

```
a = [1,2,3,4,1,1,-1]
```

```
print(a[0]) # 显示列表a的第0位的值
```

```
# 1
```

```
print(a[-1]) # 显示列表a的最末位的值
```

```
# -1
```

```
print(a[0:3]) # 显示列表a的从第0位 到 第2位(第3位之前) 的所有项的值
```

```
# [1, 2, 3]
```

```
print(a[5:]) # 显示列表a的第5位及以后的所有项的值
```

```
# [1, -1]
```

```
print(a[-3:]) # 显示列表a的倒数第3位及以后的所有项的值
```

```
# [1, 1, -1]
```

11. 元组, 列表, 字典

打印列表中的某个值的索引(index):

Solution

```
a = [1,2,3,4,1,1,-1]
print(a.index(2)) # 显示列表a中第一次出现的值为2的项的索引
# 1
```

统计列表中某值出现的次数:

Solution

```
a = [4,1,2,3,4,1,1,-1]
print(a.count(-1))
# 1
```

11. 元组, 列表, 字典

List 排序

对列表的项排序:

Solution

```
a = [4,1,2,3,4,1,1,-1]
```

```
a.sort() # 默认从小到大排序
```

```
print(a)
```

```
# [-1, 1, 1, 1, 2, 3, 4, 4]
```

```
a.sort(reverse=True) # 从大到小排序
```

```
print(a)
```

```
# [4, 4, 3, 2, 1, 1, 1, -1]
```


11. 元组, 列表, 字典

创建二维列表

一个一维的List是线性的List, 多维List是一个平面的List:

Solution

```
a = [1,2,3,4,5] # 一行五列  
multi_dim_a = [[1,2,3],  
.....[2,3,4],  
.....[3,4,5]] # 三行三列
```

11. 元组, 列表, 字典

索引

在上面定义的List中进行搜索:

Solution

```
print(a[1])
```

```
# 2
```

```
print(multi_dim_a[0][1])
```

```
# 2
```

用行数和列数来定位list中的值。这里用的是二维的列表，但可以有更多的维度。

11. 元组, 列表, 字典

创建字典

如果说List是有顺序地输出输入的话, 那么字典的存档形式则是无需顺序的, 我们来看一个例子:

在字典中, 有key和 value两种元素, 每一个key对应一个value, key是名字, value是内容。数字和字符串都可以当做key或者value, 在同一个字典中, 并不需要所有的key或value有相同的形式。这样说, List 可以说是一种key为有序数列的字典。

Solution

```
a_list = [1,2,3,4,5,6,7,8]
d1 = {'apple':1, 'pear':2, 'orange':3}
print(d1['apple']) # 1
print(a_list[0]) # 1
del d1['pear']
print(d1) # {'orange': 3, 'apple': 1}
d1['b'] = 20
print(d1) # {'orange': 3, 'b': 20, 'pear': 2, 'apple': 1}
```

11. 元组, 列表, 字典

字典存储类型

以上的例子可以对列表中的元素进行增减。在打印出整个列表时, 可以发现各个元素并没有按规律打印出来, 进一步验证了字典是一个无序的容器。

Solution

```
def func():  
    ....return 0  
  
d4 = {'apple':[1,2,3], 'pear':{'1:3, 3:'a'}}, 'orange':func}  
print(d4['pear'][3]) # a
```

字典还可以以更多样的形式出现, 例如字典的元素可以是一个List, 或者再是一个列表, 再或者是一个function。索引需要的项目时, 只需要正确指定对应的key就可以了。

12. 模块

import 模块

`import time` 指 `import time` 模块, 这个模块可以python自带, 也可以是自己安装的, 比如以后会用到`numpy`这些模块, 需要自己安装。

Solution

```
import time

print(time.localtime()) #这样就可以print 当地时间了

time.struct_time(tm_year=2016, tm_mon=12, tm_mday=23, tm_hour=14,
tm_min=12, tm_sec=48, tm_wday=4, tm_yday=358, tm_isdst=0)
```

方法二: `import time as __, __` 下划线缩写部分可以自己定义, 在代码中把`time` 定义成 `t`.

Solution

```
import time as t

print(t.localtime()) # 需要加t.前缀来引出功能
```

方法三: `from time import time, localtime` ,只import自己想要的功能.

Solution

```
from time import time, localtime

print(localtime())
```

12. 模块

方法四: `from time import *` 输入模块的所有功能

Solution

```
from time import *  
print(localtime())
```

自建一个模块

Solution

```
d=float(input('Please enter what is your initial balance: \n'))  
p=float(input('Please input what is the interest rate (as a number): \n'))  
d=float(d+d*(p/100))  
year=1  
while year<=5:  
....d=float(d+d*p/100)  
....print('Your new balance after year:',year,'is',d)  
....year=year+1  
print('your final year is',d)
```

12. 模块

调用自己的模块

新开一个脚本，`import balance`

Solution

```
import balance
```

模块存储路径说明

在Mac系统中，下载的python模块会被存储到外部路径`site-packages`，同样，我们自己建的模块也可以放到这个路径，最后不会影响到自建模块的调用。