eval 函数

eval() 函数十分强大 —— **将字符串** 当成 **有效的表达式** 来求值 并 **返回计算结果**

```python

## 基本的数学计算

In [1]: eval("1 + 1") Out[1]: 2

## 字符串重复

In [2]: eval("' * 10") Out[2]: '**********'

## 将字符串转换成列表

In [3]: type(eval("[1, 2, 3, 4, 5]")) Out[3]: list

## 将字符串转换成字典

In [4]: type(eval("{'name': 'xiaoming', 'age': 18}")) Out[4]: dict ```

### 案例 - 计算器

**需求**

1. 提示用户输入一个 **加减乘除混合运算**
2. 返回计算结果

```python input_str = input("请输入一个算术题：")

print(eval(input_str))

```

不要滥用 eval

> 在开发时千万不要使用 eval 直接转换 input 的结果

```
python __import__('os').system('ls')
```

- 等价代码

```python import os

os.system("终端命令") ```

- 执行成功，返回 0
- 执行失败，返回错误信息

## 模块和包

### 目标

- 模块
- 包
- 发布模块

### 01. 模块

#### 1.1 模块的概念

> 模块是 Python 程序架构的一个核心概念

- 每一个以扩展名 py 结尾的 Python 源代码文件都是一个 **模块**
- **模块名** 同样也是一个 **标识符**，需要符合标识符的命名规则
- 在模块中定义的 **全局变量** 、**函数、类** 都是提供给外界直接使用的 **工具**
- **模块** 就好比是 **工具包**，要想使用这个工具包中的工具，就需要先 **导入** 这个模块

#### 1.2 模块的两种导入方式

**1）import 导入**

```
python import 模块名1, 模块名2
```

> 提示：在导入模块时，每个导入应该独占一行

```
python import 模块名1 import 模块名2
```

- **导入之后**
  - 通过 模块名. 使用 **模块提供的工具** —— **全局变量、函数、类**

使用 as 指定模块的别名

> 如果模块的名字太长，可以使用 as 指定模块的名称，以方便在代码中的使用

```
python import 模块名1 as 模块别名
```

> 注意：模块别名 应该符合 大驼峰命名法

2）from...import 导入

- 如果希望 **从某一个模块** 中，导入 **部分** 工具，就可以使用 from ... import 的方式
- import 模块名 是 **一次性** 把模块中 **所有工具全部导入**，并且通过 **模块名/别名** 访问

```python

## 从 模块 导入 某一个工具

from 模块名1 import 工具名 ```

- 导入之后
  - **不需要** 通过 模块名.
  - 可以直接使用 **模块提供的工具** —— **全局变量、函数、类**

**注意**

> 如果 两个模块，存在 同名的函数，那么 后导入模块的函数，会 覆盖掉先导入的函数

- 开发时 import 代码应该统一写在 **代码的顶部**，更容易及时发现冲突
- 一旦发现冲突，可以使用 as 关键字 **给其中一个工具起一个别名**

**from...import *（知道）**

```python

## 从 模块 导入 所有工具

from 模块名1 import * ```

**注意**

> 这种方式不推荐使用，因为函数重名并没有任何的提示，出现问题不好排查

#### 1.3 模块的搜索顺序[扩展]

Python 的解释器在 **导入模块** 时，会：

1. 搜索 **当前目录** 指定模块名的文件，**如果有就直接导入**
2. 如果没有，再搜索 **系统目录**

> 在开发时，给文件起名，不要和 系统的模块文件 重名

Python 中每一个模块都有一个内置属性 __file__ 可以 **查看模块** 的 **完整路径**

**示例**

```python import random

## 生成一个 0～10 的数字

rand = random.randint(0, 10)

print(rand)

```

> 注意：如果当前目录下，存在一个 random.py 的文件，程序就无法正常执行了！

- 这个时候，Python 的解释器会 **加载当前目录** 下的 random.py 而不会加载 **系统的** random 模块

1.4 原则 —— 每一个文件都应该是可以被导入的

- 一个 **独立的 Python 文件** 就是一个 **模块**
- 在导入文件时，文件中 **所有没有任何缩进的代码** 都会被执行一遍！

实际开发场景

- 在实际开发中，每一个模块都是独立开发的，大多都有专人负责
- **开发人员** 通常会在 **模块下方 增加一些测试代码**
  - 仅在模块内使用，而被导入到其他文件中不需要执行

__name__ 属性

- `__name__` 属性可以做到，测试模块的代码 只在测试情况下被运行，而在 被导入时不会被执行！

- `__name__` 是 Python 的一个内置属性，记录着一个 **字符串**
- 如果 **是被其他文件导入的**，`__name__` 就是 **模块名**
- 如果 **是当前执行的程序** `__name__` 是 **`__main__`**

在很多 Python 文件中都会看到以下格式的代码：

```python

## 导入模块

## 定义全局变量

## 定义类

## 定义函数

## 在代码的最下方

def main(): # ... pass

## 根据 name 判断是否执行下方代码

if **name** == "**main**": main()

```

---

02. 包（Package）

概念

- **包** 是一个 **包含多个模块** 的 **特殊目录**
- 目录下有一个 **特殊的文件** `__init__.py`
- 包名的 **命名方式** 和变量名一致，**小写字母** + _

好处

- 使用 `import 包名` 可以一次性导入 **包** 中 **所有的模块**

案例演练

1. 新建一个 `hm_message` 的 **包**
2. 在目录下，新建两个文件 `send_message` 和 `receive_message`
3. 在 `send_message` 文件中定义一个 `send` 函数
4. 在 `receive_message` 文件中定义一个 `receive` 函数
5. 在外部直接导入 `hm_message` 的包

`__init__.py`

- 要在外界使用 **包** 中的模块，需要在 `__init__.py` 中指定 **对外界提供的模块列表**

```python

## 从 当前目录 导入 模块列表

from . import send*message from . import receive*message ```

## 03. 发布模块（知道）

- 如果希望自己开发的模块，**分享** 给其他人，可以按照以下步骤操作

### 3.1 制作发布压缩包步骤

**1) 创建 setup.py**

- `setup.py` 的文件

```python from distutils.core import setup

setup(name="hm*message", # 包名 version="1.0", # 版本 description="itheima's 发送和接收消息模块", # 描述信息 long*description="完整的发送和接收消息模块", # 完整描述信息 author="itheima", # 作者 author*email="itheima@itheima.com", # 作者邮箱 url="www.itheima.com", # 主页 py*modules=["hm*message.send*message", "hm*message.receive*message"])

```

有关字典参数的详细信息，可以参阅官方网站：

https://docs.python.org/2/distutils/apiref.html

2) 构建模块

```
bash $ python3 setup.py build
```

3) 生成发布压缩包

```
bash $ python3 setup.py sdist
```

> 注意，要制作哪个版本的模块，就使用哪个版本的解释器执行！

3.2 安装模块

```bash $ tar -zxvf hm_message-1.0.tar.gz

$ sudo python3 setup.py install ```

**卸载模块**

直接从安装目录下，把安装模块的 **目录** 删除就可以

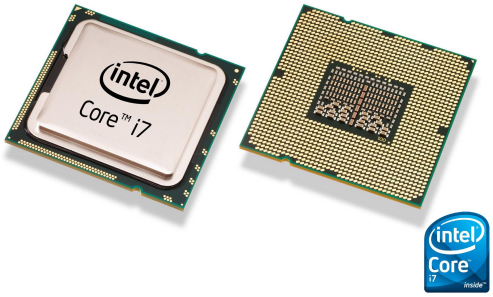
```
python $ cd /usr/local/lib/python3.5/dist-packages/ $ sudo rm -r hm_message*
```

3.3 `pip` 安装第三方模块

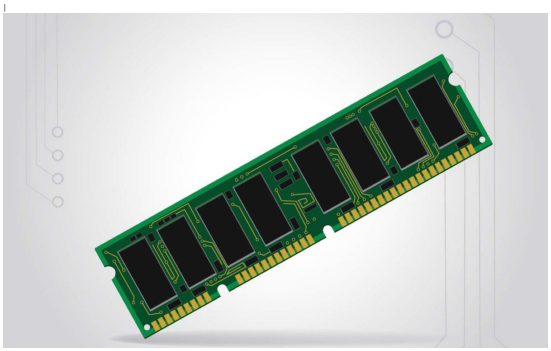
- **第三方模块** 通常是指由 **知名的第三方团队** 开发的 并且被 **程序员广泛使用** 的 Python 包 / 模块
  - 例如 `pygame` 就是一套非常成熟的 **游戏开发模块**
- `pip` 是一个现代的，通用的 Python 包管理工具
- 提供了对 Python 包的查找、下载、安装、卸载等功能

安装和卸载命令如下：

```bash

## 将模块安装到 Python 2.x 环境

$ sudo pip install pygame $ sudo pip uninstall pygame

## 将模块安装到 Python 3.x 环境

$ sudo pip3 install pygame $ sudo pip3 uninstall pygame ```

**在 `Mac` 下安装 `iPython`**

```
bash $ sudo pip install ipython
```

**在 `Linux` 下安装 `iPython`**

```
bash $ sudo apt install ipython $ sudo apt install ipython3
```

文件

目标

- 文件的概念
- 文件的基本操作
- 文件/文件夹的常用操作
- 文本文件的编码方式

01. 文件的概念

1.1 文件的概念和作用

- 计算机的 **文件**，就是存储在某种 **长期储存设备** 上的一段 **数据**
- 长期存储设备包括：硬盘、U 盘、移动硬盘、光盘...

文件的作用

将数据长期保存下来，在需要的时候使用

| CPU | 内存 | 硬盘 | | :---: | :---: | :---: | |
```



|



|

|

1.2 文件的存储方式

- 在计算机中，文件是以 **二进制** 的方式保存在磁盘上的

文本文件和二进制文件

- 文本文件
  - 可以使用 **文本编辑软件** 查看
  - 本质上还是二进制文件
  - 例如：python 的源程序
- 二进制文件
  - 保存的内容 不是给人直接阅读的，而是 **提供给其他软件使用的**
  - 例如：图片文件、音频文件、视频文件等等
  - 二进制文件不能使用 **文本编辑软件** 查看

02. 文件的基本操作

2.1 操作文件的套路

在 **计算机** 中要操作文件的套路非常固定，一共包含**三个步骤**：

1. 打开文件
2. 读、写文件
   - **读** 将文件内容读入内存
   - **写** 将内存内容写入文件
3. 关闭文件

2.2 操作文件的函数/方法

- 在 `Python` 中要操作文件需要记住 1 个函数和 3 个方法

| 序号 | 函数/方法 | 说明 | | --- | --- | --- | | 01 | open | 打开文件，并且返回文件操作对象 | | 02 | read | 将文件内容读取到内存 | | 03 | write | 将指定内容写入文件 | | 04 | close | 关闭文件 |

- `open` 函数负责打开文件，并且返回文件对象
- `read`/`write`/`close` 三个方法都需要通过 **文件对象** 来调用

2.3 read 方法 —— 读取文件

- `open` 函数的第一个参数是要打开的文件名（文件名区分大小写）
  - 如果文件 **存在**，返回 **文件操作对象**
  - 如果文件 **不存在**，会 **抛出异常**
- `read` 方法可以一次性 **读入** 并 **返回** 文件的 **所有内容**
- `close` 方法负责 **关闭文件**
  - 如果 **忘记关闭文件**，**会造成系统资源消耗**，**而且会影响到后续对文件的访问**
- **注意**：`read` 方法执行后，会把 **文件指针** 移动到 **文件的末尾**

```python
# 1. 打开 - 文件名需要注意大小写
file = open("README")

# 2. 读取
text = file.read()
print(text)

# 3. 关闭
file.close()
```

提示

- 在开发中，通常会先编写 **打开** 和 **关闭** 的代码，再编写中间针对文件的 **读/写** 操作！

文件指针（知道）

- **文件指针** 标记 **从哪个位置开始读取数据**
- **第一次打开** 文件时，通常 **文件指针会指向文件的开始位置**
- 当执行了 `read` 方法后，**文件指针** 会移动到 **读取内容的末尾**
  - 默认情况下会移动到 **文件末尾**

思考

- 如果执行了一次 `read` 方法，读取了所有内容，那么再次调用 `read` 方法，还能够获得到内容吗？

答案

- 不能
- 第一次读取之后，文件指针移动到了文件末尾，再次调用不会读取到任何的内容

2.4 打开文件的方式

- `open` 函数默认以 **只读方式** 打开文件，并且返回文件对象

语法如下：

```
python f = open("文件名", "访问方式")
```

| 访问方式 | 说明 | | --- | --- | | 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。如果文件不存在，抛出异常** | | w | 以**只写**方式打开文件。如果文件存在会被覆盖。如果文件不存在，创建新文件 | | a | 以**追加**方式打开文件。如果该文件已存在，文件指针将会放在文件的结尾。如果文件不存在，创建新文件进行写入 | | r+ | 以**读写方式打开文件。文件的指针将会放在文件的开头。如果文件不存在，抛出异常** | | w+ | 以**读写**方式打开文件。如果文件存在会被覆盖。如果文件不存在，创建新文件 | | a+ | 以**读写**方式打开文件。如果该文件已存在，文件指针将会放在文件的结尾。如果文件不存在，创建新文件进行写入 |

提示

- 频繁的移动文件指针，**会影响文件的读写效率**，开发中更多的时候会以 **只读、只写** 的方式来操作文件

写入文件示例

```python
# 打开文件
f = open("README", "w")

f.write("hello python！\n") f.write("今天天气真好")

# 关闭文件
f.close()
```

2.5 按行读取文件内容

- `read` 方法默认会把文件的 **所有内容 一次性读取到内存**
- 如果文件太大，对内存的占用会非常严重

`readline` 方法

- `readline` 方法可以一次读取一行内容
- 方法执行后，会把 **文件指针** 移动到下一行，准备再次读取

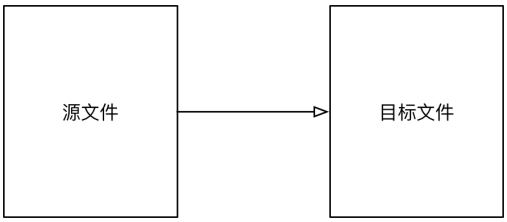
读取大文件的正确姿势

```python
# 打开文件
file = open("README")

while True: # 读取一行内容 text = file.readline()

    # 判断是否读到内容
    if not text:
        break

    # 每读取一行的末尾已经有了一个 `\n`
    print(text, end="")

# 关闭文件
file.close()
```

2.6 文件读写案例 —— 复制文件

目标

用代码的方式，来实现文件复制过程



小文件复制

- 打开一个已有文件，读取完整内容，并写入到另外一个文件

```python
# 1. 打开文件
file*read = open("README") file*write = open("README[复件]", "w")

# 2. 读取并写入文件
text = file*read.read() file*write.write(text)

# 3. 关闭文件
file*read.close() file*write.close()
```

大文件复制

- 打开一个已有文件，逐行读取内容，并顺序写入到另外一个文件

```python
# 1. 打开文件
file*read = open("README") file*write = open("README[复件]", "w")

# 2. 读取并写入文件
while True: # 每次读取一行 text = file_read.readline()

    # 判断是否读取到内容
    if not text:
        break

    file_write.write(text)

# 3. 关闭文件
file*read.close() file*write.close()
```

03. 文件/目录的常用管理操作

- 在 **终端 / 文件浏览器**、中可以执行常规的 **文件 / 目录** 管理操作，例如：
  - 创建、重命名、删除、改变路径、查看目录内容、……
- 在 `Python` 中，如果希望通过程序实现上述功能，需要导入 `os` 模块

文件操作

序号	方法名	说明	示例
01	rename	重命名文件	`os.rename(源文件名，目标文件名)`
02	remove	删除文件	`os.remove(文件名)`

目录操作

序号	方法名	说明	示例
01	listdir	目录列表	`os.listdir(目录名)`
02	mkdir	创建目录	`os.mkdir(目录名)`
03	rmdir	删除目录	`os.rmdir(目录名)`
04	getcwd	获取当前目录	`os.getcwd()`
05	chdir	修改工作目录	`os.chdir(目标目录)`
06	path.isdir	判断是否是文件	`os.path.isdir(文件路径)`

> 提示：文件或者目录操作都支持 **相对路径** 和 **绝对路径**

04. 文本文件的编码格式（科普）

- 文本文件存储的内容是基于 **字符编码** 的文件，常见的编码有 `ASCII` 编码，`UNICODE` 编码等

> Python 2.x默认使用 `ASCII` 编码格式 Python 3.x默认使用 `UTF-8` 编码格式

4.1 ASCII 编码和 UNICODE 编码

`ASCII` 编码

- 计算机中只有 `256` 个 `ASCII` 字符
- 一个 `ASCII` 在内存中占用 **1 个字节** 的空间
  - 8 个 `0/1` 的排列组合方式一共有 `256` 种，也就是 `2 ** 8`

ASCII表

（ American Standard Code for Information Interchange 美国标准信息交换代码 ）

高四位 / 低四位	ASCII控制字符 0000					ASCII控制字符 0001					ASCII打印字符 0010			ASCII打印字符 0011		ASCII打印字符 0100		ASCII打印字符 0101		ASCII打印字符 0110		ASCII打印字符 0111	
	0					1					2			3		4		5		6		7	
	十进制	字符	Ctrl	代码	转义字符	字符解释	十进制	字符	Ctrl	代码	转义字符	字符解释	十进制	字符	十进制	字符	十进制	字符	十进制	字符	十进制	字符	十进制	字符	Ctrl
0000	0		^@	NUL	\0	空字符	16	►	^P	DLE		数据链路转义	32		48	0	64	@	80	P	96	`	112	p	
0001	1	☺	^A	SOH		标题开始	17	◄	^Q	DC1		设备控制 1	33	!	49	1	65	A	81	Q	97	a	113	q	
0010	2	☻	^B	STX		正文开始	18	↕	^R	DC2		设备控制 2	34	"	50	2	66	B	82	R	98	b	114	r	
0011	3	♥	^C	ETX		正文结束	19	‼	^S	DC3		设备控制 3	35	#	51	3	67	C	83	S	99	c	115	s	
0100	4	♦	^D	EOT		传输结束	20	¶	^T	DC4		设备控制 4	36	$	52	4	68	D	84	T	100	d	116	t	
0101	5	♣	^E	ENQ		查询	21	§	^U	NAK		否定应答	37	%	53	5	69	E	85	U	101	e	117	u	
0110	6	♠	^F	ACK		肯定应答	22	▬	^V	SYN		同步空闲	38	&	54	6	70	F	86	V	102	f	118	v	
0111	7	•	^G	BEL	\a	响铃	23	↨	^W	ETB		传输块结束	39	'	55	7	71	G	87	W	103	g	119	w	
1000	8	◘	^H	BS	\b	退格	24	↑	^X	CAN		取消	40	(	56	8	72	H	88	X	104	h	120	x	
1001	9	○	^I	HT	\t	横向制表	25	↓	^Y	EM		介质结束	41	)	57	9	73	I	89	Y	105	i	121	y	
1010	A	10	^J	LF	\n	换行	26	→	^Z	SUB		替代	42	*	58	:	74	J	90	Z	106	j	122	z	
1011	B	♂	^K	VT	\v	纵向制表	27	←	^[	ESC	\e	溢出	43	+	59	;	75	K	91	[	107	k	123	{	
1100	C	12	^L	FF	\f	换页	28	∟	^\	FS		文件分隔符	44	,	60	<	76	L	92	\	108	l	124	|	
1101	D	13	♪	^M	CR	\r	回车	29	↔	^]	GS		组分隔符	45	-	61	=	77	M	93	]	109	m	125	}	
1110	E	14	♫	^N	SO		移出	30	▲	^^	RS		记录分隔符	46	.	62	>	78	N	94	^	110	n	126	~	
1111	F	15	☼	^O	SI		移入	31	▼	^-	US		单元分隔符	47	/	63	?	79	O	95	_	111	o	127	⌂	^Backspace 代码：DEL

注：表中的ASCII字符可以用 "Alt + 小键盘上的数字键 "方法输入。 2013/08/08

`UTF-8` 编码格式

- 计算机中使用 **1~6 个字节** 来表示一个 `UTF-8` 字符，涵盖了 **地球上几乎所有地区的文字**
- 大多数汉字会使用 **3 个字节** 表示
- `UTF-8` 是 `UNICODE` 编码的一种编码格式

4.2 Ptyhon 2.x 中 如何使用中文

> Python 2.x默认使用 `ASCII` 编码格式 Python 3.x默认使用 `UTF-8` 编码格式

- 在 Python 2.x 文件的 **第一行** 增加以下代码，解释器会以 `utf-8` 编码来处理 python 文件

```python
# - coding:utf8 -
```

> 这方式是官方推荐使用的！

- 也可以使用

```python
# coding=utf8
```

unicode 字符串

- 在 `Python 2.x` 中，即使指定了文件使用 `UTF-8` 的编码格式，但是在遍历字符串时，仍然会 **以字节为单位遍历** 字符串
- 要能够 **正确的遍历字符串**，在定义字符串时，需要 **在字符串的引号前**，增加一个小写字母 `u`，告诉解释器这是一个 `unicode` 字符串（使用 `UTF-8` 编码格式的字符串）

```python
# - coding:utf8 -

# 在字符串前，增加一个 u 表示这个字符串是一个 utf8 字符串
hello_str = u"你好世界"

print(hello_str)

for c in hello_str:
    print(c)
```

异常

目标

- 异常的概念
- 捕获异常
- 异常的传递
- 抛出异常

01. 异常的概念

- 程序在运行时，如果 `Python` 解释器 **遇到** 到一个错误，**会停止程序的执行，并且提示一些错误信息**，这就是 **异常**
- **程序停止执行并且提示错误信息** 这个动作，我们通常称之为：**抛出(raise)异常**

```
```

> 程序开发时，很难将 **所有的特殊情况** 都处理的面面俱到，通过 **异常捕获** 可以针对突发事件做集中的处理，从而保证程序的 **稳定性和健壮性**

02. 捕获异常

2.1 简单的捕获异常语法

- 在程序开发中，如果 **对某些代码的执行不能确定是否正确**，可以增加 `try(尝试)` 来 **捕获异常**
- 捕获异常最简单的语法格式：

```
python try:
    尝试执行的代码
except:
    出现错误的处理
```

- `try` **尝试**，下方编写要尝试代码，不确定是否能够正常执行的代码
- `except` **如果不是**，下方编写尝试失败的代码

简单异常捕获演练 —— 要求用户输入整数

```
python try:
    # 提示用户输入一个数字
    num = int(input("请输入数字："))
except:
    print("请输入正确的数字")
```

2.2 错误类型捕获

- 在程序执行时，可能会遇到 **不同类型的异常**，并且需要 **针对不同类型的异常，做出不同的响应**，这个时候，就需要捕获错误类型了

- 语法如下：

```
python try:
    # 尝试执行的代码
    pass
except 错误类型1:
    # 针对错误类型1，对应的代码处理
    pass
except (错误类型2, 错误类型3):
    # 针对错误类型2 和 3，对应的代码处理
    pass
except Exception as result:
    print("未知错误 %s" % result)
```

- 当 `Python` 解释器 **抛出异常 时，最后一行错误信息的第一个单词，就是错误类型**

异常类型捕获演练 —— 要求用户输入整数

需求

1. 提示用户输入一个整数
2. 使用 `8` 除以用户输入的整数并且输出

```
try:
    num = int(input("请输入整数："))
    result = 8 / num
    print(result)
except ValueError:
    print("请输入正确的整数")
except ZeroDivisionError:
    print("除 0 错误")
```

捕获未知错误

- 在开发时，**要预判到所有可能出现的错误**，还是有一定难度的
- 如果希望程序 **无论出现任何错误**，都不会因为 `Python` 解释器 **抛出异常而被终止**，可以再增加一个 `except`

语法如下：

```
python except Exception as result: print("未知错误 %s" % result)
```

2.3 异常捕获完整语法

- 在实际开发中，为了能够处理复杂的异常情况，完整的异常语法如下：

> 提示：
>
> - 有关完整语法的应用场景，在后续学习中，结合实际的案例会更好理解
> - 现在先对这个语法结构有个印象即可

```
python try: # 尝试执行的代码 pass except 错误类型1: # 针对错误类型1，对应的代码处理 pass except 错误类型2: # 针对错误类型2，对应的代码处理 pass except (错误类型3, 错误类型4): # 针对错误类型3 和 4，对应的代码处理 pass except Exception as result: # 打印错误信息 print(result) else: # 没有异常才会执行的代码 pass finally: # 无论是否有异常，都会执行的代码 print("无论是否有异常，都会执行的代码")
```

- `else` 只有在没有异常时才会执行的代码
- `finally` 无论是否有异常，都会执行的代码

- 之前一个演练的 **完整捕获异常** 的代码如下：

```python try: num = int(input("请输入整数：")) result = 8 / num print(result) except ValueError: print("请输入正确的整数") except ZeroDivisionError: print("除 0 错误") except Exception as result: print("未知错误 %s" % result) else: print("正常执行") finally: print("执行完成，但是不保证正确")
```

---

03. 异常的传递

- **异常的传递** —— 当 **函数/方法** 执行 **出现异常**，会 **将异常传递** 给 函数/方法 的 **调用一方**
- 如果 **传递到主程序**，仍然 **没有异常处理**，程序才会被终止

> 提示

- 在开发中，可以在主函数中增加 **异常捕获**
- 而在主函数中调用的其他函数，只要出现异常，都会传递到主函数的 **异常捕获** 中
- 这样就不需要在代码中，增加大量的 **异常捕获**，能够保证代码的整洁

需求

1. 定义函数 `demo1()` **提示用户输入一个整数并且返回**
2. 定义函数 `demo2()` 调用 `demo1()`
3. 在主程序中调用 `demo2()`

```python def demo1(): return int(input("请输入一个整数："))

def demo2(): return demo1()

try: print(demo2()) except ValueError: print("请输入正确的整数") except Exception as result: print("未知错误 %s" % result)
```

---

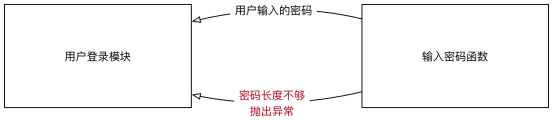
04. 抛出 `raise` 异常

4.1 应用场景

- 在开发中，除了 **代码执行出错** `Python` 解释器会 **抛出** 异常之外
- 还可以根据 **应用程序 特有的业务需求 主动抛出异常**

示例

- 提示用户 **输入密码**，如果 **长度少于 8**，抛出 **异常**



注意

- 当前函数 **只负责** 提示用户输入密码，如果 **密码长度不正确，需要其他的函数进行额外处理**
- 因此可以 **抛出异常**，由其他需要处理的函数 **捕获异常**

4.2 抛出异常

- `Python` 中提供了一个 `Exception` **异常类**
- 在开发时，如果满足 **特定业务需求时**，希望 **抛出异常**，可以：
  1. **创建** 一个 `Exception` 的 **对象**
  2. 使用 `raise` **关键字** 抛出 **异常对象**

需求

- 定义 `input_password` 函数，提示用户输入密码
- 如果用户输入长度 < 8，抛出异常
- 如果用户输入长度 >=8，返回输入的密码

```python def input_password():
```

```
# 1. 提示用户输入密码
pwd = input("请输入密码：")

# 2. 判断密码长度，如果长度 >= 8，返回用户输入的密码
if len(pwd) >= 8:
    return pwd

# 3. 密码长度不够，需要抛出异常
# 1> 创建异常对象 - 使用异常的错误信息字符串作为参数
ex = Exception("密码长度不够")

# 2> 抛出异常对象
raise ex
```

```
try: user*pwd = input*password() print(user_pwd) except Exception as result: print("发现错误：%s" % result)
```
