



Python

极夜酱

目录

1	Python 简介	1
1.1	Python 简介	1
1.2	注释	4
1.3	标识符	5
1.4	数据类型	7
1.5	数据输入	10
1.6	格式化输出	12
1.7	运算符	14
2	判断	16
2.1	逻辑运算符	16
2.2	if	17
2.3	断言	19
3	循环	20
3.1	while	20
3.2	for	24
3.3	break or continue?	29
4	数据结构	31
4.1	序列	31
4.2	列表	32
4.3	元组	39
4.4	字符串	42
4.5	字典	47
5	函数	51
5.1	函数	51
5.2	主函数	55
5.3	变量作用域	56

5.4	函数参数	59
5.5	lambda 表达式	62
5.6	递归	63
6	模块	73
6.1	模块导入	73
6.2	math 模块	78
6.3	random 模块	79
6.4	time 模块	80
6.5	calendar 模块	83
6.6	copy 模块	84
6.7	MapReduce 数据处理	88
6.8	pip 模块管理工具	90
6.9	jieba 分词	91
7	面向对象	93
7.1	面向过程与面向对象	93
7.2	类与对象	94
7.3	封装	97
7.4	构造方法与析构方法	99
7.5	继承	102
7.6	多态	106
8	文件操作	108
8.1	文件操作	108
8.2	文件缓冲	112
8.3	os 模块	114
8.4	csv 模块	116
8.5	pymysql 模块	119

Chapter 1 Python 简介

1.1 Python 简介

1.1.1 编程简介

程序 (program) 是为了让计算机执行某些操作或者解决问题而编写的一系列有序指令的集合。由于计算机只能够识别二进制数字 0 和 1，因此需要使用特殊的编程语言来描述如何解决问题过程和方法。

算法 (algorithm) 是可完成特定任务的一系列步骤，算法的计算过程定义明确，通过一些值作为输入并产生一些值作为输出。

流程图 (flow chart) 是算法的一种图形化表示方式，使用一组预定义的符号来说明如何执行特定任务。

- 圆角矩形：开始和结束
- 矩形：数据处理
- 平行四边形：输入/输出
- 菱形：分支判断条件
- 流程线：步骤

1.1.2 Python 简介

在当今的环境下 Python 是一个热门语言，在全世界的范围内，许多大学都开始使用 Python 作为基础语言的教程，同时在数据分析领域以及人工智能领域也取得了显著的成绩。



图 1.1: 计算 $\sum_{i=1}^{100} i$ 的流程图

Python 最大的特点就是简单。同样的一个功能，使用 C 语言实现需要 20 行代码，Java 需要 10 行代码，Python 的实现只需要 4 行代码，它从整体的代码量而言是非常简洁的。

- Python 具有很强的可读性，比其他语言更有特色语法结构。
- Python 是解释型语言，这意味着开发过程中没有了编译环节。
- Python 是交互式语言，这意味着可以在提示符后直接执行代码。
- Python 是面向对象语言，这意味着支持面向对象的风格和代码封装。

虽然 Python 提供有交互式的命令模式，但是在很多的情况下，对于程序的编写往往是将其定义在源文件之中，在 Python 里面所有的源文件的后缀名称必须是 .py。

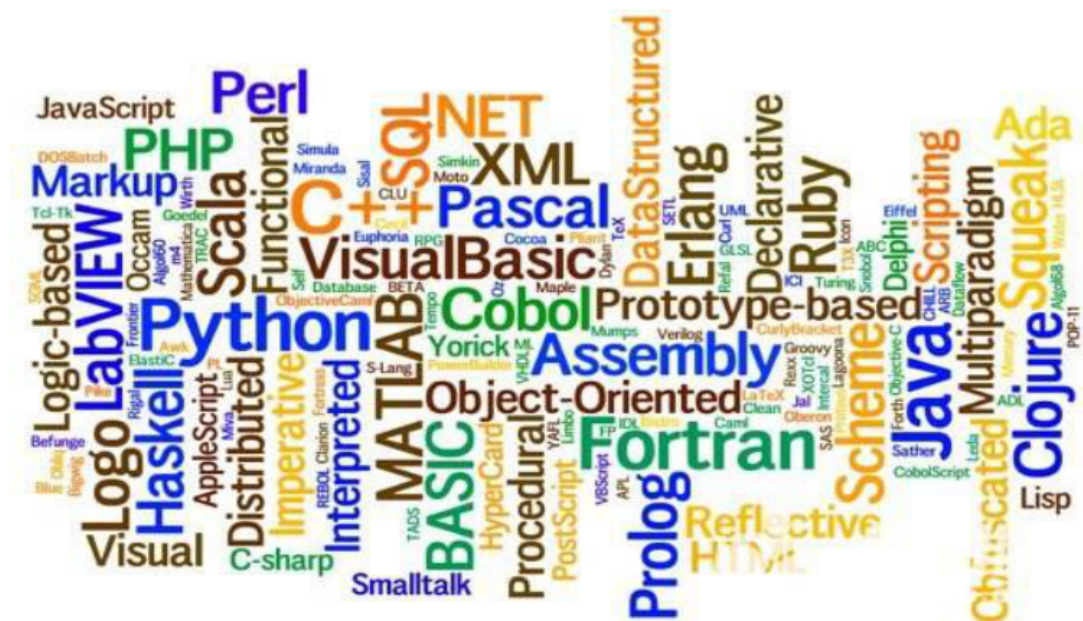


图 1.2: 常见编程语言

1.1.3 Hello World!

输出的时候使用 `print()` 函数，函数就是一个完成特定功能的代码组织结构。

Hello World!

```
1 print("Hello World!")
```

运行结果

Hello World!

1.2 注释

1.2.1 注释 (Comment)

在进行项目的开发过程中，不可能说一个项目编写完成一次后就在也不进行修改了。所以很多情况下，为了方便下一次的修改，会在一些关键性的代码上进行一些注释信息的定义，开发者根据这些注释的文字信息就可能直到这段代码的主要作用，方便代码的维护。

Python 里面的注释分为两类：

1. 单行注释：将一行内【#】之后的内容视为注释。
2. 多行注释：三引号中间的内容视为注释。

注释

```
1 """  
2 这是一条  
3 多行注释  
4 """  
5 print("Hello World!")    # 输出Hello World!
```

运行结果

Hello World!

1.3 标识符

1.3.1 标识符 (Identifier)

标识符的第一个字符必须是字母表中字母或下划线 **【_】**，标识符的其它部分由字母、数字和下划线组成，标识符对大小写敏感。标识符不可以使用保留字或关键字。标识符应该准确、顾名思义，不要使用汉语拼音。

关键字 (key word) 也称保留字，关键字是编程语言内置的一些名称，具有特殊的用处和意义。保留字不能用作于标识符名称。Python 的标准库提供了一个 keyword 模块，可以输出当前版本的所有关键字。

关键字

```
1 import keyword
2 print(keyword.kwlist)
```

False	None	True	and	as
assert	break	class	continue	def
del	elif	else	except	finally
for	from	global	if	import
in	is	lambda	nonlocal	not
or	pass	raise	return	try
while	with	yield		

表 1.1: 关键字

1.3.2 变量 (Variable)

在程序之中所谓的变量指的是可以被改变的内容，而常量指的是绝对不会被改变的内容。Python 语言最大的特点是变量都是可以被直接定义的，不需要那些复杂的数据类型的声明，直接使用变量名称即可。

所有的变量实际上都会占据内存空间，当一些变量不再使用的时候，可以使用 `del` 关键字对内存空间进行删除。变量一旦被删除了，那么后续的代码部分将无法继续使用它。

变量

```
1 num = 10
2 print(num)
3 del num
4 print(num)
```

运行结果

```
10
NameError: name 'a' is not defined
```

1.4 数据类型

1.4.1 数据类型

Python 中的变量不需要声明，但是每个变量在使用前都必须赋值，变量赋值以后该变量才会被创建。在 Python 中变量没有类型，我们所说的“类型”是变量所指的内存中对象的类型。使用 `type()` 函数可以获取变量的数据类型。

在 Python 中使用等号【=】用来给变量赋值，赋值运算符左边是一个变量名，赋值运算符右边是存储在变量中的值。

布尔是 19 世纪一位英国数学家的名字，在开发之中布尔主要用于程序的逻辑分支处理，数值包括 `True` 和 `False`。

按照各个传统编程语言的做法来讲，整数 / 整数 = 整数，但是 Python 认为这个结果应该包含有小数（整数的结果变为浮点数类型）。

数据类型

```
1 num = 123      # 整型
2 PI = 3.1415    # 浮点型
3 s = "hello"    # 字符串
4 flag = True    # 布尔
5
6 print(num)
7 print(PI)
8 print(s)
9 print(flag)
10
11 print(type(num))
12 print(type(PI))
13 print(type(s))
14 print(type(flag))
15
```

```
16 print(10 / 4)
17 print(type(10 / 4))
```

运行结果

```
123
3.1415
hello
True
<class 'int'>
<class 'float'>
<class 'str'>
<class 'bool'>
2.5
<class 'float'>
```

1.4.2 字符串 (String)

字符串是开发中最为重要的概念，一个编程语言是否好用，很大程度上也是取决于是否提供有字符串类型。Python 中可以直接使用引号（单引号或双引号）进行字符串的定义。Python 并没有字符的概念，所以对于引号表示的含义是相同的。

使用【+】运算符可以进行字符串的连接处理。

字符串连接

```
1 str = "Hello" + "World"
2 str += "!"
3 print(str)
```

运行结果

HelloWorld!

1.4.3 转义字符

在一个字符串描述的过程中，有可能会有一些特殊字符的信息。

转义字符	描述
\\	表示一个反斜杠 \
\'	表示一个单引号 '
\"	表示一个双引号 "
\n	换行
\t	横向制表符

表 1.2: 转义字符

转义字符

```
1 print("全球最大同性交友网站\n\"https://github.com\"")
```

运行结果

全球最大同性交友网站
"https://github.com"

1.5 数据输入

1.5.1 input()

input() 用于接受键盘输入。在所有编程语言里面, 输入数据的类型是字符串类型。

数据输入

```
1 data = input("输入数据: ")
2 print(data)
3 print(type(data))
```

运行结果

输入数据: 123

123

<class 'str'>

1.5.2 转换函数

很多时候可能需要的是各种类型, 例如: 整数、浮点数、布尔型, 或者将其它类型变为字符串型。如果字符串不是由数字所组成, 那么程序的执行就会产生异常。

函数	功能
int(数据)	将指定数据转为整型数据
float(数据)	将指定数据转为浮点型数据
bool(数据)	将指定数据转为布尔型数据
str(数据)	将指定数据转为字符串型数据

表 1.3: 转换函数

加法计算 (Bug 版本)

```
1 num1 = float(input("输入第一个数字: "))
2 num2 = float(input("输入第二个数字: "))
3 result = num1 + num2
4 print(num1 + "+" + num2 + "=" + result)
```

运行结果

输入第一个数字: 11.1

输入第二个数字: 22.2

TypeError: unsupported operand type(s) for +: 'float' and 'str'

加法计算（正确版本）

```
1 num1 = float(input("输入第一个数字: "))
2 num2 = float(input("输入第二个数字: "))
3 result = num1 + num2
4 print(str(num1) + "+" + str(num2) + "=" + str(result))
```

运行结果

输入第一个数字: 11.1

输入第二个数字: 22.2

11.1+22.2=33.3

1.6 格式化输出

1.6.1 格式化输出

为了解决数据的输出问题，Python 提供有格式化的输出操作。

标记	功能
%c	格式化字符
%s	格式化字符串
%d	格式化整型
%f	格式化浮点型，可以设置保留精度
%e	科学计数法，使用小写字母 e
%E	科学计数法，使用大写字母 e
%g	%f 和 %e 的简写
%G	%f 和 %E 的简写
%u	格式化无符号整型
%o	格式化无符号八进制数
%x	格式化无符号十六进制数
%X	格式化无符号十六进制数（大写字母）

表 1.4: 格式化输出

格式化字符串

```
1 name = "小灰"
2 age = 16
3 height = 175.6
4
5 print("姓名: %s, 年龄: %d, 身高: %.2f" % (name, age, height))
```

运行结果

姓名: 小灰, 年龄: 16, 身高: 175.60

1.6.2 分隔符

在默认情况下，使用 `print()` 输出数据时，都会使用换行作为分隔符号。如果不希望使用换行，则可以追加一个 `end` 参数。

分隔符

```
1 sequence = "1 2 4 8 16 32 64"  
2 print("序列: ", end='')  
3 print(sequence, end='...')
```

运行结果

序列: 1 2 4 8 16 32 64...

1.7 运算符

1.7.1 算术运算符

运算符	功能
+	两个数相加
-	得到负数或一个数减去另一个数
*	两个数相乘或是返回一个被重复若干次的字符串
/	两数相除
%	返回除法的余数
**	幂
//	整除（向下取整）

表 1.5: 算术运算符

计算圆的面积

```
1 PI = 3.14159
2 r = float(input("输入半径: "))
3 area = PI * r ** 2
4 print("面积 = %.2f" % area)
```

运行结果

输入半径: 5
面积 = 78.54

逆序三位数

```
1 num = int(input("输入一个正三位数: "))
2 a = num // 100
3 b = num // 10 % 10
4 c = num % 10
```

```
5 print("逆序: %d" % (c*100 + b*10 + a))
```

运行结果

输入一个正三位数：520

逆序：25

1.7.2 复合赋值运算符

运算符	功能
+=	a += b 等价于 a = a + b
-=	a -= b 等价于 a = a - b
*=	a *= b 等价于 a = a * b
/=	a /= b 等价于 a = a / b
%=	a %= b 等价于 a = a % b
**=	a **= b 等价于 a = a ** b
//=	a //= b 等价于 a = a // b

表 1.6: 复合赋值运算符

1.7.3 比较运算符

运算符	功能
==	等于
!=	不等于
>	大于
<	小于
>=	大于等于
<=	小于等于

表 1.7: 比较运算符

Chapter 2 判断

2.1 逻辑运算符

2.1.1 逻辑运算符

Python 中逻辑运算符有三种：

1. 逻辑与 and：当多个条件同时为真，结果为真。

条件 1	条件 2	条件 1 and 条件 2
T	T	T
T	F	F
F	T	F
F	F	F

表 2.1: 逻辑与

2. 逻辑或 or：多个条件有一个为真时，结果为真。

条件 1	条件 2	条件 1 or 条件 2
T	T	T
T	F	T
F	T	T
F	F	F

表 2.2: 逻辑或

3. 逻辑非 not：条件为真时，结果为假；条件为假时，结果为真。

条件	not 条件
T	F
F	T

表 2.3: 逻辑非

2.2 if

2.2.1 if

分支结构最大特征就是可以进行指定条件的判断处理，关键字为 if、elif、else。

每一个满足条件之后的语句都可以有多条，并且在 Python 里面是利用缩进来确定语句的关系。使用逻辑运算符可以进行若干个条件的连接。

单分支

```
1 age = 15
2 if 0 < age < 18:
3     print("未成年")
```

双分支

```
1 age = 30
2 if 0 < age < 18:
3     print("未成年人")
4 else:
5     print("成年人")
```

多分支

```
1 score = 76
2
3 if 90 <= score <= 100:
4     print("优秀")
5 elif score >= 60:
6     print("合格")
7 else:
8     print("不合格")
```

判断整数奇偶

```
1 num = int(input("输入一个正整数: "))
2
3 if num > 0:
4     if num % 2 == 0:
5         print("%d是偶数" % num)
6     else:
7         print("%d是奇数" % num)
```

运行结果

输入一个正整数: 66

66是偶数

2.3 断言

2.3.1 断言 (Assertion)

设置断言表达式后，当满足条件时程序正常执行，当断言失败时，程序会中断执行。在进行断言时可以配置错误的提示信息，否则很难知道那块代码出现了错误。

通过断言可以直接查找出程序的错误，但从另外一个角度来讲，断言由于不受到程序逻辑的控制，可能会造成许多的额外的问题，在实际的开发之中慎用。

断言

```
1 import math
2
3 print("计算三角形面积")
4 a = float(input("第一条边: "))
5 b = float(input("第二条边: "))
6 c = float(input("第三条边: "))
7
8 assert a + b > c, "边长不合法"
9 assert a + c > b, "边长不合法"
10 assert b + c > a, "边长不合法"
11
12 p = (a + b + c) / 2    # 半周长
13 area = math.sqrt(p * (p-a) * (p-b) * (p-c)) # 海伦公式
14 print("面积 = %.2f" % area)
```

运行结果

计算三角形面积

第一条边: 1

第二条边: 1

第三条边: 2

AssertionError: 边长不合法

Chapter 3 循环

3.1 while

3.1.1 while

在 while 循环中，当条件满足时重复循环体内的语句。如果条件永远为真，循环会永无止境的进行下去（死循环），因此循环体内要有改变条件的机会。

控制循环次数的方法就是设置循环变量：初值、判断、更新。

while 循环的特点是先判断、再执行，所以循环体有可能会进入一次或多次，也有可能一次也不会进入。

```
1 while condition:
2     # code
```

计算 5 个人的平均身高

```
1 total = 0
2 i = 1
3
4 while i <= 5:
5     height = float(input("输入第%d个人的身高: " % i))
6     total += height
7     i += 1
8
9 average = total / 5
10 print("平均身高: %.2f" % average)
```

运行结果

输入第1个人的身高：160.8

输入第2个人的身高：175.2

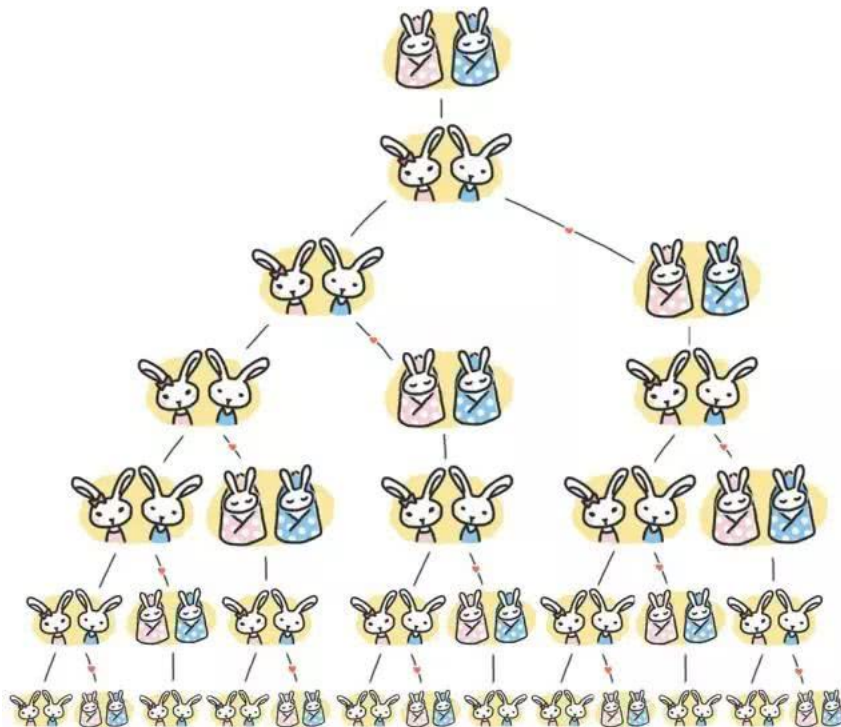
输入第3个人的身高：171.2

输入第4个人的身高：181.3

输入第5个人的身高：164

平均身高：170.5

斐波那契数列



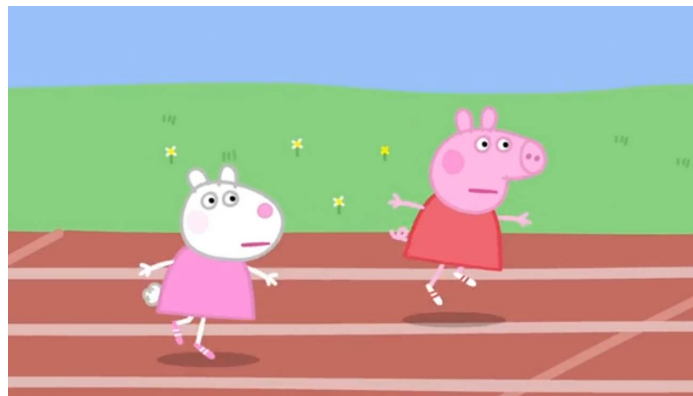
```
1 num1 = 0
2 num2 = 1
3 while num2 < 1000:
4     print(num2, end=' ')
5     num1, num2 = num2, num1 + num2
```


运行结果

1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987

3.1.2 死循环

对于循环的操作而言，一定要避免死循环所带来的问题。所谓的死循环指的就是循环结束条件没有得到正常的修改，导致程序无法结束。如果在执行中真的出现了死循环，命令行方式下可以直接按下 Ctrl+C 来中断执行。



猜数字

```
1 import random          # 随机模块
2
3 answer = random.randint(1, 100)    # 产生1-100之间的随机数
4 cnt = 0                    # 猜测次数
5
6 while True:
7     num = int(input("猜一个1-100之间的数字: "))
8     cnt += 1
9
10    if num > answer:
11        print("猜大了")
12    elif num < answer:
13        print("猜小了")
```

```
14     else:         # 猜对了就跳出循环
15         break
16
17 print("猜对了! 你一共用了%d次猜对! " % cnt)
```

运行结果

猜一个1-100之间的数字: 50
猜大了!
猜一个1-100之间的数字: 25
猜小了!
猜一个1-100之间的数字: 37
猜小了!
猜一个1-100之间的数字: 43
猜小了!
猜一个1-100之间的数字: 46
猜小了!
猜一个1-100之间的数字: 48
猜小了!
猜一个1-100之间的数字: 49
猜对了! 你一共用了7次猜对!

3.2 for

3.2.1 for

for 循环的功能是在输出指定范围内的数据操作。在每一次 for 循环里面都会自动的获取给定范围中的每一个元素，并且在 for 语句里面进行相关的处理操作。

range() 生成一个从 0 开始的数据范围，并且是按照线性的方式增长的。range() 也可以设置数据的开始值，默认从 0 开始。range() 的步长默认为 1。

for

```
1 for i in range(5):
2     print(i, end=' ')
3 print()
4
5 for i in range(10, 20):
6     print(i, end=' ')
7 print()
8
9 for i in range(1, 10, 2):
10    print(i, end=' ')
11 print()
```

运行结果

```
0 1 2 3 4
10 11 12 13 14 15 16 17 18 19
1 3 5 7 9
```

计算 1 ~ 100 的累加和

```
1 sum = 0
```

```

2 for i in range(1, 101):
3     sum += i
4 print("累加: %d" % sum)

```

运行结果

累加: 5050

计算 $1 + \frac{1}{2} + \frac{1}{3} + \dots + \frac{1}{n}$

```

1 sum = 0
2 n = int(input("输入n: "))
3
4 for i in range(1, n+1):
5     sum += 1 / i
6 print(sum)

```

运行结果

输入n: 10

2.9289682539682538

for 循环主要是通过序列的形式完成的范围设置，而字符串也属于序列。

单个的字符是一种特殊的类型，是用单引号表示字符字面量。每一个字符都有其对应的码值。

ASCII 全称 American Standard Code for Information Interchange（美国信息交换标准代码），一共定义了 128 个字符。

ASCII	字符	ASCII	字符	ASCII	字符	ASCII	字符
0	NUL	32	(space)	64	@	96	`
1	SOH	33	!	65	A	97	a
2	STX	34	"	66	B	98	b

3	ETX	35	#	67	C	99	c
4	EOT	36	\$	68	D	100	d
5	ENQ	37	%	69	E	101	e
6	ACK	38	&	70	F	102	f
7	BEL	39	'	71	G	103	g
8	BS	40	(72	H	104	h
9	HT	41)	73	I	105	i
10	LF	42	*	74	J	106	j
11	VT	43	+	75	K	107	k
12	FF	44	,	76	L	108	l
13	CR	45	-	77	M	109	m
14	SO	46	.	78	N	110	n
15	SI	47	/	79	O	111	o
16	DLE	48	0	80	P	112	p
17	DC1	49	1	81	Q	113	q
18	DC2	50	2	82	R	114	r
19	DC3	51	3	83	S	115	s
20	DC4	52	4	84	T	116	t
21	NAK	53	5	85	U	117	u
22	SYN	54	6	86	V	118	v
23	TB	55	7	87	W	119	w
24	CAN	56	8	88	X	120	x
25	EM	57	9	89	Y	121	y
26	SUB	58	:	90	Z	122	z
27	ESC	59	;	91	[123	{
28	FS	60	<	92	/	124	
29	GS	61	=	93]	125	}
30	RS	62	>	94	^	126	~
31	US	63	?	95	_	127	DEL

表 3.1: ASCII 码表

小写字母转大写

```
1 str = "Hello World!"
2
3 for s in str:
4     # 小写字母
5     if 97 <= ord(s) <= 122:      # ord(): 字符转ASCII码
6         s = chr(ord(s) - 32)    # chr(): ASCII码转字符
7     print(s, end='')
```

运行结果

HELLO WORLD!

3.2.2 嵌套循环

循环也可以进行嵌套使用。

九九乘法表

1*1=1	1*2=2	1*3=3	1*4=4	1*5=5	1*6=6	1*7=7	1*8=8	1*9=9
2*1=2	2*2=4	2*3=6	2*4=8	2*5=10	2*6=12	2*7=14	2*8=16	2*9=18
3*1=3	3*2=6	3*3=9	3*4=12	3*5=15	3*6=18	3*7=21	3*8=24	3*9=27
4*1=4	4*2=8	4*3=12	4*4=16	4*5=20	4*6=24	4*7=28	4*8=32	4*9=36
5*1=5	5*2=10	5*3=15	5*4=20	5*5=25	5*6=30	5*7=35	5*8=40	5*9=45
6*1=6	6*2=12	6*3=18	6*4=24	6*5=30	6*6=36	6*7=42	6*8=48	6*9=54
7*1=7	7*2=14	7*3=21	7*4=28	7*5=35	7*6=42	7*7=49	7*8=56	7*9=63
8*1=8	8*2=16	8*3=24	8*4=32	8*5=40	8*6=48	8*7=56	8*8=64	8*9=72
9*1=9	9*2=18	9*3=27	9*4=36	9*5=45	9*6=54	9*7=63	9*8=72	9*9=81

表 3.2: 九九乘法表

```
1 for i in range(1, 10):
2     for j in range(1, 10):
3         print("%d*%d=%d" % (i, j, i*j), end='\t')
4     print()
```

输出图案

```
1 *
2 **
3 ***
4 ****
5 *****
```

```
1 for i in range(5):
2     for j in range(i+1):
3         print("*", end=' ')
4     print()
```

3.3 break or continue?

3.3.1 循环控制

循环控制语句的作用是控制当前的循环结构是否继续向下执行，如果不进行控制，那么会根据既定的结构重复执行。如果有一些特殊的情况导致循环的执行中断，就称为循环的控制语句。循环控制语句的关键字有 `break` 和 `continue`。

`break` 的作用是跳出当前循环，执行当前循环之后的语句。`break` 只能跳出一层循环，如果是嵌套循环，那么需要按照嵌套的层次，逐步使用 `break` 来跳出。`break` 语句只能在循环体内和 `switch` 语句内使用。

`continue` 的作用是跳过本轮循环，开始下一轮循环的条件判断。`continue` 终止当前轮的循环过程，但它并不跳出循环。

break

```
1 for i in range(10):
2     if i == 5:
3         break
4     print(i, end=' ')
```

运行结果

1 2 3 4

continue

```
1 for i in range(10):
2     if i == 5:
3         continue
4     print(i, end=' ')
```


运行结果

1 2 3 4 6 7 8 9 10

Chapter 4 数据结构

4.1 序列

4.1.1 序列

Python 中序列类型包含字符串、列表、元组、字典。序列的核心意义在于可以进行多个数据的保存。

Python 在整体设计的过程之中强调的是简单化。以多数据的存储为例，在许多的编程语言里面，都是利用了数组实现了数据的存储，但是数据有一个最大的问题在于：长度是固定的，无法进行容量的扩充。

在这样的背景下，很多的开发者就不得不去独立地进行一些动态数组的开发，所以就有了数据结构的概念，而数据存储的内容多了，就需要提升数据的操作性能，C、C++、Java 等等都有这样的问题。

Python 在设计的时候充分地考虑到了这些动态性的设计问题，所以才将这些可能动态修改的内容统一地称为序列，也就是说 Python 中的序列就是一个动态（或静态）的存储，而这些存储的操作结构都是内置的，开发者可以避免数据结构的开发所造成的困难，尤其是面对与非科班的人而言。

列表是对传统数组的一种使用包装，但是与传统数组最大的不同在于，Python 中的列表的内容是允许进行动态修改的，并且 Python 中的列表也可以像传统数组那样通过索引的访问，这样就使得时间复杂度降低了许多。

4.2 列表

4.2.1 列表 (List)

列表使用 `[]` 进行列表的定义, Python 的列表是对于传统数组的更高级的实现。列表可以通过索引的形式进行访问, 索引下标是从 0 开始的, 到列表长度 - 1 结束。

在使用列表进行数据存储的时候, 虽然大部分的情况下都会使用相同的数据类型, 但是在列表里面却可以同时有不同的数据类型, 这比其它语言里面提供的数据的功能强大。虽然提供有多个数据类型的保存使得程序存储更加灵活, 但是从实际的开发角度而言, 尽量让数据类型保持一致会比较合理。

在 Python 中列表除了正向索引访问之外, 也可以进行反向索引访问。

列表

```
1 lst = [1, 2, 3]
2
3 print(lst[0])
4 print(lst[1])
5 print(lst[2])
6
7 print(lst[-1])
8 print(lst[-2])
9 print(lst[-3])
10
11 print(lst[3])      # 越界
```

运行结果

```
1
2
3
3
2
1
IndexError: list index out of range
```

序列数据可以直接使用【*】进行重复定义，或者使用【+】进行与其它序列拼接。

序列重复/拼接

```
1 lst = [1, 2, 3] * 3
2 print(lst)      # [1, 2, 3, 1, 2, 3, 1, 2, 3]
3
4 lst = [1, 2, 3] + [4, 5, 6]
5 print(lst)      # [1, 2, 3, 4, 5, 6]
```

运行结果

```
[1, 2, 3, 1, 2, 3, 1, 2, 3]
[1, 2, 3, 4, 5, 6]
```

4.2.2 成员运算符

在使用索引访问的时候都需要去考虑索引的设置错误的问题，但是使用了 for 循环之后，不再需要明确的进行索引的设置，避免了 IndexError 异常信息。

如果想要判断某一个数据是否在列表之中存在，最直接的方式就是直接进行列表的迭代操作（其它语言传统形式）。

迭代查找

```
1 lst = ["C/C++", "Java", "Python", "JavaScript"]
2 key = "Python"      # 待查找关键词
3 flag = False        # 初始假设未找到
4
5 for item in lst:
6     if item == key:
7         flag = True
8         break
9
10 if flag:
11     print("数据存在")
12 else:
13     print("数据不存在")
```

运行结果

数据存在

这种查询最大的问题在于需要进行整个列表数据的迭代，造成的结果就是时间复杂度的攀升 $O(n)$ 。在 Python 设计过程之中不希望开发者为这些复杂的执行效率犯愁，所以在使用列表判断某些数据是否存在的时候提供了专门的运算符。使用关键字 `in`（在范围内）、`not in`（不在范围内）。

成员运算符

```
1 lst = ["C/C++", "Java", "Python", "JavaScript"]
2 key = "Python"      # 待查找关键词
3
4 if key in lst:
5     print("数据存在")
6 else:
7     print("数据不存在")
```

运行结果

数据存在

4.2.3 切片 (Slicing)

切片可以截取部分的列表内容。在默认情况下，数据的分片都是依照数值为 1 的间隔进行获取的，如果有需要也可以进行步长的修改。

切片

```
1 lst = list(range(10))
2
3 print("原始列表: ", end='')
4 print(lst)
5
6 print("截取[2:7]: ", end='')
7 print(lst[2:7])
8
9 print("截取[:5]: ", end='')
10 print(lst[:5])
11
12 print("截取[3:]: ", end='')
13 print(lst[3:])
14
15 print("截取[::2]: ", end='')
16 print(lst[::2])
```

运行结果

原始列表: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

截取[2:7]: [2, 3, 4, 5, 6]

截取[:5]: [0, 1, 2, 3, 4]

截取[3:]: [3, 4, 5, 6, 7, 8, 9]

截取[::2]: [0, 2, 4, 6, 8]

4.2.4 列表操作函数

Python 中的列表设计非常到位, 它可以实现内容动态扩充, 可以进行后期的追加或者删除。

函数	功能
append(data)	在列表最后追加新内容
clear()	清除列表数据
copy()	列表拷贝
count()	统计某一个数据在列表中的出现次数
extend(列表)	为一个列表追加另外一个列表
index(data)	从列表查询某个值第一次出现的位置
insert(index, data)	向列表中指定索引位置追加新数据
pop(index)	从列表弹出并删除一个数据
remove(data)	从列表删除数据
reverse()	列表数据反转
sort()	列表数据排序

表 4.1: 列表操作函数

列表内容扩充

```
1 lst = list(range(5))    # [0, 1, 2, 3, 4]
2 lst.append(5)           # [0, 1, 2, 3, 4, 5]
```

```
3 lst.insert(2, 6)          # [0, 1, 6, 2, 3, 4, 5]
4 print(lst)
5 print("列表长度: %d" % len(lst))
```

运行结果

[0, 1, 6, 2, 3, 4, 5]

列表长度: 7

在使用 `remove()` 删除的时候，内容存在可以删除，不存在就会抛出 `ValueError` 异常信息，因此使用 `remove()` 操作之前一定要通过 `in` 判断。

如果不知道内容要进行数据的删除，最简单的原始的 Python 支持可以采用 `del` 关键字实现内容删除，而且使用 `del` 的时候只需要知道列表数据的索引即可实现。使用 `del` 关键字的确可以实现索引的删除，但是无法知道被删除了哪些数据。

`pop()` 可以根据索引删除，而后告诉用户哪些数据被删除了。

删除元素

```
1 lst = list(range(5))    # [0, 1, 2, 3, 4]
2 key = 3
3
4 if key in lst:
5     lst.remove(key)
6 print(lst)              # [0, 1, 2, 4]
7
8 del lst[2]
9 print(lst)              # [0, 1, 4]
10
11 print(lst.pop(1))       # 1
12 print(lst)              # [0, 4]
```


运行结果

[0, 1, 2, 4]

[0, 1, 4]

1

[0, 4]

列表反转/排序

```
1 lst = [7, 4, 0, 3, 6, 8, 9, 2]
2
3 print("原始列表: ", end='')
4 print(lst)
5
6 print("反转: ", end='')
7 lst.reverse()
8 print(lst)
9
10 print("排序（升序）: ", end='')
11 lst.sort()
12 print(lst)
13
14 print("排序（降序）: ", end='')
15 lst.sort(reverse=True)
16 print(lst)
```

运行结果

原始列表: [7, 4, 0, 3, 6, 8, 9, 2]

反转: [2, 9, 8, 6, 3, 0, 4, 7]

排序（升序）: [0, 2, 3, 4, 6, 7, 8, 9]

排序（降序）: [9, 8, 7, 6, 4, 3, 2, 0]

4.3 元组

4.3.1 元组 (Tuple)

列表是可以进行多个数据的保存，并且里面的内容都是可以修改和动态扩充的。与列表不一样，Python 提供了一个不可变的数据的序列——元组，元组需要使用 **【()】** 进行定义。元组定义的时候如果只有一个内容，必须要有 **【,】**。

元组

```
1 tup = (0, 1, 2, 3, 4)
2 print(tup)
3 tup[0] = 5
```

运行结果

```
(0, 1, 2, 3, 4)
```

```
TypeError: 'tuple' object does not support item assignment
```

tuple() 可以实现列表与元组的转换。

元组与列表的转换

```
1 lst = list(range(5))
2 print(lst)
3 print(type(lst))
4
5 tup = tuple(lst)
6 print(tup)
7 print(type(tup))
```

运行结果

```
[0, 1, 2, 3, 4]
<class 'list'>
(0, 1, 2, 3, 4)
<class 'tuple'>
```

4.3.2 序列统计函数

函数	功能
len()	获取序列的长度
max()	获取序列中的最大值
min()	获取序列中的最小值
sum()	计算序列中的内容总和
any()	序列中有一个为 True 结果为 True，否则为 False
all()	序列中有一个为 False 结果为 False，否则为 True

表 4.2: 序列统计函数

序列统计函数

```
1 lst = [4, 0, 1, 3, 2]
2 tup = (8, 5, 7, 9, 6)
3 str = "HelloWorld"
4
5 print("列表lst的长度 = %d" % len(lst))
6 print("元组tup的长度 = %d" % len(tup))
7 print("字符串str的长度 = %d" % len(str))
8
9 print("列表lst最大值: %d" % max(lst))
10 print("元组tup最小值 = %d" % min(tup))
11 print("字符串str最大值 = %c" % max(str))
```

```
12 print("字符串str最小值 = %c" % min(str))
13
14 print("列表lst总和: %d" % sum(lst))
15 print("元组tup总和: %d" % sum(tup))
```

运行结果

列表lst的长度 = 5
元组tup的长度 = 5
字符串str的长度 = 10
列表lst最大值: 4
元组tup最小值 = 5
字符串str最大值 = r
字符串str最小值 = H
列表lst总和: 10
元组tup总和: 35

4.4 字符串

4.4.1 字符串 (String)

字符串是由若干个字符所组成的，使用引号进行一串内容的声明。字符串也可以进行数据的分片操作和序列统计函数。

字符串

```
1 info = "url: www.baidu.com"
2
3 print("字符串长度: %d" % len(info))
4
5 if "url: " in info:
6     print(info[5:])
```

运行结果

```
字符串长度: 18
www.baidu.com
```

4.4.2 字符串格式化

在 Python 默认的语言环境中字符串格式化的处理支持。为了进一步完善对于字符串格式化的需求，又提供了 `format()`。在 `format()` 里面出现的 **【{}】** 标记里面可以定义标记名称或者数据的填充序列顺序定义。

字符串格式化

```
1 name = "小灰"
2 age = 16
3 height = 175.6
```

```

4
5 info = "姓名: %s, 年龄: %d, 身高: %.2f" % (name, age, height)
6 print(info)
7
8 info = "姓名: {}, 年龄: {}, 身高: {}".format(name, age, height)
9 print(info)

```

运行结果

姓名: 小灰, 年龄: 16, 身高: 175.60

姓名: 小灰, 年龄: 16, 身高: 175.6

4.4.3 字符串数据处理函数

函数	功能
center()	字符串居中显示
find(data)	查找到内容返回索引位置, 找不到返回-1
join(data)	字符串连接
split(data [, limit])	字符串拆分
lower()	字符串转小写
upper()	字符串转大写
capitalize()	首字母大写
replace(old, new [, limit])	字符串替换
strip()	删除左右空格

表 4.3: 字符串数据处理函数

字符串大小写转换

字符串大小写转换

```

1 str = "Hello World!"

```

```
2 print(str.upper())
3 print(str.lower())
```

运行结果

```
HELLO WORLD!
hello world!
```

字符串查找 find()

在进行数据内容查询的时候，如果内容存在就返回索引位置，不存在则返回-1。

字符串查找

```
1 str = "Hello World!"
2 print(str.find("Wor"))
3 print(str.find("Python"))
```

运行结果

```
6
-1
```

字符串连接 join()

使用一个特定的字符串连接序列中的若干内容。

字符串连接

```
1 lst = ["www", "baidu", "com"]
2 url = ".".join(lst)
3 print(url)
```

运行结果

www.baidu.com

字符串拆分 split()

字符串拆分

```
1 ip = "127.0.0.1"
2 print("ip信息: %s" % ip.split("."))
3
4 date_time = "2021/03/31 17:32:53"
5 item = date_time.split(" ")
6 print("date信息: %s" % item[0].split("/"))
7 print("time信息: %s" % item[1].split(":"))
```

运行结果

```
ip信息: ['127', '0', '0', '1']
date信息: ['2021', '03', '31']
time信息: ['17', '32', '53']
```

字符串替换 replace()

使用一个字符串去替换其它的字符串，在进行字符串替换的时候可以设置替换的次数。replace() 的替换是进行全匹配的替换操作。

字符串替换

```
1 str = "Hello World, Hello Python!"
2 print("全部替换: " + str.replace("Hello", "Hey"))
3 print("部分替换: " + str.replace("Hello", "Hey", 1))
```


运行结果

全部替换: Hey World, Hey Python!

部分替换: Hey World, Hello Python!

字符串去除前后空格 strip()

在用户键盘数据输入的过程中,有可能用户在输入信息时会添加无用的空格,导致数据判断错误。对于用户输入数据需要进行左右空格的删除。

模拟登陆

```
1 while True:
2     info = input("Username:Password >>> ")
3     if len(info) == 0 or info.find(":") == -1:
4         print("输入格式错误! ")
5     else:
6         data = info.split(":")
7         username = data[0]
8         password = data[1]
9         if username == "admin" and password == "123456":
10            print("欢迎【%s】成功登录! " % username)
11            break
12        else:
13            print("用户名或密码错误! ")
```

运行结果

Username:Password >>> hey:

用户名或密码错误!

Username:Password >>> admin

输入格式错误!

Username:Password >>> admin:123456

欢迎【admin】成功登录!

4.5 字典

4.5.1 字典 (Dictionary)

字典是一个在开发之中极为重要的类型，字典提供了非常方便地数据内容查找操作。字典的本质就是一个数据查找的序列，与之前的列表或者元组不同的地方在于，其它的结构保存数据的目的都是为了输出使用，而字典是为了查询使用。

字典严格意义上来讲属于一种哈希表的结构，在哈希表进行数据存储的时候往往都需要一个哈希算法进行数据存储位置的计算。

字典是一个二元偶对象的集合，所以里面保存的数据都是成对的，所有的数据内容都按照 `key = value` 的形式进行存放。考虑到用户使用的方便，`key` 的类型可以使数字、字符串或者是元组，但是最为常见的还是字符串。

在使用字典进行查询的时候如果指定的 `key` 不存在，则在代码执行会出现 `KeyError` 错误提示信息，也就是原生的数据查询方式是有可能抛出异常的。

字典的本质是根据 `key` 查找对应的 `value`，一旦 `key` 重复的时候，使用新的数据覆盖掉旧的数据。

字典

```
1 info = {"name": "小灰", "age": 16, "height": 175.6}
2 print(info)
3 print(info["age"])
4
5 info["height"] = 180
6 print(info)
```

运行结果

```
{'name': '小灰', 'age': 16, 'height': 175.6}  
16  
{'name': '小灰', 'age': 16, 'height': 180}
```

既然字典是一个序列，那么字典可以使用 `in` 进行判断指定 `key` 是否存在，通过这样的机制可以避免由于 `key` 不存在所带来的 `KeyError` 异常的抛出。

同样，字典也可以使用 `for` 循环迭代输出，但是迭代的知识字典中全部的 `key`。

字典迭代输出

```
1 info = {"name": "小灰", "age": 16, "height": 175.6}  
2 for key in info:  
3     print("%s: %s" % (key, info[key]))
```

运行结果

```
name: 小灰  
age: 16  
height: 175.6
```

在进行字典数据迭代输出的时候，最好的做法是每一次迭代直接返回当前完整的 `key:value` 的映射项，此时的处理就需要依赖 `items()` 来完成。

`items()` 迭代输出

```
1 info = {"name": "小灰", "age": 16, "height": 175.6}  
2 for key, value in info.items():  
3     print("%s: %s" % (key, value))
```

运行结果

name: 小灰
age: 16
height: 175.6

4.5.2 字典操作函数

函数	功能
clear()	清空字典数据
update({k:v, ...})	更新字典数据
get(key[, defaultvalue])	根据 key 获取数据
pop(key)	弹出字典中指定的 key 数据
popitem()	从字典中弹出一组映射项
keys()	返回字典中全部 key 数据
values()	返回字典中全部的 value 数据

表 4.4: 字典操作函数

字典数据更新 update()

Python 中的字典是可以进行存储数据的动态扩充的，update() 除了拥有表面上的更新之外，也拥有数据的扩充操作。

字典数据更新 update()

```
1 info = {"name": "小灰", "age": 16, "height": 175.6}
2 info.update({"city": "Shanghai", "job": "programmer"})
3 print(info)
```

运行结果

```
{'name': '小灰', 'age': 16, 'height': 175.6,  
'city': 'Shanghai', 'job': 'programmer'}
```

字典数据删除 pop()

pop() 可以根据 key 进行弹出操作。

字典数据删除

```
1 info = {"name": "小灰", "age": 16, "height": 175.6}  
2 print(info.pop("age"))  
3 print(info)
```

运行结果

```
16  
{'name': '小灰', 'height': 175.6}
```

字典数据获取 get()

可以通过 key 获取数据，如果 key 不存在则返回 None。

字典数据获取

```
1 info = {"name": "小灰", "age": 16, "height": 175.6}  
2 print(info.get("name"))  
3 print(info.get("job"))
```

运行结果

```
小灰  
None
```

Chapter 5 函数

5.1 函数

5.1.1 函数 (Function)

函数执行一个特定的任务，Python 提供了大量内置函数，例如 `print()` 用来输出字符串、`len()` 用来计算序列长度等。

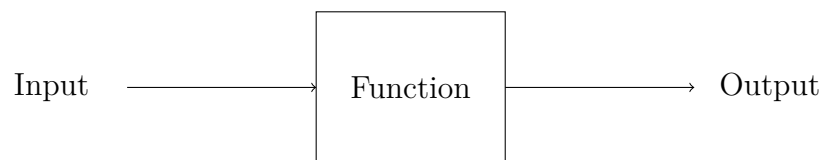


图 5.1: 函数

当调用函数时，程序控制权会转移给被调用的函数，当函数执行结束后，函数会把程序控制权交还给其调用者。

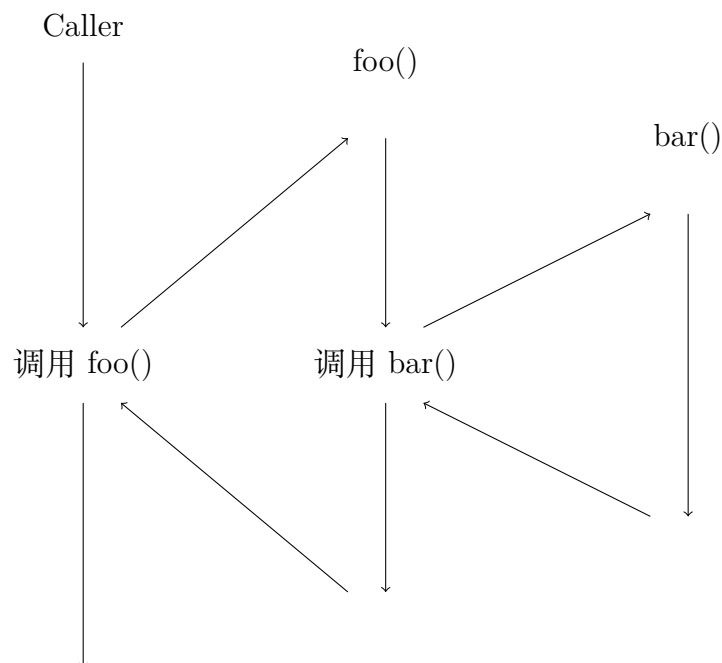


图 5.2: 函数调用

使用 `def` 关键字可以定义函数：

```
1 def func_name([param_list]):  
2     # code
```

5.1.2 函数设计方法

为什么不把所有的代码全部写在一起，还需要自定义函数呢？

使用函数有以下好处：

1. 避免代码复制，代码复制是程序质量不良的表现
2. 便于代码维护
3. 避免重复造轮子，提高开发效率

在设计函数的时候需要考虑以下的几点要素：

1. 确定函数的功能
2. 确定函数的参数
 - 是否需要参数
 - 参数个数
 - 参数类型
3. 确定函数的返回值
 - 是否需要返回值
 - 返回值类型

函数实现返回最大值

```
1 def get_max(num1, num2):  
2     # if num1 > num2:  
3     #     return num1  
4     # else:
```

```

5     #     return num2
6
7     return num1 if num1 > num2 else num2
8
9 print(get_max(4, 12))
10 print(get_max(54, 33))
11 print(get_max(0, -12))
12 print(get_max(-999, -774))

```

运行结果

```

12
54
0
-774

```

函数实现累加和

```

1 def get_sum(start, end):
2     total = 0
3     for i in range(start, end+1):
4         total += i
5     return total
6
7 print("1-100的累加和 = %d" % get_sum(1, 100))
8 print("1024-2048的累加和 = %d" % get_sum(1024, 2048))

```

运行结果

```

1-100的累加和 = 5050
1024-2048的累加和 = 1574400

```

函数实现输出 i 行 j 列由自定义字符组成的图案


```
1 def print_chars(row, col, c):
2     for i in range(row):
3         for j in range(col):
4             print(c, end='')
5         print()
6
7 print_chars(5, 10, '?')
```

运行结果

```
??????????
??????????
??????????
??????????
??????????
```

5.2 主函数

5.2.1 主函数

Python 是为数不多的直接定义完源代码就可以执行的编程语言，很多编程语言对于程序的执行都有非常严格的标准，例如 C、C++、Java 等都有主函数（主方法）来标记程序的起点。

在现实的开发之中主函数是很有必要的，可以区分出其它的结构，在模块中更需要主函数的使用。

在 Python 中如果实现主函数的定义，必须借助于全局变量 `__name__` 的返回内容，而后采用自定义的函数形式实现，返回的 `__main__` 是一个字符串，这个内容是会改变的，跟程序身处的结构有关。

在很多的编程语言都将主函数通过 `main` 这个标识符来定义，所以定义的 `main()` 是符合一般的习惯的。在进行复杂开发的时候，强烈建议使用主函数作为程序的起点。

主函数

```
1 def main():
2     pass
3
4 if __name__ == "__main__":
5     main()
```

5.3 变量作用域

5.3.1 变量作用域

一个变量会根据其自身所处的位置由不同的作用范围，不同范围内使用的变量采用就近取用的原则。

Python 的变量作用域采用 LEGB 原则：

- Local：函数内部变量名称。
- Enclosing Function Locals：外部嵌套函数变量名称。
- Global：函数所在模块或程序文件的变量名称。
- Builtin：内置模块的变量名称。

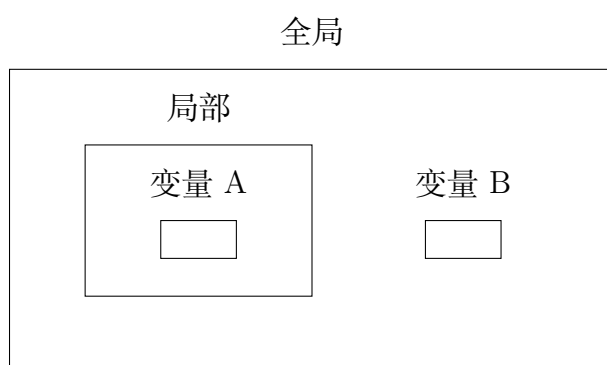


图 5.3: 变量作用域

在一个 Python 源文件之中可以存在有若干个函数（或者类），每一个函数里面有可能定义自己的变量，但是这个变量不是公共的。局部变量只能被一个函数所使用的，其它的函数都无法进行操作。

局部变量

```
1 def foo():  
2     num = 123  
3
```

```
4 def main():
5     print(num)
6
7 if __name__ == "__main__":
8     main()
```

运行结果

NameError: name 'num' is not defined

如果希望在函数中访问全局的变量，可以通过 global 关键字实现。

全局变量

```
1 num = 123      # 全局变量
2
3 def foo():
4     num = 321   # 局部变量
5     print("foo(): %d" % num)    # 321
6
7 def bar():
8     global num  # 调用全局变量
9     num = 456
10    print("bar(): %d" % num)    # 456
11
12 def main():
13     foo()
14     print(num)    # 123
15     bar()
16     print(num)    # 456
17
18 if __name__ == "__main__":
19     main()
```

运行结果

foo(): 321

123

bar(): 456

456

5.4 函数参数

5.4.1 参数默认值

在进行函数参数定义的时候，也可以设置默认值。当参数没有传递的时候就利用默认值来进行参数内容的填充，如果在参数上定义了默认值，那么该参数一定要放在参数列表的最后。

参数默认值

```
1 def set_date(year=1970, month=1, day=1):
2     print("%04d-%02d-%02d" % (year, month, day))
3
4 def main():
5     set_date(2021, 4, 1)
6     set_date(2021, 3)
7     set_date(2021)
8     set_date()
9
10 if __name__ == "__main__":
11     main()
```

运行结果

```
2021-04-01
2021-03-01
2021-01-01
1970-01-01
```

5.4.2 可变参数

在 Python 中提供有可变参数形式，所有可变参数都使用元组进行接收。

可变参数

```
1 def calculate(operator, *nums):
2     """
3     定义一个可变参数的数学计算
4     通过传入的运算符和运算数进行计算
5     Args:
6         operator (str): 运算符
7         nums (tuple): 运算数 (可变参数)
8     Return:
9         计算结果
10    """
11    if operator == '+':
12        result = 0
13        for num in nums:
14            result += num
15    elif operator == '*':
16        result = 1
17        for num in nums:
18            result *= num
19    return result
20
21 def main():
22     print("累加: %d" % calculate('+', 1, 2, 3, 4, 5))
23     print("阶乘: %d" % calculate('*', 1, 2, 3, 4, 5))
24
25 if __name__ == "__main__":
26     main()
```

运行结果

累加: 15
阶乘: 120

在进行参数传递的时候也可以使用【**】来标记关键字参数（字典）。

关键字参数

```
1 def print_scores(name, **scores):
2     print(name)
3     for key, value in scores.items():
4         print("\t|- %s: %s" % (key, value))
5
6 def main():
7     print_scores("小灰", Python=100, Java=95)
8     print_scores("小白", 数据结构=78, 算法=82)
9
10 if __name__ == "__main__":
11     main()
```

运行结果

小灰

|- Python: 100

|- Java: 95

小白

|- 数据结构: 78

|- 算法: 82

5.5 lambda 表达式

5.5.1 lambda 表达式

lambda 指的是函数式编程，函数式编程最简单的理解就是没有名字的函数。
lambda 定义简单，只使用一次。

```
1 lambda param1, param2, ...: statement
```

一般而言，lambda 函数都比较短，但是一般的普通函数内容都会比较多。

lambda 函数

```
1 def main():  
2     add = lambda x, y: x + y  
3     print(add(10, 20))  
4  
5 if __name__ == "__main__":  
6     main()
```

运行结果

30

5.6 递归

5.6.1 递归 (Recursion)

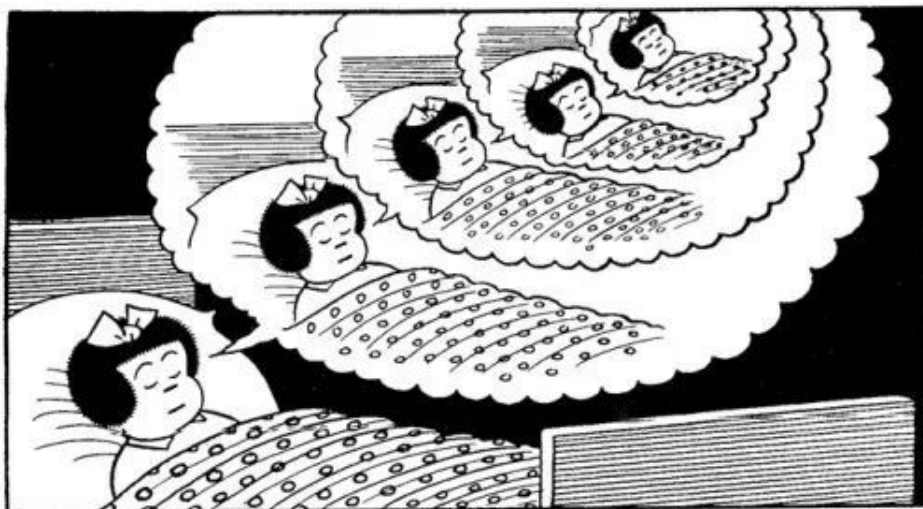
要理解递归，先得理解递归（见5.6章节）。

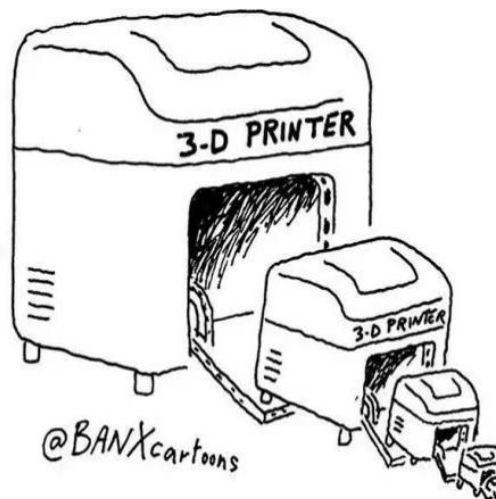
在函数的内部，直接或者间接的调用自己的过程就叫作递归。对于一些问题，使用递归可以简洁易懂的解决问题，但是递归的缺点是性能低，占用大量系统栈空间。

递归算法很多时候可以处理一些特别复杂、难以直接解决的问题。例如：

- 迷宫
- 汉诺塔
- 八皇后
- 排序
- 搜索

在定义递归函数时，一定要确定一个结束条件，否则会造成无限递归的情况，最终会导致栈溢出。







无限递归

```
1 def tell_story():
2     print("从前有座山")
3     print("山里有座庙")
4     print("庙里有个老和尚和小和尚")
5     print("老和尚在对小和尚讲故事")
6     print("他讲的故事是：")
7     tell_story()
8
9 def main():
10     tell_story()
11
12 if __name__ == "__main__":
13     main()
```

运行结果

从前有座山
山里有座庙
庙里有个老和尚和小和尚
老和尚对小和尚在讲故事
他讲的故事是：
从前有座山
山里有座庙
庙里有个老和尚和小和尚
老和尚对小和尚在讲故事
他讲的故事是：
...

递归函数一般需要定义递归的出口，即结束条件，确保递归能够在适合的地方退出。

阶乘

```
1 def factorial(n):
2     if n == 0 or n == 1:
3         return 1
4     return n * factorial(n-1)
5
6 def main():
7     print("5! = %d" % factorial(5))
8
9 if __name__ == "__main__":
10     main()
```

运行结果

5! = 120

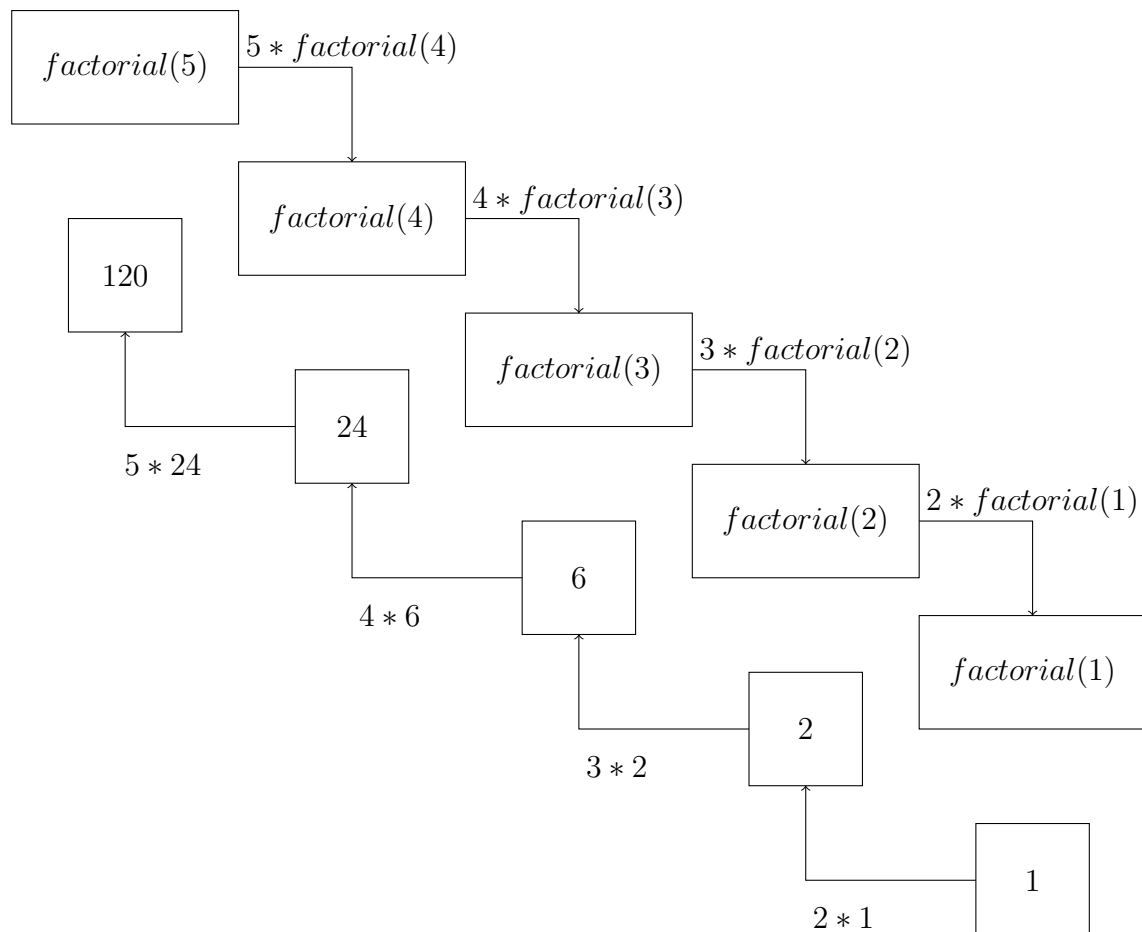


图 5.4: 阶乘

斐波那契数列（递归）

```

1 def fibonacci(n):
2     if n == 1 or n == 2:
3         return 1
4     return fibonacci(n-2) + fibonacci(n-1)
5
6 def main():
7     n = 7
8     print("斐波那契数列第%d位: %d" % (n, fibonacci(n)))
9
10 if __name__ == "__main__":
11     main()

```

运行结果

斐波那契数列第7位：13

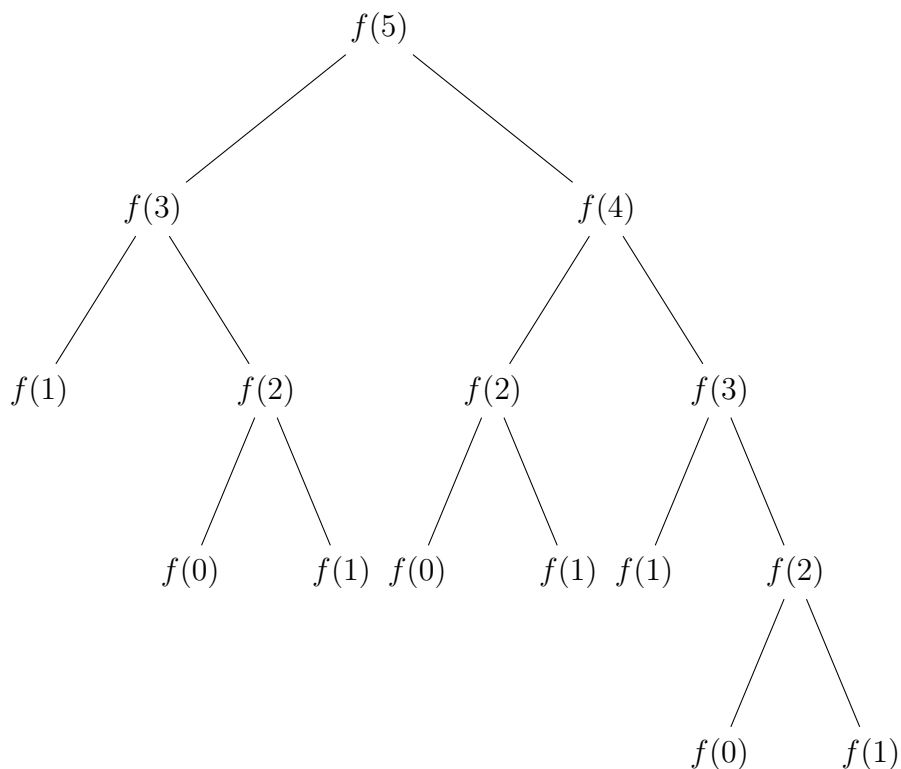


图 5.5: 递归树

斐波那契数列（迭代）

```
1 def fibonacci(n):
2     f = [0] * n
3     f[0] = f[1] = 1
4     for i in range(2, n):
5         f[i] = f[i-2] + f[i-1]
6     return f[n-1]
7
8 def main():
9     n = 7
10    print("斐波那契数列第%d位: %d" % (n, fibonacci(n)))
11
```

```

12 if __name__ == "__main__":
13     main()

```

运行结果

斐波那契数列第7位：13

阿克曼函数

$$A(m, n) = \begin{cases} n + 1 & m = 0 \\ A(m - 1, 1) & m > 0, n = 0 \\ A(m - 1, A(m, n - 1)) & m > 0, n > 0 \end{cases}$$

```

1 def A(m, n):
2     if m == 0:
3         return n + 1
4     elif m > 0 and n == 0:
5         return A(m-1, 1)
6     else:
7         return A(m-1, A(m, n-1))
8
9 def main():
10     print(A(3, 4))
11
12 if __name__ == "__main__":
13     main()

```

运行结果

125

$m \backslash n$	0	1	2	3	4	n
0	1	2	3	4	5	$n + 1$
1	2	3	4	5	6	$2 + (n + 3) - 3$
2	3	5	7	9	11	$2(n + 3) - 3$
3	5	13	29	61	125	$2^{n+3} - 3$
4	13	65533	$2^{65536} - 3$	$A(3, 2^{65536} - 3)$	$A(3, A(4, 3))$	$\underbrace{2^{2^{\cdot^{\cdot^{\cdot^2}}}}_{n+3 \text{ twos}} - 3$
5	65533	$A(4, 65533)$	$A(4, A(5, 1))$	$A(4, A(5, 2))$	$A(4, A(5, 3))$...
6	$A(5, 1)$	$A(5, A(5, 1))$	$A(5, A(6, 1))$	$A(5, A(6, 2))$	$A(5, A(6, 3))$...

表 5.1: 阿克曼函数



汉诺塔

给定三根柱子，其中 A 柱子从大到小套有 n 个圆盘，问题是如何借助 B 柱子，将圆盘从 A 搬到 C。

规则：

- 一次只能搬动一个圆盘
- 不能将大圆盘放在小圆盘上面



递归算法求解汉诺塔问题：

1. 将前 $n-1$ 个圆盘从 A 柱借助于 C 柱搬到 B 柱。
2. 将最后一个圆盘直接从 A 柱搬到 C 柱。
3. 将 $n-1$ 个圆盘从 B 柱借助于 A 柱搬到 C 柱。

```
1 move = 0          # 移动次数
2
3 def hanoi(n, src, mid, dst):
4     global move
5     """
6     汉诺塔算法
7     把 n 个盘子从 src 借助 mid 移到 dst
8     Args:
9         n (int): 层数
10        src (str): 起点柱子
11        mid (str): 临时柱子
12        dst (str): 目标柱子
13    """
14    if n == 1:
15        print("%d号盘: %c -> %c" % (n, src, dst))
16        move += 1
17    else:
18        # 把前 n-1 个盘子从 src 借助 dst 移到 mid
```

```

19     hanoi(n-1, src, dst, mid)
20     # 移动第 n 个盘子
21     print("%d号盘: %c -> %c" % (n, src, dst))
22     move += 1
23     # 把刚才的 n-1 个盘子从 mid 借助 src 移到 dst
24     hanoi(n-1, mid, src, dst)
25
26 def main():
27     hanoi(4, 'A', 'B', 'C')
28     print("步数 ==> %d" % move)
29
30 if __name__ == "__main__":
31     main()

```

运行结果

```

1号盘: A -> B
2号盘: A -> C
1号盘: B -> C
3号盘: A -> B
1号盘: C -> A
2号盘: C -> B
1号盘: A -> B
4号盘: A -> C
1号盘: B -> C
2号盘: B -> A
1号盘: C -> A
3号盘: B -> C
1号盘: A -> B
2号盘: A -> C
1号盘: B -> C
步数 ==> 15

```

Chapter 6 模块

6.1 模块导入

6.1.1 模块 (Module)

模块是进行大型项目拆分组织的一种有效技术手段，它可以将一个庞大的代码分割成若干个小的组成单元，方便进行代码的开发与维护。利用模块的划分，在进行代码维护的时候，可以保证局部的更新不影响其它的程序的运行操作。

使用 `import` 关键字可以进行模块的导入，`import` 可以同时导入多个模块，但是从开发的角度来讲，强烈建议分开导入。

```
1 import package.module [as alias] [, ...]
```

模块导入

algorithm/search.py

```
1 def sequence_search(list, key):
2     """
3     顺序查找
4     Args:
5         list (list): 待查找数组
6         key (int): 关键字
7     """
8     for i in range(len(list)):
9         if list[i] == key:
10             return i
11     return -1
12
13 def binary_search(list, key):
14     """
15     二分查找
16     Args:
```

```

17         list (list): 待查找数组
18         key (int): 关键字
19     """
20     start = 0
21     end = len(list) - 1
22     while start <= end:
23         mid = (start + end) // 2
24         if list[mid] == key:
25             return mid
26         elif list[mid] < key:
27             start = mid + 1
28         else:
29             end = mid - 1
30     return -1

```

import_as.py

```

1 import algorithm.search as search
2
3 def main():
4     list = [40, 9, 20, 93, 7, 34, 85, 91]
5     key = 34
6     print("%d所在位置: %d" % (key, search.sequence_search(list, key)))
7
8 if __name__ == "__main__":
9     main()

```

运行结果

34所在位置: 5

在 Python 里有一个作者写的 Python 开发禅道（开发的 19 条哲学），如果想看到这个彩蛋的信息，可以在 Python 的交互模式输入 `import this`。

运行结果

The Zen of Python, by Tim Peters

```
Beautiful is better than ugly.  
Explicit is better than implicit.  
Simple is better than complex.  
Complex is better than complicated.  
Flat is better than nested.  
Sparse is better than dense.  
Readability counts.  
Special cases aren't special enough to break the rules.  
Although practicality beats purity.  
Errors should never pass silently.  
Unless explicitly silenced.  
In the face of ambiguity, refuse the temptation to guess.  
There should be one-- and preferably only one --obvious way to  
do it.  
Although that way may not be obvious at first unless you're  
Dutch.  
Now is better than never.  
Although never is often better than *right* now.  
If the implementation is hard to explain, it's a bad idea.  
If the implementation is easy to explain, it may be a good idea.  
Namespaces are one honking great idea -- let's do more of those!
```

6.1.2 from-import 模块导入

使用 import 导入模块之后需要采用 module_name.func() 形式进行调用，每一次调用函数非常麻烦。from-import 导入语法可以简化调用语句。

```
1 from package.module import name [as alias] [, ...]
```

在实践中, from-import 不是良好的编程风格, 因为如果导入的变量与作用域中现有变量同名, 那么变量就会被悄悄覆盖掉。使用 import 语句的时候就不会发生这种问题, 通过 module.var 或 module.func() 获取的变量或方法不会与现有作用域冲突。

from-import 导入

algorithm/search.py

```
1 def sequence_search(list, key):
2     """
3     顺序查找
4     Args:
5         list (list): 待查找数组
6         key (int): 关键字
7     """
8     for i in range(len(list)):
9         if list[i] == key:
10            return i
11    return -1
12
13 def binary_search(list, key):
14     """
15     二分查找
16     Args:
17         list (list): 待查找数组
18         key (int): 关键字
19     """
20    start = 0
21    end = len(list) - 1
22    while start <= end:
23        mid = (start + end) // 2
24        if list[mid] == key:
25            return mid
26        elif list[mid] < key:
27            start = mid + 1
```

```
28         else:
29             end = mid - 1
30     return -1
```

from_import.py

```
1 from algorithm.search import binary_search
2
3 def main():
4     list = [7, 9, 20, 34, 40, 85, 91, 93]
5     key = 34
6     print("%d所在位置: %d" % (key, binary_search(list, key)))
7
8 if __name__ == "__main__":
9     main()
```

运行结果

34所在位置: 3

6.2 math 模块

6.2.1 math 模块

现代计算机的基础学科就是数学，如果没有数学理论作为基础，计算机是无法得到正常发展的。数学模块只提供了数学的基本计算功能，在很多的开发之中，有可能会需要使用到更加复杂的数学逻辑的时候就需要采用一些第三方模块进行数学计算了。

math 模块

```
1 import math
2
3 def main():
4     print("累加: %d" % math.fsum(range(101)))
5     print("阶乘: %d" % math.factorial(10))
6     print("乘方: %d" % math.pow(2, 10))
7     print("对数: %f" % math.log(10))
8     print("余数: %d" % math.fmod(22, 5))
9
10 if __name__ == "__main__":
11     main()
```

运行结果

```
累加: 5050
阶乘: 3628800
乘方: 1024
对数: 2.302585
余数: 2
```

6.3 random 模块

6.3.1 random 模块

随机数可以在一个指定的范围之内随机地生成一些数字供使用。例如，手机验证码发送来的数字就是使用随机数的方式生成的。

方法	功能
random()	生成一个 0 到 1 的随机浮点数: $0.0 \leq n \leq 1.0$
uniform(x, y)	生成一个在指定范围内的随机浮点数
randint(x, y)	生成一个指定范围内的随机整数 $x \leq n \leq y$
choice(sequence)	从序列中随机抽取数据
shuffle(x [, random])	将一个列表中的元素打乱
sample(sequence, k)	从指定序列中随机获取指定序列分片

表 6.1: random 模块

random 模块

```
1 import random
2
3 def main():
4     lst = [random.randint(1, 100) for _ in range(10)]
5     print("初始序列: %s" % lst)
6     print("随机抽取: ", end='')
7     for _ in range(5):
8         print(random.choice(lst), end=' ')
9
10 if __name__ == "__main__":
11     main()
```

运行结果

初始序列: [85, 83, 83, 29, 2, 93, 30, 65, 41, 54]

随机抽取: 41 93 2 54 93

6.4 time 模块

6.4.1 time 模块

time 模块是 Python 内置的一个实现时间的操作模块，用于描述日期时间的数据类型分为三种：

时间戳 (timestamp)

从 1970 年 1 月 1 日 00 时 00 分 00 秒开始的按秒计算的时间偏移量。

计算操作耗时

```
1 import time
2
3 def main():
4     start = time.time()
5     print("【开始】%s" % start)
6     sum = 0
7     for i in range(99999999):
8         sum += i
9     end = time.time()
10    print("【结束】%s" % end)
11    print("【耗时】%.2fs" % (end - start))
12
13 if __name__ == "__main__":
14     main()
```

运行结果

【开始】 1617327695.1819823

【结束】 1617327701.1566718

【耗时】 5.97s

时间元组

保存日期时间数据的元素结构对象。

属性	功能	数值
tm_year	年	yyyy
tm_mon	月	1 ~ 12
tm_mday	日	1 ~ 32
tm_hour	时	0 ~ 23
tm_min	分	0 ~ 59
tm_sec	秒	0 ~ 61 (60 或 61 是闰秒)
tm_wday	一周第几天	0 ~ 6 (0 表示周一)
tm_yday	一年第几天	1 ~ 366

表 6.2: 时间元组

时间戳与时间元组的转换

```
1 import time
2
3 def main():
4     current_time = time.time()
5     current_time_tuple = time.localtime(current_time)
6     print("时间戳转换时间元组: " + str(current_time_tuple))
7
8 if __name__ == "__main__":
9     main()
```

运行结果

时间戳转换时间元组: `time.struct_time(tm_year=2021, tm_mon=4, tm_mday=2, tm_hour=9, tm_min=45, tm_sec=16, tm_wday=4, tm_yday=92, tm_isdst=0)`

格式化日期时间

可以按照指定的标记进行格式化处理。时间戳与时间元组更多情况下还是描述程序层次上的概念，格式化日期时间可以给出人们都认可的显示格式。

格式化日期时间

```
1 import time
2
3 def main():
4     current_time = time.time()
5     current_time_tuple = time.localtime(current_time)
6
7     print(time.strftime("%Y-%m-%d %H:%M:%S", current_time_tuple))
8     print("date: %s" % time.strftime("%F", current_time_tuple))
9     print("time: %s" % time.strftime("%T", current_time_tuple))
10
11 if __name__ == "__main__":
12     main()
```

运行结果

```
2021-04-02 09:49:01
date: 2021-04-02
time: 09:49:01
```

6.5 calendar 模块

6.5.1 calendar 模块

calendar 模块可以进行年历和日历的显示，同时也可以实现闰年的判断。

calendar 模块

```
1 import calendar
2
3 def main():
4     calendar.setfirstweekday(calendar.SUNDAY)
5     print(calendar.month(2021, 4))
6
7     print("2020是闰年吗? %s" % calendar.isleap(2020))
8     print("2000-3000年间闰年数量: %d" % calendar.leapdays(2000, 3000))
9
10    print(calendar.calendar(2021))
11
12 if __name__ == "__main__":
13     main()
```

6.6 copy 模块

6.6.1 copy 模块

copy 是一个专门进行内容复制的处理模块。拷贝分为浅拷贝（shallow copy）和深拷贝（deep copy）两种：

- 浅拷贝：只是复制第一层的内容，而更深入的数据嵌套关系不会拷贝。
- 深拷贝：会进行完整的复制。

6.6.2 引用

引用的本质在于将同一块内存空间，交给不同的对象进行同时操作，当一个对象修改了内存数据之后，其它对象的内存数据也会同时发生改变。

```
1 a = {1:[1, 2, 3]}
2 b = a
```

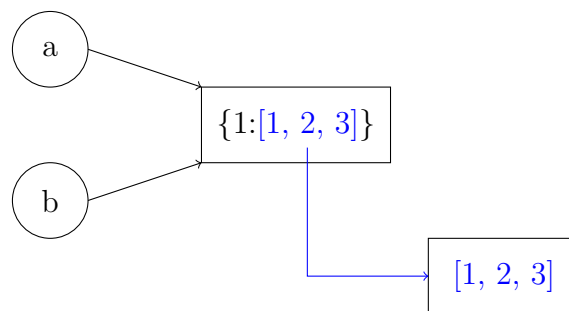


图 6.1: 引用

引用传递

```
1 def main():
2     info = dict(name="小灰", age=16, skills=["Python", "C/C++"])
3     copy_info = info          # 引用传递
4     copy_info["skills"].append("Java")
```

```

5     print(info)
6
7 if __name__ == "__main__":
8     main()

```

运行结果

```
{'name': '小灰', 'age': 16, 'skills': ['Python', 'C/C++', 'Java']}
```

6.6.3 浅拷贝

拷贝和引用传递是不同的，拷贝是将原始的内存的数据进行一份复制，而后为其分配单独的对象的指向。

```

1 a = {1:[1, 2, 3]}
2 b = a.copy()

```

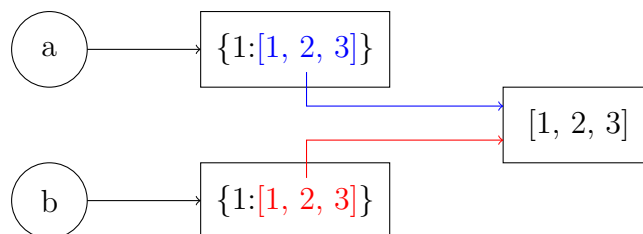


图 6.2: 浅拷贝

浅拷贝

```

1 import copy
2
3 def main():
4     info = dict(name="小灰", age=16, skills=["Python", "C/C++"])
5     copy_info = copy.copy(info)    # 浅拷贝
6     copy_info.pop("age")
7     copy_info["skills"].append("Java")

```



```

8     print(info)
9     print(copy_info)
10
11 if __name__ == "__main__":
12     main()

```

运行结果

```

{'name': '小灰', 'age': 16, 'skills': ['Python', 'C/C++', 'Java']}
{'name': '小灰', 'skills': ['Python', 'C/C++', 'Java']}

```

6.6.4 深拷贝

```

1 a = {1:[1, 2, 3]}
2 b = copy.deepcopy(a)

```

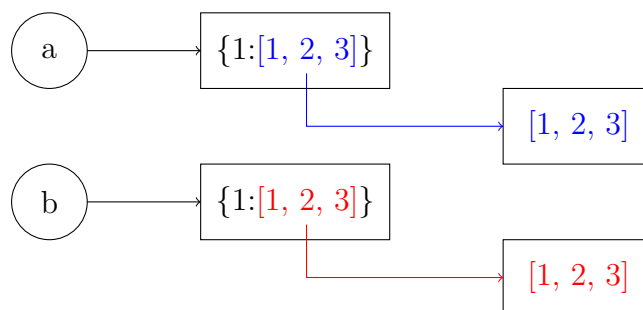


图 6.3: 深拷贝

深拷贝

```

1 import copy
2
3 def main():
4     info = dict(name="小灰", age=16, skills=["Python", "C/C++"])
5     copy_info = copy.deepcopy(info)    # 深拷贝
6     copy_info.pop("age")

```

```
7     copy_info["skills"].append("Java")
8     print(info)
9     print(copy_info)
10
11 if __name__ == "__main__":
12     main()
```

运行结果

```
{'name': '小灰', 'age': 16, 'skills': ['Python', 'C/C++']}
{'name': '小灰', 'skills': ['Python', 'C/C++', 'Java']}
```

6.7 MapReduce 数据处理

6.7.1 MapReduce

Python 在数据分析领域上使用非常广泛，并且实现简单。在 Python 中可以进行大量数据的快速处理，进行数据的过滤、分析操作。在数据量小的情况下可以方便地使用 for 循环进行数据的逐个处理，但是在数据量大的情况下，就需要使用一些特定的处理函数进行过滤、分析、统计等操作。

在 Python 中默认提供有了 filter()、map()，但是要进行统计处理，则需要导入 reduce()。

函数	功能
filter(function, sequence)	对传入的序列数据进行过滤
map(function, sequence)	对传入的序列数据进行处理
reduce(function, sequence)	对传入的序列数据进行统计

表 6.3: MapReduce 数据处理函数

在进行数据处理的过程之中都需要有一个处理函数，这个处理函数就定义了数据该如何进行处理或统计，一般而言这样的函数都比较短，所以大部分情况下都可以利用 lambda 函数来完成。

MapReduce 数据处理

```
1 from functools import reduce
2
3 def main():
4     lst = list(range(10))
5
6     filter_lst = list(filter(lambda x: x % 2 == 0, lst))
7     print("过滤出偶数: %s" % filter_lst)
8
9     map_lst = list(map(lambda x: x ** 2, filter_lst))
10    print("平方: %s" % map_lst)
```

```
11
12     result = reduce(lambda x, y: x+y, map_lst)
13     print("求和: %d" % result)
14
15 if __name__ == "__main__":
16     main()
```

运行结果

过滤出偶数: [0, 2, 4, 6, 8]

平方: [0, 4, 16, 36, 64]

求和: 120

6.8 pip 模块管理工具

6.8.1 pip

Python 本地有一些系统模块开发者可以直接进行使用，但是开发者仅仅是依靠系统模块是不够的，需要大量的使用第三方模块。为了解决这些模块的管理问题，在 Python 中内置了 pip 管理工具。通过此工具可以直接连接到 Python 远程服务模块仓库，通过仓库下载所需要的模块。

在 Python 安装的时候会自动进行 pip 工具的相关安装，输入 `pip -help` 查看 pip 的相关命令选项。

功能	命令
搜索模块	<code>pip search 模块名</code>
安装模块	<code>pip install 模块名</code>
查看已安装模块	<code>pip list</code>
列出过期模块	<code>pip list --outdated</code>
更新模块	<code>pip install --upgrade 模块名</code>
卸载模块	<code>pip uninstall 模块名</code>

表 6.4: pip 命令

6.9 jieba 分词

6.9.1 jieba

分词是一种数学的应用，它可以直接根据词语之间的数学关系进行文字或单词的抽象。例如对“中华人民共和国”进行分词处理，可以拆分为“中华”、“华人”、“人民”、“共和”、“共和国”、“中华人民共和国”。如果没有分词，就无法进行搜索引擎的开发。

jieba 是在中文自然语言处理中用得最多的工具包之一，它以分词起家，目前已经能够实现包括分词、词性标注以及命名实体识别等多种功能。

考虑到分词的效果与性能，在 jieba 组件中提供有 3 种分词模式：

1. 精确模式：将句子进行最精确的切分，分词速度相对较低。
2. 全模式：基于词汇列表将句子中所有可以成词的词语都扫描出来，该模式处理速度非常快，但是不能有效解决歧义的问题。
3. 搜索引擎模式：在精确模式的基础上，对长词进行再次切分，该模式适用于搜索引擎构建索引的分词。

统计《西游记》中出现次数最多的 20 个词语

```
1 import jieba
2
3 PATH = "西游记.txt"      # 文件路径
4
5 def main():
6     word_frequency = {}   # 词频表
7
8     # 打开文件
9     with open(file=PATH, mode="r", encoding="UTF-8") as file:
10         line = file.readline() # 读取一行数据
11         while line:
```

```

12         words = jieba.lcut(line)    # 分词
13         for word in words:
14             if len(word) == 1: # 舍弃长度为1的词
15                 continue
16             else:
17                 # dict.get(key, default=None)
18                 word_frequency[word]
19                 = word_frequency.get(word, 0) + 1
20         line = file.readline()
21
22     # 获取所有数据项
23     items = list(word_frequency.items())
24     # 根据出现次数降序排序
25     items.sort(key=lambda x: x[1], reverse=True)
26
27     # 取前20项
28     for i in range(20):
29         word, count = items[i]
30         print("%s: %s" % (word, count))
31
32 if __name__ == "__main__":
33     main()

```

Chapter 7 面向对象

7.1 面向过程与面向对象

7.1.1 面向过程 (Procedure Oriented)

面向过程是一种以过程为中心的编程思想，以什么正在发生为主要目标进行编程，分析出解决问题所需要的步骤，然后用函数把这些步骤一步一步实现，使用的时候一个一个依次调用。

C 语言就是一种面向过程的编程语言，但是面向过程的缺陷是数据和函数并不完全独立，使用两个不同的实体表示信息及其操作。

7.1.2 面向对象 (Object Oriented)

面向对象是相对于面向过程来讲的，面向对象方法把相关的数据和方法组织为一个整体来看待，从更高的层次来进行系统建模，更贴近事物的自然运行模式。

在面向对象中，把构成问题的事物分解成各个对象，建立对象的目的不是为了完成一个步骤，而是为了描述某个事物在整个解决问题的步骤中的行为。

Java、C++、Python 等都是面向对象的编程语言，面向对象的优势在于只是用一个实体就能同时表示信息及其操作。

面向对象三大特性：

1. 封装 (encapsulation)：数据和代码捆绑，避免外界干扰和不确定性访问。
2. 继承 (inheritance)：让某种类型对象获得另一类型对象的属性和方法。
3. 多态 (polymorphism)：同一事物表现出不同事物的能力。

7.2 类与对象

7.2.1 类与对象

类（class）表示同一类具有相同特征和行为的对象的集合，类定义了对应的属性和方法。

对象（object）是类的实例，对象拥有属性和方法。

类的设计需要使用关键字 `class`，类名是一个标识符，遵循大驼峰命名法。类中可以包含属性和方法。其中，属性通过变量表示，又称实例变量；方法用于描述行为，又称实例方法。

在程序之中如果需要使用类，那么一般都会通过对象来进行操作。

```
1 obj_name = class_name([param])
```

当实例化了一个对象之后，就可以通过此对象进行类中成员的访问：

- 对象. 属性：访问类中的属性内容，如果程序中访问了没有定义的实例属性，那么将引发 `AttributeError` 异常。
- 对象. 方法 ()：调用类中的方法。对于类中每一个方法的当前对象都会有 Python 自己来负责该对象的传入，这一操作不是由用户负责的。

类和对象

```
1 class Person:
2     name = ""
3     age = 0
4     gender = ""
5
6     def eat(self):
7         print("吃饭")
8
```

```
9     def sleep(self):
10         print("睡觉")
11
12 def main():
13     person = Person()
14
15     person.name = "小灰"
16     person.age = 16
17     person.gender = "男"
18
19     print("姓名: %s, 年龄: %d, 性别: %s" % (
20         person.name, person.age, person.gender))
21     person.eat()
22     person.sleep()
23
24 if __name__ == "__main__":
25     main()
```

运行结果

姓名: 小灰, 年龄: 16, 性别: 男
吃饭
睡觉

7.2.2 垃圾回收机制

引用传递的本质在于将同一块空间修改权力交由不同的对象来完成, 在这样的处理之中就有可能产生垃圾空间。在 Python 的引用数据处理之中都会存在有一个引用计数器, 当引用计数器为 0 的时候就表示该对象已经成为了垃圾, 等待进行回收。

垃圾回收机制

```
1 class Person:
2     pass
3
4 def main():
5     person1 = Person()
6     person2 = Person()
7
8     print("【引用传递前地址】 person1: %d, person2: %d" % (
9         id(person1), id(person2)))
10    person2 = person1
11    print("【引用传递后地址】 person1: %d, person2: %d" % (
12        id(person1), id(person2)))
13
14 if __name__ == "__main__":
15     main()
```

运行结果

```
【引用传递前地址】 person1: 1988221852552, person2: 1988222686536
【引用传递后地址】 person1: 1988221852552, person2: 1988221852552
```

在开发之中，实际上对于垃圾空间应该尽可能少的产生，虽然 Python 提供有垃圾收集机制，但是垃圾的回收与释放依然需要占用系统资源。

7.3 封装

7.3.1 封装 (Encapsulation)

封装是面向对象方法的重要原则，就是把对象的属性和方法结合为一个独立的整体，并尽可能隐藏对象的内部实现细节。

封装可以认为是一个保护屏障，防止该类的数据被外部类随意访问。要访问该类的数据，必须通过严格的接口控制。合适的封装可以让代码更容易理解和维护，也加强了程序的安全性。

实现封装的步骤：

1. 修改属性的可见性来限制对属性的访问，一般限制为 `private`。
2. 对每个属性提供对外的公共方法访问，也就是提供一对 `setter` / `getter`，用于对私有属性的访问。

封装

```
1 class Person:
2     def set_name(self, name):
3         self.__name = name
4
5     def get_name(self):
6         return self.__name
7
8     def set_age(self, age):
9         self.__age = age
10
11    def get_age(self):
12        return self.__age
13
14 def main():
15     person = Person()
```

```
16     person.set_name("小灰")
17     person.set_age(17)
18     print("姓名: %s, 年龄: %d" % (
19         person.get_name(), person.get_age()))
20
21 if __name__ == "__main__":
22     main()
```

运行结果

姓名: 小灰, 年龄: 17

7.4 构造方法与析构方法

7.4.1 构造方法 (Constructor)

构造方法也是一个方法，用于实例化对象，在实例化对象的时候调用。一般情况下，使用构造方法是为了在实例化对象的同时，给一些属性进行初始化赋值。

构造方法和普通方法的区别：

1. 构造方法的名称必须为 `__init__()`。
2. 构造方法没有返回值。
3. 一个类中只允许定义最多 1 个构造方法。

如果一个类中没有写构造方法，系统会自动提供一个无参构造方法，以便实例化对象。

构造方法

```
1 class Person:
2     def __init__(self, name, age):
3         self.__name = name
4         self.__age = age
5
6     def get_info(self):
7         return "姓名: %s, 年龄: %d" % (self.__name, self.__age)
8
9 def main():
10     person = Person("小灰", 17)
11     print(person.get_info())
12
13 if __name__ == "__main__":
14     main()
```

运行结果

姓名：小灰，年龄：17

7.4.2 析构方法 (Destructor)

在对象实例化的时候会触发构造方法的执行，与之对应的操作称为析构，当对象不再使用的时候进行某些收尾处理的操作。析构方法名称也要要求，必须为 `__del__()`。

析构方法

```
1 class Person:
2     def __init__(self):
3         print("构造方法被执行了")
4
5     def __del__(self):
6         print("析构方法被执行了")
7
8 def main():
9     person = Person()
10    del person
11
12 if __name__ == "__main__":
13     main()
```

运行结果

构造方法被执行了
析构方法被执行了

7.4.3 匿名对象

对象的名称只是一个地址的信息，真正的内容是保存在内存中的，如果不需要名称就可以使用匿名对象。匿名对象使用完成之后由于没有其它对象进行引用，那么就有可能被垃圾回收，同时也会调用析构方法。

如果一个对象要被反复使用，那么可以定义有名对象；如果一个对象只是用一次，就采用匿名对象完成操作即可。

匿名对象

```
1 class Person:
2     def __init__(self, name, age):
3         self.__name = name
4         self.__age = age
5
6     def get_info(self):
7         return "姓名: %s, 年龄: %d" % (self.__name, self.__age)
8
9 def main():
10     print(Person("小灰", 17).get_info())
11
12 if __name__ == "__main__":
13     main()
```

运行结果

姓名: 小灰, 年龄: 17

7.5 继承

7.5.1 继承 (Inheritance)

继承是面向对象的三大特征之一，程序中的继承是类与类之间的特征和行为的一种赠予或获取。两个类之间的继承必须满足“is a”的关系。子类继承自父类，父类也称基类或超类，子类也称派生类。

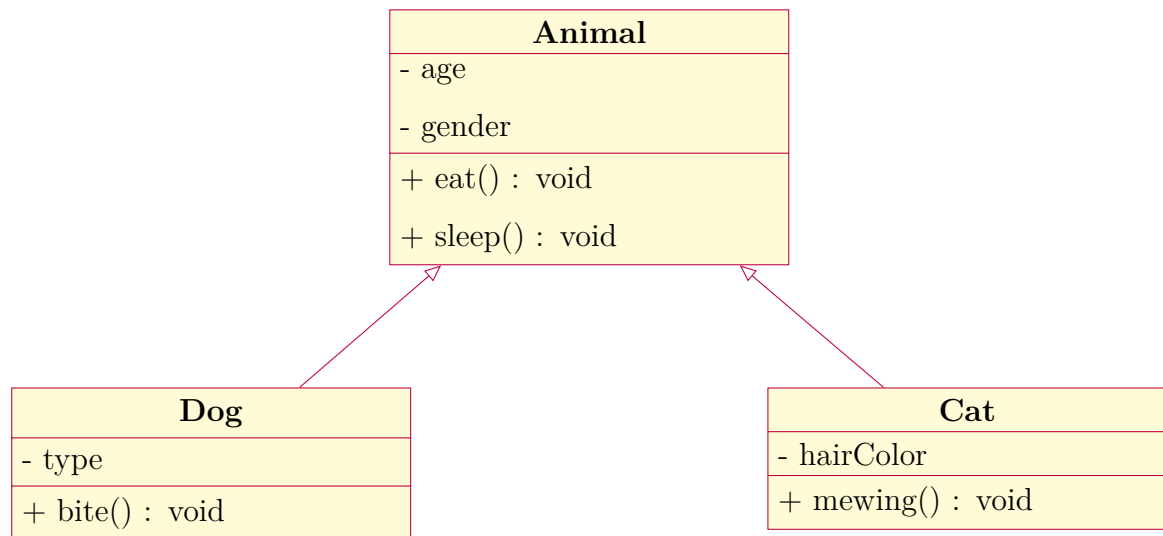


图 7.1: 继承

产生继承关系后，子类可以使用父类中的属性和方法，也可以定义子类独有的属性和方法。

```
1 class subclass(superclass1, ...):
2     # code
```

在进行继承的时候，子类会继承父类之中全部定义的结构。但是对于构造方法的继承是比较特殊的，需要考虑两种情况：

1. 当父类定义了构造方法，但是子类没有定义构造方法时，实例化子类对象会自动调用父类中提供的无参构造方法。
2. 当子类定义了构造方法时，将不再默认调用父类中的任何构造方法，但是可以手动调用。如果有需要也可以通过 `super` 类的实例实现子类调用父类结构的需求。

继承

```
1 class Animal:
2     def __init__(self, name, age):
3         self.__name = name
4         self.__age = age
5
6     def set_name(self, name):
7         self.__name = name
8
9     def get_name(self):
10        return self.__name
11
12    def set_age(self, age):
13        self.__age = age
14
15    def get_age(self):
16        return self.__age
17
18    def eat(self):
19        print("吃饭")
20
21    def sleep(self):
22        print("睡觉")
23
24 class Dog(Animal):
25     def __init__(self, name, age, type):
26         super().__init__(name, age)
27         self.__type = type
28
29     def set_type(self, type):
30         self.__type = type
31
32     def get_type(self):
33         return self.__type
34
35     def bite(self):
```

```

36         print("咬人")
37
38 def main():
39     dog = Dog("狗子", 3, "哈士奇")
40
41     print("姓名: %s, 年龄: %d, 品种: %s" % (
42         dog.get_name(), dog.get_age(), dog.get_type()))
43
44     dog.eat()
45     dog.sleep()
46     dog.bite()
47
48 if __name__ == "__main__":
49     main()

```

运行结果

姓名：狗子，年龄：3，品种：哈士奇
吃饭
睡觉
咬人

7.5.2 多继承

多继承指一个子类可以同时继承多个父类的内容，在多继承实现中只需要编写多个父类的名称即可。利用多继承的最大优势在于可以在进行子类操作的时候将多个父类中定义的结构全部保留继续使用。

多继承

```

1 class Date:
2     def set_date(self, year=1970, month=1, day=1):
3         self.__year = year
4         self.__month = month

```

```

5         self.__day = day
6
7     def get_date(self):
8         return "%04d/%02d/%02d" % (
9             self.__year, self.__month, self.__day)
10
11 class Time:
12     def set_time(self, hour=0, minute=0, second=0):
13         self.__hour = hour
14         self.__minute = minute
15         self.__second = second
16
17     def get_time(self):
18         return "%02d:%02d:%02d" % (
19             self.__hour, self.__minute, self.__second)
20
21 class DateTime(Date, Time):
22     def __init__(self, year=1970, month=1, day=1,
23                 hour=0, minute=0, second=0):
24         super().set_date(year, month, day)
25         super().set_time(hour, minute, second)
26
27     def __repr__(self):
28         return super().get_date() + " " + super().get_time()
29
30 def main():
31     date_time = DateTime(2021, 4, 6, 14, 38, 40)
32     print(date_time)
33
34 if __name__ == "__main__":
35     main()

```

运行结果

2021/04/06 14:38:40

7.6 多态

7.6.1 多态 (Polymorphism)

多态是同一个行为具有多个不同表现形式或形态的能力。例如可以把一只哈士奇，当成它的父类去看待，因此哈士奇是一只狗、一个动物或一个生物。

在类继承的结构之中，很难保证父类中的某些操作方法可以被子类继续拿来使用。这个时候子类为保留住原始的方法名称，同时也为了可以对功能实现进一步的扩充，就可以利用方法覆写。

多态

```
1 import math
2
3 class Shape:
4     def get_area(self):
5         pass
6
7 class Rectangle(Shape):
8     def __init__(self, width, length):
9         self.__width = width
10        self.__length = length
11
12    def get_area(self):
13        return self.__length * self.__width
14
15 class Circle(Shape):
16     def __init__(self, radius):
17         self.__radius = radius
18
19    def get_area(self):
20        return math.pi * self.__radius ** 2
21
22 def shape_area(obj):
```

```
23     if isinstance(obj, Shape):
24         return obj.get_area()
25
26 def main():
27     print("长方形面积: %.2f" % shape_area(Rectangle(6, 11)))
28     print("圆形面积: %.2f" % shape_area(Circle(5)))
29
30 if __name__ == "__main__":
31     main()
```

运行结果

长方形面积: 66.00

圆形面积: 78.54

Chapter 8 文件操作

8.1 文件操作

8.1.1 文件操作

计算机对于数据的存储一般可以通过文件的形式来完成。Python 中直接提供有文件的 I/O（Input/Output）处理函数操作，能够方便地实现读取和写入。

`open()` 的功能是进行文件的打开，在进行文件打开的时候如果不设置任何的模式类型，则默认为 `r`（只读模式）。

```
1 def open(file, mode='r', buffering=None, encoding=None,  
2         errors=None, newline=None, closefd=True  
3 )
```

打开模式	功能
r	使用只读模式打开文件，此为默认模式
w	写模式，如果文件存在则覆盖，文件不存在则创建
x	写模式，新建一个文件，如果该文件已存在则会报错
a	内容追加模式
b	二进制模式
t	文本模式（默认）
+	打开一个文件进行更新（可读可写）

表 8.1: 文件打开模式

如果以只读的模式打开文件，并且文件路径不存在的话，就会出现 `FileNotFoundError` 的错误信息。

文件操作

```

1 def main():
2     file = open(file="test.txt", mode="w")
3     print("文件名称: %s" % file.name)
4     print("访问模式: %s" % file.mode)
5     print("文件状态: %s" % file.closed)
6     print("关闭文件...")
7     file.close()
8     print("文件状态: %s" % file.closed)
9
10 if __name__ == "__main__":
11     main()

```

运行结果

```

文件名称: test.txt
访问模式: w
文件状态: False
关闭文件...
文件状态: True

```

8.1.2 文件读写

当使用 `open()` 打开一个文件后，就可以使用创建的文件对象进行读写操作。

方法	功能
<code>def close(self)</code>	关闭文件资源
<code>def flush(self)</code>	强制刷新缓冲区
<code>def read(self, n: int = -1)</code>	默认读取全部，也可设置读取个数
<code>def readlines(self, hint: int = -1)</code>	读取所有数据行，以列表形式返回
<code>def readline(self, limit: int = -1)</code>	读取每行数据，也可设置读取个数
<code>def write(self, s: AnyStr)</code>	文件写入
<code>def writelines(self, lines, List[AnyStr])</code>	写入一组数据

表 8.2: 文件读写

既然所有的文件对象最终都需要被开发者关闭，那么可以结合 with 语句实现自动的关闭处理。通过 with 实现所有资源对象的连接和释放是在 Python 中编写资源操作的重要技术手段，通过这样的操作可以极大地减少和优化代码结构。

使用读模式打开文件后，可以使用循环读取每一行的数据内容。Python 在进行文件读取操作的时候也可以进一步简化操作。文件对象本身是可以迭代的，在迭代的时候是以换行符进行分割，每次迭代就读取到一行数据内容。

读取文件

data.txt

```
1 小灰    16
2 小白    17
3 小黄    21
```

read_file.py

```
1 def main():
2     with open(file="data.txt", mode="r", encoding="utf-8") as file:
3         for line in file:
4             print(line, end='')
5
6 if __name__ == "__main__":
7     main()
```

运行结果

```
小灰 16
小白 17
小黄 21
```

写入文件

```
1 def main():
2     with open(file="data.txt", mode="w", encoding="utf-8") as file:
```

```
3     info = {"小灰": 16, "小白": 17, "小黄": 21}
4     for name, age in info.items():
5         file.write("%s\t%d\n" % (name, age))
6
7 if __name__ == "__main__":
8     main()
```

运行结果 data.txt

小灰 16

小白 17

小黄 21

8.2 文件缓冲

8.2.1 文件缓冲

在使用 `open()` 创建一个文件对象的时候，默认情况下是不会启用缓冲的。在 `open()` 中提供的 `buffering` 参数描述的就是文件处理缓冲的定义，在进行文件写入的时候利用缓冲可以避免频繁的 I/O 资源占用。

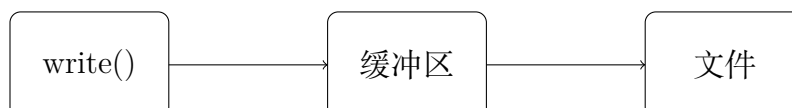


图 8.1: 文件缓冲

开启缓冲可以提高写入效率，`buffering` 参数设置有 3 种类型：

1. 全缓冲 (`buffering > 1`)：当标准 I/O 缓存被填满后才会进行真正 I/O 操作，全缓冲的典型代表就是对磁盘文件的读写操作。
2. 行缓冲 (`buffering = 1`)：在 I/O 操作中遇见换行符时才执行真正的 I/O 操作，例如在使用网络聊天工具时所编辑的文字在没有发送前是不会进行 I/O 操作的。
3. 不缓冲 (`buffering = 0`)：直接进行终端设备的 I/O 操作，数据不经过缓冲保存。

如果想要观察到行缓冲的使用特点，就不能直接使用 `with` 语句，因为 `with` 最后会执行 `close()` 的关闭操作，而一旦关闭，则缓冲的内容会全部进行输出。

使用 `flush()` 进行缓冲区的强制清空，一旦强制清空之后，缓冲区的内容将全部输出。每次使用 `close()` 关闭文件流的时候默认情况下也会调用 `flush()` 进行缓冲区的清空处理。

文件缓冲

```
1 import os
2
```

```
3 def main():
4     file = open(file="data.txt", mode="w",
5                 encoding="utf-8", buffering=1)
6     file.write("This is a test.")
7     os.system("pause") # 程序暂停
8     file.flush()       # 强制清空缓冲区
9     os.system("pause") # 程序暂停
10    file.close()
11
12 if __name__ == "__main__":
13     main()
```

运行结果 data.txt

This is a test.

8.3 os 模块

8.3.1 os 模块

os 模块是 Python 用于与操作系统进行交互的一个操作模块，这个模块提供有大量与系统相关的处理函数，开发者可以直接通过 Python 程序进行操作系统的功能调用。

方法	功能
getcwd()	获取当前的工作目录
chdir(path)	修改工作目录
system()	执行操作系统命令
symlink(src, dst)	创建软链接
link(src, dst)	创建硬链接

表 8.3: os 模块

8.3.2 os.path 子模块

os.path 是 os 模块之中的一个子模块，该子模块的核心作用在于进行路径处理操作。Python 的程序代码本身是强调跨平台的，既然要进行跨平台的开发，尤其是在 I/O 路径的处理上就特别要引起注意。在 Windows 系统下路径分隔符使用的是 **【\】**，而在 Linux 系统下使用的路径分隔符是 **【/】**。所以在进行程序编写的时候就必须考虑到不同平台的设计问题。

使用 os.path 模块中提供的一系列函数可以针对给定的路径进行拆分处理以及判断和取得数据信息。

Python 需要考虑不同操作系统的跨平台的特点，所以对于访问路径需要进行适当的变更，根据不同的操作系统使用不同的路径分隔符。如果每一次都判断操作系统就过于繁琐了，可以直接使用 os.path 中提供的变量来完成。

变量	功能
curdir	表示当前文件夹 【.】 ，一般可以省略
pardir	上一层文件夹 【..】
sep	系统路径分隔符，Windows 为 【\】 ，Linux 为 【/】
extsep	文件名称和后缀之间的间隔符 【.】

表 8.4: 路径分隔符

获取路径信息

```

1 import os
2
3 PATH = "code" + os.sep + "第10章 文件操作" + os.sep \
4       + "10.3 os模块" + os.sep + "data.txt"
5
6 def main():
7     if os.path.exists(PATH):
8         print("绝对路径: %s" % os.path.abspath(PATH))
9         print("文件名称: %s" % os.path.basename(PATH))
10        print("文件大小: %s" % os.path.getsize(PATH))
11        print("当前路径是否为文件: %s" % os.path.isfile(PATH))
12        print("当前路径是否为目录: %s" % os.path.isdir(PATH))
13
14 if __name__ == "__main__":
15     main()

```

运行结果

绝对路径: C:\Users\Administrator\Desktop\Python\code\第10章 文件操作\10.3 os模块\data.txt

文件名称: data.txt

文件大小: 15

当前路径是否为文件: True

当前路径是否为目录: False

8.4 csv 模块

8.4.1 csv 文件

CSV（Comma-Separated Values，逗号分隔值/字符分隔值）是一种文件的格式，在该类型的文件里面一般会保存多个数据信息的内容，但是每一个数据信息一定都有各自的组成部分，用这样的文件进行数据采集内容的记录。CSV 文件是跟人工智能和数据分析有直接联系的一种数据存储文件。

CSV 是一种以纯文件方式进行数据记录的存储格式，在 CSV 文件内容使用不同的数据行记录数据的内容，每行数据使用特定的符号（一般是逗号）进行数据项的拆分，这样就形成了一种相对简单且通用的数据格式。在实际开发中利用 CSV 数据格式可以方便实现大数据系统中对于数据采集结果的信息记录，也可以方便进行数据文件的传输，同时 CSV 文件格式也可以被 Excel 工具所读取。

CSV 文件是可以通过 Excel 工具打开的，当一个 CSV 文件被创建之后，在 Windows 系统中会自动和 Excel 软件进行关联。

8.4.2 csv 读写操作

在 Python 中直接提供有 csv 模块，利用这个模块可以方便地实现数据的写入和读取操作，在 CSV 文件内容一般对于不同的数据项都要使用逗号分隔。除了数据之外，在 CSV 文件内容还可以设置文件标题。

写入 csv 文件

```
1 import csv
2 import random
3
4 HEADER = ["Location", "Longitude", "Latitude"]
5
6 def main():
7     # 如果不使用newline，那么每行记录之间就会多出一个空行
```

```

8     with open(file="location.csv", mode="w",
9               newline="", encoding="utf-8") as file:
10         csv_writer = csv.writer(file)      # 创建csv写入对象
11         csv_writer.writerow(HEADER)        # 写入头部信息
12         for i in range(1, 11):
13             longitude = round(random.random() * 180, 3) # [0, 180)
14             latitude = round(random.random() * 90, 3)  # [0, 90)
15             csv_writer.writerow(["loc-%d" % i, longitude, latitude])
16
17 if __name__ == "__main__":
18     main()

```

运行结果 location.csv

```

Location,Longitude,Latitude
loc-1,176.165,35.458
loc-2,12.729,56.247
loc-3,6.605,45.14
loc-4,15.123,53.435
loc-5,131.984,11.927
loc-6,155.038,35.681
loc-7,98.772,15.125
loc-8,70.991,30.328
loc-9,152.967,30.372
loc-10,96.362,76.798

```

读取 csv 文件

```

1 import csv
2
3 def main():
4     with open(file="location.csv", mode="r",
5               newline="", encoding="utf-8") as file:
6         csv_reader = csv.reader(file) # 创建csv读取对象

```



```
7         header = next(csv_reader)          # 读取标题行
8         print(header)
9         for row in csv_reader:
10             print(row)
11
12 if __name__ == "__main__":
13     main()
```

运行结果

```
['Location', 'Longitude', 'Latitude']
['loc-1', '176.165', '35.458']
['loc-2', '12.729', '56.247']
['loc-3', '6.605', '45.14']
['loc-4', '15.123', '53.435']
['loc-5', '131.984', '11.927']
['loc-6', '155.038', '35.681']
['loc-7', '98.772', '15.125']
['loc-8', '70.991', '30.328']
['loc-9', '152.967', '30.372']
['loc-10', '96.362', '76.798']
```

8.5 pymysql 模块

8.5.1 pymysql 模块

Python 中针对 MySQL 数据库的开发操作提供有 pymysql (Python3) 和 mysqldb (Python2) 这两个核心模块。

对于数据库的开发操作，首先需要解决的就是数据库的连接。在 Python 中可以直接利用 pymysql.connect() 工厂函数获取数据库连接对象，这样就会返回一个 pymysql.connections.Connection 类的对象实例，通过此实例可以进行数据库的操作。

在进行 connect() 连接的时候必须明确地设置数据库连接的主机名称、端口号、用户名、密码、数据库名称。

MySQL 数据库操作

```
1 import pymysql
2 import traceback
3
4 INSERT = "INSERT INTO info VALUES \
5         (3, 'Henry', 3.6)"
6 SEARCH = "SELECT * FROM info"
7
8 def main():
9     try:
10         # 连接数据库
11         conn = pymysql.connect(
12             host="localhost",
13             port=3306,
14             charset="UTF8",
15             user="root",
16             password="mysqladmin",
17             database="student"
18         )
```

```

19     print("MySQL数据库连接成功")
20     print("数据库版本: %s" % conn.get_server_info())
21
22     db = conn.cursor()          # 获取数据库操作对象
23     db.execute(INSERT)          # 执行SQL语句
24     conn.commit()               # 提交事务，否则不会更新
25     print("影响数据行数: %d" % db.rowcount)
26
27     db.execute(SEARCH)
28     for row in db.fetchall():   # 查询结果
29         print(row)
30 except Exception:
31     print(traceback.format_exc())
32 finally:
33     db.close()
34
35 if __name__ == "__main__":
36     main()

```

运行结果

```

MySQL数据库连接成功
数据库版本: 8.0.26
影响数据行数: 1
(1, 'Terry', 3.7)
(2, 'Lily', 4.2)
(3, 'Henry', 3.6)

```