

Python语言程序设计

第3章 基本数据类型





前课复习

Python基本语法元素

- 缩进、注释、命名、变量、保留字
- 数据类型、字符串、整数、浮点数、列表
- 赋值语句、分支语句、函数
- input()、print()、eval()、 print()格式化



Python基本图形绘制

- 从计算机技术演进角度看待Python语言
- 海龟绘图体系及import保留字用法
- penup()、pendown()、pensize()、pencolor()
- fd()、circle()、seth()
- 循环语句：for和in、range()函数



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



保留字

```
#TempConvert.py
```

```
TempStr = input("请输入带有符号的温度值: ")
```

```
if TempStr[-1] in ['F', 'f']:
```

```
    C = (eval(TempStr[0:-1]) - 32)/1.8
```

```
    print("转换后的温度是{:.2f}C".format(C))
```

```
elif TempStr[-1] in ['C', 'c']:
```

```
    F = 1.8*eval(TempStr[0:-1]) + 32
```

```
    print("转换后的温度是{:.2f}F".format(F))
```

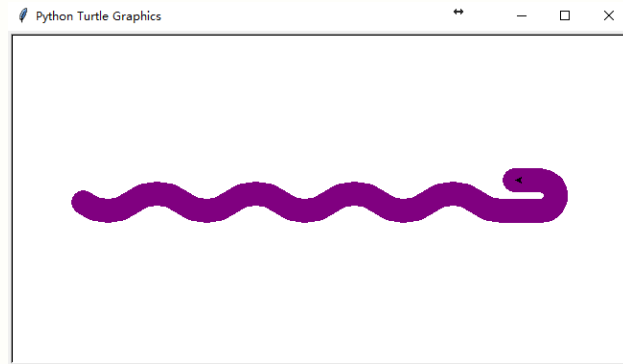
```
else:
```

```
    print("输入格式错误")
```

温度转换



```
import turtle
turtle.setup(650, 350, 200, 200)
turtle.penup()
turtle.fd(-250)
turtle.pendown()
turtle.pensize(25)
turtle.pencolor("purple")
turtle.seth(-40)
for i in range(4):
    turtle.circle(40, 80)
    turtle.circle(-40, 80)
turtle.circle(40, 80/2)
turtle.fd(40)
turtle.circle(16, 180)
turtle.fd(40 * 2/3)
turtle.done()
```



Python蟒蛇绘制





本课概要

第3章 基本数据类型



- 3.1 数字类型及操作
- 3.2 实例3: 天天向上的力量
- 3.3 字符串类型及操作
- 3.4 模块2: time库的使用
- 3.5 实例4: 文本进度条



第3章 基本数据类型

方法论

- Python数字及字符串类型

实践能力

- 初步学会编程进行字符类操作



Python语言程序设计

3.1 数字类型及操作



数字类型及操作



- 整数类型
- 浮点数类型
- 复数类型
- 数值运算操作符
- 数值运算函数



计算机对数字的识别和处理有两个基本要求：确定性和高效性

正确性：程序能够正确且无歧义地解读数据所代表的类型含义

高效性：程序能够为数字运算提供较高的计算速度，同时具备较少的存储空间代价



语言程序设计



整数类型

整数类型

与数学中整数的概念一致

- 可正可负，没有取值范围限制
- `pow(x,y)`函数：计算 x^y ，想算多大算多大

```
>>> pow(2,100)
```

```
1267650600228229401496703205376
```

```
>>> pow(2,pow(2,15))
```

```
1415461031044954789001553.....
```

理论上没有限制，实际上取值范围受限于运行python程序的计算机内存大小

整数类型

4种进制表示形式

- 十进制 : 1010, 99, -217
- 二进制 , 以0b或0B开头 : 0b010, -0B101
- 八进制 , 以0o或0O开头 : 0o123, -0O456
- 十六进制 , 以0x或0X开头 : 0x9a, -0X89

关于Python整数，就需要知道这些

- 整数无限制 `pow()`
- 4种进制表示形式



浮点数类型



浮点数类型

与数学中实数的概念一致

- 带有小数点及小数的数字
- 浮点数取值范围和小数精度都存在限制，但常规计算可忽略
- 取值范围数量级约 -10^{307} 至 10^{308} ，精度数量级 10^{-16}

Python浮点数的数值范围和小数精度受不同计算机系统的限制

```
>>>import sys  
>>>sys.float_info
```

浮点数类型

浮点数间运算存在不确定尾数，不是bug

>>> 0.1 + 0.3

0.4

>>> 0.1 + 0.2

0.30000000000000004

不确定尾数

浮点数在超过15位数字计算中产生的误差与计算机内部采用的二进制运算有关，使用浮点数无法进行极高精度的数学运算

浮点数类型

浮点数间运算存在不确定尾数，不是bug

0.1

53位二进制表示小数部分，约 10^{-16}

0.000110011001100110011001100110011001100110011001100110011010 (二进制表示)

0.1000000000000000055511151231257827021181583404541015625 (十进制表示)

二进制表示小数，可以无限接近，但不完全相同

0.1 + 0.2

结果无限接近0.3，但可能存在尾数

浮点数类型

浮点数间运算存在不确定尾数

```
>>> 0.1 + 0.2 == 0.3
```

```
False
```

```
>>> round(0.1+0.2, 1) == 0.3
```

```
True
```

round()返回浮点数的四舍五入值

浮点数类型

浮点数间运算存在不确定尾数

- **round(x, d) : 对x四舍五入, d是小数截取位数**
- **浮点数间运算与比较用round()函数辅助**
- **不确定尾数一般发生在 10^{-16} 左右, round()十分有效**

浮点数类型

浮点数可以采用科学计数法表示

- 使用字母e或E作为幂的符号，以10为基数，格式如下：

$\langle a \rangle e \langle b \rangle$ 表示 $a * 10^b$

- 例如：**4.3e-3** 值为0.0043 **9.6E5** 值为960000.0

关于Python浮点数，需要知道这些

- 取值范围和精度基本无限制
- 运算存在不确定尾数 `round()`
- 科学计数法表示



复数类型

复数类型

与数学中复数的概念一致

如果 $x^2 = -1$ ，那么 x 的值是什么？

- 定义 $j = \sqrt{-1}$ ，以此为基础，构建数学体系
- $a+bj$ 被称为复数，其中， a 是实部， b 是虚部

复数类型

复数实例

$$z = \underline{1.23\text{e-}4} + \underline{5.6\text{e}+89}j$$

- 实部是什么？ `z.real` 获得实部
- 虚部是什么？ `z.imag` 获得虚部

复数类型中的实数部分和虚数部分的数值都是浮点类型



数值运算操作符

数值运算操作符

操作符是完成运算的一种符号体系

操作符及使用	描述
$x + y$	加，x与y之和
$x - y$	减，x与y之差
$x * y$	乘，x与y之积
x / y	除，x与y之商 10/3结果是3.3333333333333335
$x // y$	整数除，x与y之整数商 10//3结果是3

数值运算操作符

操作符是完成运算的一种符号体系

操作符及使用	描述
$+ x$	x本身
$- x$	x的负值
$x \% y$	余数，模运算 10%3结果是1
$x ** y$	幂运算，x的y次幂， x^y
	当y是小数时，开方运算10**0.5结果是 $\sqrt{10}$

数值运算操作符

二元操作符有对应的增强赋值操作符

增强操作符及使用	描述
x op = y	即 $x = x \text{ op } y$, 其中 , op 为二元操作符
	$x \text{ += } y$ $x \text{ -= } y$ $x \text{ *= } y$ $x \text{ /= } y$ $x \text{ //= } y$ $x \text{ %= } y$ $x \text{ **= } y$
	>>> x = 3.1415
	>>> x **= 3 # 与 x = x **3 等价 31.006276662836743

数字类型的关系

类型间可进行混合运算，生成结果为"最宽"类型

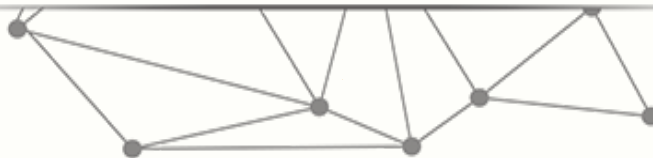
- 三种类型存在一种逐渐"扩展"或"变宽"的关系：

整数 -> 浮点数 -> 复数

- 例如： $123 + 4.0 = 127.0$ （整数+浮点数 = 浮点数）



数值运算函数



数值运算函数

一些以函数形式提供的数值运算功能

函数及使用	描述
<code>abs(x)</code>	绝对值，x的绝对值 <code>abs(-10.01)</code> 结果为 10.01
<code>divmod(x,y)</code>	商余，(x//y, x%y)，同时输出商和余数 <code>divmod(10, 3)</code> 结果为 (3, 1)
<code>pow(x, y[, z])</code>	幂余，(x**y)%z，[.]表示参数z可省略 <code>pow(3, pow(3, 99), 10000)</code> 结果为 4587

数值运算函数

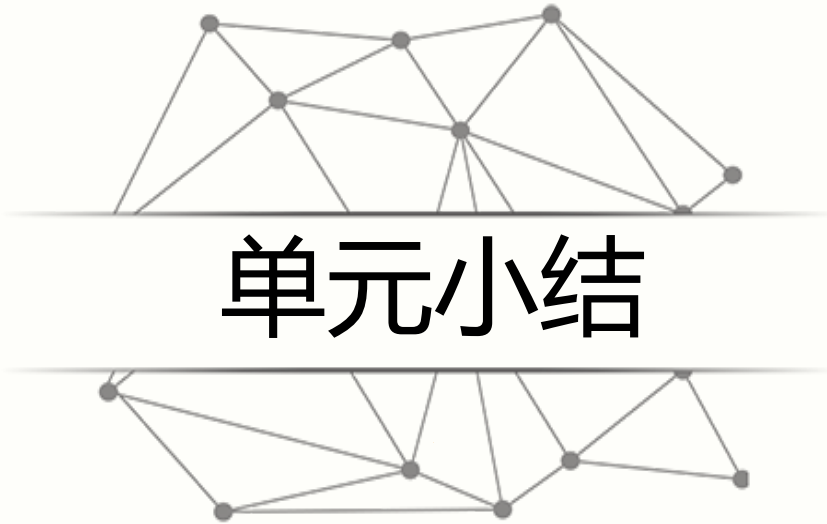
一些以函数形式提供的数值运算功能

函数及使用	描述
<code>round(x[, d])</code>	四舍五入，d是保留小数位数，默认值为0 <code>round(-10.123, 2)</code> 结果为 -10.12
<code>max(x₁, x₂, ..., x_n)</code>	最大值，返回x ₁ , x ₂ , ..., x _n 中的最大值，n不限 <code>max(1, 9, 5, 4, 3)</code> 结果为 9
<code>min(x₁, x₂, ..., x_n)</code>	最小值，返回x ₁ , x ₂ , ..., x _n 中的最小值，n不限 <code>min(1, 9, 5, 4, 3)</code> 结果为 1

数值运算函数

一些以函数形式提供的数值运算功能

函数及使用	描述
<code>int(x)</code>	将x变成整数，舍弃小数部分 <code>int(123.45)</code> 结果为123； <code>int("123")</code> 结果为123
<code>float(x)</code>	将x变成浮点数，增加小数部分 <code>float(12)</code> 结果为12.0； <code>float("1.23")</code> 结果为1.23
<code>complex(x)</code>	将x变成复数，增加虚数部分 <code>complex(4)</code> 结果为 $4 + 0j$



单元小结

数字类型及操作

- 整数类型的无限范围及4种进制表示
- 浮点数类型的近似无限范围、小尾数及科学计数法
- +、-、*、/、//、%、**、二元增强赋值操作符
- abs()、divmod()、pow()、round()、max()、min()
- int()、float()、complex()



Python语言程序设计

3.2 实例3: 天天向上的力量





"天天向上的力量"问题分析

天天向上的力量

基本问题：持续的价值

- 一年365天，每天进步1%，累计进步多少呢？

$$1.01^{365}$$

- 一年365天，每天退步1%，累计剩下多少呢？

$$0.99^{365}$$

需求分析

天天向上的力量

好好學習
天天向上
毛澤東

- 数学公式可以求解，似乎没必要用程序
- 如果是“三天打鱼两天晒网”呢？
- 如果是“双休日又不退步”呢？



"天天向上的力量"第一问

天天向上的力量

问题1：1‰的力量

- 一年365天，每天进步1‰，累计进步多少呢？

$$1.001^{365}$$

- 一年365天，每天退步1‰，累计剩下多少呢？

$$0.999^{365}$$

天天向上的力量

问题1：1‰的力量

```
#DayDayUpQ1.py
```

```
dayup = pow(1.001, 365)
```

```
daydown = pow(0.999, 365)
```

```
print("向上：{:.2f}，向下：{:.2f}".format(dayup, daydown))
```

编写上述代码，并保存为DayDayUpQ1.py文件

天天向上的力量

问题1：1‰的力量

>>> (运行结果)

向上：1.44，向下：0.69

$$1.001^{365} = 1.44$$

$$0.999^{365} = 0.69$$

1‰的力量，接近2倍，不可小觑哦



天天向上的力量

问题2：5‰和1%的力量

- 一年365天，每天进步5‰或1%，累计进步多少呢？

$$1.005^{365} \quad 1.01^{365}$$

- 一年365天，每天退步5‰或1%，累计剩下多少呢？

$$0.995^{365} \quad 0.99^{365}$$

天天向上的力量

问题2：5‰和1%的力量

```
#DayDayUpQ2.py
```

```
dayfactor = 0.005
```

使用变量的好处：一处修改即可

```
dayup = pow(1+dayfactor, 365)
```

```
daydown = pow(1-dayfactor, 365)
```

```
print("向上：{:.2f}，向下：{:.2f}".format(dayup, daydown))
```

编写上述代码，并保存为DayDayUpQ2.py文件

由于每天努力的因此根据需求的不同而不断变化，因此，引用dayfactor变量表示这个值
这种改变的好处是每次只需要修改这个变量值即可，而不用修改后续与该变量相关位置的代码

天天向上的力量

问题2：5‰和1%的力量

>>> (5‰运行结果)

向上：6.17，向下：0.16

$$1.005^{365} = 6.17$$

$$0.995^{365} = 0.16$$

5‰的力量，惊讶！

>>> (1%运行结果)

向上：37.78，向下：0.03

$$1.01^{365} = 37.78$$

$$0.99^{365} = 0.03$$

1%的力量，惊人！



"天天向上的力量"第三问

天天向上的力量

问题3：工作日的力量

- 一年365天，一周5个工作日，每天进步1%
- 一年365天，一周2个休息日，每天退步1%
- 这种工作日的力量，如何呢？

1.01^{365} (数学思维)  **for..in.. (计算思维)**

由于水平值并非每天都乘以相同系数，因此，采用循环方式实现

天天向上的力量

```
#DayDayUpQ3.py
```

```
dayup = 1.0
```

```
dayfactor = 0.01
```

```
for i in range(365):
```

```
    if i % 7 in [6,0]:
```

```
        dayup = dayup*(1-dayfactor)
```

```
    else:
```

```
        dayup = dayup*(1+dayfactor)
```

```
print("工作日的力量：{: .2f} ".format(dayup))
```

采用循环模拟365天的过程

抽象 + 自动化

天天向上的力量

问题3：工作日的力量

>>> (运行结果)

工作日的力量：4.63

$$1.001^{365} = 1.44 \quad 1.005^{365} = 6.17 \quad 1.01^{365} = 37.78$$

尽管工作日提高1%，但总体效果介于1‰和5‰的力量之间



"天天向上的力量"第四问

天天向上的力量

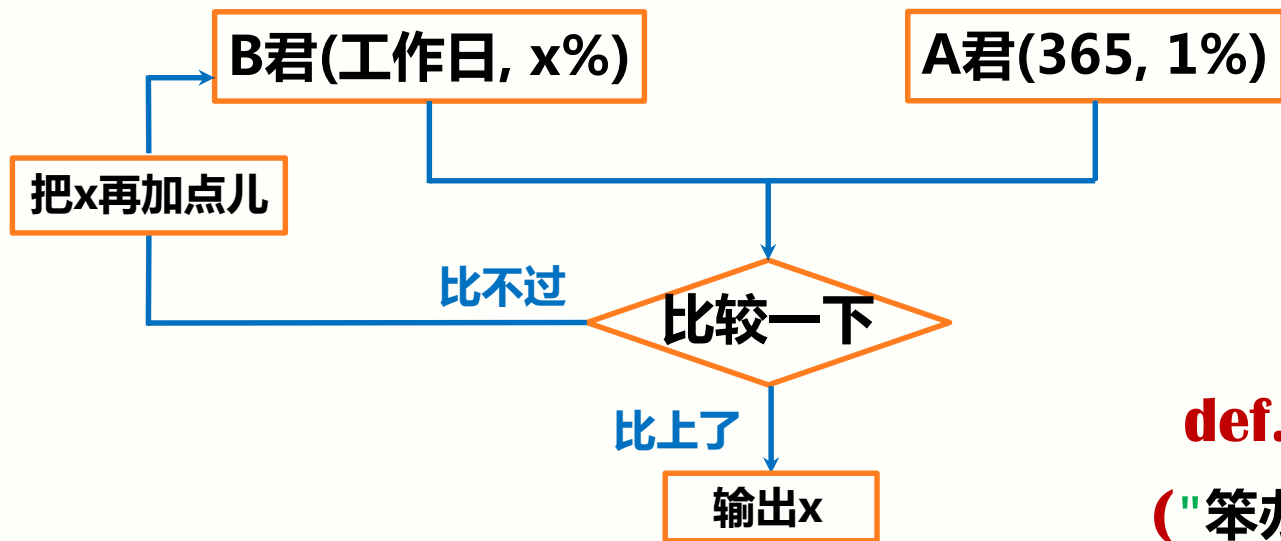
问题4：工作日的努力

- 工作日模式要努力到什么水平，才能与每天努力1%一样？
- A君：一年365天，每天进步1%，不停歇
- B君：一年365天，每周工作5天休息2天，休息日下降1%，要多努力呢？

for..in.. (计算思维) ➡ **def..while.. ("笨办法"试错)**

天天向上的力量

问题4：工作日的努力



def..while..
(**"笨办法"试错**)

天天向上的力量

```
#DayDayUpQ4.py
```

```
def dayUP(df):
```

```
    dayup = 1
```

```
    for i in range(365):
```

```
        if i % 7 in [6,0]:
```

```
            dayup = dayup*(1 - 0.01)
```

```
        else:
```

```
            dayup = dayup*(1 + df)
```

```
    return dayup
```

```
dayfactor = 0.01
```

```
while dayUP(dayfactor) < 37.78:
```

```
    dayfactor += 0.001
```

```
print("工作日的努力参数是：{:.3f}".format(dayfactor))
```

根据df参数计算工作日力量的函数

参数不同，这段代码可共用

def保留字用于定义函数

while保留字判断条件是否成立

条件成立时循环执行

天天向上的力量

问题4：工作日的努力

>>> (运行结果)

工作日的努力参数是：0.019

$$1.01^{365} = 37.78$$

$$1.019^{365} = 962.89$$

工作日模式，每天要努力到1.9%，相当于365模式每天1%的效果！



"天天向上的力量"举一反三

```
#DayDayUpQ3.py
```

```
dayup = 1.0
```

```
dayfactor = 0.01
```

```
for i in range(365):
```

```
    if i % 7 in [6,0]:
```

```
        dayup = dayup*(1-dayfactor)
```

```
    else:
```

```
        dayup = dayup*(1+dayfactor)
```

```
print("工作日的力量：{: .2f} ".format(dayup))
```

for..in.. (计算思维)

#DayDayUpQ4.py

def dayUP(df):

dayup = 1

for i in range(365):

if i % 7 in [6,0]:

dayup = dayup*(1 - 0.01)

else:

dayup = dayup*(1 + df)

return dayup

dayfactor = 0.01

while dayUP(dayfactor) < 37.78:

dayfactor += 0.001

print("工作日的努力参数是：{:.3f}".format(dayfactor))

def..while..

("笨办法"试错)

举一反三

天天向上的力量

- 实例虽然仅包含8-12行代码，但包含很多语法元素
- 条件循环、计数循环、分支、函数、计算思维
- 清楚理解这些代码能够快速入门Python语言

举一反三

问题的变化和扩展

- 工作日模式中，如果休息日不下降呢？
- 如果努力每天提高1%，休息时每天下降1‰呢？
- 如果工作3天休息1天呢？

举一反三

问题的变化和扩展

- "三天打鱼，两天晒网"呢？
- "多一份努力"呢？（努力比下降多一点儿）
- "多一点懈怠"呢？（下降比努力多一点儿）

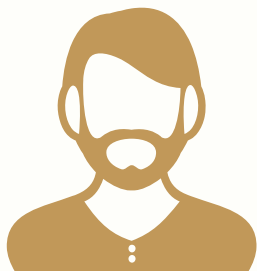
好好学习
天天向上
毛泽东

Python语言程序设计

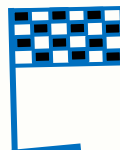
3.3 字符串类型及操作



字符串类型及操作



- 字符串类型的表示
- 字符串操作符
- 字符串处理函数
- 字符串处理方法
- 字符串类型的格式化





字符串类型的表示

字符串

由0个或多个字符组成的有序字符序列

- 字符串由一对单引号或一对双引号表示

"请输入带有符号的温度值: " 或者 'c'

- 字符串是字符的有序序列，可以对其中的字符进行索引

"请" 是 "请输入带有符号的温度值: " 的第0个字符

字符串

字符串有 2类共4种 表示方法

- 由一对单引号或双引号表示，仅表示单行字符串

"请输入带有符号的温度值：" 或者 'C'

- 由一对三单引号或三双引号表示，可表示多行字符串

''' Python

语言 '''

Python语言为何提供 2类共4种 字符串表示方式？

字符串

字符串有 2类共4种 表示方法

- 如果希望在字符串中包含双引号或单引号呢？

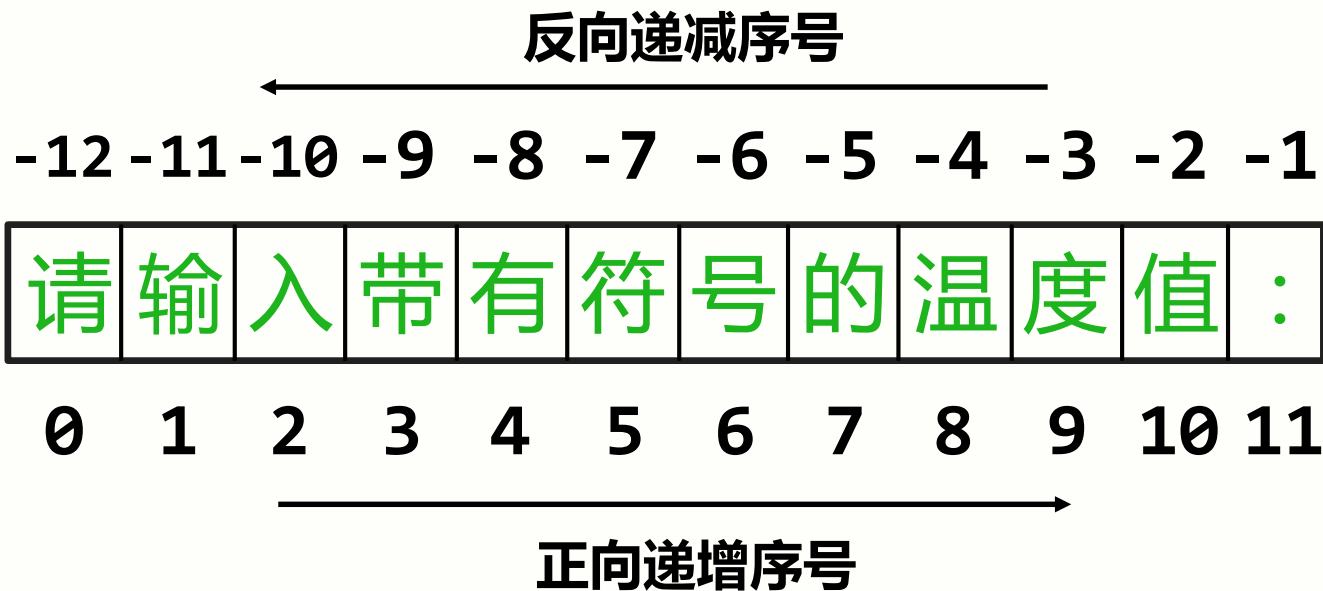
'这里有个双引号("' 或者 "这里有个单引号(')"

- 如果希望在字符串中既包括单引号又包括双引号呢？

''' 这里既有单引号(')又有双引号 (") '''

字符串的序号

正向递增序号 和 反向递减序号



字符串的使用

使用[]获取字符串中一个或多个字符

- 索引：返回字符串中单个字符 <字符串>[M]

"请输入带有符号的温度值: "[0] 或者 TempStr[-1]

- 切片：返回字符串中一段字符串 <字符串>[M: N]

"请输入带有符号的温度值: "[1:3] 或者 TempStr[0:-1]

字符串切片高级用法

使用[M: N: K]根据步长对字符串切片

- <字符串>[M: N] , M缺失表示**至开头** , N缺失表示**至结尾**

"〇一二三四五六七八九十"[:3] 结果是 "〇一二"

- <字符串>[M: N: K] , 根据步长K对字符串切片

"〇一二三四五六七八九十"[1:8:2] 结果是 "一三五七"

"〇一二三四五六七八九十"[::-1] 结果是 "十九八七六五四三二一〇"

1.字符串切片：从字符串中取出相应的元素，重新组成一个新的字符串

语法：字符串[开始元素下标 : 结束元素下标 : 步长] # 字符串的每个元素都有正负两种下标

步长：切片间隔以及切片方向,默认值是1；实际意义为从开始取一个数据，跳过步长的长度，再取一个数据，一直到结束索引

步长为正值： 开始索引默认为0， 结束索引默认为最后是len()+1，从开始索引从左往右走；步长为负值，开始索引默认为-1， 结束索引默认为开始，不能认为是0，也不能认为是-1，从开始索引从右往左走；



例如：L=list(range(10)) #L中的元素是0-9

案例一：L[::1]的值

结果：[0,1,2,3,4,5,6,7,8,9]

这里开始下标=>0,结束下标=>9或者开始下标=>-10,结束下标=>-1

案例二：L[::-1]的值

结果：[9,8,7,6,5,4,3,2,1,0]

这里开始下标=>9,结束下标=>0或者开始下标=>-1,结束下标=>-10

案例三：L[-1:1]的值

结果：[] L[-1:1]翻译过来变为L[-1:1:1],由于最后一个元素后面找不到坐标为1的，故返回为空

案例四：L[-1:1:-1]的值

结果：[9,8,7,6,5,4,3,2]

字符串的特殊字符

转义符 \

- 转义符表达特定字符的本意

"这里有个双引号(\"")" 结果为 这里有个双引号(")

- 转义符形成一些组合，表达一些不可打印的含义

"\b" 回退 "\n" 换行(光标移动到下行首) "\r" 回车(光标移动到本行首)



字符串操作符



字符串操作符

由0个或多个字符组成的有序字符序列

操作符及使用	描述
$x + y$	连接两个字符串x和y
$n * x$ 或 $x * n$	复制n次字符串x
$x \text{ in } s$	如果x是s的子串，返回True，否则返回False

字符串操作符

获取星期字符串

- 输入：1-7的整数，表示星期几
- 输出：输入整数对应的星期字符串
- 例如：输入3，输出 星期三

字符串操作符

获取星期字符串

```
#WeekNamePrintV1.py
```

```
weekStr = "星期一星期二星期三星期四星期五星期六星期日"
```

```
weekId = eval(input("请输入星期数字(1-7) : "))
```

```
pos = (weekId - 1 ) * 3
```

```
print(weekStr[pos: pos+3])
```

通过在字符串中截取适当子串实现星期名称的查找
问题的关键在于找出子串的剪切位置

字符串操作符

获取星期字符串

```
#WeekNamePrintV2.py
```

```
weekStr = "一二三四五六日"
```

```
weekId = eval(input("请输入星期数字(1-7) : "))
```

```
print("星期" + weekStr[weekId-1])
```



字符串处理函数

字符串处理函数

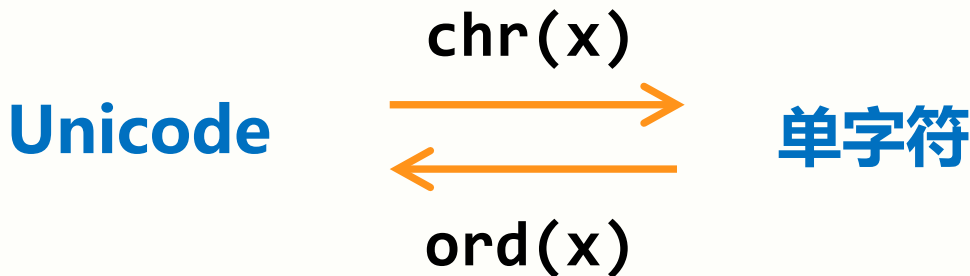
一些以函数形式提供的字符串处理功能

函数及使用	描述
len(x)	长度，返回字符串x的长度，英、中文字符都是1个 len("一二三456") 结果为 6
str(x)	任意类型x所对应的字符串形式 str(1.23)结果为"1.23" str([1,2])结果为"[1,2]"
hex(x) 或 oct(x)	整数x的十六进制或八进制小写形式字符串 hex(425)结果为"0x1a9" oct(425)结果为"0o651"

字符串处理函数

一些以函数形式提供的字符串处理功能

函数及使用	描述
<code>chr(x)</code>	x为Unicode编码，返回其对应的字符
<code>ord(x)</code>	x为字符，返回其对应的Unicode编码



Unicode编码

Python字符串的编码方式

- 统一字符编码，即覆盖几乎所有字符的编码方式
- 从0到1114111 (0x10FFFF)空间，每个编码对应一个字符
- Python字符串中每个字符都是Unicode编码字符

编码：指每个字符在计算机中可以表示为一个数字
字符串以编码序列方式存储在计算机中，python字符串中每个字符都使用Unicode编码表示

Unicode编码

一些有趣的例子

```
>>> "1 + 1 = 2 " + chr(10004)
```

```
'1 + 1 = 2 ✓'
```

```
>>> "这个字符𐄂的Unicode值是：" + str(ord("𐄂"))
```

```
'这个字符𐄂的Unicode值是： 9801'
```

```
>>> for i in range(12):
```

```
    print(chr(9800 + i), end="")
```

```
𐄂𐄃𐄄𐄅𐄆𐄇𐄈𐄉𐄊𐄋𐄌𐄍
```

包含end="作为print()的一个参数，会使该函数关闭“在输出中自动包含换行”的默认行为。



字符串处理方法

字符串处理方法

"方法"在编程中是一个专有名词

- **"方法"特指<a>.()风格中的函数()**
- **方法本身也是函数，但与<a>有关，<a>.()风格使用**
- **字符串或字符串变量是<a>，存在一些可用方法**

在python解释器内部，所有数据类型都采用面向对象方式实现，封装为一个类
字符串也是一个类，它具有类似<a>.()形式的字符串处理函数，在面向对象中，这类函数被称为“方法”

字符串处理方法

一些以方法形式提供的字符串处理功能

方法及使用 1/3	描述
<code>str.lower()</code> 或 <code>str.upper()</code>	返回字符串的副本，全部字符小写/大写 <code>"AbCdEfGh".lower()</code> 结果为 <code>"abcdefgh"</code>
<code>str.split(sep=None)</code>	返回一个列表，由str根据sep被分隔的部分组成 <code>"A,B,C".split(",")</code> 结果为 <code>['A','B','C']</code>
<code>str.count(sub)</code>	返回子串sub在str中出现的次数 <code>"an apple a day".count("a")</code> 结果为 4

字符串处理方法

一些以方法形式提供的字符串处理功能

方法及使用 2/3	描述
<code>str.replace(old, new)</code>	返回字符串str副本，所有old子串被替换为new <code>"python".replace("n","nAIA")</code> 结果为 <code>"pythonAIA"</code>
<code>str.center(width[,fillchar])</code>	字符串str根据宽度width居中，fillchar可选 <code>"python".center(20,"=")</code> 结果为 <code>'=====python====='</code>

`str.center(width[,fillchar])`方法返回长度为width的字符串，其中，str处于新字符串中心的位置，两侧新增字符采用fillchar填充，当width小于字符串长度时，返回str

字符串处理方法

一些以方法形式提供的字符串处理功能

方法及使用 3/3	描述
<code>str.strip(chars)</code>	从str中去掉在其左侧和右侧chars中列出的字符 <code>"= python= ".strip(" =np")</code> 结果为 <code>"ytho"</code> #该方法只能删除开头或是结尾的字符，不能删除中间部分的字符
<code>str.join(iter)</code>	在iter变量除最后元素外每个元素后增加一个str <code>", ".join("12345")</code> 结果为 <code>"1,2,3,4,5"</code> #主要用于字符串分隔等



字符串类型的格式化

字符串类型的格式化

格式化是对字符串进行格式表达的方式

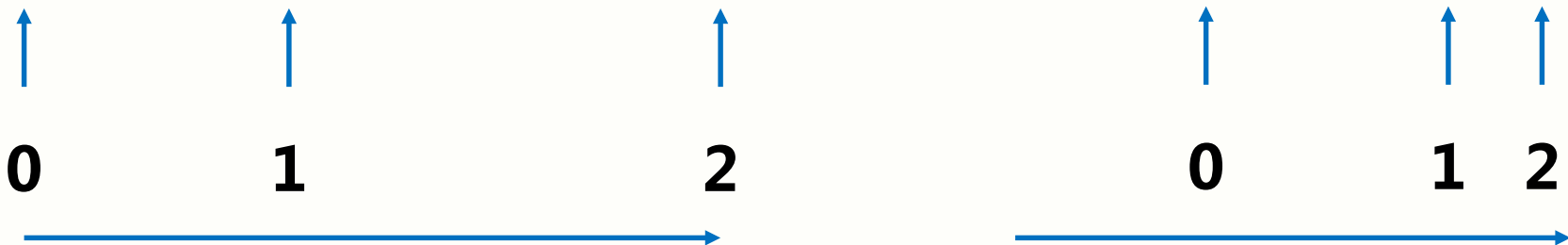
- 字符串格式化使用.format()方法，用法如下：

<模板字符串>.format(<逗号分隔的参数>)

字符串类型的格式化

槽

"{ } : 计算机{ } 的CPU占用率为{ }%".format("2018-10-10", "C", 10)



字符串中槽{}的默认顺序

format()中参数的顺序

模板字符串由一系列槽组成，用来控制修改字符串中嵌入值出现的位置
基本思想是将format()方法中逗号分隔的参数按照序号关系替换到模板字符串的槽中

字符串类型的格式化

槽

The diagram illustrates the mapping of format specifiers in the string `"{1}:计算机{0}的CPU占用率为{2}%"` to the arguments in the `.format()` call. Blue lines with arrows show the following connections: from `{1}` to the first argument `"2018-10-10"`, from `{0}` to the second argument `"C"`, and from `{2}` to the third argument `10`.

```
"{1}:计算机{0}的CPU占用率为{2}%" .format("2018-10-10", "C", 10)
```

如果大括号中没有序号，则按出现顺序替换
如果大括号中指定了使用参数的序号，按照序号对应参数替换

format()方法的格式控制

槽内部对格式化的配置方式

{ <参数序号> : <格式控制标记> }

用来控制参数显示时的格式

:	<填充>	<对齐>	<宽度>	< , >	< .精度>	<类型>
引导 符号	用于填充的 单个字符	< 左对齐 > 右对齐 ^ 居中对齐	槽设定的输 出宽度	数字的千位 分隔符	浮点数小数 精度 或 字 符串最大输 出长度	整数类型 b, c, d, o, x, X 浮点数类型 e, E, f, %

format()方法的格式控制

```
Python 3.5.4 Shell
File Edit Shell Debug Options
Python 3.5.4 (v3.5.4:3f56838, Aug 14 2015) on win32
Type "copyright", "credits" or "help()" to get more help.
>>> "{0:=20}".format("PYTHON")
'=====PYTHON====='
>>> "{0:=^21}".format("PYTHON")
'*****PYTHON*****'
>>>
```

:	<填充>	<对齐>	<宽度>	<, >	<.精度>	<类型>
引导符号	用于填充的单个字符	< 左对齐 > 右对齐 ^ 居中对齐	槽设定的输出宽度	<pre>>>> "{0:=^20}".format("PYTHON") '=====PYTHON=====' >>> "{0:*>20}".format("BIT") '*****BIT' >>> "{:10}".format("BIT") 'BIT'</pre>		

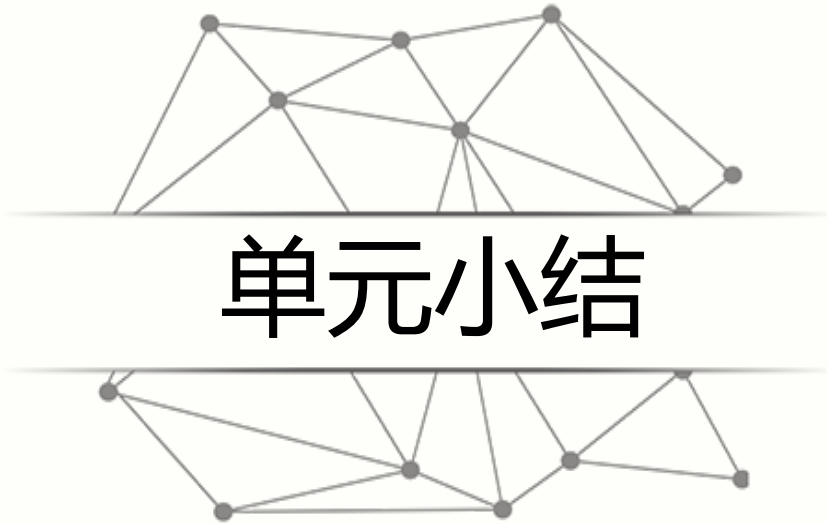
- ✓ <宽度>指当前槽的设定输出字符宽度，如果该槽对应的format()参数长度比<宽度>设定值大，则使用参数实际长度；如果该值的实际位数小于指定宽度，则位数将被默认以空格字符补充
- ✓ <对齐>指参数在宽度内输出时的对齐方式
- ✓ <填充>值宽度内除了参数外的字符采用什么方式表示，默认采用空格

format()方法的格式控制

:	<填充>	<对齐>	<宽度>	<,>	<.精度>	<类型>
<pre>>>> "{0:,.2f}".format(12345.6789)</pre> <pre>'12,345.68'</pre>				数字的千位 分隔符	浮点数小数 精度或字 符串最大输 出长度	整数类型 b, c, d, o, x, X 浮点数类型 e, E, f, %
<pre>>>> "{0:b},{0:c},{0:d},{0:o},{0:x},{0:X}".format(425)</pre> <pre>'110101001,Σ,425,651,1a9,1A9'</pre>						
<pre>>>> "{0:e},{0:E},{0:f},{0:%}".format(3.14)</pre> <pre>'3.140000e+00,3.140000E+00,3.140000,314.000000%'</pre>						

✓ <,>用于显示数字类型的千位分隔符
✓ <.精度>表示两个含义，对于浮点数，精度表示小数部分输出的有效位数。对于字符串，精度表示输出的最大长度
✓ <类型>表示输出整数和浮点数类型的格式规则

c: 输出整数对应的Unicode字符 f: 输出浮点数的标准浮点形式 %: 输出浮点数的百分比形式



单元小结

字符串类型及操作

- 正向递增序号、反向递减序号、<字符串>[M:N:K]
- +、*、in、len()、str()、hex()、oct()、ord()、chr()
- .lower()、.upper()、.split()、.count()、.replace()
- .center()、.strip()、.join()、.format()格式化



Python语言程序设计

3.4 模块2: time库的使用





time库基本介绍

time库概述

time库是Python中处理时间的标准库

- 计算机时间的表达

```
import time
```

- 提供获取系统时间并格式化输出功能

```
time.<b>()
```

- 提供系统级精确计时功能，用于程序性能分析

time库概述

time库包括三类函数

- 时间获取 : `time()` `ctime()` `gmtime()`
- 时间格式化 : `strftime()` `strptime()`
- 程序计时 : `sleep()`, `perf_counter()`



时间获取

函数	描述
time()	获取当前时间戳，即计算机内部时间值，浮点数 <code>>>>time.time()</code> <code>1516939876.6022282</code>
ctime()	获取当前时间并以易读方式表示，返回字符串 <code>>>>time.ctime()</code> <code>'Wed Sep 9 17:01:42 2020'</code>

时间获取

函数	描述
gmtime()	<p>获取当前时间，表示为计算机可处理的时间格式</p> <pre>>>>time.gmtime() time.struct_time(tm_year=2018, tm_mon=1, tm_mday=26, tm_hour=4, tm_min=11, tm_sec=16, tm_wday=4, tm_yday=26, tm_isdst=0)</pre>

tm_isdst :传入的时间是否是 DST(夏令时)



时间格式化

时间格式化

将时间以合理的方式展示出来

- **格式化：类似字符串格式化，需要有展示模板**
- **展示模板由特定的格式化控制符组成**
- **strftime()方法**

时间格式化

函数	描述
strftime(tpl, ts)	<p>tpl是格式化模板字符串，用来定义输出效果</p> <p>ts是计算机内部时间类型变量</p> <pre>>>>t = time.gmtime() >>>time.strftime("%Y-%m-%d %H:%M:%S",t) '2018-01-26 12:55:20'</pre>

格式化控制符

格式化字符串	日期/时间说明	值范围和实例
%Y	年份	0000~9999，例如：1900
%m	月份	01~12，例如：10
%B	月份名称	January~December，例如：April
%b	月份名称缩写	Jan~Dec，例如：Apr
%d	日期	01~31，例如：25
%A	星期	Monday~Sunday，例如：Wednesday

格式化控制符

格式化字符串	日期/时间说明	值范围和实例
%a	星期缩写	Mon~Sun , 例如 : Wed
%H	小时 (24h制)	00~23 , 例如 : 12
%I	小时 (12h制)	01~12 , 例如 : 7
%p	上/下午	AM, PM , 例如 : PM
%M	分钟	00~59 , 例如 : 26
%S	秒	00~59 , 例如 : 26

时间格式化

```
>>>t = time.gmtime()
```

```
>>>time.strftime("%Y-%m-%d %H:%M:%S",t)
```



'2018-01-26 12:55:20'



```
>>>timeStr = '2018-01-26 12:55:20'
```

```
>>>time.strptime(timeStr, "%Y-%m-%d %H:%M:%S")
```

strptime函数根据指定的格式把一个时间字符串解析为时间元组

时间格式化

函数	描述
<p><code>strptime(str, tpl)</code> 根据指定的格式把一个时间字符串解析为时间元组</p>	<p>str是字符串形式的时间值 tpl是格式化模板字符串，用来定义输入效果</p> <pre>>>>timeStr = '2018-01-26 12:55:20' >>>time.strptime(timeStr, "%Y-%m-%d %H:%M:%S") time.struct_time(tm_year=2018, tm_mon=1, tm_mday=26, tm_hour=4, tm_min=11, tm_sec=16, tm_wday=4, tm_yday=26, tm_isdst=0)</pre>



程序计时应用



程序计时

程序计时应用广泛

- 程序计时指测量起止动作所经历时间的过程
- 测量时间：`perf_counter()`
- 产生时间：`sleep()`

程序计时

函数	描述
perf_counter()	<p>返回一个CPU级别的精确时间计数值，单位为秒 由于这个计数值起点不确定，连续调用差值才有意义</p> <pre>>>>start = time.perf_counter() 318.66599499718114 >>>end = time.perf_counter() 341.3905185375658 >>>end - start 22.724523540384666</pre>

程序计时

函数	描述
sleep(s)	<p>s拟休眠的时间，单位是秒，可以是浮点数</p> <pre>>>>def wait(): time.sleep(3.3)</pre> <p>>>>wait() #程序将等待3.3秒后再退出</p>

Python语言程序设计

3.5 实例4: 文本进度条





"文本进度条"问题分析

文本进度条

用过计算机的都见过

- 进度条什么原理呢？



75%

进度条是计算机处理任务或执行软件中常用的增强用户体验的重要手段
能够实时显示任务或软件的执行进度



需求分析

文本进度条

- 采用字符串方式打印可以动态变化的文本进度条
- 进度条需要能在一行中逐渐变化

问题分析

如何获得文本进度条的变化时间？

- 采用sleep()模拟一个持续的进度
- 似乎不那么难

由于程序执行速度远超过人眼的视觉停留时间，直接进行输出几乎是瞬间完成，不利于观察
因此，在执行的过程中，可以使用sleep函数将当前程序暂时挂起



"文本进度条"简单的开始

简单的开始

#TextProBarV1.py

```
import time
```

```
scale = 10
```

```
print("-----执行开始-----")
```

```
for i in range(scale+1):
```

```
    a = '*' * i
```

```
    b = '.' * (scale - i)
```

```
    c = (i/scale)*100
```

```
    print("{:^3.0f}%[{}->{}]" .format(c,a,b))
```

```
    time.sleep(0.1)
```

```
print("-----执行结束-----")
```

```
-----执行开始-----  
 0 %[->.....]  
10 %[*->.....]  
20 %]**->.....]  
30 %***->.....]  
40 %****->.....]  
50 %*****->.....]  
60 %*****->....]  
70 %*****->...]  
80 %*****->..  
90 %*****->.  
100%[*****->]  
-----执行结束-----
```

默认情况，print()函数在输出结尾处自动产生一个\n换行符，从而让光标自动移动到下一行行首



"文本进度条"单行动态刷新

单行动态刷新

刷新的关键是 \r

- 刷新的本质是：用之后打印的字符覆盖之前的字符
- 不能换行：print()需要被控制
- 要能回退：打印后光标退回到之前的位置 \r

单行动态刷新

#TextProBarV2.py

import time

for i in range(101):

print("\r{:3}%".format(i), end="")

time.sleep(0.1)

包含end="作为print()的一个参数，会使该函数关闭“在输出中自动包含换行”的默认行为。

```
= RESTART: C:/Users/Admin/AppData/Local/Programs/Python/Python35/20200909.py =  
0% 1% 2% 3% 4% 5% 6% 7% 8% 9% 10% 11% 12% 13% 14% 15% 16% 17% 18% 19%  
20% 21% 22% 23% 24% 25% 26% 27% 28% 29% 30% 31% 32% 33% 34% 35% 36% 37% 38% 39%  
40% 41% 42% 43% 44% 45% 46% 47% 48% 49% 50% 51% 52% 53% 54% 55% 56% 57% 58% 59%  
60% 61% 62% 63% 64% 65% 66% 67% 68% 69% 70% 71% 72% 73% 74% 75% 76% 77% 78% 79%  
80% 81% 82% 83% 84% 85% 86% 87% 88% 89% 90% 91% 92% 93% 94% 95% 96% 97% 98% 99%  
100%
```

IDLE屏蔽了\r功能

单行动态刷新

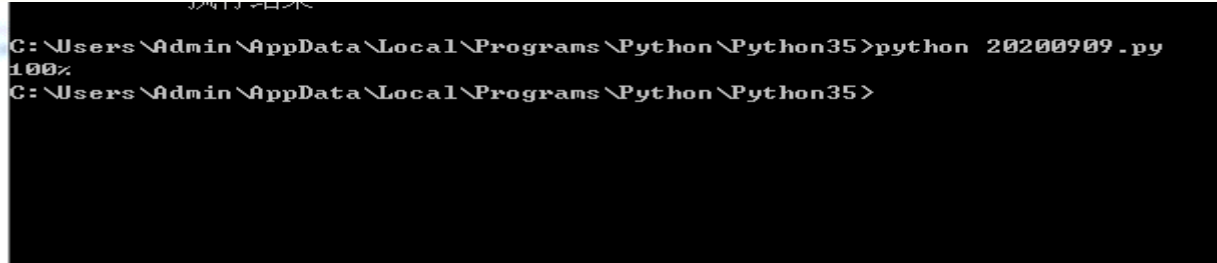
```
#TextProBarV2.py
```

```
import time
```

```
for i in range(101):
```

```
    print("\r{:3}%".format(i), end="")
```

```
    time.sleep(0.1)
```



```
C:\Users\Admin\AppData\Local\Programs\Python\Python35>python 20200909.py  
100%  
C:\Users\Admin\AppData\Local\Programs\Python\Python35>
```

命令行执行



"文本进度条"实例完整效果

完整效果

```
#TextProBarV3.py
```

```
import time
```

```
scale = 50
```

```
print("执行开始".center(scale//2, "-")) #字符串str根据宽度width居中，fillchar可选
```

```
start = time.perf_counter()
```

```
for i in range(scale+1):
```

```
    a = '*' * i
```

```
    b = '.' * (scale - i)
```

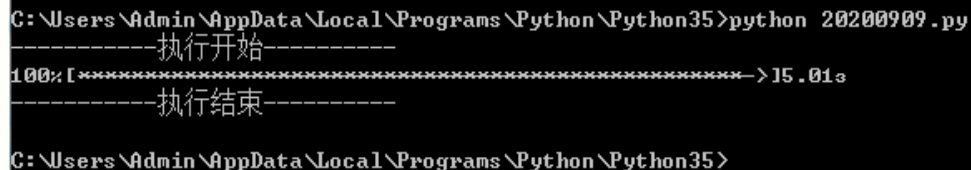
```
    c = (i/scale)*100
```

```
    dur = time.perf_counter() - start
```

```
    print("\r{:^3.0f}%[{}->{}]{:.2f}s".format(c,a,b,dur),end='')
```

```
    time.sleep(0.1)
```

```
print("\n"+"执行结束".center(scale//2, '-'))
```



```
C:\Users\Admin\AppData\Local\Programs\Python\Python35>python 20200909.py
-----执行开始-----
100%[*----->]15.01s
-----执行结束-----

C:\Users\Admin\AppData\Local\Programs\Python\Python35>
```



"文本进度条"举一反三

#TextProBarV3.py

import time

scale = 50

print("执行开始".center(scale//2, "-"))

start = time.perf_counter()

for i in range(scale+1):

 a = '*' * i

 b = '.' * (scale - i)

 c = (i/scale)*100

 dur = time.perf_counter() - start

 print("\r{:^3.0f}%[{}->{}]{:.2f}s".format(c,a,b,dur),end='')

 time.sleep(0.1)

print("\n"+"执行结束".center(scale//2, '-'))

举一反三

计算问题扩展

- 文本进度条程序使用了perf_counter()计时
- 计时方法适合各类需要统计时间的计算问题
- 例如：比较不同算法时间、统计程序运行时间

举一反三

进度条应用

- 在任何运行时间需要较长的程序中增加进度条
- 在任何希望提高用户体验的应用中增加进度条
- 进度条是人机交互的纽带之一

举一反三

文本进度条的不同设计函数

设计名称	趋势	设计函数
Linear	Constant	$f(x) = x$
Early Pause	Speeds up	$f(x) = x + (1 - \sin(x * \pi * 2 + \pi / 2)) / -8$
Late Pause	Slows down	$f(x) = x + (1 - \sin(x * \pi * 2 + \pi / 2)) / 8$
Slow Wavy	Constant	$f(x) = x + \sin(x * \pi * 5) / 20$
Fast Wavy	Constant	$f(x) = x + \sin(x * \pi * 20) / 80$

举一反三

文本进度条的不同设计函数

设计名称	趋势	设计函数
Power	Speeds up	$f(x) = (x + (1-x) * 0.03)^2$
Inverse Power	Slows down	$f(x) = 1 + (1-x)^{1.5} * -1$
Fast Power	Speeds up	$f(x) = (x + (1-x)/2)^8$
Inverse Fast Power	Slows down	$f(x) = 1 + (1-x)^3 * -1$

