

Table of Contents

Python语言	1.1
Python开发环境	1.2
计算机组成	1.2.1
编程语言和Python	1.2.2
Python语言介绍	1.2.3
Python开发环境搭建	1.2.4
Python基础语法	1.3
注释	1.3.1
变量	1.3.2
分支语句	1.3.3
循环语句	1.3.4
函数	1.3.5
容器	2.1
字符串	2.1.1
列表	2.1.2
元组	2.1.3
字典	2.1.4
集合	2.1.5
案例-员工管理系统	2.1.6

基础课程概述

本阶段课程分为三大部分, 分别如下:

- Python开发环境搭建
- Python语法规则

1 Python开发环境

学习目标:

1. 能够说出计算机有那两部分组成
2. 能够说出操作系统的作用
3. 能够说出编程语言的作用
4. 能够说出解释器的作用
5. 能够说出Python解释器种类
6. 能够说出目前Python主流的两大版本是哪些
7. 能够说出在不同系统上搭建Python开发环境的流程
8. 知道Python语言历史、优缺点、应用领域

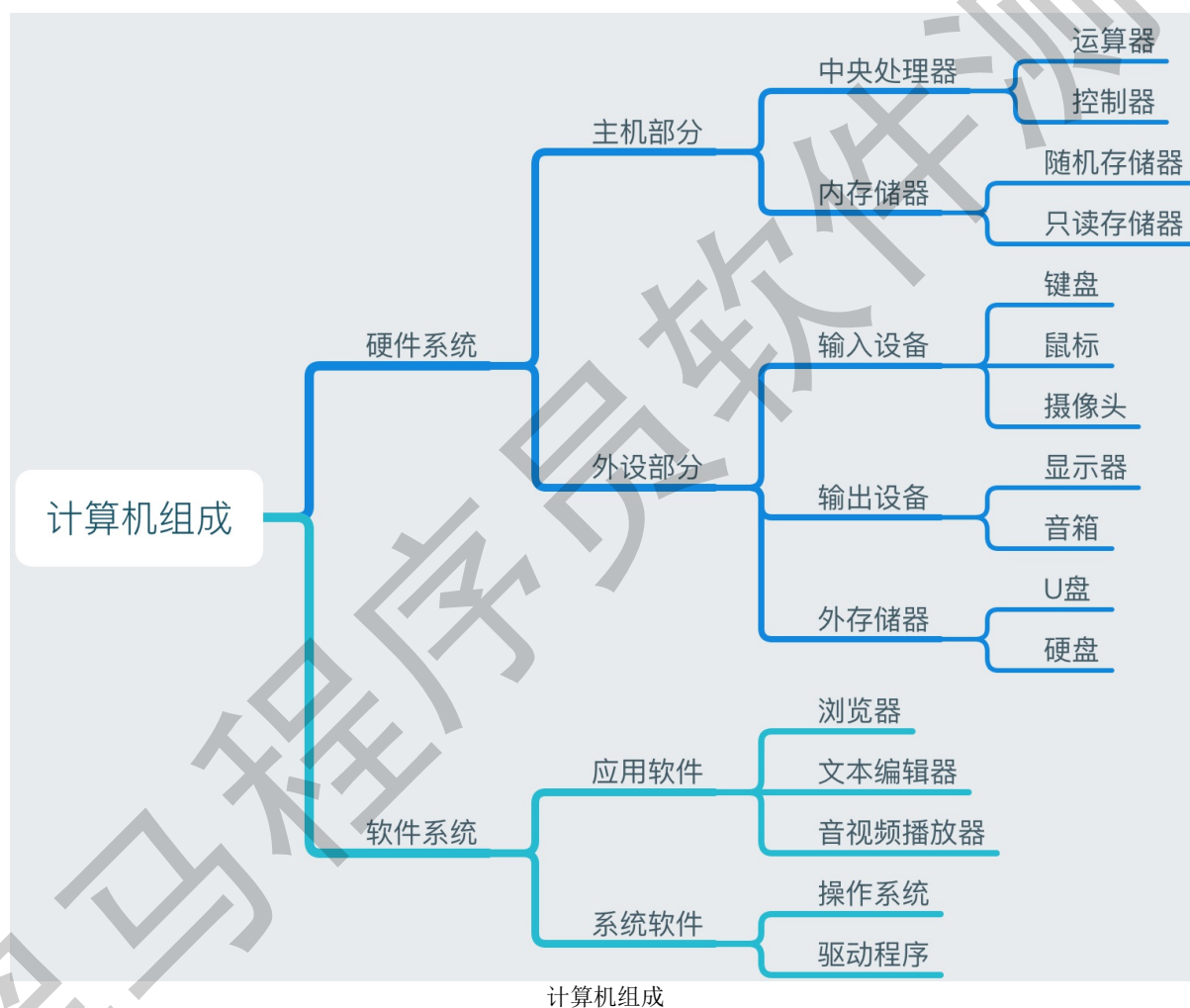
1.1 计算机组成

学习目标:

1. 能够说出计算机有那两部分组成
2. 能够说出操作系统的作用

计算机是可以进行数值计算和逻辑运算, 并且具有存储功能的电子机器.

计算机由硬系统件和软件系统组成.



1.1.1 硬件系统

主要分为主机和外设两部分, 是指那些构成计算机系统的物理实体, 它们主要由各种各样的电子器件和机电装置组成.

- 运算器: 负责数据的算术运算和逻辑运算, 即数据的加工处理.
- 控制器: 是整个计算机的中枢神经, 分析程序规定的控制信息, 并根据程序要求进行控制, 协调计算机各部分组件工作及内存与外设的访问等.

- 运算器和控制器统称中央处理器（即CPU）。
- 存储器：实现记忆功能的部件，用来存储程序、数据和各种信号、命令等信息，并在需要时提供这些信息。
- 输入设备：实现将程序、原始数据、文字、字符、控制命令或现场采集的数据等信息输入到计算机。
- 输出设备：实现将计算机处理后生成的中间结果或最后结果（各种数据符号及文字或各种控制信号等信息）输出出来。

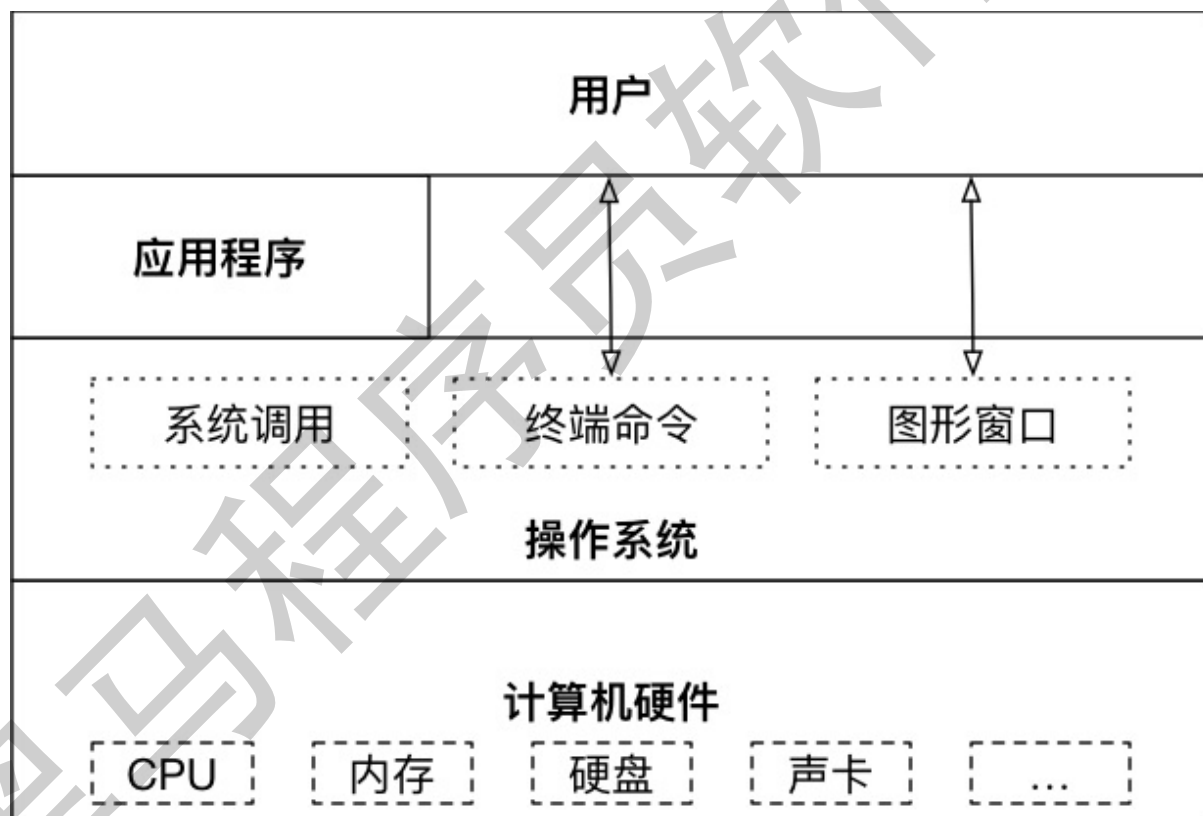
1.1.2 软件系统

主要分为系统软件和应用软件，是指计算机运行所需的各种各样的计算机程序。

系统软件的任务是既要保证计算机硬件的正常工作，又要使计算机硬件的性能得到充分发挥，并且为计算机用户提供一个比较直观、方便和友好的使用界面。

1.1.2.1 操作系统

- 没有安装操作系统的计算机，通常被称为裸机
- 如果想在裸机上运行自己所编写的程序，就必须用机器语言书写程序。



操作系统功能示意图

操作系统提供以下功能:

- 给用户间接操作硬件的方式
 - 图形窗口方式
 - 终端命令方式
- 给开发者提供的间接操作硬件的方式
 - 系统调用

简言之: 主要作用是管理好硬件设备, 并为用户和开发者提供一个简单的接口, 以便于使用。

1.1.2.2 驱动程序

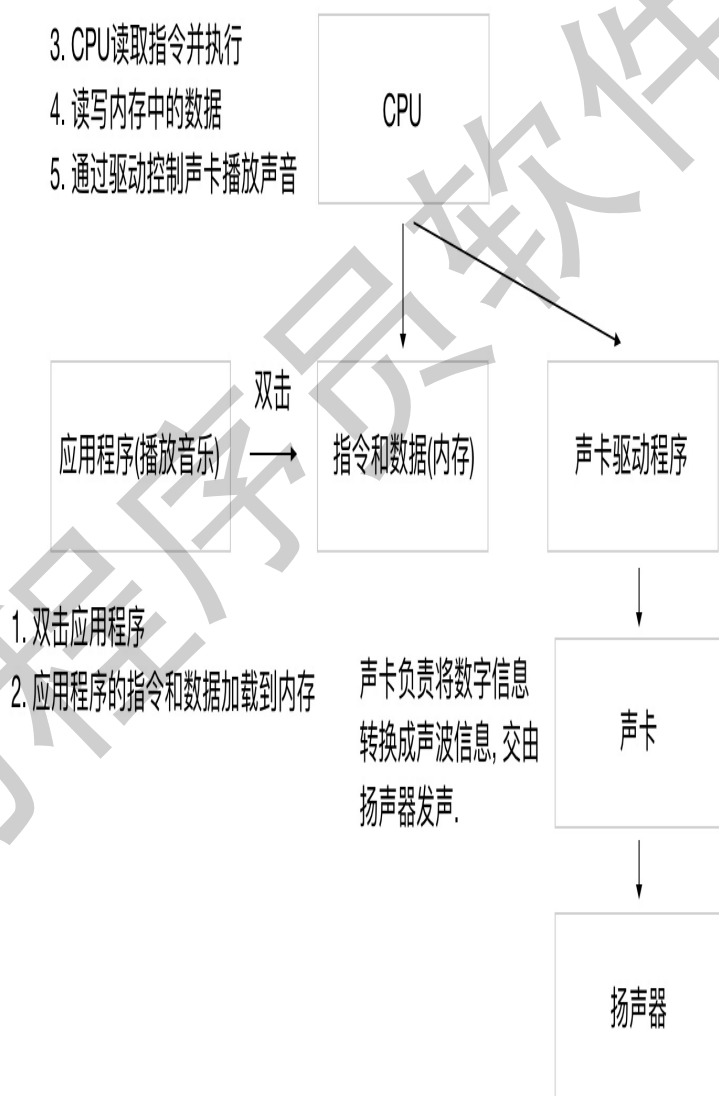
驱动程序: 驱动程序指的是设备驱动程序, 是一种可以使计算机和设备通信的特殊程序. 操作系统通过这个程序操作和控制硬件设备工作, 如果某设备的驱动程序没有正确安装, 该设备则无法工作. 所以一般操作系统安装完毕之后, 首要就是要安装硬件设备的驱动程序, 不过大多数情况下, 我们并不需要安装驱动程序, 例如硬件、显示器、光驱就不需要安装驱动程序, 而显卡、声卡、摄像头、打印机等就需要安装驱动程序.

比如, 让声卡播放音乐, 它首先会发送响应的指令到声卡驱动程序, 声卡驱动程序接受到后, 马上将其翻译成声卡才能听懂电子信号命令, 从而让声卡播放音乐.

简言之, 驱动程序提供了硬件到操作系统的一个接口以及协调二者之间的关系. 扮演者硬件和操作系统之间的一个桥梁的作用.

1.1.3 计算机运行程序的过程

我们从计算机如何播放音乐, 来了解计算机执行程序的过程.



程序运行过程

1. 双击应用程序, 应用程序中的指令和数据就会加载到内存中.
2. CPU从内存中获取指令并执行, 在内存中存储运行之后的数据.
3. CPU控制硬件进行相应的操作.

1.1.4 小结

1. 计算机是能够进行数值运算、逻辑运算, 并且具有存储功能的电子设备.
2. 计算机由硬件系统和软件系统构成.
3. 计算机中所有程序的运行都是在内存中进行的, 暂时存放CPU中的运算数据.
4. 操作系统的作用就是来管理硬件, 为普通用户和开发者提供一种间接控制硬件的方式.
5. 操作系统为普通用户提供了终端、图形化操作硬件的方式.
6. 操作系统为开发者提供了系统调用操作硬件的方式.
7. 驱动程序扮演操作系统和硬件之间的桥梁.

1.1.5 思考

- 我们可以直接控制硬件吗? 缺点是什么?
- 为什么程序运行时, 要将程序数据存储在内存中? 内存的作用是什么?
- 计算机中程序的执行流程是什么样的?

1.2 编程语言

学习目标:

1. 能够说出编程语言的作用
2. 能够说出解释器的作用
3. 能够说出Python解释器种类

1.2.1 编程语言

我们如何根据自己的需求控制硬件? 编程语言

编程语言(计算机语言)是人们为了控制计算机, 而设计的一种符号和文字的组合, 从而实现向计算机发出指令.

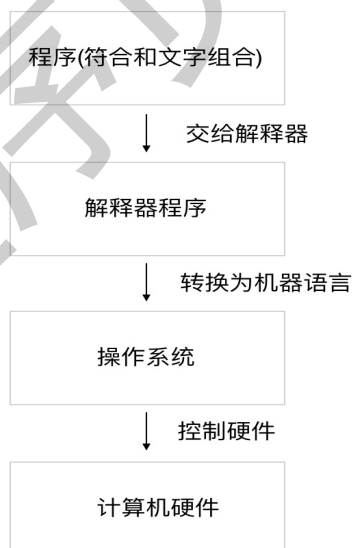
- 形式是符号和文字的组合.
- 目的是为了控制计算机硬件.

Python语言就是一种编程语言, 由符号和文字组成的, 使用Python语言的目的就是为了控制计算机硬件进行工作.

1.2.2 解释器

编程语言是文字和符号的组合, CPU只能认识机器指令, 机器指令的表现方式就是0和1的组合. 这显然很矛盾?

此时, 需要一个中间角色负责将文字和符号转换为机器指令, 这个负责转换的角色叫做解释器. 解释器本质上就是一个运行在操作系统上的应用程序.



Python语言如果想要被CPU读懂, 也需要一个中间的翻译程序.

1.2.3 Python 语言解释器

Python语言是解释型程序, 也就是说Python语言是读一行解释执行一行的方式进行工作的. 解释器是一个程序, 那么可以使用其他的程序来编写这个解释器.

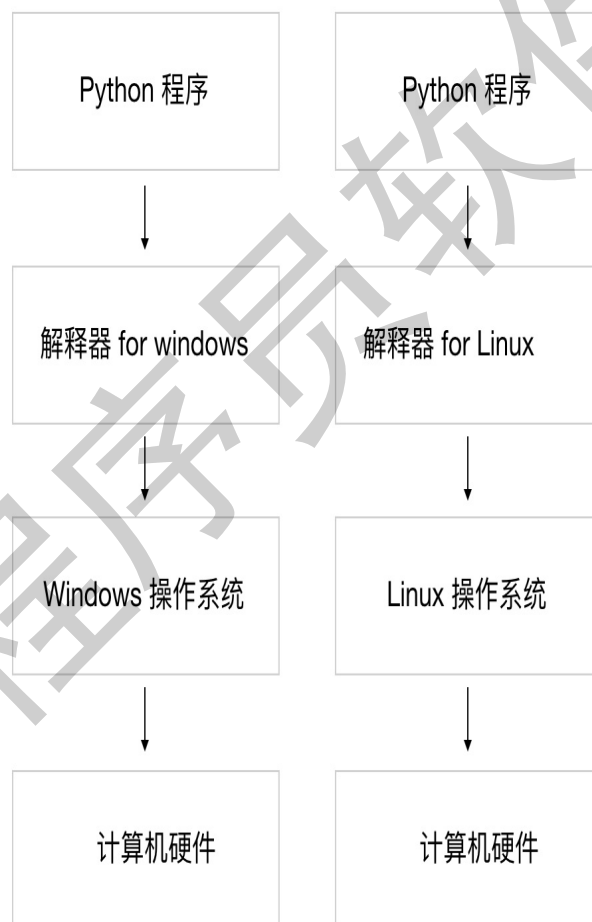
- 使用C语言编写的CPython解释器(官方版本).
- Java语言编写的Jython解释器.
- C#语言编写的IronPython解释器.
- Python语言编写的PyPy解释器.

1.2.4 小结

1. Python语言是符合和文字的组合, 目的是用来向计算机发送指令, 控制计算机工作.
2. Python解释器是运行在操作系统之上的一个特殊功能的应用程序.
3. 解释器负责将编程语言的符合和文字转换为计算机能够识别的计算机语言.
4. Python解释器程序可以由不同的语言的编写, 官方解释器使用C语言编写, 称之为CPython解释器.

1.2.4 思考

我们在Windows上编写的Python程序是否可以在Linux、Mac或者其他种类的操作系统上运行呢? 如果希望在其他操作系统上运行, 我们需要做哪些工作呢?



答案图示

1.3 Python 语言介绍

学习目标

1. 能够说出 Python 语言应用领域

1.3.1 Python 作者简介

Python 的作者, Guido von Rossum (吉多·范·罗苏姆, 中国Python程序员都叫他 龟叔), 荷兰人. 1982年, 龟叔从阿姆斯特丹大学获得了数学和计算机硕士学位. 然而, 尽管他算得上是一位数学家, 但他更加享受计算机带来的乐趣. 用他的话说, 虽然拥有数学和计算机双料资质, 他总趋向于做计算机相关的工作, 并热衷于做任何和编程相关的事情.



1.3.2 Python 语言的发展历史

- 80 年代个人电脑浪潮, 电脑配置很低, 所以大家都使用类似于 C 语言这样的程序语言, 但使用 C 语言使得程序员必须像计算机一样思考, 写出符合机器口味的程序. 不利于我们使用人思考问题的方式解决问题, 对于一些复杂问题, 会使得编码比较复杂.
- 吉多希望编码简单, 功能又强大. 于是在 1991 年, 诞生了第一个 C 语言实现的 Python 解释器.
- 计算机硬件越来越强大, Python 又容易使用, 所以许多人开始转向 Python. 这些来自不同领域的开发者, 将不同领域的优点带给了 Python.
- 从 Python 2.0 开始, Python 转为完全开源的开发方式, Python 也获得了更加高速的发展.

1.3.3 Python 语言的版本

Python 目前有两个版本, Python2 和 Python3, 最新版分别为 2.7.15 和 3.6.5, 其中 Python2 截止到 2020 年停止更新.

1.3.4 Python 语言的优缺点

- 易学. Python 有极其简单的语法, 学习极其容易上手.
- 开源.
- 可移植性. Python 已经被移植在许多平台上. 这些平台包括 Linux、Windows、Mac OS、FreeBSD、Windows CE 甚至还有 Symbian、Android 平台.
- 丰富的库. Python 标准库确实很庞大. 除了标准库以外, 还有许多其他高质量的库.

May 2018	May 2017	Change	Programming Language	Ratings	Change
1	1		Java	16.380%	+1.74%
2	2		C	14.000%	+7.00%
3	3		C++	7.668%	+2.92%
4	4		Python	5.192%	+1.64%
5	5		C#	4.402%	+0.95%
6	6		Visual Basic .NET	4.124%	+0.73%
7	9	▲	PHP	3.321%	+0.63%
8	7	▼	JavaScript	2.923%	-0.15%
9	-	▲	SQL	1.987%	+1.99%
10	11	▲	Ruby	1.182%	-1.25%
11	14	▲	R	1.180%	-1.01%
12	18	▲	Delphi/Object Pascal	1.012%	-1.03%
13	8	▼	Assembly language	0.998%	-1.86%
14	16	▲	Go	0.970%	-1.11%
15	15		Objective-C	0.939%	-1.16%
16	17	▲	MATLAB	0.929%	-1.13%
17	12	▼	Visual Basic	0.915%	-1.43%
18	10	▼	Perl	0.909%	-1.69%
19	13	▼	Swift	0.907%	-1.37%
20	31	▲	Scala	0.900%	+0.18%

2018年5月全球编程语言排行榜

1.3.5 Python 语言应用领域

- Web应用开发.
- 网络爬虫.
- 桌面软件.
- 操作系统管理.
- ...

1.3.6 小结

1. Python 的作者叫吉多.
2. 第一个 Python 解释器诞生于 1991 年.
3. Python 目前存在两大版本, Python2 和 Python3, 主流版本为 Python3.

4. Python 简单、易学、开源、扩展性强、有丰富的库.
5. Python 可以用来做 Web 网站、网络爬虫、操作系统管理.

1.4 Python开发环境搭建

学习目标

1. 能够说出在不同系统上搭建 Python 开发环境的流程

Python 程序开发一般包含两部分, 编写 Python 程序和运行 Python 程序, 所以一个 Python 开发环境主要包含两部分:

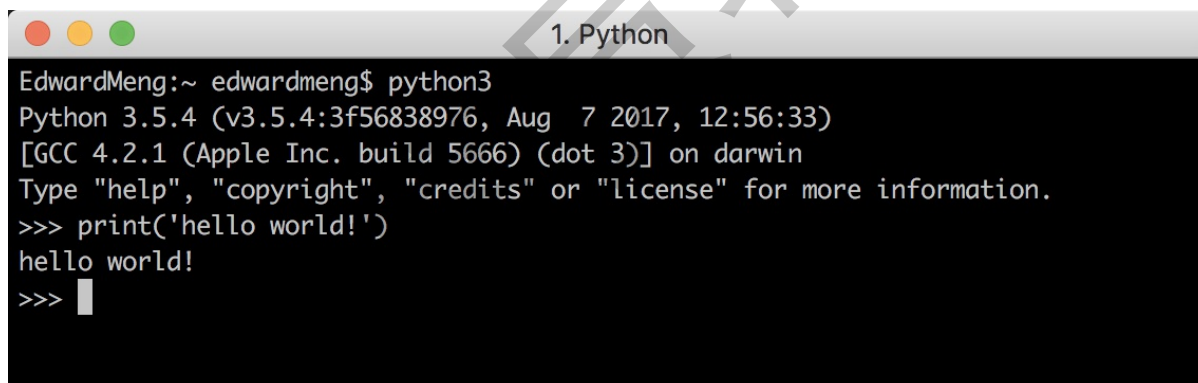
- 编辑Python代码的编辑器.
- 运行Python代码的解释器.

1.4.1 Python解释器 + 普通文本编辑器

普通文本编辑器我们可以使用 Windows 系统自带的 txt 文本编辑器、notepad++、sublime、editplus、ue 等等. 任何能够进行文本编辑的软件都可以作为 Python 程序开发的代码编辑器.

1.4.2 Python解释器 + 交互式终端

在安装 Python 解释器时安装了交互式终端. 我们可以通过在命令行窗口中, 输入 Python 或者 Python2 或者 Python3 进入不同 Python 版本的交互式终端.



```
1. Python
EdwardMeng:~ edwardmeng$ python3
Python 3.5.4 (v3.5.4:3f56838976, Aug 7 2017, 12:56:33)
[GCC 4.2.1 (Apple Inc. build 5666) (dot 3)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> print('hello world!')
hello world!
>>> 
```

交互式终端图示

1.4.3 Python解释器 + 集成开发环境(IDE)

集成开发环境(IDE, Integrated Development Environment)是用于提供程序开发环境的应用程序, 一般包括代码编辑器、编译器、调试器和图形用户界面等工具. 集成了代码编写功能、分析功能、编译功能、调试功能等一体化的开发软件服务套. 所有具备这一特性的软件或者软件套(组)都可以叫集成开发环境.

我们使用 PyCharm 这款集成开发环境(IDE). 该软件提供了 Windows、Linux、Mac三个版本, 可依据实际开发平台选择.

PyCharm的具体使用, 演示内容如下:

1. PyCharm 在 Windows、Linux、Mac 都有对应版本, 基础班可使用免费的社区版本.
2. 如何创建项目, 注意不同版本的 PyCharm 在创建项目时指定解释器. 希望学生的版本和讲师的版本统一.
3. 介绍 PyCharm 编辑器的各个项目结构区域、代码编辑区域、以及如何执行一个python 程序.

4. 介绍如何配置 PyCharm 的字体、颜色、风格, 以及如何配置项目解释器.

1.4.4 小结

1. Python 的开发环境包含两部分: Python 解释器和代码编辑器.
2. 编写 Python 程序可以使用普通文本编辑器、交互式终端环境、集成开发环境.
3. 我们在开发中主要使用集成开发环境 PyCharm.

1.4.5 思考

请问: 在 Linux 操作系统上编写 Python 程序, 我们需要做哪些准备工作? 安装哪些软件?

2. Python基础语法

学习目标:

1. 能够说出注释的作用以及使用注释的语法
2. 能够说出什么是标识符、什么是关键字
3. 能够说出变量的作用
4. 能够说出变量类型的作用
5. 能够说出不同类型的数据之间的运算规则
6. 能够说出 Python 中的运算符的种类
7. 能够说出 print 函数的作用
8. 能够说出 input 函数的作用
9. 能够说出为什么要进行类型转换
10. 能够说出在 Python 中如何定义变量
11. 能够说出 if 分支语句的作用
12. 能够说出 if 语句的语法格式
13. 能够说出 while 循环语句的作用
14. 能够说出 while 循环的语法格式
15. 能够说出 break 在循环语句中的作用
16. 能够说出 continue 在循环语句中的作用
17. 能够说出函数的作用
18. 能够说出函数定义的语法格式
19. 能够说出函数编写的单一职责原则是什么
20. 能够说出函数文档的作用

2.1 注释

学习目标:

1. 能够说出注释的作用
2. 能够说出注释的语法格式
3. 能够说出添加和取消注释的快捷方式

2.1.1 注释的作用

注释是编写程序时, 写程序的人给一个语句、程序段、函数等的解释或提示, 能提高程序代码的可读性.

注释就是对代码的解释和说明, 其目的是让人们能够更加轻松地了解代码.

如何在程序文件中添加的解释说明文字, 不会被解释器当做程序代码运行?

我们需要告诉解释器那一行是注释.

2.1.2 注释语法格式

1. 单行注释

```
# 这是注释内容
print('hello world!')
print('hello world!') # 在代码后也可以编写注释
```

2. 多行注释

```
"""
代码完成的功能是, 打印输出hello world
1. 首先调用 print 函数
2. 给 print 函数传入要打印输出的参数
"""
print('hello world!')
```

3. 快捷键注释

快捷键增加注释: `ctrl + /`

2.1.3 小结

1. 注释的作用是解释说明代码.
2. 注释分为单行注释和多行注释.
3. PyCharm 可以使用 `ctrl + /` 注释快捷键.
4. 注释的恰当用法是弥补我们无法用代码表达意图. 当代码修改时, 程序员要坚持维护自己的注释.

黑马程序员软件测试

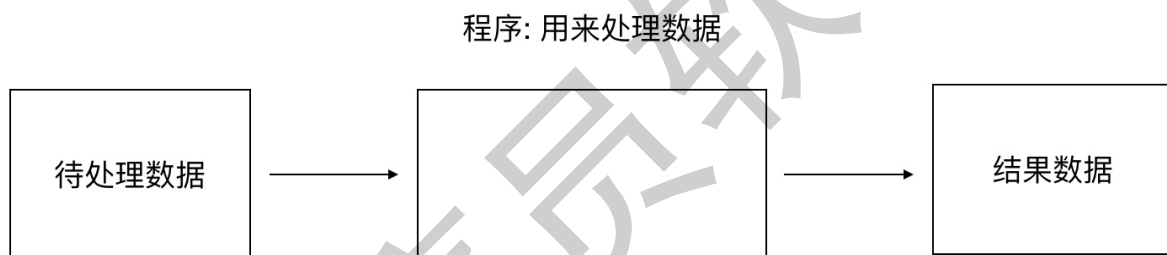
2.2 变量

学习目标:

1. 能够说出什么是标识符、什么是关键字
2. 能够说出变量的作用
3. 能够说出变量类型的作用
4. 能够说出不同类型的数据之间的运算规则
5. 能够说出 Python 中的运算符的种类
6. 能够说出 print 函数的作用
7. 能够说出 input 函数的作用
8. 能够说出为什么要进行类型转换
9. 能够说出在 Python 中如何定义变量

2.2.1 变量的作用

编写程序的目的就是将待处理的数据, 经过程序计算, 得出结果数据.



计算器举例:

1. 我们通过键盘输入的操作数.
2. 程序是否需要获得键盘输入的数据? 获得数据之后是否要将数据临时保存, 便于后续计算?
3. 输出运算结果.

变量是在程序运行过程中, 临时存储程序所需要计算的数据.

2.2.2 变量定义语法

那么在 Python 如何定义一个变量呢?

变量在程序中表现为一个唯一不重复的名字, 只需定义一个名字, 给这个名字变量赋值即可.

注意: 变量意味着存储的数据是可以变化的.

```
# 定义一个变量, 名字为 val, 这个变量临时存储的值为 100
val = 100
# 下面将这个变量的值, 更改为其他的值 200
val = 200
```

注意: 这里的等号(=), 叫做赋值运算符, 表示将=号后面的数据存储到名字为=号前面的名字变量里.

在取变量名时有什么需要注意的地方吗?

1. 标识符由字母、下划线和数字组成，且数字不能开头.
2. python中的标识符是区分大小写的.
3. 变量名一般用小写加下划线组成.
4. 不能和关键字和已有的名字冲突.

什么是关键字? 就是已经被 Python 占用的一些名字.

2.2.3 变量的类型

我们临时存储数据的目的是为了计算, 获取最终结果. 数据在运算过程中, 不同类型的数据之间的运算规则是不同的.

例如: 两个整数的运算规则和一个整数、一个是'abc'字符串运算规则是不一样的.

也就是说在数据运算过程中, 不同的数据类型约束了数据之间的运算规则.

下面我们先了解下, 在 Python 中的数据都有哪些类型?



- 数字类型: 整数和小数
- 字符串类型: 除了数字类型的数据, 我们处理更多的就是字符串类型数据, 例如 Word 中编辑文本, 其实就是在处理文本信息, 这些文本信息就是字符串类型.
在 Python 程序里, 无论任何字符写到两个单引号或者双引号内部, 我们称之为字符串. 例如: 'abcd'
- 布尔类型: 用于表示逻辑运算结果, 该变量只有两个值, True 或 False.

注意: Python中定义变量时不需要指定类型, Python 会根据变量的值来推导变量的类型. 我们可使用 `type()` 函数来查看变量的类型.

函数指的是一个单独的功能. `type` 函数的功能就是为了获得变量的类型.

2.2.3 不同类型之间的运算规则

我们一般不会将数字类型的数据和布尔类型的数据进行运算, 也不会将字符串类型的数据和布尔类型进行运算, 这样做毫无意义.

我们需要借由 Python 提供的算术运算符来完成变量之间的运算, Python 提供了两种: 算术运算符和复合运算符.

- 算术运算符

运算符	描述	实例
+	加	10 + 20 = 30
-	减	10 - 20 = -10
*	乘	10 * 20 = 200

/	除	10 / 20 = 0.5
//	取整除	返回除法的整数部分（商）9 // 2 输出结果 4
%	取余数	返回除法的余数 9 % 2 = 1
**	幂	又称次方、乘方，2 ** 3 = 8

• 复合赋值运算符

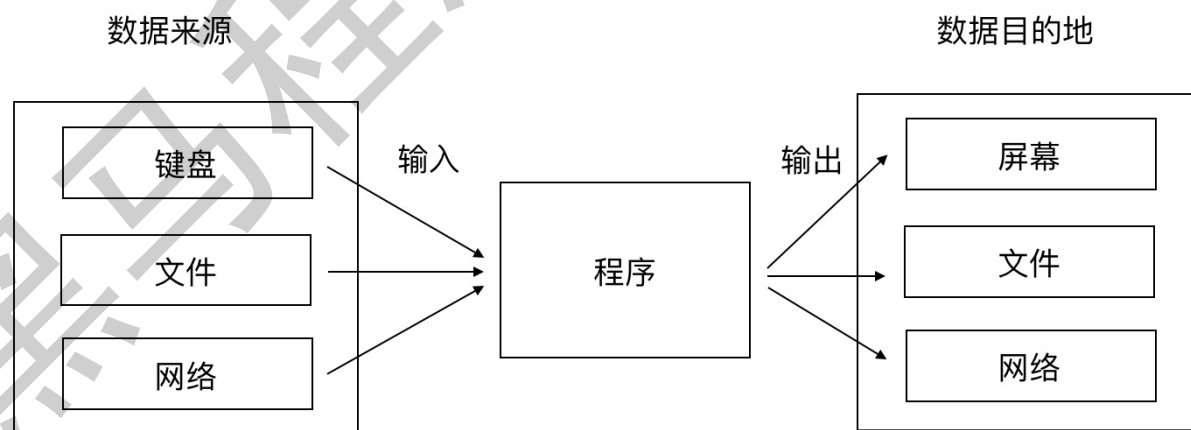
运算符	描述	实例
=	简单的赋值运算符	<code>c = a + b</code> 将 <code>a + b</code> 的运算结果赋值为 <code>c</code>
+=	加法赋值运算符	<code>c += a</code> 等效于 <code>c = c + a</code>
-=	减法赋值运算符	<code>c -= a</code> 等效于 <code>c = c - a</code>
*=	乘法赋值运算符	<code>c = a</code> 等效于 <code>c = c * a</code>
/=	除法赋值运算符	<code>c /= a</code> 等效于 <code>c = c / a</code>
//=	取整除赋值运算符	<code>c //= a</code> 等效于 <code>c = c // a</code>
%=	取模 (余数)赋值运算符	<code>c %= a</code> 等效于 <code>c = c % a</code>
**=	幂赋值运算符	<code>c = a</code> 等效于 <code>c = c ** a</code>

注意:

1. 数字和数字之间可以进行所有的运算
2. 数字和字符串之间只能进行乘法运算.
3. 字符串和字符串之间可以进行加法运算.

2.2.4 变量的输入和输出

输入和输出, 都是相对于程序而言的. 输入和输出, 简称 I/O(input/output)



1. 从键盘读取数据到程序中, 并且从程序中将数据显示到屏幕, 叫做标准输入和输出.
2. 从文件读取数据到程序中, 并且从程序中将数据存储到文件, 叫做文件输入和输出.
3. 从网络读取数据到程序中, 并且从程序中将数据发送到网络, 叫做网络输入和输出(就业班学习).

我们本小节学习标准输入和输出. 我们知道数据的接收和发送需要依赖于计算机操作系统来控制硬件设备来完成, 内部实现机制很复杂, 但 **Python** 将这些复杂的步骤封装起来, 给了我们一种极其简单的实现方式. 通过调用 **print** 函数和 **input** 函数来完成.

函数, 可以理解为封装了某一个功能, 我们不必关心功能如何实现, 只需要怎么使用即可.

2.2.4.1 print 函数使用

print 用于向屏幕输出数据. 分为普通输出和格式化输出.

1. 普通输出变量

```
# 定义一个整数类型变量
my_number = 100
# 定义一个字符串类型变量
my_string = 'hello itcast'

# 输出两个变量
print(my_number)
print(my_string)
```

注意: 变量名不能加引号

2. 格式化输出变量

格式化输出就是让数据按照一定的格式输出, 例如: 我的名字是xxx.

进行格式化输出的流程;

1. 先定义输出格式.
2. 在格式中填充数据.

案例: 已知有数据: **name** = '司马二狗', **age** = 30, **salary** = 19988.78, 请按照 "我的名字是xxx, 我的年龄是xxx, 我的工资是xxx" 的格式将变量输出.

定义输出格式, 需要用到格式化占位符. 用来暂时替代不确定的值.

% 被称为 格式化操作符, 专门用于处理字符串中的格式

- 包含 **%** 的字符串, 被称为 格式化字符串
- **%** 和不同的 字符 连用, 不同类型的数据 需要使用 不同的格式化字符

常用格式化字符	含义
%s	字符串
%d	有符号十进制整数, %06d 表示输出的整数显示位数, 不足的地方使用 0 补全
%f	浮点数, %.2f 表示小数点后只显示两位
%%	输出 %

```
# 定义名字、年龄、工资变量
name = '司马二狗'
age = 30
salary = 19988.78
# 格式化字符串
format_string = '我的名字是%s, 我的年龄是%d, 我的工资是%.2f' % (name, age, salary)
# 输出格式化后的字符串
```

```
print(format_string)
```

3. 格式化输出练习

1. 定义字符串变量 `name`，输出 我的名字叫 小明，请多多关照！
2. 定义整数变量 `student_no`，输出 我的学号是 **000001**
3. 定义小数 `price`、`weight`、`money`，输出 苹果单价 **9.00** 元 / 斤，购买了 **5.00** 斤，需要支付 **45.00** 元

```
print("我的名字叫 %s，请多多关照！" % name)
print("我的学号是 %06d" % student_no)
print("苹果单价 %.02f 元 / 斤，购买 %.02f 斤，需要支付 %.02f 元" % (price, weight, money))
```

2.2.4.2 input 函数使用

`input` 函数主要用于从键盘获取数据. 但是需要注意的是, 无论我们输入的数据是小数、整数、还是字符串, 该函数统统当做字符串类型来获取.

```
# 获得键盘输入
your_name = input('请输入您的名字:')
# 输出内容
print(your_name)
```

`input` 函数获得键盘输入内容, 并将内容存储在 `your_name` 变量中.

课堂练习-个人名片

- 在控制台依次提示用户输入：姓名、公司、职位、电话、邮箱
- 按照以下格式输出：

```
*****
公司名称

姓名（职位）

电话：电话
邮箱：邮箱
*****
```

实现代码如下：

```
"""
在控制台依次提示用户输入：姓名、公司、职位、电话、电子邮箱
"""
name = input("请输入姓名：")
company = input("请输入公司：")
title = input("请输入职位：")
phone = input("请输入电话：")
email = input("请输入邮箱：")

print("*" * 50)
print(company)
print()
print("%s (%s)" % (name, title))
print()
print("电话： %s" % phone)
print("邮箱： %s" % email)
```

```
print("*" * 50)
```

2.2.5 变量的类型转换

什么叫做变量类型转换？

变量类型转换就是将变量的类型由一种类型转换为另外一种类型，例如将变量从数字类型转换为字符串类型。

为什么需要类型转换呢？

由于各种原因，我们在拿到数据之后，数据的类型和我们的预期不相符，导致我们无法进行相关的计算(数据类型决定了数据之间的运算规则)。此时我们需要先将数据的类型转换为我们预期的类型，再进行计算。

我们下面通过一个案例，来理解类型转换的作用。

我们现在完成一个计算器案例，要求用户输入左操作数和右操作数，并对两个数进行加法计算，输出计算结果。

```
# 输入左操作数
left_number = input('请输入一个数字:')
# 输入右操作数
right_number = input('请输入一个数字:')
# 对两个数进行加法计算
result = left_number + right_number
# 输出计算结果
print('计算结果是:', result)
```

运行结果可能不是我们的预期，因为 `input` 接收的任何数据都当做了 `str` 类型来处理。此时如果想要进行数值运算，就必须将字符串转换为数字类型，再进行计算。因为类型不同，运算规则不同。

我们可以使用以下函数完成变量类型的转换：

1. `int(val)`，将变量 `val` 转换为 `int` 类型。
2. `float(val)`，将变量 `val` 转换为 `float` 类型。
3. `str(val)`，将变量 `val` 转换为 `str` 类型。

代码修改如下：

```
# 输入左操作数
left_number = input('请输入一个数字:')
# 输入右操作数
right_number = input('请输入一个数字:')
# 将字符串类型变量转换为数字类型
left_number_int = int(left_number)
right_number_int = int(right_number)
# 对两个数进行加法计算
result = left_number_int + right_number_int
# 输出计算结果
print('计算结果是:', result)
```

2.2.6 小结

1. 变量是在程序运行过程中临时保存所需要的数据。
2. 变量的名字规则. 2.1 标识符由字母、下划线和数字组成，且数字不能开头. 2.2 python中的标识符是区分大小写的. 2.3 变量名一般用小写加下划线组成. 2.4 不能和关键字和已有的名字冲突。
3. 变量的类型决定了变量之间的运算规则. 3.1 字符串和字符串只能进行加法运算，拼接字符串. 3.2 数字和数字之间可以进行所有的数学运算. 3.3 数字和字符串之间只能进行乘法运算，复制指定次数字符串。

4. Python常见的变量类型有: 数字类型、字符串类型、布尔类型.
5. Python中的输入和输出分为: 标准输入和输出、文件输入和输出、网络输入和输出.
6. `print` 是标准输出函数, 用于向屏幕打印数据.
7. `input` 是标准输入函数, 用于获得键盘输入的数据.
8. 在某些情况下, 我们拿到的数据并不是我们预期类型, 此时就需要我们将该数据从一种类型转换成我们预期的类型, 以便于完成计算. 需要注意的是, 进行类型转换的前提是该类型的数据能够转换为预期类型.

2.3.7 思考

1. "5" 和 5 是等价的吗?

2.3 分支语句

学习目标:

1. 能够说出 if 分支语句的作用
2. 能够说出 if 语句的语法格式
3. 能够说出什么是 BUG

2.3.1 BUG

BUG 原意为臭虫, 在计算机领域, 指 导致程序 不能正常执行, 或者 执行结果不是预期的 错误. BUG是程序员在开发时非常常见的, 初学者常见错误的原因包括:

1. 手误.
2. 对技术点理解不足.
3. 业务思路不熟练.

在学习语言时, 不仅要学会语法, 还要学会如何认识 and 解决错误的方法. 每一个程序员都必备的能力:

1. 编码能力.
2. 解决错误能力.

2.3.2 IF分支语句

先看一个系统登录的案例:

```
# 请输入您的用户名
username = input('请输入您的用户名:')
# 请输入您的密码
password = input('请输入您的密码:')
# 打印欢迎信息
print('欢迎 %s 登录系统!' % username)
```

以上的程序在执行过程中, 无论你输入的是什么, 都显示出来欢迎登录系统的信息提示. 实际上, "欢迎登录系统" 这个提示信息是否要打印出来, 取决于用户名和密码是否正确, 也就是说代码要有选择性的去执行, 而不是最开始无论你写了多少行代码, 都会执行.

如何在程序中, 实现这种有选择的执行某些特定代码呢?

使用 if 分支语句.

2.3.2.1 if 分支语法

1. 单个分支语法格式

```
if 条件一:
    执行一行或多行特定代码
```

如果条件成立则执行 if 下面的代码, 不成立则不会执行.

案例代码:

```
a = 10
if a > 5:
    print('a > 5')
```

2. 两个分支语法格式

```
if 条件一:
    执行一行或多行特定代码
elif 条件二:
    执行一行或多行特定代码
```

如果 if 条件成立, 则执行 if 下面的代码 如果 elif 条件成立, 则执行 elif 下面的代码 如果 if elif 条件同时成立, 则 执行第一个满足条件的分支

案例代码:

```
name = 'Obama'

if name == 'Obama':
    print('我是 Obama!')
elif name == 'Trump':
    print('我是 Trump!')
```

```
if 条件一:
    执行一行或多行特定代码
else:
    执行一行或多行特定代码
```

如果 if 条件成立, 则执行 if 下面的代码 如果 if 条件不成立, 则执行 else 下面的代码 if 和 else 代码必定会执行其中一个

案例代码:

```
name = 'Obama'

if name == 'Obama':
    print('我是 Obama!')
else:
    print('我是其他人!')
```

注意: else 后面不需要写条件.

3. 多个分支语法格式

```
if 条件一:
    执行一行或多行特定代码
elif 条件二:
    执行一行或多行特定代码
elif 条件三:
    执行一行或多行特定代码
elif 条件四:
    执行一行或多行特定代码
```

如果 if 条件成立, 则执行 if 下面的代码 如果 elif 条件成立, 则执行 elif 下面的代码 如果 if elif 条件同时成立, 则 执行第一个满足条件的分支

案例代码:

```
day = input('请输入1-7的数字:')

if day == '1':
    print('今天是星期一')
elif day == '2':
    print('今天是星期二')
elif day == '3':
    print('今天是星期三')
elif day == '4':
    print('今天是星期四')
elif day == '5':
    print('今天是星期五')
elif day == '6':
    print('今天是星期六')
elif day == '7':
    print('今天是星期日')
```

```
if 条件一:
    执行一行或多行特定代码
elif 条件二:
    执行一行或多行特定代码
elif 条件三:
    执行一行或多行特定代码
elif 条件四:
    执行一行或多行特定代码
else:
    执行一行或多行特定代码
```

如果 if 条件成立, 则执行 if 下面的代码 如果 elif 条件成立, 则执行 elif 下面的代码 如果 if elif 都不满足条件, 则执行 else 下面的代码

案例代码:

```
if day == '1':
    print('今天是星期一')
elif day == '2':
    print('今天是星期二')
elif day == '3':
    print('今天是星期三')
elif day == '4':
    print('今天是星期四')
elif day == '5':
    print('今天是星期五')
elif day == '6':
    print('今天是星期六')
elif day == '7':
    print('今天是星期日')
else:
    print('无法确定星期几')
```

2.3.2.2 if 条件成立

分支语句的条件如何表示呢?

成立则表示结果为真(True), 不成立则表示结果为假(False).

分支条件判断的场景一般为: a大于b、a小于b、a等于b、a大于等于b、a小于等于b、a不等于b等等.

运算符	描述
==	检查两个操作数的值是否 相等，如果是，则条件成立，返回 True
!=	检查两个操作数的值是否 不相等，如果是，则条件成立，返回 True
>	检查左操作数的值是否 大于 右操作数的值，如果是，则条件成立，返回 True
<	检查左操作数的值是否 小于 右操作数的值，如果是，则条件成立，返回 True
>=	检查左操作数的值是否 大于或等于 右操作数的值，如果是，则条件成立，返回 True
<=	检查左操作数的值是否 小于或等于 右操作数的值，如果是，则条件成立，返回 True

登录案例的代码可修改为:

```
# 请输入您的用户名
username = input('请输入您的用户名:')
# 请输入您的密码
password = input('请输入您的密码:')

# 判断密码是否正确
if username == 'admin':
    # 再判断密码是否合法
    if password == 'admin':
        # 打印欢迎信息
        print('欢迎 %s 登录系统!' % username)
    else:
        print('用户名或者密码不正确!')
else:
    print('用户名或者密码不正确')
```

注意: 编写if语句代码时, 注意缩进

2. 多个条件之间的关系

上面代码在 if 语句中又嵌套了一个 if 语句, 能否有更简单的写法呢?

我们可以将多个条件并列写在 if 后面, 此时需要表示多个条件之间的关系, 需要逻辑运算符.

运算符	逻辑表达式	描述
and	x and y	只有 x 和 y 的值都为 True, 才会返回 True 否则只要 x 或者 y 有一个值为 False, 就返回 False
or	x or y	只要 x 或者 y 有一个值为 True, 就返回 True 只有 x 和 y 的值都为 False, 才会返回 False
not	not x	如果 x 为 True, 返回 False 如果 x 为 False, 返回 True

```
a = 10
b = 20
c = 30
d = 40

# and 两个条件都为真, 则结果为真
result = a > b and c < d
print('result:', result)

# or 有一个条件为真, 则结果为真
result = a > b or c < d
print('result:', result)
```

```
# not 如果条件为真，则结果就为假
result = not (a > b)
print('result:', result)
```

我们继续修改上面代码为:

```
# 请输入您的用户名
username = input('请输入您的用户名:')
# 请输入您的密码
password = input('请输入您的密码:')

# 判断密码是否正确
if username == 'admin' and password == 'admin':
    # 打印欢迎信息
    print('欢迎 %s 登录系统!' % username)
else:
    print('用户名或者密码不正确')
```

2.3.3 练习-猜拳游戏

```
import random

user_quan = int(input('请出拳 石头(0)、剪刀(1)、布(2):'))

computer_quan = random.randint(0, 2)
if (user_quan == 0 and computer_quan == 1) or \
    (user_quan == 1 and computer_quan == 2) or \
    (user_quan == 2 and computer_quan == 0):
    print('您赢了!')
elif user_quan == computer_quan:
    print('平局!')
else:
    print('您输了!')
```

2.3.4 小结

1. if 语句的作用可以实现选择执行某些特定代码。
2. if 语句的条件结果为真, 则会执行对应分支下的代码。
3. if 条件的运算符包含比较关系运算符、逻辑关系运算符。
 - i. 比较关系运算符用于构建单个条件。
 - ii. 逻辑关系运算符可用于表示多个条件之间的关系. 5.1 and: 多个条件都为真, 则整个条件结果为真. 5.2 or: 多个条件中有一个条件为真, 则整个条件结果为真. 5.3 not: 对条件取反, 如果条件为真, 则结果为假, 反之则反。
4. BUG 在程序中不可避免, 要学会解决错误. 多积累。

2.4 循环语句

学习目标:

1. 能够说出 **while** 循环语句的作用
2. 能够说出 **while** 循环的语法格式
3. 能够说出 **break** 在循环语句中的作用
4. 能够说出 **continue** 在循环语句中的作用

2.4.1 while 循环

我们的猜拳游戏只能玩一次, 然后就需要重新启动程序. 我们在玩游戏时, 并不会每次玩都需要重新启动程序.

从语法角度, 某些代码需要重复去执行. 如何解决部分代码重复执行的问题?

Python 提供了 **while** 循环语法用于支持特定代码重复执行.

1. while 循环的语法格式

```
while 条件:
    重复执行的一行或多行代码
```

Python 每次会判断 **while** 关键字后面的条件是否为真, 如果为真, 则执行 **while** 下面的一行或多行代码, 直到不满足条件, 循环执行结束.

注意: 如果条件永远满足, 则意味着循环永远会被执行, 叫做死循环, 这是无意义的.

2. while 循环课堂练习

1. 计算1-100之间的累加和.

```
i = 1
sum = 0
while i <= 100:
    sum = sum + i
    i += 1

print("1~100的累积和为:%d" % sum)
```

1. 计算1-100之间所有的偶数累加和.

```
i = 1
sum = 0
while i <= 100:
    if i % 2 == 0:
        sum = sum + i
    i += 1

print("1~100的累积和为:%d" % sum)
```

2.4.2 break 和 continue

我们下面实现一个需求, 用户输入名字, 并显示名字, 当输入 **stop** 时, 停止输入.

```
name = ''
while name != 'stop':
    name = input('请输入一个名字(stop停止输入):')
    if name != 'stop':
        print(name)
```

我们可以在循环体内部使用 **IF** 语句配合 **break** 关键字来实现. 那么 **break** 是什么意思? 当循环体执行到 **break** 语句时就会马上退出循环.

```
while True:

    name = input('请输入一个名字(stop停止输入):')
    if name == 'stop':
        # 退出循环
        break
    else:
        print(name)
```

这么写的好处是什么呢?

如果 **while** 条件比较简单的话, 我们可以将循环退出条件写到 **while** 后面, 但有时一个循环退出的条件比较复杂, 也就是有多个条件, 写到 **while** 后面会增加阅读理解难度. 我们可以将条件写到循环内部, 会更加容易理解.

现在我的需求出来了, 我在做累加的时候, 希望碰到偶数的时候不累加. 那么该如何实现呢?

解决这个问题, 我们只需要让循环体的变量 **i** 等于 偶数 时, 跳过那一次循环, 并不退出循环, 就可以实现我们想要的请求了.

```
i = 1
sum = 0

while i < 100:

    if i % 2 == 0:
        # 一定要加这一句, 否则会进入死循环
        i += 1
        # 如果 i 为偶数则跳过循环
        continue

    sum = sum + i
    i += 1

print("1~100的累积和为:%d" % sum)
```

2.4.3 小结

1. **while** 循环用于特定代码重复执行.
2. **break** 语句用于退出循环.
3. **continue** 语句可以终止当前次循环.
4. **while** 循环要避免死循环出现.

黑马程序员软件测试

2.5 函数

学习目标:

1. 能够说出函数的作用
2. 能够说出函数定义的语法格式
3. 能够说出函数编写的单一职责原则是什么
4. 能够说出函数文档的作用

2.5.1 函数的作用

请问: 我家里种地需要锄头, 我是每次去锄地时重新做一把锄头, 还是提前做好一把锄头, 需要时直接拿来用?

很显然, 每次重新做都是重复劳动, 浪费时间. 所以我们选择提前做好一把锄头, 每次直接拿来用, 这里面就是一种复用的思想. 程序设计思想来源于生活, 所以在程序中也有复用的思想, 只不过复用的是代码.

我们的代码是完成某些固定任务, 如果需要频繁解决这个问题, 那么解决这个问题的代码就可以提前写好, 需要解决该问题时, 直接拿来用.

例如: 我在代码文件的多个地方都需要计算1-100累加和这个功能, 那么怎么解决这个问题呢?

最简单的方式就是将咱们之前的代码拷贝到需要这个功能的地方. 这么写也很明显带来一个问题, 如果这个累加和的功能实现改变了, 意味着所有地方都需要修改, 增加了代码的维护量.

怎么解决呢? 通过将这个功能封装成一个单独的功能代码块, 只要需要该功能, 直接使用该功能代码块, 这个功能代码块我们也叫做函数.

简言之, 函数的作用就是将常用的代码封装起来, 便于以后使用.

2.5.2 函数语法格式

在 Python 中, 函数使用 `def` 关键字来定义, 包含函数的名字(功能的名字), 函数的实现(实现功能的代码).

函数的行为分为: 函数定义和函数调用.

1. 函数定义是实现函数功能的过程.
2. 函数调用是使用功能.

注意: 函数不调用是不会自动执行的.

```
def 函数名():  
    一行或者多行代码
```

那么1-100这个功能我们就可以写成这样的一个函数(功能代码块).

```
def my_sum():  
    i = 1  
    s = 0  
    while i <= 100:  
        s = s + i  
        i += 1  
  
    print("1~100的累积和为:%d" % s)
```

```
# 函数调用
my_sum()
```

当使用该功能时直接调用该函数即可。

2.5.2.1 函数的参数

再思考一个问题: 我们发现这个函数只能计算1-100的累加和, 我们能否让函数的功能支持从指定开始到结束的数字累加和?

我们可以将我们要开始和结束数字传递给函数, 让函数按照我们传递的参数来计算。

```
def my_sum(start, end):
    my_start = start
    my_end = end
    my_sum = 0
    while my_start <= my_end:
        my_sum = my_sum + my_start
        my_start += 1

    print("%d~%d的累积和为:%d" % (start, end, my_sum))

# 函数调用
my_sum(2, 50)
```

函数参数的作用就是让函数依据我们给定的值来进行运算, 这样可以增强函数的通用性. 函数可以有多个参数。

例如: 我们想要编写一个具有加法功能函数, 很显然需要两个参数。

```
def my_add(num1, num2):
    result = num1 + num2
    print('num1 + num2 =', result)

my_add(10, 20)
```

我们在调用函数时传递的真实数据叫做实参, 函数参数叫做形参, 形参只是代表真实传递的数值。

多个函数参数再传递时是从左向右传递的。当然, 在 Python 中我们也可以指定某个值给那个形参。

```
def my_add(num1, num2):
    result = num1 + num2
    print('num1 + num2 =', result)

my_add(num1=10, num2=20)
my_add(num2=20, num1=10)
```

1. 按照从左向右的顺序传递叫做位置参数。
2. 按照形参名字传递叫做关键字参数。

能否在调用函数时既传递位置参数, 又传递关键字参数呢?

```
def my_add(num1, num2, num3, num4):
    result = num1 + num2 + num3 + num4
    return result

my_add(100, 200, 300, num2=10)
```

可以,只需要保证位置参数在关键字参数之前即可。

2.5.2.2 函数的返回值

请思考下面两个问题:

1. 现在我们的函数在计算完结果之后, 把结果放到哪里了?
2. 我的程序需要继续使用函数的结计算果来进行下一步计算, 该怎么办?

使用 `return` 语句将函数的运行结果返回给函数的调用者。

```
def my_add(num1, num2):  
    result = num1 + num2  
    return result  
  
# 使用一个变量保存函数执行的结果  
my_add_result = my_add(10, 20)  
# 使用结果进行下一步计算  
finish_result = my_add_result + 100  
# 输出最终结果  
print('最终结果:', finish_result)
```

`print`函数 和 `return` 的区别是什么?

`print` 只负责将内容输出到屏幕显示. 而 `return` 会将函数计算结果, 返回给函数的调用者。

比如: 函数类似于一个工厂, 我们将工厂生产所需要的材料通过参数的形式传递给工厂, 工厂使用我们传递的材料生产出产品。

`print` 相当于生产完产品后, 说了一句 "产品生产完毕", 但是并不会把产品给用户。

而 `return` 相当于生产完产品之后, 并将产品交到用户手里. 那么用户就需要用一个篮子来装产品(用变量来保存函数的返回值). 当然, 虽然工厂将产品给了用户, 用户也可以不要(忽略函数的返回值)。

关于 `return` 注意下以下几点。

1. 只要函数执行碰到 `return` 就会停止执行。
2. 函数中可以编写多个 `return`, 但有且只有一个 `return` 会执行。
3. `return` 后面可以跟上要返回的值, 也单独使用相当于 `return None`。
4. `break` 用在循环中, 用来终止循环执行. `return` 用在函数中, 用来终止函数执行。

2.5.2.3 局部变量和全局变量

1. 全局变量: 在函数外部定义的变量. 全局指的是该变量在当前 `python` 文件范围内是可见的. 全局变量可以被当前 `python` 文件内的所有函数直接使用。
2. 局部变量: 在函数内部定义的变量. 该变量只能在定义的函数内部使用。

```
# 定义全局变量  
g_val = 100  
  
# 在函数内部可以访问全局变量  
def my_function1():  
    print(g_val)
```

```
# 在函数内部定义局部变量 my_val
def my_function2():
    my_val = 100

# 尝试输出 my_function2 函数中定义的局部变量
def my_function3():
    print(my_val)

# 函数调用
my_function1()
my_function2()
my_function3()
```

如果局部变量和全局变量命名冲突, Python 解释器会怎么做?

```
total_value = 100

def my_function():
    total_value = 200
    print('total_value:', total_value)

my_function()
```

请问: 上面的代码输入结果是什么?

Python 解释器会在函数内部搜索变量 `total_value`, 如果找到了就直接使用, 如果找不到则到全局范围内搜索。

2.5.2.4 函数的缺省参数(默认参数)

默认参数指的是当函数调用中省略了实参时默认使用的值。

默认参数的语法与使用:

1. 在函数声明或定义时, 直接对参数赋值. 这就是设置形参的默认参数.
2. 在函数调用时, 省略部分或全部的参数. 这时可以用默认参数来代替.

```
def my_function(a, b=20, c=30):
    return a+b+c

my_function(10)
my_function(10, 100)
my_function(10, 100, 1000)
```

注意: 带有默认值的参数一定要位于参数列表的最后面。

2.5.3 函数文档及作用

函数也需要添加注释, 方便函数功能、参数以及返回值的含义能够被调用者知悉. 但普通的单行多行注释, 需要查看函数定义时才能看到, 有没有一种方式能够在调用时快捷查看函数相关信息?

DocString 是一个重要的工具, 由于它帮助你的程序文档更加简单易懂。

```

# 单行函数文档字符串
def my_function(param):
    """函数做了什么事, 返回什么结果."""

    return param + 10

# 多行函数文档字符串
def my_add(num1, num2):
    """计算两个整数的和.

    :param int num1: 加法运算的左操作数
    :param int num2: 加法运算的右操作数
    :return 返回两个操作数相加的结果
    """

    result = num1 + num2

    return result

```

我们可以通过 `ctrl + q` 快捷键可以查看函数信息, 也可以通过 `help()` 函数来查看函数信息.

2.5.4 单一职责原则

请问: 在工作中一个人负责的职责越多还是越好? 同理问, 编写一个函数, 函数能完成的功能越多还是越好?

单一职责原则说的是一个函数只负责一个事情. 这是因为, 如果一个函数承担的职责过多, 就等于把这些职责混合在一起, 一个职责的变化可能会影响其它职责的能力.

1. 简称 单一职责原则的英文名称是 **Single Responsibility Principle**, 简称 **RSP**.

2. 定义 就一个函数而言, 应该仅有一个引起它变化的原因, 简单的说, 一个函数中应该是一组相关性很高的的封装. 即一个类只负责一项职责, 而不应该同时负责多个职责.

3. 问题 比如 **C** 函数负责两个不同的职责 **D1** 和 **D2**. **D1** 功能需求发生变化时, 更改 **C** 函数, 有可能使原本正常运行的 **D2** 发生错误, 代码耦合性太高, 较复杂.

4. 解决 把功能独立出来, 让它们满足单一职责原则. 比如创建两个函数 **C1** 和 **C2**, **C1** 完成功能 **D1**, **C2** 完成功能 **D2**. 任何一个功能出现问题都不会造成另一个功能出问题.

2.5.5 小结

1. 函数是实现代码复用的一种技术, 可以减少代码冗余.
2. 函数定义不会执行代码, 函数调用会执行代码.
3. 函数使用 `def` 来定义, 函数调用时候使用 "函数名(参数...)".
4. 函数调用时, 如果位置参数和关键字参数并存, 位置参数必须在关键字参数前面.
5. 函数的参数叫做形参, 调用函数时传递的数值叫做实参.
6. 函数内部定义的变量叫做局部变量, 函数外部定义的变量叫做全局变量. 6.1 局部变量只能在函数内部使用, 函数外无法使用. 6.2 全局变量可以在当前 `python` 文件定义的所有函数中访问. 6.3 全局范围指的是整个 `Python` 文件范围.
7. 函数文档的作用解释说明函数, 并可以通过 `ctrl + q` 或者 `help()` 函数快速查阅.
8. 函数的编写要遵循的单一职责原则, 即一个函数只负责一个事情.
9. `return` 用于将函数的计算结果返回给函数的调用者, 使用时需要注意以下几点: 9.1 只要函数执行碰到 `return` 就会停止执行. 9.2 函数中可以编写多个 `return`, 但有且只有一个 `return` 会执行. 9.3 `return` 后面可以跟上要返回的值, 也单独使用相当于 `return None`. 9.4 `break` 用在循环中, 用来终止循环执行. `return` 用在函数中, 用来终止函

数执行.

2.5.6 思考

1. 一个函数是否可以没有参数, 如果可以, 为什么? 如果不可以, 为什么?
2. 一个函数是否可以没有返回值, 如果可以, 为什么? 如果不可以, 为什么?

3 容器

学习目标:

1. 能够说出容器类型有什么用
2. 能够说出常用 Python 容器的名字
3. 能够说出切片语法的用途
4. 能够说出容器中的索引指的是什么
5. 能够说出如何定义一个字符串
6. 能够说出字符串容器的特性
7. 能够说出至少5个字符串方法名字和作用
8. 能够使用切片语法获得指定索引区间的子串
9. 能够说出如何使用 `while` 和 `for` 循环来遍历字符串
10. 能够说出如何定义一个列表
11. 能够说出列表容器和字符串容器的区别
12. 能够说出至少5个列表方法名字和作用
13. 能够使用切片语法获得列表指定索引区间的元素
14. 能够说出如何使用 `while` 和 `for` 循环来遍历列表中的元素
15. 能够说出如何定义一个列表
16. 能够说出元组和列表的区别
17. 能够说出如何使用 `while` 和 `for` 循环来遍历元组中的元素
18. 能够说出元组支持哪些操作
19. 能够说出如何定义一个字典
20. 能够说出字典和列表的区别
21. 能够说出如何使用 `for` 循环来遍历列表中的键、值和键值对
22. 能够说出字典键和值的特点

为什么要学习容器类型? 容器类型有什么用?

答: 之前学习的变量类型都只能存储单一元素, 如果我要存储全班100个学生的成绩, 需要定义100个变量. 将所有的元素放在同一个容器中, 一个容器变量就可以存储多个元素. 减少了变量的定义.

为什么容器类型提供了多个类型?

答: 每种容器针对了不同的使用场景, 有的容器方便操作、有的容器查询效率比较高、有的元素可以保证元素唯一、有的元素可以保证数据只读等等. 学校就是一个容器, 学校有多种, 有学习动漫的、有学习IT编程的、有学习英语的等等, 虽然都是学校但是所做的事情不同.

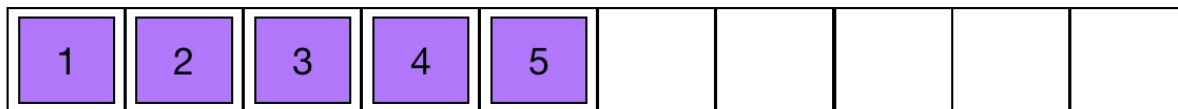
容器分类?

为了便于学习, 我们根据不同容器的特性, 将常用容器分为序列式容器和非序列式容器.

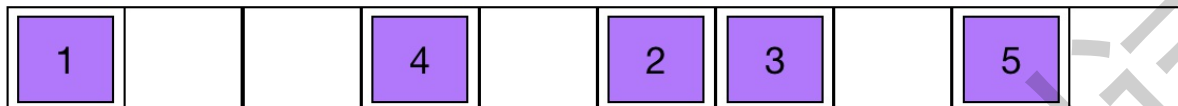
1. 序列式容器中的元素在存放时都是连续存放的, 也就是序列式容器中, 除了第一个元素的前面没有元素, 最后一个元素的后面没有元素, 其他所有的元素前后都有一个元素. 包括字符串、列表、元组.

2. 非序列式容器在存储元素时不是连续存放的, 容器中的任何一个元素前后都可能没有元素. 包括字典、集合.

序列式容器



非序列式容器



- 序列式容器支持根据索引(下标)访问元素, 而非序列式容器不支持索引(下标)的方式访问元素.
- 序列式容器支持切片操作, 而非序列式容器不支持切片操作.

什么是索引?

在序列式容器中, 会给每一个元素赋予一个编号, 该编号从 0 开始计算. 第一个元素的索引就为 0, 第二个元素的索引就为 1, 以此类推, 支持索引的容器可以使用 0 1 ... 来获得某个位置的元素.

什么是切片?

通过索引可以获取序列式容器中的某个元素, 切片语法主要用于获得一个指定索引区间的多个元素, 例如获取从索引值为 0 到索引值为 5 之间的所有元素.

如何学习容器?

容器用来存储多个元素, 针对元素的操作提供了一些操作方法, 比如添加一个元素、删除一个元素、修改一个元素、对容器中的元素排序等等.

学习容器类型就是在学习容器的特点、以及容器对元素的操作.

上面所说的 "方法", 就是我们所说所学的函数, 本质上 "方法"和"函数"指的是同一个东西, 只不过我们将某个类型专属的一些函数叫做方法.

3.1 字符串

学习目标:

1. 能够说出如何定义一个字符串
2. 能够说出字符串容器的特性
3. 能够说出如何对字符串中的子串进行替换
4. 能够说出如何对字符串中的字符串进行查找
5. 能够说出如何去除字符串两侧的空格
6. 能够说出如何判断字符串是否全部为字母
7. 能够说出字符串如何根据某个分隔符进行切分
8. 能够使用切片语法获得指定索引区间的子串
9. 能够说出如何使用 `while` 和 `for` 循环来遍历字符串

3.1.1 字符串语法格式

我们知道数据是有类型的, 现实生活中很多数据都仅仅是一个字符序列, 计算机如何表示字符序列的信息, 使用字符串类型.

如何定义字符串?

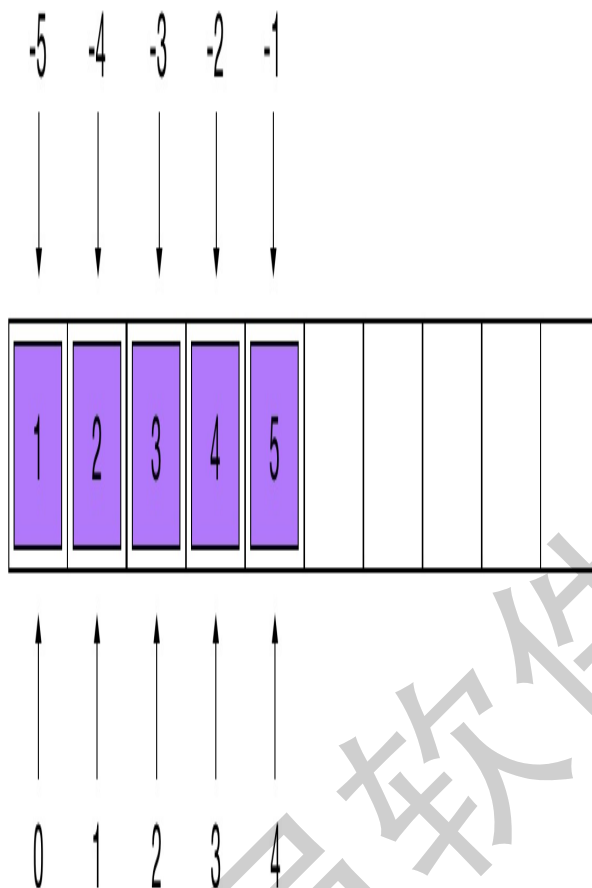
1. 字符串使用一对单引号来定义.
2. 字符串使用一对双引号来定义.
3. 字符串使用一对三引号来定义.

一般我们在定义字符串时候, 使用两个单引号或者两个双引号, 很少使用三引号.

3.1.2 字符串操作

3.1.2.1 字符串遍历

字符串属于序列式容器, 支持依据索引的操作.



我们可以使用 `while` 循环来访问字符串容器中的每一个字符元素。

注意: 序列式容器的索引都是以 **0** 开始的, 并不是从 **1** 开始。

```
my_string = '我叫做司马狗剩,我今年10岁了!'
i = 0
while i < len(my_string):
    print(my_string[i], end=' ')
    i += 1
```

Python 是一门简单易用的语言, 对于容器的遍历, 提供了另外一种简单方式, `for` 循环。

```
my_string = '我叫做司马狗剩,我今年10岁了!'
for ch in my_string:
    print(ch, end=' ')
```

3.1.2.2 字符串替换

我们现在已经存储了一首诗歌:

```
poetry = '远看泰山黑乎乎, 上头细来下头粗。 茹把泰山倒过来, 下头细来上头粗。'
```

诗歌中的茹正确写法应该是如, 我们需要用程序解决这个问题, 如何做?

1. 查找到错别字茹。
2. 将错别字替换成正确的如字。

我们可使用字符串的 `replace` 方法完成上面两步. 该方法默认会将字符串中所有指定字符或子串替换为新的字符串, 我们可以指定第三个参数, 替换多少次.

```
poetry = '远看泰山黑乎乎, 上头细来下头粗. 茹把泰山倒过来, 下头细来上头粗. 茹'
# 将所有的 '茹' 替换为 '如'
right_poetry = poetry.replace('茹', '如')
# 只替换第一次出现的 '茹'
right_poetry = poetry.replace('茹', '如', 1)
```

3.1.2.3 字符串查找和切片

现在有一邮箱地址如下:

```
user_email = 'simagousheng@itcast.cn'
```

我们希望从邮箱地址字符串中获取用户名和邮箱后缀名, 那么这个问题如何解决?

1. 由分析可知, @符号之前为用户名, @符号之后的内容为邮箱后缀名.
2. 首先获得 @ 符号的位置, 从开始位置截取到 @ 符号位置, 即可获得用户名.
3. 从 @ 符号位置开始截取到字符串最后, 即可获得邮箱后缀名.

如何获得 @ 符号的位置?

我们可以使用字符串提供的 `find` 方法, 该方法可返回查找字符串第一次出现的位置, 查找字符串不存在则会返回-1.

备注: `find` 方法默认从字符串开始位置(0位置)开始查找, 我们也可以指定从哪个位置范围开始查找, 设置 `find` 的第二个参数表示从哪个位置开始查找, 第三个参数, 表示查找结束位置.

```
poetry = '远看泰山黑乎乎, 上头细来下头粗. 茹把泰山倒过来, 下头细来上头粗.'
# 从 10 位置开始查找 '上'
position = poetry.find('上', 10, 100)
```

如何获得指定范围的字符串?

字符串属于序列式容器, 可以根据索引获得某一个字符, 也可以根据由两个索引标识的区间获得区间内的字符序列.

```
poetry = '远看泰山黑乎乎, 上头细来下头粗. 茹把泰山倒过来, 下头细来上头粗.'
# 从0位置开始到7位置之前, 不包含7位置字符
print(poetry[0: 7])
# 起始位置不写, 默认就是0
print(poetry[: 7])
# 从0位置开始到最后, 结束位置不写默认字符串最后一个位置的下一个位置.
print(poetry[9:])
# 步长, 每隔2个字符选取一个字符, 组成一个序列
print(poetry[0: 7: 2])
# 如果步长为负数, 那么起始位置参数和结束位置参数就会反过来.
print(poetry[6:: -1])
# 位置也可以使用负数
print(poetry[-3: -1])
print(poetry[-3:])
print(poetry[::-1])
```

下面我们看看如何解决这个问题?

```
user_email = 'simagousheng@itcast.cn'
# 查找 @ 位置
position = user_email.find('@')
```

```
# 根据 position 截取用户名和邮箱后缀
user_name = user_email[: position]
mail_suffix = user_email[position + 1:]
```

另外一种解决该问题的思路.

1. 先根据 @ 符号将邮箱分割成两部分.
2. 分别获取每一部分, 即可得到用户名和邮箱后缀.

```
user_email = 'simagousheng@itcast.cn'
# 判断 user_email 是否有多个 @
at_count = user_email.count('@')
if at_count > 1:
    print('邮箱地址不合法, 出现了多个@符号!')
else:
    # 根据 @ 将字符串截取为多个部分
    result = user_email.split('@')
    # 输出用户名和邮箱后缀
    print(result[0], result[1])
```

疑惑: 老师解决一个问题你给我们讲了好几个方法, 我应该用那个?

这个问题非常简单, 你如何思考这个问题, 根据你解决思路, 选择来使用即可. 这些方法无优劣之分.

3.1.2.4 字符串去除两侧空格、是否为字母

我们经常在各个网站进行会员注册, 一般注册的处理流程如下:

1. 获得用户输入的注册用户名.
2. 用户在输入用户名时, 可能在用户名两个不小心输入多个空格. 我们需要去除用户名两侧的空格.
3. 判断用户名是否全部为字母(用户名的组成由我们来规定, 这里我们规定必须是字母)
4. 处理完毕之后, 显示注册成功.

```
# 获得用户注册用户名
register_username = input('请输入您的用户名:')
# 去除用户名两侧的空格
register_username = register_username.strip()
# 判断字符串是否全部为字母
if register_username.isalpha():
    print('恭喜您:', register_username, '注册成功!')
else:
    print('注册失败!')
```

3.1.3 小结

1. 字符串一般使用两个双引号或两个单引号来定义.
2. 字符串容器特点: 元素不能修改, 并且只能由一系列字符组成.
3. 字符串是序列式容器, 支持下标索引和切片操作, 索引支持正数和负数.
4. 切片语法由开始索引、结束索引、步长组成, 语法格式如: `my_str[start: end: step]` 4.1 开始索引省略默认为0. 4.2 结束索引省略默认为最后一个元素的索引的下一个索引. 4.3 步长省略默认为 1. 4.4 步长为负数时, 开始索引就变成结束索引, 结束索引就变成开始索引. 4.5 切片的索引区间为左闭右开.
5. 字符串遍历可以使用 `while` 循环, 也可以使用 `for` 循环.
6. 字符串的 `find` 方法用于查找指定子串是否存在, 存在则返回出现的索引位置, 否则返回-1.
7. 字符串的 `replace` 方法用于替换字符串中的指定子串, 注意, 不会修改原字符串. 会返回一个替换后的新字符串.
8. 字符串的 `count` 方法返回指定子串出现的次数.

9. 字符串的 `split` 方法根据指定的分割字符串, 将原字符串分割成多个部分, 以列表形式返回.
10. 字符串的 `strip` 方法去除字符串两侧空格.
11. 字符串的 `isalpha` 方法判断字符串是否全部为字母组成.

3.2 列表

学习目标:

1. 能够说出如何定义一个列表
2. 能够说出列表容器和字符串容器的区别
3. 能够说出如何向列表中添加元素
4. 能够说出如何删除列表中的一个元素
5. 能够说出如何对列表中的元素进行排序
6. 能够说出如何对列表中元素进行查询
7. 能够说出如何判断列表中是否存在某个元素
8. 能够使用切片语法获得列表指定索引区间的元素
9. 能够说出如何使用 `while` 和 `for` 循环来遍历列表中的元素

3.2.1 列表语法格式

字符串容器中存放的元素只能是字符序列, 并且字符串容器中的元素不能修改, 如果需要存储的数据并非单一类型, 并且需要频繁修改, 如何解决?

我们可以使用列表容器类型, 列表中存储的元素可以是多种数据类型, 甚至存储的数据类型都不一样, 并且列表支持对元素的修改、删除等操作.

列表也是一个序列式容器, 同样支持索引和切片语法.

```
# 创建空列表
my_list = []
# 创建带有元素的列表
my_list = [10, 20, 30]

# 通过索引来访问列表中元素
print(my_list[0])
# 也可以通过索引来修改元素
my_list[0] = 100
# 通过切片语法获得区间元素
print(my_list[1:])

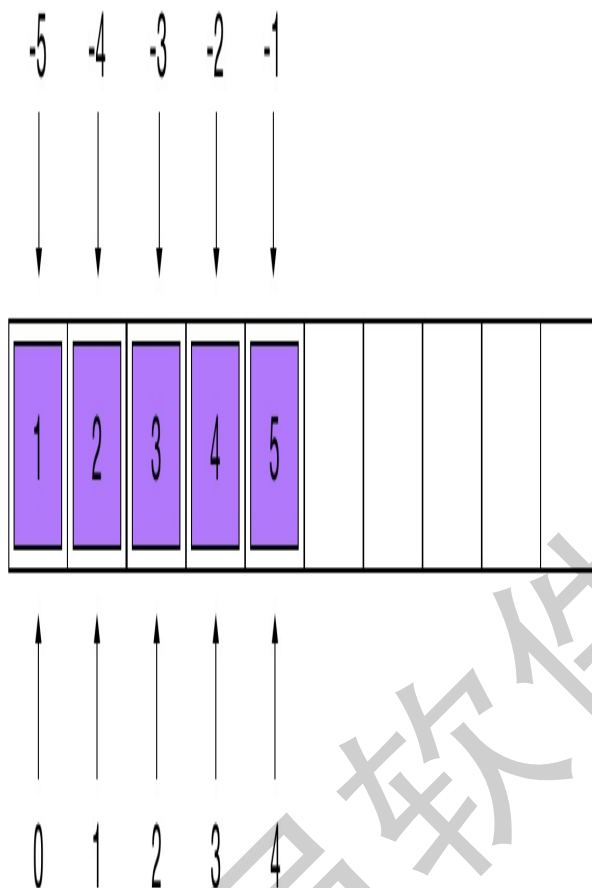
# 列表可存储不同类型的数据
my_list = ['John', 18, True]

# 列表中也存储列表
my_list = [[10, 20], [30, 40], [50, 60]]
```

注意: 列表中支持存储不同类型的数据, 如果有没有特殊需求, 建议存储相同类型数据. 这样可以对数据应用统一的操作.

3.2.2 列表操作

3.2.2.1 列表遍历



```
my_list = [1, 2, 3, 4, 5]
i = 0
while i < len(my_list):
    print(my_list[i], end=' ')
    i += 1
```

我们也可以使用 `for` 循环来简化列表的遍历操作.

```
my_list = [1, 2, 3, 4, 5]
for val in my_list:
    print(val, end=' ')
```

3.2.2.2 列表查找和修改

已知: 列表中包含 5 个元素, 分别为: 10、20、30、40、50. 需要将列表中 40 这个元素替换成 100, 如何实现?

1. 列表不存在类似字符串 `replace` 的方法.
2. 查询 40 这个元素在列表中的索引位置.
3. 根据索引位置修改元素为 100.

`index` 方法可以根据值查找, 查找到返回该值元素所在的位置, 查找失败会报错, 程序终止. 我们可以先使用 `count` 方法可以统计值出现的次数, 如果不为0, 再使用 `index` 方法.

```
my_list = [10, 20, 30, 40, 50]
# 要修改的值
old_value = 40
```



```

# 更新的新值
new_value = 100
# 判断要修改的值是否存在
if my_list.count(old_value):
    # 获得指定值的位置
    position = my_list.index(old_value)
    # 根据值来修改元素
    my_list[position] = new_value

print(my_list)

```

如果我们只是关心值是否存在, 而并不关心出现多少次, 可以使用 **in** 或者 **not in** 运算符.

1. **in** 可以判断元素是否存在, 存在返回 **True**, 不存在返回 **False**.
2. **not in** 可以判断元素是否存在, 不存在返回 **True**, 存在返回 **False**.

以上代码可修改为:

```

my_list = [10, 20, 30, 40, 50]
# 要修改的值
old_value = 40
# 更新的新值
new_value = 100
# 判断要修改的值是否存在
if old_value in my_list:
    # 获得指定值的位置
    position = my_list.index(old_value)
    # 根据值来修改元素
    my_list[position] = new_value

print(my_list)

```

3.2.2.3 列表的插入和删除元素

列表是一个容器, 我们可以向容器中添加和删除元素.

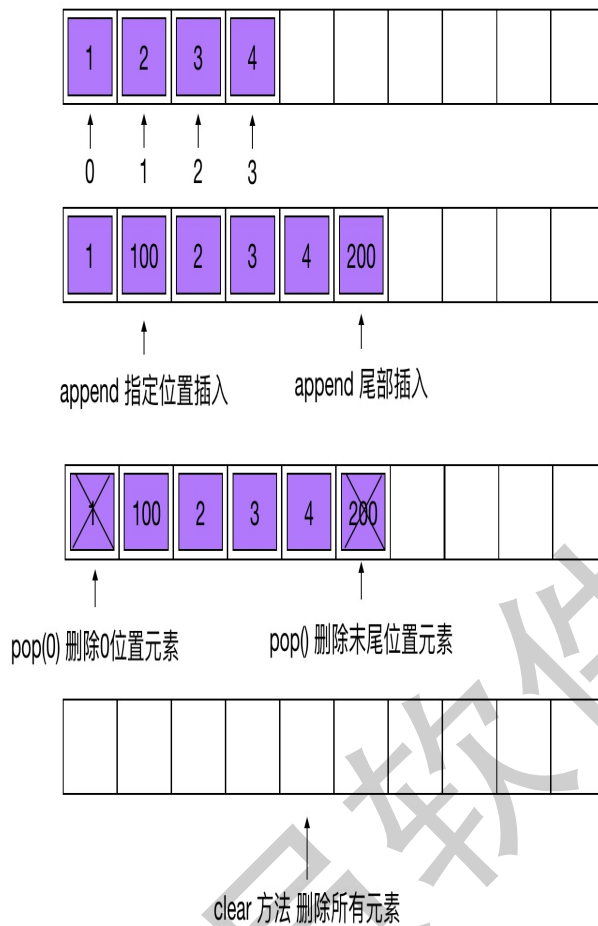
插入元素分为:

1. 插入单一元素. 1.1 尾插. 1.2 指定位置插入.
2. 插入一个列表(多个元素).

Python 提供了 **append** 方法, 用于向列表尾部添加元素, **insert** 方法用于向列表指定的索引位置插入元素, **extend** 方法用于将另外一个列表中的所有元素追加到当前列表的尾部.

删除分为两种:

1. 根据值删除, 使用 **remove** 方法. 该方法只能删除第一次出现的值.
2. 根据索引删除, 使用 **pop** 方法, 该方法不传递索引时默认删除最后一个元素, 传递索引则根据索引删除元素.



插入元素分为两种：尾部插入 指定位置插入

```
# 创建空列表
my_list = []
# 向列表中添加元素
my_list.append(10)
my_list.append('Obama')
# 在指定位置插入元素
my_list.insert(1, 20)
# 输出内容
print(my_list)
# 创建新的列表
new_list = [100, 200, 300]
# 合并两个列表
my_list.extend(new_list)
print(my_list)
```

删除分为两种：根据值删除，根据位置(索引)删除

```
my_list = [10, 20, 30, 20]
# 根据索引删除
my_list.pop(2)
print(my_list)
# 根据值删除
my_list.remove(20)
print(my_list)
# 删除所有元素
my_list.clear()
# 显示列表中还剩下多少元素
print(len(my_list))
```

3.2.2.4 列表元素排序

排序指的是记录按照要求排列. 排序算法在很多领域得到相当地重视.

列表提供了相关方法来对列表中的元素进行排序. 分别是:

1. 将列表中的元反转.
2. 列表中的元素升序(从小到大)、降序排列(从大到小).

```
import random

my_list = []
# 产生一个包含10个随机数的列表
i = 0
while i < 10:
    random_number = random.randint(1, 100)
    my_list.append(random_number)
    i += 1

# 打印列表中的元素
print('my_list:', my_list)
# 对列表中的反转
my_list.reverse()
# 打印列表中的元素
print('my_list:', my_list)
# 对列表中的元素排序, 默认升序
my_list.sort()
print('my_list:', my_list)
# 对列表中的元素排序, 降序排列
my_list.sort(reverse=True)
print('my_list:', my_list)
```

3.2.2.5 列表练习

一个学校, 有3个办公室, 现在有 8 位老师等待工位的分配, 请编写程序, 完成随机的分配.

思路分析如下:

1. 待分配的 8 位老师需要存储, 我们可以用列表来暂时存储 8 位老师.
2. 一个学校中包含了多个办公室, 学校可用列表来表示, 学校中又包含了多个办公室, 每个办公室里可能有多个老师, 办公室仍然可用列表来表示.
3. 从待分配老师列表中取出数据, 随机产生办公室编号, 将该老师分配到该办公室.
4. 打印各个办公室中的老师列表.

```
import random

# 定义一个列表用来存储8位老师的名字
teacher_list = []
i = 0
while i < 8:
    teacher_name = '老师' + str(i + 1)
    teacher_list.append(teacher_name)
    i += 1

# 定义学校并包含3个办公室
school = [[], [], []]
# 获取每个老师并随机分配办公室
for teacher in teacher_list:
    office_number = random.randint(0, 2)
    school[office_number].append(teacher)

# 打印各个办公室的老师列表
```

```
for office in school:
    for teacher in office:
        print("%s" % teacher, end=' ')
    print('\n' + '*' * 20)
```

3.2.3 小结

1. 列表一般使用一对中括号 [] 来定义.
2. 列表容器的特点: 可以存储任意类型的数据, 并且元素可以修改.
3. 列表中存储的元素类型可不相同, 但建议存储同样的类型.
4. 列表是序列式容器, 支持下标索引和切片操作.
5. 列表遍历可以使用 while 循环, 也可以使用 for 循环.
6. 列表可通过 in 或 not in 运算符来判断是否存在某个元素.
7. 列表的 append 方法用于向列表尾部添加元素.
8. 列表的 insert 方法用于向列表指定索引位置添加元素.
9. 列表的 extend 方法用于将一个列表中所有元素添加到当前列表的尾部.
10. 列表的 pop 方法默认删除尾部元素, 如果设置索引参数, 可删除该索引位置的元素.
11. 列表的 reverse 方法可以将列表中元素逆序.
12. 列表的 sort 方法可以将列表中的元素升序或者降序排列.

3.2.4 思考

1. 如何使用列表存储以下三个人的信息?
1.1 姓名: 奥巴马 年龄: 50 性别: 男 身份证: 1088 1.2 姓名: 希拉里 年龄: 56 性别: 女 身份证: 2782 1.3 姓名: 特朗普 年龄: 60 性别: 男 身份证: 3368
2. 假如列表中存储了上述信息 10 亿条, 现在需要根据身份证号从列表中查找该身份证号所对应的信息, 请问最差情况下, 需要多少次才能找到目标数据?

3.3 元组

学习目标:

1. 能够说出如何定义一个列表
2. 能够说出元组和列表的区别
3. 能够说出如何使用 **while** 和 **for** 循环来遍历元组中的元素
4. 能够说出元组支持哪些操作

3.3.1 元组语法和方法

Python的元组与列表类似, 不同之处在于元组的元素不能修改. 元组使用小括号来定义, 列表使用方括号来定义.

由于元组不支持修改, 所以元组只支持遍历、查找操作.

元组同样属于序列式容器, 支持索引和切片语法.

1. 查询元素: **count**、**index**
2. 遍历操作: **while**、**for**

```
# 定义元组
my_tuple = (10, 20, 30)
my_tuple = ((10, 20, 30), (100, 200, 300))

# 遍历
for ele in my_tuple:
    for val in ele:
        print(val)

# 查找
my_tuple = (10, 20, 30)
# 判断元素是否存在
if my_tuple.count(20) > 0:
    index = my_tuple.index(20)
    print('元素的位置:', index)

if 20 in my_tuple:
    index = my_tuple.index(20)
    print('元素的位置:', index)
```

注意: 如果定义的元素中只有一个元素, 需要额外添加一个逗号在元素后.

```
my_tuple = (10,)
my_tuple = ((10, 20, 30), )
my_tuple = ((10, ), )
```

3.3.2 小结

1. 元组使用一对小括号来定义, 在定义之后不允许对元素进行修改.
2. 元组中只有一个元素时, 需在最尾部添加一个逗号.
3. 元组是序列式容器, 支持索引、切片操作.
4. 元组比列表更节省空间.

黑马程序员软件测试

3.4 字典

学习目标:

1. 能够说出如何定义一个字典
2. 能够说出字典和列表的区别
3. 能够说出如何使用 `for` 循环来遍历列表中的键、值和键值对
4. 能够说出字典键和值的特点

列表中存储的元素都是值, 在列表中根据某个关键字去查找数据效率较低. 为了解决该方面的问题, Python 提供了字典这种容器类型, 字典中存储的每一个元素都是键值对, 并且在字典中根据键(关键字)去查找某个元素的效率非常高.

为了实现高的查询效率, 字典被实现成了一种非序列式容器, 也就导致字典无法根据索引获得元素, 同样也不支持切片操作.

3.4.1 字典语法格式

字典是另一种可存储任意类型对象. 字典中的每一个元素都是一个 "键值对", 键值之间用冒号(:)分割, 每个字典元素(键值对)之间用逗号(,)分割, 整个字典包括在花括号 {} 中, 格式如下所示:

```
my_dict = {key1: value1, key2: value2, key3: value3}
```

字典键和值的特点:

1. 键一般是唯一的, 如果重复最后的一个键值对会替换前面的, 键的类型一般情况下使用字符串、数字类型. **2.** 值不需要唯一, 可以为任何的数据类型.

3.4.2 字典操作

3.4.2.1 访问元素

字典中根据键获得值的操作, Python 提供了两种方式:

1. 直接通过键来获得, 但当键不存在时, 会抛出错误.
2. 通过 `get` 方法来根据键获得值, 如果键不存在则会返回 `None`, 该返回默认值也可自定义.

```
person = {'name': 'Obama', 'age': 18, 'sex': '男'}
# 如果 key 不存在会报错
print(person['name'])
# 如果 key 不存在可设置默认值
print(person.get('gender', 'default'))
```

3.4.2.2 添加和修改元素

在字典中修改元素, 直接通过键来修改即可. 这时需要注意, 如果键不存在, 默认为新增元素操作, 只有键存在时, 才为修改操作.

```
person = {'name': 'Obama', 'age': 18, 'sex': '男'}
# 如果 key 不存在则为添加新元素
```

```
person['salay'] = 12000
# 如果 key 存在则为修改元素
person['age'] = 20
```

3.4.2.3 删除元素

Python 中字典中元素的删除, 需要根据键来进行, 我们可以使用 `pop` 方法根据 `key` 来删除字典中的元素.

```
person = {'name': 'Obama', 'age': 18, 'sex': '男'}
# 删除某个元素
person.pop('name')
# 清空字典
person.clear()
```

3.4.2.4 遍历元素

由于字典是非序列式容器, 无法通过逐个获取元素, 所以遍历字典的方式就是先将字典转换成类似列表的形式, 再对其进行遍历. 在获得字典的列表时, 我们有以下三个方案:

1. 获得字典键的列表, 通过字典的 `keys` 方法.
2. 获得字典值的列表, 通过字典的 `values` 方法.
3. 获得字典的键值对列表, 通过字典的 `items` 方法.

```
person = {'name': 'Obama', 'age': 18, 'sex': '男'}
# 获得字典的值列表
print(person.values())
# 获得字典的键列表
print(person.keys())
# 获得字典的键值对列表
print(list(person.items()))

for key, value in person.items():
    print(key, value)
```

3.4.3 小结

1. 字典通过一对花括号 `"{}"` 来定义, 每一个元素都是一个键值对.
2. 字典不支持索引、切片操作.
3. 字典根据键查询元素的效率非常高.
4. 字典的键一般情况下是数字、字符串等, 键必须唯一不重复.
5. 字典的值可以重复, 任意类型.
6. `for` 循环无法直接遍历字典, 需要先将字典转换为类似列表那样能够被迭代的类型.
7. 字典的 `get` 方法可以根据键获得值, 如果键不存在返回默认值.
8. 字典的 `pop` 方法可以根据键来删除字典中某个元素.
9. 字典的 `clear` 方法可用来清空字典.
10. 字典的 `keys` 方法可以返回一个由字典的键组成的列表.
11. 字典的 `values` 方法可以返回一个由字典的值组成的列表.
12. 字典的 `items` 方法将每一个键值对存放到元组中, 然后将元组列表返回.

黑马程序员软件测试

3.5 集合

学习目标:

1. 能够说出如何创建集合
2. 能够说出字典和集合的区别
3. 能够说出如何向集合中添加元素
4. 能够说出如何删除集合中的某个元素
5. 能够说出如何使用 for 循环来遍历集合
6. 能够说出如何计算两个集合的交集
7. 能够说出如何计算两个集合的并集

set集合是一个无序不重复元素集。由于set是一个无序集合，set并不记录元素位置，所以不支持下标操作和切片操作。

3.5.1 创建集合

```
# 1. 创建一个空的set集合
my_set = set()

# 2. 创建一个包含元素的集合
my_set = {10, 20, 30, 40}
print(my_set)

# 3. 用一个容器来创建集合
# 注意: set会剔除重复元素
my_set = set([1, 2, 3, 4, 5, 5])
print(my_set)

# 4. 创建一个唯一元素的字符集合
my_set = set("hello world!")
print(my_set)
```

3.5.2 集合添加元素

向set集合中添加元素，可以使用add()函数和update()函数，add()可以一次添加一个元素，update()函数可以一次添加多个元素。

```
# 创建一个空的集合
my_set = set()
# add()函数向set中添加元素
my_set.add(10)
my_set.add(20)
my_set.add(30)
# 打印set集合
print(my_set)

# update()函数添加多个元素
my_set.update([60, 40, 80, 90])
my_set.update((160, 140, 180, 190))
my_set.update("hello")
# 如果添加的元素是一个字典，那么将字典的key添加到集合中
```

```
# my_set.update({"name": "smith", "age": 1030})
print(my_set)
```

3.5.3 集合删除元素

删除set集合中的元素可以使用pop()、remove()函数、discard()函数

1. pop()函数会删除set集合中的任意一个元素，如果set集合为空，会抛出KeyError错误。
2. remove(element)函数从集合中删除一个元素，如果元素不存在，会抛出KeyError错误。
3. discard(val)函数删除集合中的一个元素，如果不存在，则不做任何事。

```
my_set = set([9, 2, 3, 4, 7])
# 删除任意一个元素
my_set.pop()
print(my_set)

# 删除指定元素
my_set.remove(4)
print(my_set)

# 删除元素
my_set.discard(3)
print(my_set)
```

3.5.4 集合遍历

```
# 创建一个空的集合
my_set = set([1, 2, 3, 4])
# 遍历set集合
for value in my_set:
    print(value, end="|")
```

3.5.5 集合交集和并集

```
my_set1 = set([1, 2, 3, 4, 5])
my_set2 = set([3, 4, 5, 6, 7])

# 1. 求两个集合的并集
new_set1 = my_set1.union(my_set2)
# 或者
new_set2 = my_set1 | my_set2
print(new_set1)
print(new_set2)

# 2. 求两个集合的交集
new_set3 = my_set1.intersection(my_set2)
# 或者
new_set4 = my_set1 & my_set2
print(new_set3)
print(new_set4)
```

3.5.6 set应用: 统计字符个数

```
# 统计字符串中字符的个数
my_string = input("请输入任意字符串:")
# 先对字符串去重
```

```
new_string = set(my_string)
# 字典记录字符出现次数
my_count = {}
# 遍历new_string
for ch in new_string:
    my_count[ch] = my_string.count(ch)
# 输出结果
print(my_count)
```

3.5.7 小结

1. 集合使用一对花括号定义, 每一个元素是任意类型的对象, 不是键值对.
2. 集合不支持切片、索引操作.
3. 集合中的元素唯一且不重复.
4. 集合支持 for 循环遍历.
5. 集合的 `add` 方法可以向集合中添加一个元素.
6. 集合的 `update` 方法可以向集合中添加一个容器的元素.
7. 集合的 `pop` 方法删除set集合中的任意一个元素, 如果set集合为空, 会抛出`KeyError`错误.
8. 集合的 `remove` 方法从集合中删除一个元素, 如果元素不存在, 会抛出`KeyError`错误.
9. 集合的 `discard` 方法删除集合中的一个元素, 如果不存在, 则不做任何事.
10. 集合的 `union` 方法可以返回两个集合的并集.
11. 集合的 `intersection` 方法可以返回两个集合的交集.

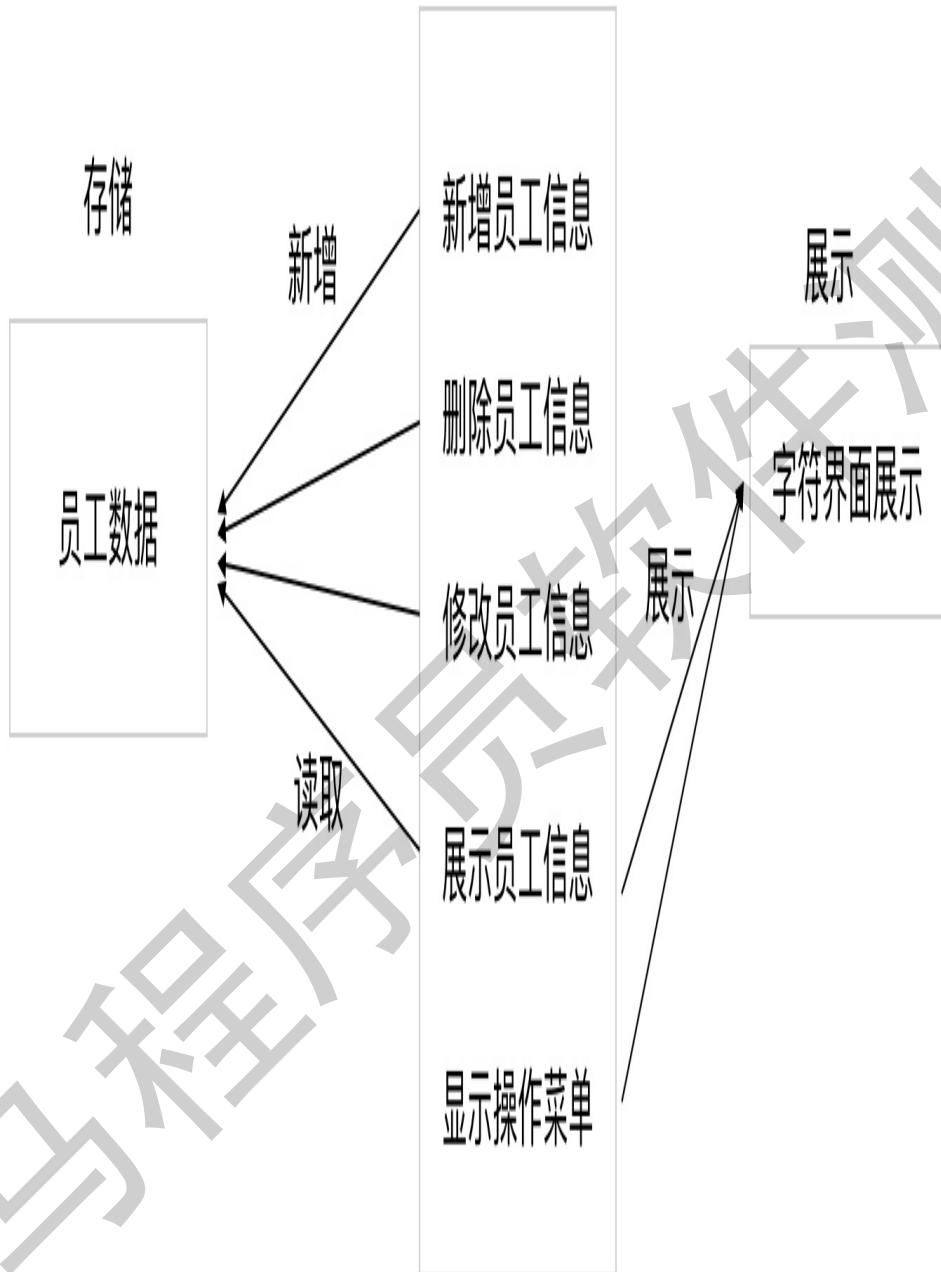
3.6 案例-员工管理系统

3.6.0 思路分析

员工管理系统可依据职责不同分为3部分:

1. 数据存储
2. 业务实现
3. 数据呈现

业务



从图上可分析出, 业务部分实现了对员工数据的增删改查等操作, 员工数据的存储问题决定了业务实现部分的实现, 业务部分的实现决定了数据呈现部分代码的编写.

1. 员工数据存储问题

员工信息: 编号、姓名、工资、性别

我们首要解决的问题就是数据的存储问题, 我们这里要考虑的是数据以什么样的容器来存储.

从业务的删除操作、修改操作来讲, 我们都需要根据键(员工编号)快速定位员工信息, 达到修改和删除的目的.

所学容器中, 字典可以根据某个不重复的关键字来快速定位元素. 所以我们使用字典存储员工信息, 字典的键为员工编号, 字典的值为员工信息. 存储结构如下:

```
employee = {'1001': 员工信息, '1002': 员工信息}
```

员工信息的存储我们使用何种类型?

因为员工信息是多个维度的信息, 所以本身也是个容器, 从所学容器中, 得出以下三种存储方案:

```
# 1. 使用字典
employee = {'1001': ['Obama', 10000, '男'], '1002': ['Trump', 12000, '男']}
# 2. 使用元组
employee = {'1001': ('Obama', 10000, '男'), '1002': ('Trump', 12000, '男')}
# 3. 使用字典
employee = {'1001': {'name': 'Obama', 'salary': 10000, 'sex': '男'}, '1002': {'name': 'Trump', 'salary': 12000, 'sex': '男'}}
```

1. 使用元组的话, 会导致数据无法修改, 不能达到我们的预期, 所以这个方案被否决.
2. 使用列表的话, 可以实现我们的要求, 但是相较于字典的话, 操作列表中的元素就必须知道列表中名字在第几个索引、年龄在第几个索引、性别在第几个索引, 并且在实现代码中通过索引获取相应的姓名、年龄、性别等数据可读性较差.

最终我字典嵌套字典的存储结构.

```
# 3. 使用字典
employee = {'1001': {'name': 'Obama', 'salary': 10000, 'sex': '男'}, '1002': {'name': 'Trump', 'salary': 12000, 'sex': '男'}}
```

2. 业务实现步骤

1. 首先显示操作菜单.
2. 获得用户输入的菜单编号.
3. 根据菜单编号选择不同的操作执行.
4. 重复上面3个步骤.

3.6.1 搭建业务框架

我们根据分析出的业务实现步骤, 先将员工管理系统的实现步骤框架搭建起来, 此时并不实现具体的功能细节.

```
# 存储员工信息
employee = {}

def show_menu():
    """显示系统菜单"""
    pass

def add_new_info():
    """添加新的员工"""
    pass

def remove_info():
    """删除员工信息"""
```

```

pass

def show_all_info():
    """打印所有员工信息"""
    pass

def edit_info():
    """修改员工信息"""
    pass

def main():

    while True:

        # 1. 打印菜单
        show_menu()
        # 2. 等待用户输入
        user_operate = input('请输入您的操作:')
        # 3. 根据用户选择做相应的事情
        if user_operate == '1':
            add_new_info()
        elif user_operate == '2':
            remove_info()
        elif user_operate == '3':
            edit_info()
        elif user_operate == '4':
            show_all_info()
        elif user_operate == '5':
            print('欢迎再次使用本系统!')
            break
        else:
            print('您的输入有误, 请重新输入!')

main()

```

3.6.2 菜单功能实现

```

def show_menu():
    """显示系统菜单"""

    print("*" * 30)
    print("员工管理系统 v1.0")
    print(" 1:添加员工信息")
    print(" 2:删除员工信息")
    print(" 3:修改员工信息")
    print(" 4:显示所有信息")
    print(" 5:退出员工系统")
    print("*" * 30)

```

3.6.3 新增功能实现

1. 获得输入的员工信息.
2. 将员工信息依据 "键:值" 存储到字典中, 每个字典表示一个员工的完整信息.
3. 以员工编号为键, 员工信息为值, 将员工信息存储到 `employee` 字典中.

```

def add_new_info():
    """添加新的员工"""

```



```

em_number = input('请输入员工编号:')
em_name = input('请输入员工姓名:')
em_salary = input('请输入员工工资:')
em_gender = input('请输入员工性别:')
# 构建员工信息字典
em_info = {'name': em_name, 'salary': em_salary, 'gender': em_gender}
# 以员工的编号作为键, 存储员工信息
employee[em_number] = em_info

```

3.6.4 删除功能实现

我们删除员工信息, 根据员工的编号来删除, 实现思路如下:

1. 获得要删除的员工编号.
2. 判断员工编号是否存在, 如果不存在则终止函数执行.
3. 如果员工信息存在, 则根据键删除员工信息.

```

def remove_info():
    """删除员工信息"""

    em_number = input('请输入要删除的员工编号:')
    # 判断员工编号是否存在
    if em_number not in employee.keys():
        print('您输入的员工编号不存在!')
        return

    del employee[em_number]
    print('员工编号为: %s 的员工信息被删除!' % em_number)

```

3.6.5 修改功能实现

我们根据员工编号, 修改该员工的信息, 具体思路如下:

1. 获得要修改的员工编号.
2. 判断员工编号是否存在, 如果不存在则终止修改函数执行.
3. 首先显示员工的对应信息, 并提示用户输入修改之后的值: 3.1 如果直接回车, 表示用户无任何输入, 则表示不修改. 3.2 如果用户输入值, 则将对信息修改为新输入的值.

```

def edit_info():
    """修改员工信息"""

    em_number = input('请输入要修改的员工编号:')

    if em_number not in employee.keys():
        print('您输入的员工编号不存在!')
        return

    new_name = input('编号为 %s 的员工姓名为 %s, 你要修改为:' % (em_number, employee[em_number]['name']))
    new_salary = input('编号为 %s 的员工工资为 %s, 你要修改为:' % (em_number, employee[em_number]['salary']))
    new_gender = input('编号为 %s 的员工性别为 %s, 你要修改为:' % (em_number, employee[em_number]['gender']))

    if new_name != '':
        employee[em_number]['name'] = new_name

    if new_salary != '':
        employee[em_number]['salary'] = new_salary

    if new_gender != '':

```

```
employee[em_number]['gender'] = new_gender

print('编号为 %s 的员工信息修改成功!' % em_number)
```

3.6.6 显示功能实现

直接遍历 `employee` 字典, 注意字典中存储的每一个元素都是一个键, 键为员工编号, 值为字典类型, 存储的是员工的信息.

```
def show_all_info():
    """打印所有员工信息"""

    print('*' * 30)
    for em_num, em_info in employee.items():
        print('%s\t%s\t\t%s\t%s' % (em_num, em_info['name'], em_info['salary'], em_info['gender']))
```