# 7.25

## 1. 编写计算机视觉（CV）markdown文档

查阅资料后了解什么是计算机视觉，以及他的主要方向

计算机视觉

- 模式识别
- 图像处理

- 图像理解

# 7.26

## 1. 完善昨日CV.md

查阅嵌入式视觉及嵌入式AI的资料

嵌入式AI

- 嵌入式视觉
- 边缘AI处理
- 多协议与物联网集成
- 模型优化与压缩（模型剪枝、权重量化、知识蒸馏）
- 安全性和隐私保护
- 深度学习的嵌入式部署

## 2. 学习markdown文档

在B站和CSDN上了解markdown的语法

markdown常用语法

1. **目录**

# 标题1

## 标题2

#### 标题6

3. **列表**
4. 列表1

- 列表2

- [x] 待办1
- [ ] 待办2

***列表***

自定义列表1

自定义列表2

4. **文本**
   文本 **文本** **文本**
   文本 ~~文本~~ 文本

> 文本

5. **分割线**

6. **下划线**
   下划线
7. **注释**
   这里有一个 注释
8. **脚注**
   这里有一个脚注 [1]

9. **表格**
   | 列1 | 列2
   | --- | ---
   | 值1 | 值2
10. **代码&代码块**

```
printf("code\n");
```

```
if(1)
    printf("code\n);
```

11. **链接**
    CSDN
    https://www.csdn.net/
12. **插图**

# 8.2

## 1. 继续完善CV.md

完善嵌入式AI的细节内容

## 2. 学习下载Anacoonda

- anaconda是一个方便快捷的包管理器和环境管理器，继承了相当多的工具和库，并且有良好的跨平台性。jupyter notebook 、 numpy、pandas 、tensorflow或pytorch 、opencv均可在其中找到
- 简单了解Anaconda的命令行

# 8.6

# 1. 学习jupyter notebook

- jupyter notebook集编写、运行、展示于一体，以网页的形式打开，可以在网页页面中“直接”编写代码和运行代码，代码的运行结果也会直接在代码块下显示。如在编程过程中需要编写说明文档，可在同一个页面中直接编写，便于作及时的说明和解释。
- 将md撰写平台迁移至jupyter noteboook

# 2. 了解ipython

- IPython是一种强化的Python解释器，提供了比默认的Python解释器更丰富的功能和增强的交互性。IPython具有许多功能，包括代码补全、语法高亮、历史记录浏览、命令自动完成、内置的帮助和文档查看器等。
- 基本功能
  i. **Tab 补全**
  - 在shell中输入表达式时，只要按下Tab键，当前命名空间中任何与输入的字符串相匹配的变量(对象或者函数等)就会被找出来

  ii. **? 查看帮助**
  - ? 显示方法说明信息，不包含python代码实现的显示
  - ?? 不但显示方法说明信息，还包含python代码实现的显示

  iii. **hist 查询历史**
  - hist命令之后加上-n，即hist -n也可以显示出输入的序号
  - _，__，___和_i，_ii，_iii变量保存着最后三个输出和输入对象。_n和_in(这里的n表示具体的数字)变量返回第n个输出和输入的历史命令

  iv. **%pylab 调用Numpy和Matplotli**
  - %pylab命令可以使Numpy和matplotlib中的科学计算功能生效，这些功能被称为基于向量和矩阵的高效操作，它能够让我们在控制台进行交互式计算和动态绘图

  v. **%run 运行脚本**
    - 所有文件都可以通过%run命令当做Python程序来运行，输入%run 路径+python文件名称即可(相当于把整个文件加载进来，所用文件中的变量和函数均可直接使用)

  vi. **%time 测量运行时间**
    - % 测试单行代码
    - %% 测试多行代码
    - time 测量单次运行时间
    - timeit 测量多次运行时间，取均值

# 8.7

# 1. 学习python语法

- 基本语法
  i. **注释**（`#` `'''` `"""`）

```
# 第一注释
'''
第二注释
'''
"""
第三注释
"""
print ("Hello, Python!")
```

  ii. **使用缩进代表代码块**
    - 缩进的空格数是可变的，但是同一个代码块的语句必须包含相同的缩进空格数
  iii. **多行语句**
    - 用 \ (反斜杠)隔开，但在 `[]` ，`{}` ，或 `()` 中的多行语句，不需要使用反斜杠
  iv. **数字类型**
    - python中数字有四种类型：*整数*、*布尔型*、*浮点数*和*复数*
    - *int (整数)*, 如 1, 只有一种整数类型 int，表示为长整型，为*不可变数据类型*
      - 0b(B) 引导二进制、0o(O) 引导八进制、0x(X) 引导十六进制
    - *bool (布尔)*, 如 True
    - *float (浮点数)*, 如 1.23、3E-2，为*不可变数据类型*
      - 函数 `round(a+b,x)` 限制结果位数(保留x位小数)
    - *complex (复数)*, 如 1 + 2j、 1.1 + 2.2j
      - .real表示实部, .imag表示虚部
    - 函数 `type()` 查看变量对相应的数据类型
  v. 同行用 ； 隔开可显示多条语句

# 8.8、8.9

# 1. 学习python语法

- 基本语法
  i. **字符串**

- python字符串不区分单双引号 `'` & `"`，且可用三引号 `'''` 或 `"""` 表示跨行字符串***（8.25补档，发现多行字符串和注释的形式相同，实际上Python解释器仍然会处理这些多行字符串（例如，检查字符串内的语法错误），但它们不会产生输出，因此可以用于注释的目的）***
- 反斜杠 `\` 表示转义， `r(R)` 可阻止转义，如 `r"this is a line with \n"` 中 `\n` 会显示，而不是表现为换行
- 字符串可用 `+` 连接，用 `*` 重复表示，用 `in` & `not in` 判断是否为子串，用 `len()` 计算长度（包括空格）
- 字符串(从左往右)头部索引为0，尾部索引为-1
- 字符串切片 `str[start:end]`，其中 start（包含）是切片开始的索引，end（不包含）是切片结束的索引,遵循**左闭右开**原则。
- 字符串的切片可以加上步长参数 step，语法格式如下： `str[start:end:step]`
- 示例如下：

```
str = '123456789'
print(str)                 # 输出字符串
print(str[0:-1])           # 输出第一个到倒数第二个的所有字符
print(str[0])              # 输出字符串第一个字符
print(str[2:5])            # 输出从第三个开始到第六个的字符（不包含）
print(str[2:])             # 输出从第三个开始后的所有字符
print(str[1:5:2])          # 输出从第二个开始到第五个且每隔一个的字符（步长为2）
print(str * 2)             # 输出字符串两次
print(str + '你好')        # 连接字符串
print("你好" in str)       # 判断 “你好” 是否为 str 的字串，是返回 “true”，否返回 “false”,“not ir
print('hello\nrunoob')     # 使用反斜杠(\)+n转义特殊字符
print(r'hello\nrunoob')    # 在字符串前面添加一个 r，表示原始字符串，不会发生转义
```

ii. **输出函数 `print()`**

- 完整形式为 `print(value, ..., sep='', end='\n', file=None)`
- `chr()` 将输入的ASCII码转化为字符； `ord()` 将字符转化为对应的ASCII码
- 使用 `%` 格式化输出，python3.6 之后版本可使用f-string 格式化字符串，以 `f` 开头，后面跟着字符串，字符串中的表达式用大括号 `{}` 包起来，它会将变量或表达式计算后的值替换进去
- 示例如下：

```
name = 'Runoob'
'Hello %s' % name # 'Hello Runoob'

name = 'Runoob'
f'Hello {name}'  # 替换变量 'Hello Runoob'
f'{1+2}'         # 使用表达式 '3'

w = {'name': 'Runoob', 'url': 'www.runoob.com'}
f'{w["name"]}: {w["url"]}' # 'Runoob: www.runoob.com'
```

- python附带大量字符串内建函数，使用格式为str.name(value,...,...)

# 8.12 8.13 8.23

# 1. python语法

- 基本语法
  i. **前言**
  - 序列是Python中最基本的数据结构。序列中的每个值都有对应的位置值，称之为索引，第一个索引是0，第二个索引是1，依此类推。Python有6个序列的内置类型，但最常见的是列表和元组。列表都可以进行的操作包括***索引、切片、加、乘、检查成员***。此外，Python已经内置确定序列的长度以及确定最大和最小的元素的方法

  ii. **列表**
  - 列表是最常用的Python数据类型，它可以作为一个方括号内的逗号分隔值出现。列表的数据项不需要具有相同的类型，而且创建一个列表，只要把逗号分隔的不同的数据项并使用方括号括起来即可
  - 列表的索引和切片与字符串相同， `+` `*` `in` `len()` `+=` 等运算与字符串相同，以及列表的嵌套（即列表的元素仍是列表）

```
a = ['a', 'b', 'c']
n = [1, 2, 3]
x = [a, n]
x           # [['a', 'b', 'c'], [1, 2, 3]]
x[0]        # ['a', 'b', 'c']
x[0][1]     # 'b'
```

  - 列表包括以下函数
    - `len(list)` (求列表元素个数)
    - `max(list)` (返回列表元素最大值)
    - `min(list)` (返回列表元素最小值)
    - `list(seq)` (将元组转换成列表)
  - 列表还包括以下方法
    - `list.append(obj)` (将指定元素添加到列表尾部)
    - `list.count(obj)` (计算指定元素出现次数)
    - `list.extend(seq)` (将序列连接至列表尾部)
    - `list.index(obj)` (找出该元素第一次出现的索引位置)
    - `list.insert(index,obj)` (在指定位置插入指定元素)
    - `list.reverse()` (列表反向)
    - `list.clear()` (清空)
    - `list.copy()` (复制)
    - ……
  - 列表还可使用 `del list[index]` 语句来删除指定元素

- 列表比较需要引入 `operator` 模块的 `eq` 方法，即 `operqtor.eq(list1, list2)`

iii. **元组**

- 与列表类似，元组的各元素间用逗号隔开，所有元素位于小括号 `()` 内，**也可不使用括号而直接用逗号，分割各元素**，注意如果元组内只有一个元素，该元素后仍要使用逗号(否则括号将被当作运算符)
- 元组仍能进行索引、切片、加、乘等操作
- 元组的元素无法被修改和删除，修改和删除的操作仅针对元组整体
- 元组包括以下函数
  - `len(tuple)`
  - `max(tuple)`
  - `min(tuple)`
  - `tuple(iterable)`

iv. **字典**

- 整个字典被包括在花括号 `{}` 下，字典的元素由键值对组成，键值对内部由冒号隔开 `:` ，对与对之间由逗号 `,` 隔开，其中键必须是唯一的，且只能取不可变数据类型(如果使用两个一样的键，第二个键值会被记住)，而值是不唯一的，且可取任何数据类型
- 字典的更改类似C语言中数组的更改，不同的是数组的下标变成字典的键
- 使用 `del()` 方法删除字典元素和字典整体
- 内置函数
  - `len(dict)`
  - `str(dict)` (输出字典，以打印的字符串形式表示)
  - `type(variable)` (返回输入的变量类型)
- 内置方法
  - `dict.copy()`
  - `dict.clear()`
  - `dict.fromkeys(seq[, value])` (创建一个新字典，以序列seq中元素做字典的键，value为字典所有键对应的初始值)
  - `dict.get(key, default=None)` `dict.setdefault(key, default=None)` (后者与前者区别在于如果查找的键不在字典里，后者会生成该键并将值设为None)
  - `key in dict`
  - `dict.items()` `dict.keys()` `dict.values()` (返回视图对象，不是列表，不支持索引和更改，可以使用 `list()` 来转换为列表)
  - `dict.update(dict2)` (把字典参数dict2的key/value(键值对)更新到字典dict里)(狗尾续貂doge)
  - `dict.pop(key[,default])` (删除key所对应的值，返回被删除的值，如果key不存在且默认值default没有指定，则触发KeyError异常)
  - `dict.popitem()` (返回并删除字典中的最后一对键值,即**LIFO**规则)
- ***补充知识 ———— 赋值与复制***

  a. 直接赋值
  - `a = b` (实际是对象的别名)

  b. 浅复制(copy)

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

  a[1].append(4)
  a[1].remove(1)     # a = {1: [2, 3, 4]} b = {1: [2, 3, 4]}

  a.update({2: [5]}) # a = {2: [5]} b = {1: [2, 3, 4]}
  ```

  (拷贝父对象，不会拷贝对象的内部的子对象)

c. 深复制(deepcopy)

- ```
  a = {1: [1,2,3]}
  import copy
  c = copy.deepcopy(a) # c = {1: [1,2,3]}

  a[1].append(4)       # a = {1: [1, 2, 3, 4]}, c = {1: [1, 2, 3]}
  ```

  (引入copy模块的deepcopy方法，完全拷贝了父对象及其子对象)

v. **集合**

- 集合是一个元素无序不重复的序列，使用大括号 `{}` 创建集合，元素之间用逗号 , 分隔，或者也可以使用 `set()` 函数创建集合(注意创建一个空集合必须用 `set()` 而不是 `{}` ， 因为 `{}` 是用来创建一个空字典)
- 集合间运算
  a. 差运算 -
  b. 并运算 |
  c. 交运算 &
  d. 对称差运算 ^
- 方法
  - `set.add(elem)` `set.update(elem\set)`
  - `set.copy()`
  - `set.clear()`
  - `set.remove()` `set.discard()`
  - `set.disjoint(set)` (是否没有交集)
  - `set.issubset(set)` (前者是否是后者的子集)
  - `set.issuperset(set)` (前者是否是后者的超集)
  - `set.union(set1, set2, ……)` (返回所有集合的并集)
  - `set.difference(set1, set2, ……)` `set.difference_update(set1, set2, ……)` (前者返回一个移除相同元素的集合，后者在原集合上直接删除元素，无返回值)
  - `set.intersection(set1, set2, ……)` `set.intersection_update(set1, set2, ……)` (前者返回交集，后者把原集合转换成交集)
  - `set.sysmmetric_difference()` `set.symmetric_difference_update()` (前者返回对称差集，后者把原集合转化成对称差集)

# 8.24 8.25

# 1. python语法

- 基本语法
  - i. **条件控制**
  - `if – else` 语句
    - ```
      if condition_1:
          statement_block_1
      elif condition_2_1 | condition_2_1: # 可以设置多个匹配条件，条件使用 | 隔开
          statement_block_2
      else:
          statement_block_3
      ```
  - `match – case` 语句
    - ```
      match expression:
          case pattern1:
              statement_block_1
          case pattern2 if condition_2: # 可以使用if关键字在case中添加条件
              statement_block_2
          case _:                       # _ 通配符表示其他情况，相当于default
              statement_block_3
      ```
  - ii. **循环语句**
  - `while` 循环
    - ```
      while <expr>:                     # expr条件语句为true则执行statement(s)语句块，如果为fa
          <statement(s)>
      else:                             # else语句为可选项additional_statement(s)
          <additional_statement(s)>
      ```
  - `for` 循环
    - ```
       for <variable> in <sequence>:
           <statements>
       else:                             # else语句为可选项，当循环执行完毕后，会执行else子句中
           <statements>                  # 如果在循环过程中遇到了break语句，则会中断循环，此时不
      ```
  - `range()` 函数
    - 完整形式为 `range(start, stop, step)`，计数从start开始，默认是0，到stop结束，不包括stop（相当于**左闭右开**），步长可以为负值

- `range()` 函数返回的是一个**可迭代对象(iterable)**,但可通过 `list()` `tuple()` 等函数转换为需要的序列

- `break` & `continue` 语句
  - 与C语言中的高度相似，需要注意的是 `break` 跳出循环后不再执行任一条循环的else语句
- `pass` 语句
  - `pass` 是空语句，是为了保持程序结构的完整性。 `pass` 不做任何事情，一般用做占位语句

iii. **迭代**

- 基本概念
  - 指的是重复执行某一过程或计算步骤，直到满足特定的条件为止，其过程通常包含三个主要部分：初始化、迭代步骤（或称为循环体）和终止条件
- ***迭代与循环***
  - 迭代可以被认为是特殊的循环或变化的循环（循环指在满足条件的情况下，重复执行同一段代码），因为迭代的结果会影响到下一次迭代的执行，而狭义的循环中执行的结果往往是相同的（所以我认为的循环原来是包括迭代的）
- ***迭代与递归***
  - 递归是函数直接或间接调用自身来实现循环，而迭代是通过函数内某段代码实现循环；递归的代码简洁易懂，但是对内存的要求较高，而迭代避免了函数的多次调用和内存的额外分配，因而效率较高
- **迭代器（迭代器对象）**
  - 因为 **迭代器(Iterator)** 一词的指代对象较多，可以有迭代器对象、迭代器协议或迭代器模式等，因此以下专指 **迭代器对象**
  - **迭代器对象**是一个可以记住遍历的位置的对象，它从集合的第一个元素开始访问，直到所有的元素被访问完结束，只能往前不会后退
  - **迭代器对象**内置两个方法 `__iter__()` 和 `__next__()`
    a. `__iter__()`： 返回迭代器对象本身
    b. `__next__()`： 返回容器中的下一个元素
       - 为防止出现无限循环的情况，我们通常在 `__next__()` 方法中接入 `StopIteration` 异常，当迭代器没有更多元素返回时抛出此异常
- **可迭代对象(Iterable)**
  - 内部定义 `__iter__()` 函数的对象，python中绝大多数容器对象都是可迭代的
  - **迭代器对象**一定是**可迭代对象**，且只有**迭代器对象**才可以进入for循环或者迭代过程
- 示例如下：

```
class MyNumbers:
  def __init__(self):
      self.a = 1

  def __iter__(self):      # __iter__()函数绝大多数情况返回本身即可，初始化内容交给__ini
    return self

  def __next__(self):
    if self.a <= 20:
      x = self.a
      self.a += 1
      return x
    else:                  # 限定迭代20次
      raise StopIteration

myclass = MyNumbers()
myiter = iter(myclass)     # myiter属于迭代器对象，可进入for循环

for x in myiter:
  print(x)
```

# 8.25 8.26 8.27

# 1. python语法

- 有点难的语法
  - i. **错误与异常**
  - **语法错误(解析错)**
    - 通常包括拼写错误、缩进问题、缺少冒号、括号或引号不匹配、使用关键字作为变量名等问题
  - **异常**
    - python中异常一般分为**内建异常**和**用户自定义异常**
      - a. **内建异常**都继承自 `BaseException` 类，主要有以下几种：
        - `ZeroDivisionError`： 尝试**除以零**时引发
        - `ValueError`： 当函数接收到一个**不适当或不合理的参数**时引发（不是类型错误，而是值错误）
        - `TypeError`： 当操作或函数作用于**不适当类型**的对象时引发
        - `NameError`： 尝试访问一个**未定义的变量**时引发
        - `IndexError`： 使用序列中**不存在的索引***时引发

- `KeyError`：请求字典中**不存在的键**时引发
- `AttributeError`：尝试访问**对象的属性或方法**时，该对象没有这样的属性或方法
- `ImportError`：无法导入**模块或包**时引发
- `RuntimeError`：在程序执行期间遇到的任何错误，它不是由其他特定异常类表示的

b. **自定义异常**通过创建一个继承自 `Exception` 类（或它的任何子类）的新类来实现，允许开发者为特定的错误情况定义清晰、易于理解的异常类型

- 示例如下：

```
class MyCustomError(Exception):     # 一个自定义的异常类
    def __init__(self, message="这是一个自定义错误"):
        self.message = message
        super().__init__(self.message)

try:
    raise MyCustomError("出错了！")  # 使用自定义异常
except MyCustomError as e:
    print(e)
```

- `assert` 语句(断言)
  - `assert <expression> [, arguments]`
  - 用于判断一个表达式，在表达式条件为 `false` 的时候触发 `AssertionError` 异常，断言可以在条件不满足程序运行的情况下直接返回错误，而不必等待程序运行后出现崩溃的情况
  - `assert` 语句主要用于调试目的，帮助开发者发现代码中的逻辑错误。在生产环境中，不建议依赖 `assert` 来处理所有可能的错误情况，因为通过命令行参数 `-O`（优化模式） 运行Python程序时，所有的 `assert` 语句都会被自动忽略
- `try\excrpt...else...finally` 语句(**异常处理**)

```
try:
    statement_1     # 执行try子句
except Error:       # 如果没有异常发生，忽略except子句，try子句执行后结束
    statement_2
    # 如果在执行try子句的过程中发生了异常，那么余下的部分将被忽略。如果异常的类型和e
else:
    statement_3     # else子句将在try子句没有发生任何异常的时候执行
finally:
    statement_4     # finally语句无论异常是否发生都会执行
```

- `raise` 语句(抛出异常)
  - `raise [Exception [, args [, traceback]]]`
  - `Exception`：这是必需的参数，表示要抛出的异常类型。它必须是一个异常类或异常的实例。
  - `args`：这是一个可选参数，用于传递给异常对象的参数。如果异常类有多个初始化参数，则需要提供这些参数。

- `traceback`：这是一个可选参数，用于显示异常的原始追踪信息（即异常发生的位置）。这个参数在大多数情况下不需要手动指定，Python会自动处理它

- `with` 语句(上下文管理器)
  - `with <expression> as <variable>:`
  - 能够写入 `with` 语句的必须实现以下两个方法 `__enter__()` & `__exit__()`
    a. `__enter__()`：在进入 `with` 代码块之前被调用，并返回该上下文管理器的对象（通常是 `self`），这个返回值将被赋值给 `as` 子句后面的变量
    b. `__exit__()`：在离开 `with` 代码块时调用。它接收三个参数，分别代表异常的类型、值和追踪信息（traceback）。如果代码块正常执行，这三个参数都将是 `None`。如果代码块中发生了异常，并且 `__exit__` 方法返回 `True`，则异常会被忽略（即被捕获并处理掉了），否则异常会被正常抛出
- **示例**

```
class MyContextManager:
    def __enter__(self):
        print("Entering the context")
        return self                     # 通常返回自身或相关资源对象

    def __exit__(self, exc_type, exc_val, exc_tb):
        print("Exiting the context")
        if exc_type is ValueError:      # 忽略ValueError异常
            print("Ignoring ValueError")
            return True
        return False                    # 对于其他异常，不忽略

with MyContextManager() as cm:          # 使用上下文管理器
    raise ValueError("Something went wrong")
    print("This line would not be reached if the ValueError was not ignored")  # 注意：
# 输出：
# Entering the context
# Exiting the context
# Ignoring ValueError

# 注意：由于 ValueError 被忽略了，所以程序没有因为异常而中断
```

ii. **面向对象编程(OOP)**

- **类(Class)**
  - 用来描述具有相同的属性和方法的对象的集合，具有内置的方法（函数）和属性（变量）
  - 使用 `class` 关键字定义
  - 命名遵循**大驼峰法**(组成名称的各个单词首字母均大写)
- **对象(Object)**
  - 这里的对象特指**类实例(Instance)**，是**类**实例化的结果
  - 语法格式为 `Object = ClassName(,args,)`
- **属性(Attribute)**
  - **类变量**：在类体中，所有函数之外定义的变量，可通过类或者类实例来访问

        - **局部变量**：在类体中，所有函数内部里面以 `变量名 = 变量值` 的方式定义的变量
        - **实例变量**：在类体中，所有函数内部以 `self.变量名` 的方式定义的变量，实例变量只能通过对象名访问，无法通过类名访问
        - **私有变量**： `__private_attrs` 两个下划线开头，声明该属性为私有，不能在类的外部被使用或直接访问,可以通过 `get()` 函数或
    - **方法(Method)**
        - **类实例方法**：至少含一个 `self` 参数，用于绑定调用此方法的实例对象
            - 使用类对象直接调用实例方法，无需手动给 `self` 参数传值（绑定方法） `odj.method(,args,)`；使用类名调用实例方法，需要手动给 `self` 参数传值（非绑定方法） `ClassName.method(obj,args,)`
        - **类方法**：采用 `@classmethod` 修饰，至少包含一个 `cls` 参数
        - **类静态方法**：采用 `@staticmethod` 修饰，和函数唯一的区别是静态方法定义在类这个空间中，而函数则定义在程序所在的空间中，且无需任何特殊参数
        - **私有方法**： `__private_method` 两个下划线开头，声明该方法为私有方法，只能在类的内部调用，不能在类的外部调用
    - **类间关系**

        a. **继承**
        - 定义一个类 **（子类或派生类）** 来继承另一个类 **（父类或基类）** 的属性和方法
        - **子类或派生类**继承**父类或基类**的属性和方法时可以**单继承或多继承，若是父类中有相同的方法名，而在子类使用时未指定，python从左到右查找多个父类中是否包含方法

        > (钻石继承和MFO有点复杂，时间不够暂时没完全搞懂)

        b. **组合**

        c. **关联**

        d. **依赖**

# 8.27 8.28

# 1. python语法

- 有点难的语法

    i. **Numpy**
    - **数据类型对象(dtype)**
        - 直接查看数组的 `dtype`
        - 使用 `ndarry.astype()` 修改数组的 `dtype`
        - 从内置类型(int、float等)创建 `dtype`
        - 创建包含多个字段的结构化 `dtype` ，这样生成的数组每个元素可以看做成一个“记录”，包含多个字段，每个字段都可以存储不同类型的数据，并且可以直接访问数组对象的字段
    - **数组属性**

- `ndarray.ndim`： 数组的秩(rank)，即数组的维度数量或轴的数量
- `ndarray.shape`： 数组的维度，表示数组在每个轴上的大小
- `ndarray.dtype`： 数组中元素的数据类型

- **数组创建**
  - `numpy.array`： 完整版为 `numpy.array(object, dtype = None, copy = True, order = None, subok = False, ndmin = 0)`，还可以将Python的列表（list）、元组（tuple）等序列类型的数据转换为数组
    - `object`： 数组或嵌套的数列（多维数组创建与C语言相同）
    - `dtype`： 数组元素的数据类型，可选
    - `copy`： 对象是否需要复制，可选
    - `order`： 创建数组的样式，C为行方向，F为列方向，A为任意方向（默认）
  - `numpy.empty`： `numpy.empty(shape, dtype = float, order = 'C')`，用来创建一个指定形状、数据类型且未初始化的数组
    - `shape`： 数组形状
  - `numpy.zeros`： `numpy.zeros(shape, dtype = float, order = 'C')`，创建指定大小的数组，数组元素以0来填充
  - `numpy.ones`： `numpy.ones(shape, dtype = None, order = 'C')`，同上，元素为1
  - `numpy.zeros_like`： `numpy.zeros_like(a, dtype=None, order='K', subok=True, shape=None)`，创建一个与给定数组具有相同形状的数组，数组元素以0来填充
    - `a`： 给定要创建相同形状的数组
  - `numpy.ones_like`： `numpy.ones_like(a, dtype=None, order='K', subok=True, shape=None)`，同上，元素填充1
  - `numpy.fromiter`： `numpy.fromiter(iterable, dtype, count=-1)`，利用可迭代对象建立一维数组
- **数组的线性代数运算**
  - `numpy.dot()`： `numpy.dot(a, b, out=None)`，计算两个数组的点积
  - `numpy.inner()`： 函数返回一维数组的向量内积。对于更高的维度，它返回最后一个轴上的和的乘积
  - `numpy.linalg.det()`： 函数返回给定数组的行列式（float 或者 complex）
  - `numpy.linalg.solve()`： 函数给出矩阵形式的线性方程的解

ii. **OpenCV**

- `cv.VideoCapture()`
  - 参数可以是设备索引或者一个视频文件路径，
- `cap.isOpened()`
  - 检查视频捕获是否已成功打开
- `cap.read()`
  - 从视频捕获设备（如摄像头）或视频文件中读取下一帧
- `cap.open()`
  - 初始化一个视频捕获对象，通常是通过指定一个设备的索引（比如0代表计算机的默认摄像头）或者一个视频文件的路径来完成的，这个对象应当是一个 `VideoCapture` 对象
- `cv.imshow()`

        - 在窗口中显示图像，窗口自动适应图像的大小，第一个参数为窗口名，第二个为显示对象
    - `cv.imwrite()`
        - 保存图像，第一个参数是文件名，第二个参数是要保存的图像
    - `cv.destroyAllWindows()`
        - 销毁创建的所有窗口。如果想销毁任意指定窗口，使用函数 `cv.destroyWindow()`，参数是确切的窗口名
    - `cv.waitKey()`
        - 函数用于等待键盘输入,检查是否有按键被按下，参数表示等待时间（毫秒）
    - `cap.release()`
        - 释放与 `VideoCapture` 对象关联的资源，避免内存溢出

# 8.29 8.30 8.31 9.1

# 单片机入门

> 学习内容见STM32.md

1. 脚注在这里鸭~ ↩