

基础语法(3)

函数

函数是什么

编程中的函数和数学中的函数有一定的相似之处.

数学上的函数, 比如 $y = \sin x$, x 取不同的值, y 就会得到不同的结果.

编程中的函数, 是一段 可以被重复使用的代码片段 .

代码示例: 求数列的和, 不使用函数

```
# 1. 求 1 - 100 的和
sum = 0
for i in range(1, 101):
    sum += i
print(sum)

# 2. 求 300 - 400 的和
sum = 0
for i in range(300, 401):
    sum += i
print(sum)

# 3. 求 1 - 1000 的和
sum = 0
for i in range(1, 1001):
    sum += i
print(sum)
```

可以发现, 这几组代码基本是相似的, 只有一点点差异. 可以把重复代码提取出来, 做成一个函数

实际开发中, 复制粘贴是一种不太好的策略. 实际开发的重复代码可能存在几十份甚至上百份.

一旦这个重复代码需要被修改, 那就得改几十次, 非常不便于维护.

代码示例: 求 数列 的和, 使用函数

```
# 定义函数
def calcSum(beg, end):
    sum = 0
    for i in range(beg, end + 1):
        sum += i
    print(sum)

# 调用函数
sum(1, 100)
sum(300, 400)
sum(1, 1000)
```

可以明显看到, 重复的代码已经被消除了.

语法格式

创建函数/定义函数

```
def 函数名(形参列表):
    函数体
    return 返回值
```

调用函数/使用函数

```
函数名(实参列表)          // 不考虑返回值
返回值 = 函数名(实参列表) // 考虑返回值
```

- 函数定义并不会执行函数体内容, 必须要调用才会执行. 调用几次就会执行几次

```
def test1():
    print('hello')
```

如果光是定义函数, 而不调用, 则不会执行.

- 函数必须先定义, 再使用.

```
test3()          # 还没有执行到定义, 就先执行调用了, 此时就会报错.
```

```
def test3():
    print('hello')
```

NameError: name 'test3' is not defined

动漫里释放技能之前, 需要大喊招式的名字, 就是 "先定义, 再使用".



函数参数

在函数定义的时候, 可以在 () 中指定 "形式参数" (简称 **形参**), 然后在调用的时候, 由调用者把 "实际参数" (简称 **实参**) 传递进去.

这样就可以做到一份函数, 针对不同的数据进行计算处理.

考虑前面的代码案例:

```
def calcSum(beg, end):  
    sum = 0  
    for i in range(beg, end + 1):  
        sum += i  
    print(sum)  
  
sum(1, 100)  
sum(300, 400)  
sum(1, 1000)
```

上面的代码中, `beg, end` 就是函数的形参. `1, 100` / `300, 400` 就是函数的实参.

在执行 `sum(1, 100)` 的时候, 就相当于 `beg = 1, end = 100`, 然后在函数内部就可以针对 1-100 进行运算.

在执行 `sum(300, 400)` 的时候, 就相当于 `beg = 300, end = 400`, 然后在函数内部就可以针对 300-400 进行运算.

实参和形参之间的关系, 就像签合同一样.

甲方(出卖人): **汤老湿**

身份证号码:

乙方(买受人): **蔡徐坤**

身份证号码:

甲、乙、双方依据自愿、诚信原则,经友好协商,达成如下一致意见:

第一条 各方承诺及保证

甲方承诺并保证:拥有座落于_____房屋的所有权,该项所有权是完整的,不存在共有权、抵押权、承租权等权利瑕疵或负担。将诚信履行本合同书的项下的合同义务。

乙方承诺并保证:将诚信履行本合同书的项下的合同义务。

第二条 标的物

甲方自愿转让的房屋座落于_____

房屋内现有装修及物品全部随房屋转让并不再另外计价。

第三条 价款

本合同书第二条项下标的物全部计价_____。

各方确认:在甲方及时履行本合同书第六条义务的前提下,本条第一款的价款为甲方转让标的物的净价,即本合同书履行过程中产生的税费由乙方承

甲方, 乙方 这就相当于形参. 汤老湿, 蔡徐坤 就是实参.

```
def 签合同(甲方, 乙方):  
    合同内容....  
  
签合同('汤老湿', '蔡徐坤')  
签合同('汤老湿', '鹿晗')  
签合同('汤老湿', '吴磊')
```

注意:

- 一个函数可以有一个形参,也可以有多个形参,也可以没有形参.
- 一个函数的形参有几个,那么传递实参的时候也得传几个. 保证个数要匹配.

```
def test(a, b, c):  
    print(a, b, c)  
  
test(10)
```

```
TypeError: test() missing 2 required positional arguments: 'b' and 'c'
```

- 和 C++ / Java 不同, Python 是动态类型的编程语言, 函数的形参不必指定参数类型. 换句话说, 一个函数可以支持多种不同类型的参数.

```
def test(a):  
    print(a)  
  
test(10)  
test('hello')  
test(True)
```

```
10
hello
True
```

函数返回值

函数的参数可以视为是函数的 "输入", 则函数的返回值, 就可以视为是函数的 "输出".

此处的 "输入", "输出" 是更广义的输入输出, 不是单纯指通过控制台输入输出.

我们可以把函数想象成一个 "工厂". 工厂需要买入原材料, 进行加工, 并生产出产品.

函数的参数就是原材料, 函数的返回值就是生产出的产品.

下列代码

```
def calcSum(beg, end):
    sum = 0
    for i in range(beg, end + 1):
        sum += i
    print(sum)

calc(1, 100)
```

可以转换成

```
def calcSum(beg, end):
    sum = 0
    for i in range(beg, end + 1):
        sum += i
    return sum

result = calcSum(1, 100)
print(result)
```

这两个代码的区别就在于, 前者直接在函数内部进行了打印, 后者则使用 `return` 语句把结果返回给函数调用者, 再由调用者负责打印.

我们一般倾向于第二种写法.

实际开发中我们的一个通常的编程原则, 是 "逻辑和用户交互分离". 而第一种写法的函数中, 既包含了计算逻辑, 又包含了和用户交互(打印到控制台上). 这种写法是不太好的, 如果后续我们需要的是把计算结果保存到文件中, 或者通过网络发送, 或者展示到图形化界面里, 那么第一种写法的函数, 就难以胜任了.

而第二种写法则专注于做计算逻辑, 不负责和用户交互. 那么就很容易把这个逻辑搭配不同的用户交互代码, 来实现不同的效果.

- 一个函数中可以有多条 `return` 语句

```
# 判定是否是奇数
def isOdd(num):
    if num % 2 == 0:
        return False
    else:
        return True

result = isOdd(10)
print(result)
```

- 执行到 `return` 语句, 函数就会立即执行结束, 回到调用位置.

```
# 判定是否是奇数
def isOdd(num):
    if num % 2 == 0:
        return False
    return True

result = isOdd(10)
print(result)
```

如果 `num` 是偶数, 则进入 `if` 之后, 就会触发 `return False`, 也就不会继续执行 `return True`

- 一个函数是可以一次返回多个返回值的. 使用 `,` 来分割多个返回值.

```
def getPoint():
    x = 10
    y = 20
    return x, y

a, b = getPoint()
```

- 如果只想关注其中的部分返回值, 可以使用 `_` 来忽略不想要的返回值.

```
def getPoint():
    x = 10
    y = 20
    return x, y

_, b = getPoint()
```

变量作用域

观察以下代码

```
def getPoint():  
    x = 10  
    y = 20  
    return x, y  
  
x, y = getPoint()
```

在这个代码中, 函数内部存在 `x, y`, 函数外部也有 `x, y`.

但是这两组 `x, y` 不是相同的变量, 而只是恰好有一样的名字.

变量只能在所在的函数内部生效.

在函数 `getPoint()` 内部定义的 `x, y` 只是在函数内部生效. 一旦出了函数的范围, 这两个变量就不再生效了.

```
def getPoint():  
    x = 10  
    y = 20  
    return x, y  
  
getPoint()  
print(x, y)
```

```
NameError: name 'x' is not defined
```

在不同的作用域中, 允许存在同名的变量

虽然名字相同, 实际上是不同的变量.

```
x = 20  
  
def test():  
    x = 10  
    print(f'函数内部 x = {x}')  
test()  
print(f'函数外部 x = {x}')
```

```
函数内部 x = 10  
函数外部 x = 20
```

注意:

- 在函数内部的变量, 也称为 "局部变量"
- 不在任何函数内部的变量, 也称为 "全局变量"

如果函数内部尝试访问的变量在局部不存在, 就会尝试去全局作用域中查找

```
x = 20

def test():
    print(f'x = {x}')

test()
```

x = 20

如果是想在函数内部, 修改全局变量的值, 需要使用 global 关键字声明

```
x = 20

def test():
    global x
    x = 10
    print(f'函数内部 x = {x}')

test()
print(f'函数外部 x = {x}')
```

函数内部 x = 10
函数外部 x = 10

如果此处没有 `global`, 则函数内部的 `x = 10` 就会被视为是创建一个局部变量 `x`, 这样就和全局变量 `x` 不相关了.

`if / while / for` 等语句块不会影响到变量作用域

换言之, 在 `if / while / for` 中定义的变量, 在语句外面也可以使用.

```
for i in range(1, 10):
    print(f'函数内部 i = {i}')

print(f'函数外部 i = {i}')
```


函数执行过程

- 调用函数才会执行函数体代码, 不调用则不会执行.
- 函数体执行结束(或者遇到 return 语句), 则回到函数调用位置, 继续往下执行.

```
def test():  
    print("执行函数内部代码")  
    print("执行函数内部代码")  
    print("执行函数内部代码")  
  
print("1111")  
test()  
print("2222")  
test()  
print("3333")
```

1111

执行函数内部代码

执行函数内部代码

执行函数内部代码

2222

执行函数内部代码

执行函数内部代码

执行函数内部代码

3333

这个过程还可以使用 PyCharm 自带的调试器来观察.

- 点击行号右侧的空白, 可以在代码中插入 **断点**
- 右键, Debug, 可以按照调试模式执行代码. 每次执行到断点, 程序都会暂停下来.
- 使用 **Step Into** (F7) 功能可以逐行执行代码.

链式调用

前面的代码很多都是写作

```
# 判定是否是奇数
def isOdd(num):
    if num % 2 == 0:
        return False
    else:
        return True

result = isOdd(10)
print(result)
```

实际上也可以简化写作

```
print(isodd(10))
```

把一个函数的返回值, 作为另一个函数的参数, 这种操作称为 **链式调用**.

这是一种比较常见的写法.

嵌套调用

函数内部还可以调用其他的函数, 这个动作称为 "嵌套调用".

```
def test():
    print("执行函数内部代码")
    print("执行函数内部代码")
    print("执行函数内部代码")
```

test 函数内部调用了 print 函数, 这里就属于嵌套调用.

一个函数里面可以嵌套调用任意多个函数.

函数嵌套的过程是非常灵活的.

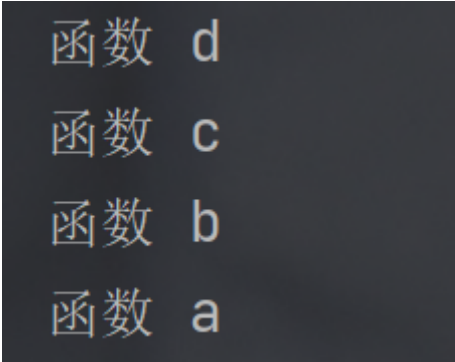
```
def a():
    print("函数 a")

def b():
    print("函数 b")
    a()

def c():
    print("函数 c")
    b()

def d():
    print("函数 d")
    c()

d()
```



```
函数 d
函数 c
函数 b
函数 a
```

如果把代码稍微调整, 打印结果则可能发生很大变化.

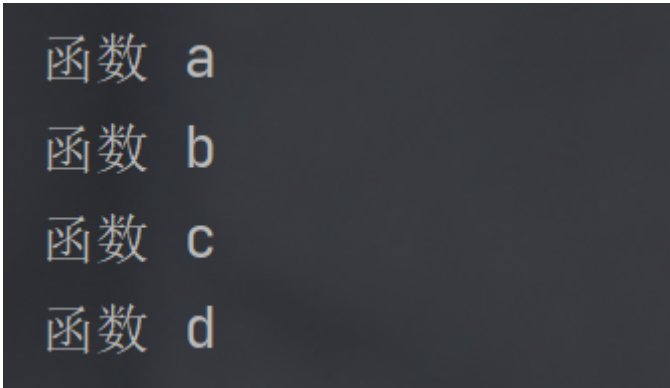
```
def a():
    print("函数 a")

def b():
    a()
    print("函数 b")

def c():
    b()
    print("函数 c")

def d():
    c()
    print("函数 d")

d()
```



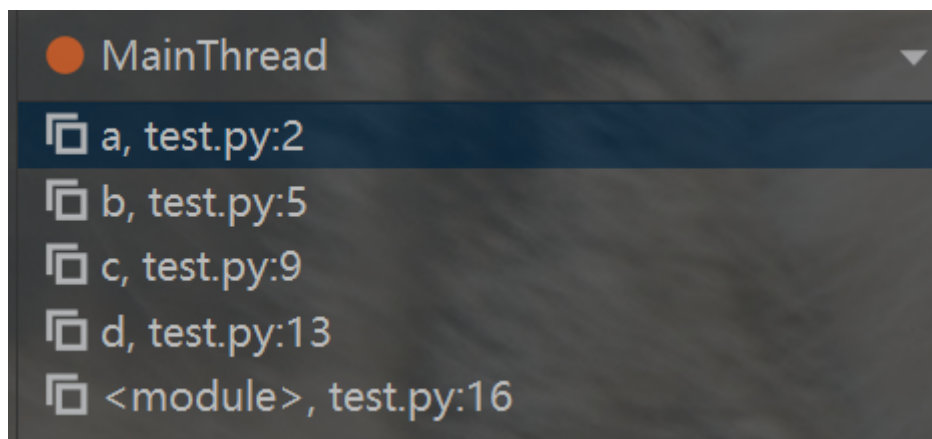
```
函数 a
函数 b
函数 c
函数 d
```

注意体会上述代码的执行顺序, 可以通过画图的方式来理解.

函数之间的调用关系, 在 Python 中会使用一个特定的数据结构来表示, 称为 **函数调用栈**. 每次函数调用, 都会在调用栈里新增一个元素, 称为 **栈帧**.

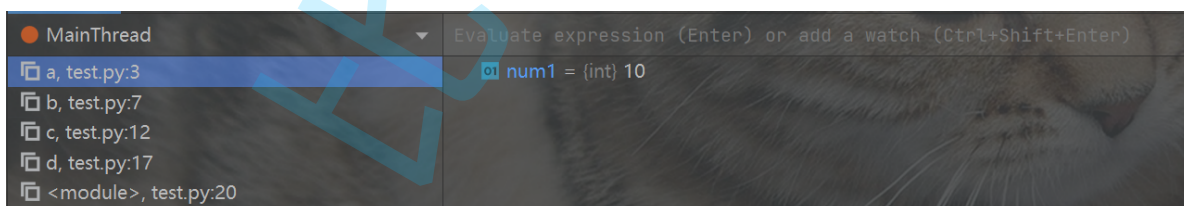
可以通过 PyCharm 调试器看到函数调用栈和栈帧.

在调试状态下, PyCharm 左下角一般就会显示出函数调用栈.



每个函数的局部变量, 都包含在自己的栈帧中

```
def a():  
    num1 = 10  
    print("函数 a")  
  
def b():  
    num2 = 20  
    a()  
    print("函数 b")  
  
def c():  
    num3 = 30  
    b()  
    print("函数 c")  
  
def d():  
    num4 = 40  
    c()  
    print("函数 d")  
  
d()
```



选择不同的栈帧, 就可以看到各自栈帧中的局部变量.

思考: 上述代码, a, b, c, d 函数中的局部变量名各不相同. 如果变量名是相同的, 比如都是 `num`, 那么这四个函数中的 `num` 是属于同一个变量, 还是不同变量呢?

函数递归

递归是 嵌套调用 中的一种特殊情况, 即一个函数嵌套调用自己.

代码示例: 递归计算 5!

```
def factor(n):  
    if n == 1:  
        return 1  
    return n * factor(n - 1)  
  
result = factor(5)  
print(result)
```

上述代码中, 就属于典型的递归操作. 在 factor 函数内部, 又调用了 factor 自身.

注意: 递归代码务必要保证

- 存在递归结束条件. 比如 `if n == 1` 就是结束条件. 当 `n` 为 1 的时候, 递归就结束了.
- 每次递归的时候, 要保证函数的实参是逐渐逼近结束条件的.

如果上述条件不能满足, 就会出现 "无限递归". 这是一种典型的代码错误.

```
def factor(n):  
    return n * factor(n - 1)  
  
result = factor(5)  
print(result)
```

RecursionError: maximum recursion depth exceeded

如前面所描述, 函数调用时会在函数调用栈中记录每一层函数调用的信息.

但是函数调用栈的空间不是无限大的. 如果调用层数太多, 就会超出栈的最大范围, 导致出现问题.

递归的优点

- 递归类似于 "数学归纳法", 明确初始条件, 和递推公式, 就可以解决一系列的问题.
- 递归代码往往代码量非常少.

递归的缺点

- 递归代码往往难以理解, 很容易超出掌控范围
- 递归代码容易出现栈溢出的情况
- 递归代码往往可以转换成等价的循环代码. 并且通常来说循环版本的代码执行效率要略高于递归版本.

实际开发的时候, 使用递归要慎重!

参数默认值

Python 中的函数, 可以给形参指定默认值.

带有默认值的参数, 可以在调用的时候不传参.

代码示例: 计算两个数字的和

```
def add(x, y, debug=False):  
    if debug:  
        print(f'调试信息: x={x}, y={y}')    return x + y  
  
print(add(10, 20))  
print(add(10, 20, True))
```

此处 `debug=False` 即为参数默认值. 当我们不指定第三个参数的时候, 默认 `debug` 的取值即为 `False`.

带有默认值的参数需要放到没有默认值的参数的后面

```
def add(x, debug=False, y):  
    if debug:  
        print(f'调试信息: x={x}, y={y}')    return x + y  
  
print(add(10, 20))
```

```
def add(x, debug=False, y):  
    ^  
SyntaxError: non-default argument follows default argument
```

关键字参数

在调用函数的时候, 需要给函数指定实参. 一般默认情况下是按照形参的顺序, 来依次传递实参的.

但是我们也可以通过 **关键字参数**, 来调整这里的传参顺序, 显式指定当前实参传递给哪个形参.

```
def test(x, y):  
    print(f'x = {x}')    print(f'y = {y}')  
test(x=10, y=20)  
test(y=100, x=200)
```

```
x = 10
y = 20
x = 200
y = 100
```

形如上述 `test(x=10, y=20)` 这样的操作, 即为 **关键字参数**.

小结

函数是编程语言中的一个核心语法机制. Python 中的函数和大部分编程语言中的函数功能都是基本类似的.

我们当下最关键要理解的主要就是三个点:

- 函数的定义
- 函数的调用
- 函数的参数传递

我们在后续的编程中, 会广泛的使用到函数. 大家在练习的过程中再反复加深对于函数的理解.

列表和元组

列表是什么, 元组是什么

编程中, 经常需要使用变量, 来保存/表示数据.

如果代码中需要表示的数据个数比较少, 我们直接创建多个变量即可.

```
num1 = 10
num2 = 20
num3 = 30
.....
```

但是有的时候, 代码中需要表示的数据特别多, 甚至也不知道要表示多少个数据. 这个时候, 就需要用到列表.

列表是一种让程序猿在代码中批量表示/保存数据的方式

就像我们去超市买辣条, 如果就只是买一两根辣条, 那咱们直接拿着辣条就走了.

但是如果一次买个十根八根的, 这个时候用手拿就不好拿, 超市老板就会给我们个袋子.

这个袋子, 就相当于 **列表**

元组和列表相比, 是非常相似的, 只是列表中放哪些元素可以修改调整, 元组中放的元素是创建元组的时候就设定好的, 不能修改调整.

列表就是买散装辣条, 装好了袋子之后, 随时可以把袋子打开, 再往里多加辣条或者拿出去一些辣条.

元组就是买包装辣条, 厂家生产好了辣条之后, 一包就是固定的这么多, 不能变动了.



散装辣条
(列表)



包装辣条
(元组)

创建列表

- 创建列表主要有两种方式. `[]` 表示一个空的列表.

```
alist = []  
  
alist = list()  
  
print(type(alist))
```

- 如果需要往里面设置初始值, 可以直接写在 `[]` 当中.

可以直接使用 `print` 来打印 `list` 中的元素内容.

```
alist = [1, 2, 3, 4]  
print(alist)
```

- 列表中存放的元素允许是不同的类型. (这一点和 C++ Java 差别较大).

```
alist = [1, 'hello', True]  
print(alist)
```

因为 `list` 本身是 Python 中的内建函数, 不宜再使用 `list` 作为变量名, 因此命名为 `alist`

访问下标

- 可以通过下标访问操作符 `[]` 来获取到列表中的任意元素.

我们把 `[]` 中填写的数字, 称为 下标 或者 索引 .


```
alist = [1, 2, 3, 4]
print(alist[2])
```

3

注意: 下标是从 0 开始计数的, 因此下标为 2, 则对应着 3 这个元素.

- 通过下标不光能读取元素内容, 还能修改元素的值.

```
alist = [1, 2, 3, 4]
alist[2] = 100
print(alist)
```

- 如果下标超出列表的有效范围, 会抛出异常.

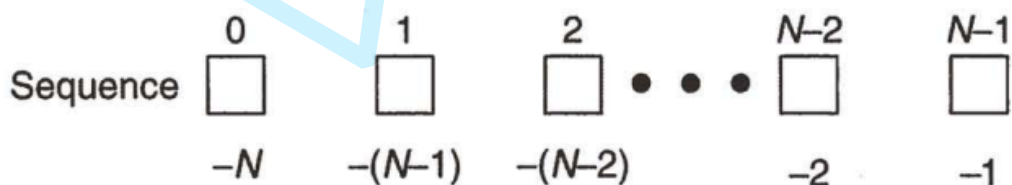
```
alist = [1, 2, 3, 4]
print(alist[100])
```

IndexError: list index out of range

- 因为下标是从 0 开始的, 因此下标的有效范围是 $[0, \text{列表长度} - 1]$. 使用 `len` 函数可以获取到列表的元素个数.

```
alist = [1, 2, 3, 4]
print(len(alist))
```

- 下标可以取负数. 表示 "倒数第几个元素"



$N == \text{length of sequence} == \text{len}(\text{sequence})$

```
alist = [1, 2, 3, 4]
print(alist[3])
print(alist[-1])
```

`alist[-1]` 相当于 `alist[len(alist) - 1]`

切片操作

通过下标操作是一次取出里面第一个元素.

通过切片, 则是一次取出一组连续的元素, 相当于得到一个 **子列表**

- 使用 `[:]` 的方式进行切片操作.

```
alist = [1, 2, 3, 4]
print(alist[1:3])
```

```
[2, 3]
```

`alist[1:3]` 中的 `1:3` 表示的是 `[1, 3)` 这样的由下标构成的前闭后开区间.

也就是从下标为 1 的元素开始(2), 到下标为 3 的元素结束(4), 但是不包含下标为 3 的元素.

所以最终结果只有 `[2, 3]`

- 切片操作中可以省略前后边界

```
alist = [1, 2, 3, 4]
print(alist[1:])      # 省略后边界, 表示获取到列表末尾
print(alist[:-1])     # 省略前边界, 表示从列表开头获取
print(alist[:])       # 省略两个边界, 表示获取到整个列表.
```

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

- 切片操作还可以指定 "步长", 也就是 "每访问一个元素后, 下标自增几步"

```
alist = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
print(alist[::1])
print(alist[::2])
print(alist[::3])
print(alist[::5])
```

```
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
[1, 3, 5, 7, 9]
[1, 4, 7, 10]
[1, 6]
```

- 切片操作指定的步长还可以是负数, 此时是从后往前进行取元素. 表示 "每访问一个元素之后, 下标自减几步"

```
alist = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
print(alist[::-1])
print(alist[::-2])
print(alist[::-3])
print(alist[::-5])
```

```
[10, 9, 8, 7, 6, 5, 4, 3, 2, 1]
[10, 8, 6, 4, 2]
[10, 7, 4, 1]
[10, 5]
```

- 如果切片中填写的数字越界了, 不会有负面效果. 只会尽可能的把满足条件的元素过去到.

```
alist = [1, 2, 3, 4]
print(alist[100:200])
```

```
[]
```

遍历列表元素

"遍历" 指的是把元素一个一个的取出来, 再分别进行处理.

- 最简单的办法就是使用 for 循环

```
alist = [1, 2, 3, 4]

for elem in alist:
    print(elem)
```

- 也可以使用 for 按照范围生成下标, 按下标访问

```
alist = [1, 2, 3, 4]

for i in range(0, len(alist)):
    print(alist[i])
```

- 还可以使用 while 循环. 手动控制下标的变化

```
alist = [1, 2, 3, 4]

i = 0
while i < len(alist):
    print(alist[i])
    i += 1
```

新增元素

- 使用 `append` 方法, 向列表末尾插入一个元素(尾插).

```
alist = [1, 2, 3, 4]
alist.append('hello')
print(alist)
```

```
[1, 2, 3, 4, 'hello']
```

- 使用 `insert` 方法, 向任意位置插入一个元素

`insert` 第一个参数表示要插入元素的下标.

```
alist = [1, 2, 3, 4]
alist.insert(1, 'hello')
print(alist)
```

```
[1, 'hello', 2, 3, 4]
```

PS: 什么是 "方法" (method)

方法其实就是函数. 只不过函数是独立存在的, 而方法往往要依附于某个 "对象".

像上述代码 `alist.append`, `append` 就是依附于 `alist`, 相当于是 "针对 `alist` 这个列表, 进行尾插操作".

查找元素

- 使用 `in` 操作符, 判定元素是否在列表中存在. 返回值是布尔类型.

```
alist = [1, 2, 3, 4]
print(2 in alist)
print(10 in alist)
```

```
True
False
```

- 使用 `index` 方法, 查找元素在列表中的下标. 返回值是一个整数. 如果元素不存在, 则会抛出异常.

```
alist = [1, 2, 3, 4]
print(alist.index(2))
print(alist.index(10))
```

删除元素

- 使用 `pop` 方法删除最末尾元素

```
alist = [1, 2, 3, 4]
alist.pop()
print(alist)
```

```
[1, 2, 3]
```

- `pop` 也能按照下标来删除元素

```
alist = [1, 2, 3, 4]
alist.pop(2)
print(alist)
```

```
[1, 2, 4]
```

- 使用 `remove` 方法, 按照值删除元素.

```
alist = [1, 2, 3, 4]
alist.remove(2)
print(alist)
```

```
[1, 3, 4]
```

连接列表

- 使用 `+` 能够把两个列表拼接在一起.

此处的 `+` 结果会生成一个新的列表. 而不会影响到旧列表的内容.

```
alist = [1, 2, 3, 4]
blist = [5, 6, 7]
print(alist + blist)
```

```
[1, 2, 3, 4, 5, 6, 7]
```

- 使用 `extend` 方法, 相当于把一个列表拼接 to 另一个列表的后面.

`a.extend(b)`, 是把 `b` 中的内容拼接 to `a` 的末尾. 不会修改 `b`, 但是会修改 `a`.

```
alist = [1, 2, 3, 4]
blist = [5, 6, 7]
alist.extend(blist)
print(alist)
print(blist)
```

```
[1, 2, 3, 4, 5, 6, 7]
[5, 6, 7]
```

关于元组

元组的功能和列表相比, 基本是一致的.

元组使用 `()` 来表示.

```
atuple = ( )
atuple = tuple()
```

元组不能修改里面的元素, 列表则可以修改里面的元素

因此, 像读操作, 比如访问下标, 切片, 遍历, `in`, `index`, `+` 等, 元组也是一样支持的.

但是, 像写操作, 比如修改元素, 新增元素, 删除元素, `extend` 等, 元组则不能支持.

另外, 元组在 Python 中很多时候是默认的集合类型. 例如, 当一个函数返回多个值的时候.

```
def getPoint():
    return 10, 20

result = getPoint()
print(type(result))
```

```
<class 'tuple'>
```

此处的 `result` 的类型, 其实是元组.

问题来了, 既然已经有了列表, 为啥还需要有元组?

元组相比于列表来说, 优势有两方面:

- 你有一个列表, 现在需要调用一个函数进行一些处理. 但是你有不是特别确认这个函数是否会把你的列表数据弄乱. 那么这时候传一个元组就安全很多.
- 我们马上要讲的字典, 是一个键值对结构. 要求字典的键必须是 "可hash对象" (字典本质上也是一个hash表). 而一个可hash对象的前提就是不可变. 因此元组可以作为字典的键, 但是列表不行.

小结

列表和元组都是日常开发最常用到的类型. 最核心的操作就是根据 [] 来按下标操作.

在需要表示一个 "序列" 的场景下, 就可以考虑使用列表和元组.

如果元素不需要改变, 则优先考虑元组.

如果元素需要改变, 则优先考虑列表.

字典

字典是什么

字典是一种存储 **键值对** 的结构.

啥是**键值对**? 这是计算机/生活中一个非常广泛使用的概念.

把 键(key) 和 值(value) 进行一个一对一的映射, 然后就可以根据键, 快速找到值.

举个栗子, 学校的每个同学, 都会有一个唯一的学号.

知道了学号, 就能确定这个同学.

此处 "学号" 就是 "键", 这个 "同学" 就是 "值".

创建字典

- 创建一个空的字典. 使用 {} 表示字典.

```
a = { }  
b = dict()  
  
print(type(a))  
print(type(b))
```

- 也可以在创建的同时指定初始值
- 键值对之间使用 , 分割, 键和值之间使用 : 分割. (冒号后面推荐加一个空格).
- 使用 print 来打印字典内容

```
student = { 'id': 1, 'name': 'zhangsan' }  
print(student)
```

```
{'id': 1, 'name': 'zhangsan'}
```

- 为了代码更规范美观, 在创建字典的时候往往会把多个键值对, 分成多行来书写.

```
student = {  
    'id': 1,  
    'name': 'zhangsan'  
}
```

- 最后一个键值对, 后面可以写逗号, 也可以不写.

```
student = {  
    'id': 1,  
    'name': 'zhangsan',  
}
```

查找 key

- 使用 `in` 可以判定 key 是否在字典中存在. 返回布尔值.

```
student = {  
    'id': 1,  
    'name': 'zhangsan',  
}  
  
print('id' in student)  
print('score' in student)
```

```
True  
False
```

- 使用 `[]` 通过类似于取下标的方式, 获取到元素的值. 只不过此处的 "下标" 是 key. (可能是整数, 也可能是字符串等其他类型).

```
student = {  
    'id': 1,  
    'name': 'zhangsan',  
}  
  
print(student['id'])  
print(student['name'])
```



```
1
zhangsan
```

- 如果 key 在字典中不存在, 则会抛出异常.

```
student = {
    'id': 1,
    'name': 'zhangsan',
}

print(student['score'])
```

```
KeyError: 'score'
```

新增/修改元素

使用 `[]` 可以根据 key 来新增/修改 value.

- 如果 key 不存在, 对取下标操作赋值, 即为新增键值对

```
student = {
    'id': 1,
    'name': 'zhangsan',
}

student['score'] = 90
print(student)
```

```
{'id': 1, 'name': 'zhangsan', 'score': 90}
```

- 如果 key 已经存在, 对取下标操作赋值, 即为修改键值对的值.

```
student = {
    'id': 1,
    'name': 'zhangsan',
    'score': 80
}

student['score'] = 90
print(student)
```

```
{'id': 1, 'name': 'zhangsan', 'score': 90}
```

删除元素

- 使用 pop 方法根据 key 删除对应的键值对.

```
student = {  
    'id': 1,  
    'name': 'zhangsan',  
    'score': 80  
}  
  
student.pop('score')  
print(student)
```

```
{'id': 1, 'name': 'zhangsan'}
```

遍历字典元素

- 直接使用 for 循环能够获取到字典中的所有的 key, 进一步的就可以取出每个值了.

```
student = {  
    'id': 1,  
    'name': 'zhangsan',  
    'score': 80  
}  
  
for key in student:  
    print(key, student[key])
```

```
id 1  
name zhangsan  
score 80
```

取出所有 key 和 value

- 使用 keys 方法可以获取到字典中的所有的 key

```
student = {  
    'id': 1,  
    'name': 'zhangsan',  
    'score': 80  
}  
  
print(student.keys())
```

```
dict_keys(['id', 'name', 'score'])
```

此处 `dict_keys` 是一个特殊的类型, 专门用来表示字典的所有 key. 大部分元组支持的操作对于 `dict_keys` 同样适用.

- 使用 `values` 方法可以获取到字典中的所有 value

```
student = {  
    'id': 1,  
    'name': 'zhangsan',  
    'score': 80  
}  
  
print(student.values())
```

```
dict_values([1, 'zhangsan', 80])
```

此处 `dict_values` 也是一个特殊的类型, 和 `dict_keys` 类似.

- 使用 `items` 方法可以获取到字典中所有的键值对.

```
student = {  
    'id': 1,  
    'name': 'zhangsan',  
    'score': 80  
}  
  
print(student.items())
```

```
dict_items([('id', 1), ('name', 'zhangsan'), ('score', 80)])
```

此处 `dict_items` 也是一个特殊的类型, 和 `dict_keys` 类似.

合法的 key 类型

不是所有的类型都可以作为字典的 key.

字典本质上是一个 **哈希表**, 哈希表的 key 要求是 "可哈希的", 也就是可以计算出一个哈希值.

- 可以使用 `hash` 函数计算某个对象的哈希值.
- 但凡能够计算出哈希值的类型, 都可以作为字典的 key.

```
print(hash(0))
print(hash(3.14))
print(hash('hello'))
print(hash(True))
print(hash(()))          # ( ) 是一个空的元组
```

```
0
322818021289917443
-232464653545525718
1
5740354900026072187
```

- 列表无法计算哈希值.

```
print(hash([1, 2, 3]))
```

```
TypeError: unhashable type: 'list'
```

- 字典也无法计算哈希值

```
print(hash({'id': 1}))
```

```
TypeError: unhashable type: 'dict'
```

小结

字典也是一个常用的结构, 字典的所有操作都是围绕 key 来展开的.

需要表示 "键值对映射" 这种场景时就可以考虑使用字典.

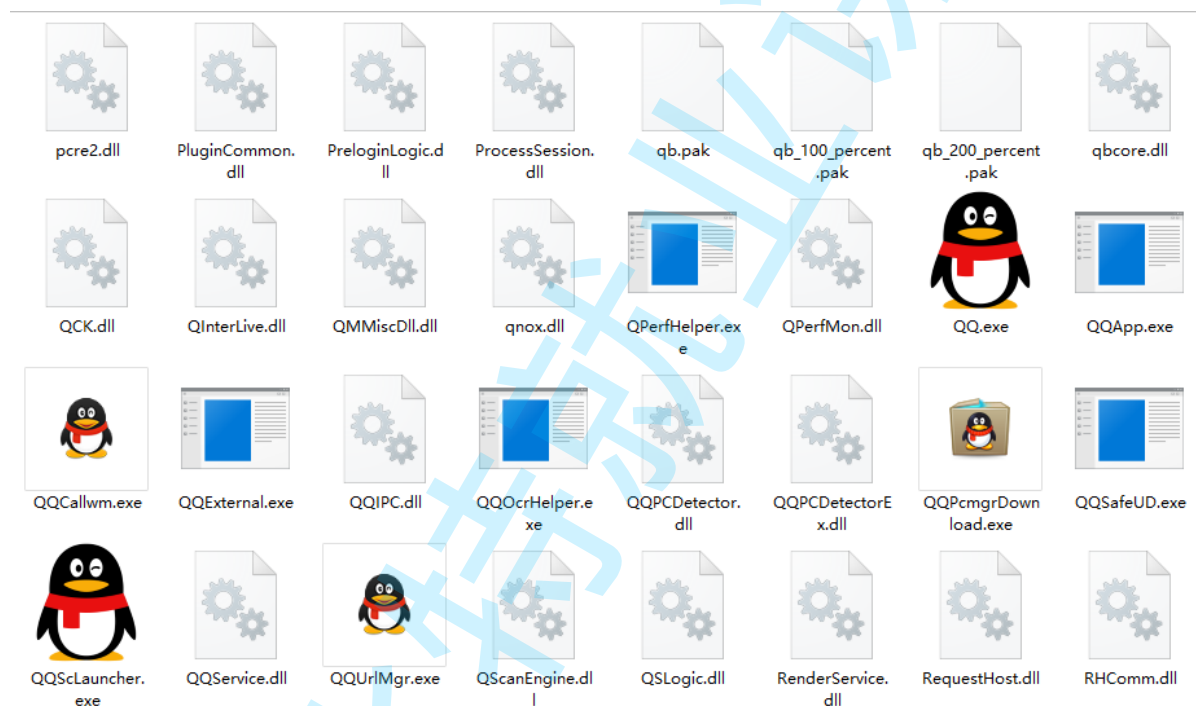
文件

文件是什么

变量是把数据保存到内存中. 如果程序重启/主机重启, 内存中的数据就会丢失.

要想能让数据被持久化存储, 就可以把数据存储到硬盘中. 也就是在 **文件** 中保存.

在 Windows "此电脑" 中, 看到的内容都是 文件.



通过文件的后缀名, 可以看到文件的类型. 常见的文件的类型如下:

- 文本文件 (txt)
- 可执行文件 (exe, dll)
- 图片文件 (jpg, gif)
- 视频文件 (mp4, mov)
- office 文件 (.ppt, docx)
-

咱们课堂上主要研究最简单的文本文件.

文件路径

一个机器上, 会存在很多文件, 为了让这些文件更方面的被组织, 往往会使用很多的 "文件夹"(也叫做**目录**) 来整理文件.

实际一个文件往往是放在一系列的目录结构之中的.

为了方便确定一个文件所在的位置, 使用 **文件路径** 来进行描述.

例如, 上述截图中的 QQ.exe 这个文件, 描述这个文件的位置, 就可以使用路径

`D:\program\qq\Bin\QQ.exe` 来表示.

- D: 表示 **盘符**. 不区分大小写.
- 每一个 `\` 表示一级目录. 当前 QQ.exe 就是放在 "D 盘下的 program 目录下的 qq 目录下的 Bin 目录中".
- 目录之间的分隔符, 可以使用 `\` 也可以使用 `/`. 一般在编写代码的时候使用 `/` 更方便.

上述以 盘符 开头的路径, 我们也称为 **绝对路径**.

除了绝对路径之外, 还有一种常见的表示方式是 **相对路径**. 相对路径需要先指定一个基准目录, 然后以基准目录为参照点, 间接的找到目标文件. 咱们课堂上暂时不详细介绍.

描述一个文件的位置, 使用 **绝对路径** 和 **相对路径** 都是可以的. 对于新手来说, 使用 **绝对路径** 更简单更好理解, 也不容易出错.

文件操作

要使用文件, 主要是通过文件来保存数据, 并且在后续把保存的数据读取出来.

但是要想读写文件, 需要先 "打开文件", 读写完毕之后还要 "关闭文件".

1. 打开文件

使用内建函数 `open` 打开一个文件.

```
f = open('d:/test.txt', 'r')
```

- 第一个参数是一个字符串, 表示要打开的文件路径
- 第二个参数是一个字符串, 表示打开方式. 其中 `r` 表示按照读方式打开. `w` 表示按照写方式打开. `a` 表示追加写方式打开.
- 如果打开文件成功, 返回一个文件对象. 后续的读写文件操作都是围绕这个文件对象展开.
- 如果打开文件失败(比如路径指定的文件不存在), 就会抛出异常.

```
FileNotFoundError: [Errno 2] No such file or directory:
```

2. 关闭文件

使用 `close` 方法关闭已经打开的文件。

```
f.close()
```

使用完毕的文件要记得及时关闭!

一个程序能同时打开的文件个数, 是存在上限的。

```
flist = []
count = 0
while True:
    f = open('d:/test.txt', 'r')
    flist.append(f)
    count += 1
    print(f'count = {count}')
```

```
OSError: [Errno 24] Too many open files: 'd:/test.txt'
```

如上面代码所示, 如果一直循环的打开文件, 而不去关闭的话, 就会出现上述报错。

当一个程序打开的文件个数超过上限, 就会抛出异常。

注意: 上述代码中, 使用一个列表来保存了所有的文件对象。如果不进行保存, 那么 Python 内置的垃圾回收机制, 会在文件对象销毁的时候自动关闭文件。

但是由于垃圾回收操作不一定及时, 所以我们写代码仍然要考虑手动关闭, 尽量避免依赖自动关闭。

3. 写文件

文件打开之后, 就可以写文件了。

- 写文件, 要使用写方式打开, `open` 第二个参数设为 `'w'`
- 使用 `write` 方法写入文件。

```
f = open('d:/test.txt', 'w')
f.write('hello')
f.close()
```

hello

用记事本打开文件, 即可看到文件修改后的内容。

- 如果是使用 `'r'` 方式打开文件, 则写入时会抛出异常。

```
f = open('d:/test.txt', 'r')
f.write('hello')
f.close()
```

io.UnsupportedOperation: not writable

- 使用 'w' 一旦打开文件成功, 就会清空文件原有的数据.
- 使用 'a' 实现 "追加写", 此时原有内容不变, 写入的内容会存在于之前文件内容的末尾.

```
f = open('d:/test.txt', 'w')
f.write('hello')
f.close()

f = open('d:/test.txt', 'a')
f.write('world')
f.close()
```

helloworld

- 针对已经关闭的文件对象进行写操作, 也会抛出异常.

```
f = open('d:/test.txt', 'w')
f.write('hello')
f.close()
f.write('world')
```

ValueError: I/O operation on closed file.

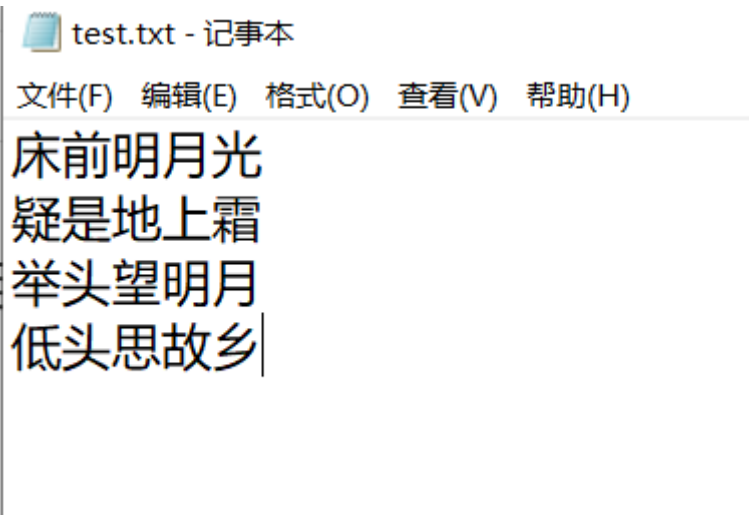
4. 读文件

- 读文件内容需要使用 'r' 的方式打开文件
- 使用 read 方法完成读操作. 参数表示 "读取几个字符"

```
f = open('d:/test.txt', 'r')
result = f.read(2)
print(result)
f.close()
```

你好

- 如果文件是多行文本, 可以使用 for 循环一次读取一行.



先构造一个多行文件.

```
f = open('d:/test.txt', 'r')
for line in f:
    print(f'line = {line}')
f.close()
```

```
line = 床前明月光
line = 疑是地上霜
line = 举头望明月
line = 低头思故乡
```

注意: 由于文件里每一行末尾都自带换行符, `print` 打印一行的时候又会默认加上一个换行符, 因此打印结果看起来之间存在空行.

使用 `print(f'line = {line}', end='')` 手动把 `print` 自带的换行符去掉.

- 使用 `readlines` 直接把文件整个内容读取出来, 返回一个列表. 每个元素即为一行.

```
f = open('d:/test.txt', 'r')
lines = f.readlines()
print(lines)
f.close()
```

```
['床前明月光\n', '疑是地上霜\n', '举头望明月\n', '低头思故乡']
```

此处的 `\n` 即为换行符。

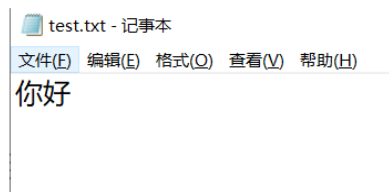
关于中文的处理

当文件内容存在中文的时候, 读取文件内容不一定就顺利。

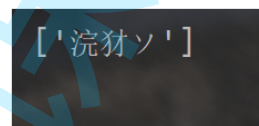
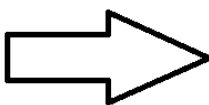
同样上述代码, 有的同学执行时可能会出现异常

```
UnicodeDecodeError: 'gbk' codec can't decode byte 0x89 in position 14: illegal multibyte sequence
```

也有的同学可能出现乱码。



文件的内容



代码读取并打印的内容

计算机表示中文的时候, 会采取一定的编码方式, 我们称为 "字符集"

所谓 "编码方式", 本质上就是使用数字表示汉字。

我们知道, 计算机只能表示二进制数据. 要想表示英文字母, 或者汉字, 或者其他文字符号, 就都要通过编码。

最简单的字符编码就是 `ascii`. 使用一个简单的整数就可以表示英文字母和阿拉伯数字。

但是要想表示汉字, 就需要一个更大的码表。

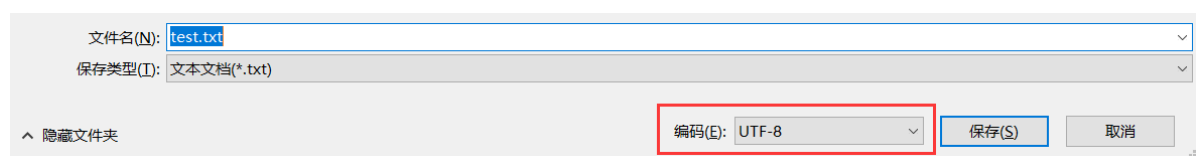
一般常用的汉字编码方式, 主要是 **GBK** 和 **UTF-8**

必须要保证文件本身的编码方式, 和 Python 代码中读取文件使用的编码方式匹配, 才能避免出现上述问题。

Python3 中默认打开文件的字符集跟随系统, 而 Windows 简体中文版的字符集采用了 GBK, 所以如果文件本身是 GBK 的编码, 直接就能正确处理。

如果文件本身是其他编码(比如 UTF-8), 那么直接打开就可能出现上述问题

使用记事本打开文本文件, 在 "菜单栏" -> "文件" -> "另存为" 窗口中, 可以看到当前文件的编码方式。



- 如果此处的编码为 `ANSI`, 则表示 GBK 编码。
- 如果此处为 `UTF-8`, 则表示 UTF-8 编码。

此时修改打开文件的代码, 给 `open` 方法加上 `encoding` 参数, 显式的指定为和文本相同的字符集, 问题即可解决.

```
f = open('d:/test.txt', 'r', encoding='utf8')
```

PS: 字符编码问题, 是编程中一类比较常见, 又比较棘手的问题. 需要对于字符编码有一定的理解, 才能从容应对.

同学们可以参考腾讯官方账号发表的帖子, 详细介绍了里面的细节. <https://zhuanlan.zhihu.com/p/46216008>

使用上下文管理器

打开文件之后, 是容易忘记关闭的. Python 提供了 **上下文管理器**, 来帮助程序猿自动关闭文件.

- 使用 `with` 语句打开文件.
- 当 `with` 内部的代码块执行完毕后, 就会自动调用关闭方法.

```
with open('d:/test.txt', 'r', encoding='utf8') as f:  
    lines = f.readlines()  
    print(lines)
```