一些简单的基础知识

print函数用于实现输出

例如：

- print("hello,world!")输出hello,world!
- print("123")输出123
- a = 100
- print(a)输出100等内容
- 对了，print(print("hello,world!"))会输出hello,world!和None你知道为什么吗？awa
- 这是因为print()函数本身没有返回值，所以会出现None

if，elif，else条件语句（也称选择语句，控制语句）

用法如下：

```
if 条件:
    语句A
else:
    语句B
#翻译：当条件为True时，执行语句A，否则执行语句B
if 条件1:
    语句A
elif 条件2:
    语句B
else:
    语句C
#翻译：当条件1为True时，执行语句A，否则当条件2为True时，执行语句B，否则执行语句C
#条件语句也可以嵌套
score = 85
if score >= 60:
    if score >= 90:
        print("优秀")
    elif score >= 80:
        print("良好")
    else:
        print("及格")
else:
    print("不及格")
#elif约等于else+if
```

for，while循环语句

其中while就是单纯的根据条件进行循环操作 例如：

- while 条件:当且仅当条件为True时循环

for语句就很复杂了 语法有这些：

```
for _ in {这里是一些东西，可以是列表，字典，甚至生成器和函数}:
    # 代码块
#也就是说可以这样解释
for 变量 in 可迭代对象:
    # 代码块
#可迭代对象包括：列表、元组、字典、集合、字符串、生成器、range()、以及任何实现了
__iter__() 方法的对象
#还有
result = [x * x for x in 可迭代对象] #这会得到一个列表
#这种奇怪的写法
```

条件可以是一个**表达式** 如：`i < 10`，`i <= 20`，`a == b`等。 这些表达式会给出一个布尔值(bool类型)，包括：True和False两种（注意，在一些语言中True和False可以被数字替代，一般是0为Flase，其余数字为True（**包括浮点数！**）） 条件也可以是一个**函数** 例如：

```
def is_odd(n):
    if n % 2 == 1:
        return True
    else:
        return False
if is_odd(n):
    print("这是一个奇数")
else:
    print("这是一个偶数")
```

一些不怎么简单的知识

1.三元表达式

```
{语句A} if 条件 else {语句B}
#约等于
if 条件:
    {语句A}
else:
    {语句B}
#所以这只是方便理解，实际上并不完全是这样
```

2.短路效应

```
{语句A} if 条件 else {语句B}
#如果条件为True，语句B就不会被求值，可以避免除0错误或一些其它的东西
#or表达式也可以，比如
1 or 1/0
#这里1/0就不会被计算
```

3.字典（键值对）

我不太了解字典，知道字典有键(key)和值(value) 比如：

```
# 多种方式创建字典
dict_a = {"name": "Alice", "age": 25}  # 直接创建
dict_b = dict(name="Bob", age=30)       # 使用dict()
dict_c = {k: v for k, v in [("a", 1), ("b", 2)]}  # 推导式

d = {"name": "Alice", "age": 25}
# 访问
d["name"]        # "Alice"
d.get("name")    # "Alice"（安全，不存在返回None）
d.get("sex", "未知")  # "未知"（可设默认值）

# 修改/添加
d["age"] = 26            # 修改
d["city"] = "北京"       # 添加新键值对

# 删除
d.pop("age")             # 删除并返回值
del d["name"]            # 直接删除
```

4.递归与迭代

啊，这个有点复杂。 一个简单区分递归与迭代的方式，就是：“递归：把问题拆成更小的同类问题，函数自己调自己；迭代：通过循环重复执行，逐步推进直到完成” 典型的迭代例子，就是tree()函数和数硬币 迭代的话，很常见，就不写了awa

5.lambda（匿名函数）

lambda 函数通常用于编写简单的、单行的函数，通常在需要函数作为参数传递的情况下使用，例如在`map()`、`filter()`等函数中。

```
f = lambda: "Hello, world!"
print(f())  # 输出: Hello, world!

x = lambda a : a + 10
print(x(5)) #输出15

x = lambda a, b : a * b
print(x(5, 6)) #输出30

numbers = [1, 2, 3, 4, 5]
squared = list(map(lambda x: x**2, numbers))
print(squared)  # 输出: [1, 4, 9, 16, 25]

numbers = [1, 2, 3, 4, 5, 6, 7, 8]
```

```
even_numbers = list(filter(lambda x: x % 2 == 0, numbers))
print(even_numbers)  # 输出：[2, 4, 6, 8]
```

面向对象

不怎么了解owo 只知道oop需要class，class里有实例变量（对象，可以被.表达式调用）和方法（可以被.表达式调用的函数），并且和Java不同，python的class需要使用**init**函数进行初始化 例如

```
1. 类的基本结构
class Student:
    # 类变量（所有实例共享）
    school = "清华大学"

    # 构造方法（初始化实例变量）
    def __init__(self, name, age):
        self.name = name  # 实例变量
        self.age = age    # 实例变量

    # 实例方法
    def introduce(self):
        print(f"我是{self.name}，今年{self.age}岁")

# 创建实例
stu = Student("小明", 20)
stu.introduce()           # 我是小明，今年20岁
print(stu.school)         # 清华大学

2. self 是什么?
class Demo:
    def __init__(self, value):
        self.value = value  # self 就是"当前实例本身"

    def show(self):
        print(self.value)

d = Demo(100) #self = d , value = 100
d.show()  # Python自动把 d 作为 self 传进去
# 等价于 Demo.show(d)
```