变量的命名

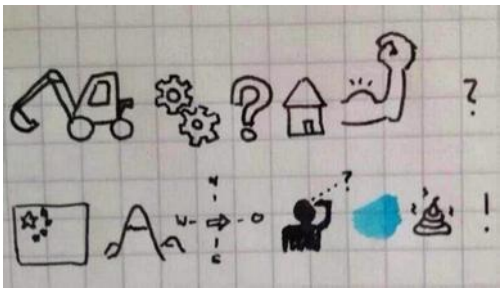
目标

- 标识符和关键字
- 变量的命名规则

0.1 标识符和关键字

1.1 标识符

> 标示符就是程序员定义的 **变量名、函数名**
>
> **名字** 需要有 **见名知义** 的效果，见下图：



- 标示符可以由 **字母、下划线** 和 **数字** 组成
- **不能以数字开头**
- **不能与关键字重名**

思考：下面的标示符哪些是正确的，哪些不正确为什么？

```
fromNo12 from#12 my_Boolean my-Boolean Obj2 2ndObj myInt My_tExt _test test!32 haha(da)tt jack_rose jack&rose GUI G.U.I
```

1.2 关键字

- **关键字** 就是在 `Python` 内部已经使用的标识符
- **关键字** 具有特殊的功能和含义
- 开发者 **不允许定义和关键字相同的名字的标示符**

通过以下命令可以查看 `Python` 中的关键字

```
python In [1]: import keyword In [2]: print(keyword.kwlist)
```

> 提示：**关键字的学习及使用**，会在后面的课程中不断介绍
>
> - `import` **关键字** 可以导入一个“工具包”
> - 在 `Python` 中不同的工具包，提供有不同的工具

02. 变量的命名规则

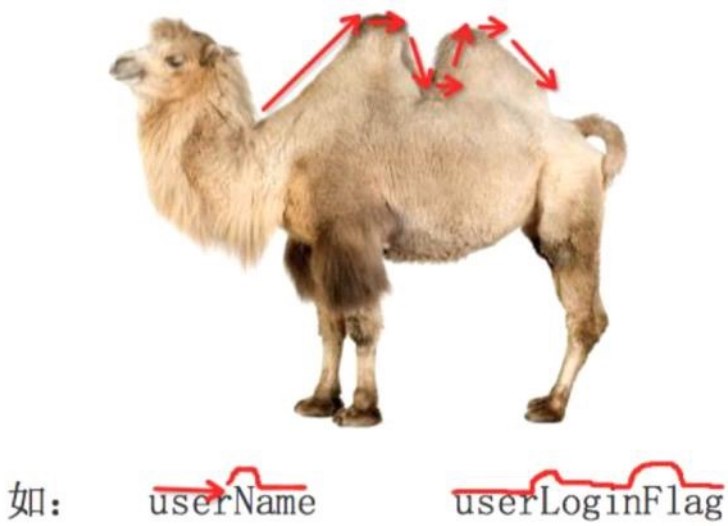
> **命名规则** 可以被视为一种 **惯例**，并无绝对与强制 目的是为了 **增加代码的识别和可读性**

注意 `Python` 中的 **标识符** 是 **区分大小写的**

1. 在定义变量时，为了保证代码格式，`=` 的左右应该各保留一个空格
2. 在 `Python` 中，如果 **变量名** 需要由 **二个** 或 **多个单词** 组成时，可以按照以下方式命名
    1. 每个单词都使用小写字母
    2. 单词与单词之间使用 **_下划线** 连接
    3. 例如：`first_name`、`last_name`、`qq_number`、`qq_password`

驼峰命名法

- 当 **变量名** 是由二个或多个单词组成时，还可以利用驼峰命名法来命名
- **小驼峰式命名法**
  - 第一个单词以小写字母开始，后续单词的首字母大写
  - 例如：`firstName`、`lastName`
- **大驼峰式命名法**
  - 每一个单词的首字母都采用大写字母
  - 例如：`FirstName`、`LastName`、`CamelCase`



判断（if）语句

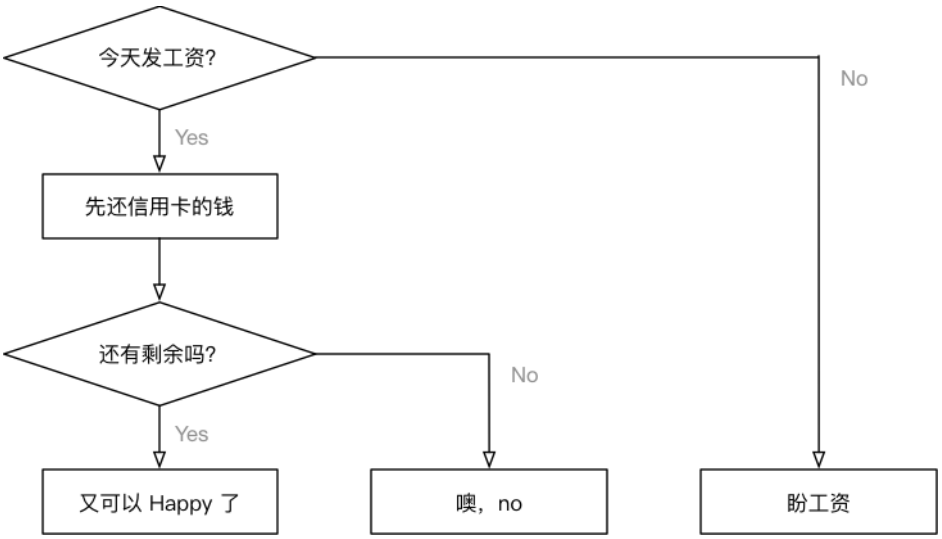
目标

- 开发中的应用场景
- if 语句体验
- if 语句进阶
- 综合应用

01. 开发中的应用场景

生活中的判断几乎是无所不在的，我们每天都在做各种各样的选择，如果这样？如果那样？……

程序中的判断




```python if 今天发工资:

```
先还信用卡的钱

if 有剩余:

    又可以happy了，O(∩_∩)O哈哈~

else:

    噢，no。。。还的等30天
```

else:

```
盼着发工资
```

```

### 判断的定义

- 如果 **条件满足**，才能做某件事情，

- 如果 **条件不满足**，就做另外一件事情，或者什么也不做

> 正是因为有了判断，才使得程序世界丰富多彩，充满变化！
>
> **判断语句** 又被称为 “分支语句”，正是因为有了判断，才让程序有了很多的分支

## 02. if 语句体验

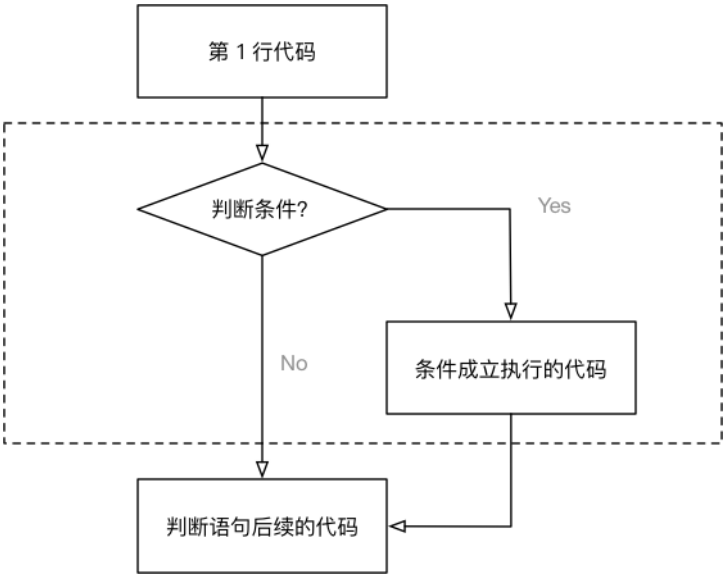
### 2.1 if 判断语句基本语法

在 `Python` 中，**if 语句** 就是用来进行判断的，格式如下：

```
python if 要判断的条件：条件成立时，要做的事情 ……
```

> 注意：代码的缩进为一个 `tab` 键，或者 4 个空格 —— **建议使用空格**
>
> - 在 Python 开发中，Tab 和空格不要混用！

我们可以把整个 if 语句看成一个完整的代码块



2.2 判断语句演练 —— 判断年龄

需求

1. 定义一个整数变量记录年龄
2. 判断是否满 18 岁 （**>=**）
3. 如果满 18 岁，允许进网吧嗨皮

```python
# 1. 定义年龄变量
age = 18

# 2. 判断是否满 18 岁
# if 语句以及缩进部分的代码是一个完整的代码块
if age >= 18:
    print("可以进网吧嗨皮……")

# 3. 思考！- 无论条件是否满足都会执行
print("这句代码什么时候执行?")
```

注意：

- `if` 语句以及缩进部分是一个 **完整的代码块**

2.3 else 处理条件不满足的情况

思考

在使用 `if` 判断时，只能做到满足条件时要做的事情。那如果需要在 **不满足条件的时候**，做某些事情，该如何做呢？

答案

`else`，格式如下：

`python if 要判断的条件： 条件成立时，要做的事情 …… else： 条件不成立时，要做的事情 ……`

注意：

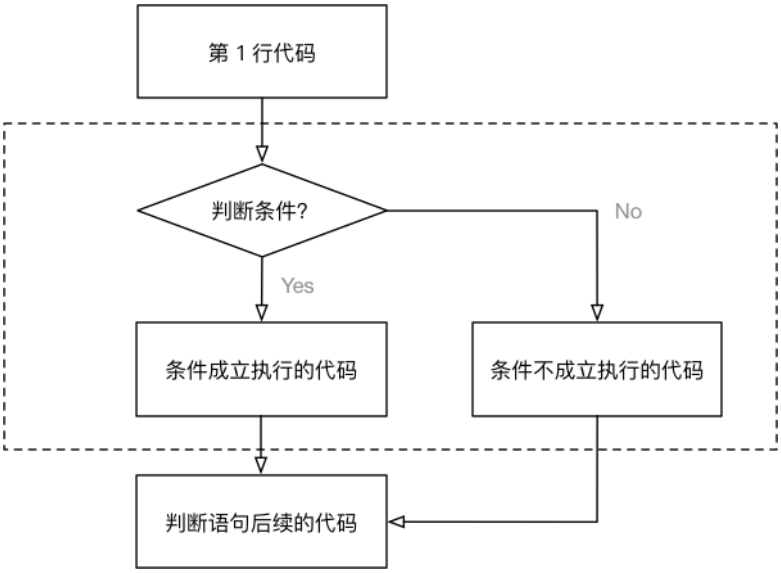
- `if` 和 `else` 语句以及各自的缩进部分共同是一个 **完整的代码块**

2.4 判断语句演练 —— 判断年龄改进

需求

1. 输入用户年龄
2. 判断是否满 18 岁 （**>=**）
3. 如果满 18 岁，允许进网吧嗨皮
4. 如果未满 18 岁，提示回家写作业

```python
# 1. 输入用户年龄
age = int(input("今年多大了？"))
# 2. 判断是否满 18 岁
# if 语句以及缩进部分的代码是一个完整的语法块
if age >= 18:
    print("可以进网吧嗨皮……")
else:
    print("你还没长大，应该回家写作业！")
# 3. 思考！- 无论条件是否满足都会执行
print("这句代码什么时候执行?")
```



03. 逻辑运算

- 在程序开发中，通常 **在判断条件时**，会需要同时判断多个条件
- 只有多个条件都满足，才能够执行后续代码，这个时候需要使用到 **逻辑运算符**
- **逻辑运算符** 可以把 **多个条件** 按照 **逻辑** 进行 **连接**，变成 **更复杂的条件**
- Python 中的 **逻辑运算符** 包括：**与 and / 或 or / 非 not** 三种

3.1 and

`条件1 and 条件2`

- **与 / 并且**
- 两个条件同时满足，返回 `True`
- 只要有一个不满足，就返回 `False`

| 条件 1 | 条件 2 | 结果 | | :---: | :---: | :---: | | 成立 | 成立 | 成立 | | 成立 | 不成立 | 不成立 | | 不成立 | 成立 | 不成立 | | 不成立 | 不成立 | 不成立 |

3.2 `or`

`条件1 or 条件2`

- **或 / 或者**
- 两个条件只要有一个满足，返回 `True`
- 两个条件都不满足，返回 `False`

| 条件 1 | 条件 2 | 结果 | | :---: | :---: | :---: | | 成立 | 成立 | 成立 | | 成立 | 不成立 | 成立 | | 不成立 | 成立 | 成立 | | 不成立 | 不成立 | 不成立 |

3.3 `not`

`not 条件`

- **非 / 不是**

| 条件 | 结果 | | :---: | :---: | | 成立 | 不成立 | | 不成立 | 成立 |

逻辑运算演练

1. 练习1: 定义一个整数变量 `age`，编写代码判断年龄是否正确
   - 要求人的年龄在 0-120 之间
2. 练习2: 定义两个整数变量 `python_score`、`c_score`，编写代码判断成绩
   - 要求只要有一门成绩 > 60 分就算合格
3. 练习3: 定义一个布尔型变量 `is_employee`，编写代码判断是否是本公司员工
   - 如果不是提示不允许入内

答案 1：

```python
# 练习1: 定义一个整数变量 age，编写代码判断年龄是否正确
age = 100
# 要求人的年龄在 0-120 之间
if age >= 0 and age <= 120: print("年龄正确") else: print("年龄不正确")
```

答案 2：

```python
# 练习2: 定义两个整数变量 python_score、c_score，编写代码判断成绩
python_score = 50 c_score = 50
# 要求只要有一门成绩 > 60 分就算合格
if python_score > 60 or c_score > 60: print("考试通过") else: print("再接再厉！") ```
```

答案 3：

```python
# 练习3: 定义一个布尔型变量 is_employee，编写代码判断是否是本公司员工
is_employee = True
# 如果不是提示不允许入内
if not is_employee: print("非公勿内") ```
```

04. if 语句进阶

4.1 `elif`

- 在开发中，使用 `if` 可以 **判断条件**
- 使用 `else` 可以处理 **条件不成立** 的情况
- 但是，如果希望 **再增加一些条件，条件不同，需要执行的代码也不同** 时，就可以使用 `elif`
- 语法格式如下：

```
python if 条件1: 条件1满足执行的代码 …… elif 条件2: 条件2满足时，执行的代码 …… elif 条件3: 条件3满足时，执行的代码 …… else: 以上条件都不满足时，执行的代码 ……
```

- 对比逻辑运算符的代码

```
python if 条件1 and 条件2: 条件1满足 并且 条件2满足 执行的代码 ……
```

注意

1. `elif` 和 `else` 都必须和 `if` 联合使用，而不能单独使用
2. 可以将 `if`、`elif` 和 `else` 以及各自缩进的代码，看成一个 **完整的代码块**

elif 演练 —— 女友的节日

需求

1. 定义 `holiday_name` 字符串变量记录节日名称
2. 如果是 **情人节** 应该 **买玫瑰 / 看电影**
3. 如果是 **平安夜** 应该 **买苹果 / 吃大餐**
4. 如果是 **生日** 应该 **买蛋糕**
5. 其他的日子每天都是节日啊……

``` holiday_name = "平安夜"

if holiday*name* == *"情人节": print("买玫瑰") print("看电影") elif holiday*name == "平安夜": print("买苹果") print("吃大餐") elif holiday_name == "生日": print("买蛋糕") else: print("每天都是节日啊……")

```

4.2 `if` 的嵌套

> **elif** 的应用场景是：**同时** 判断 **多个条件**，所有的条件是 **平级** 的

- 在开发中，使用 `if` 进行条件判断，如果希望 **在条件成立的执行语句中** 再 **增加条件判断**，就可以使用 **if 的嵌套**
- **if 的嵌套** 的应用场景就是：**在之前条件满足的前提下，再增加额外的判断**

- **if 的嵌套** 的语法格式，**除了缩进之外** 和之前的没有区别
- 语法格式如下：

```python if 条件 1: 条件 1 满足执行的代码 ……

if 条件 1 基础上的条件 2:
    条件 2 满足时，执行的代码
    ……

# 条件 2 不满足的处理
else:
    条件 2 不满足时，执行的代码

# 条件 1 不满足的处理

else: 条件1 不满足时，执行的代码 …… ```
```

if 的嵌套 演练 —— 火车站安检

需求

1. 定义布尔型变量 `has_ticket` 表示是否有车票
2. 定义整型变量 `knife_length` 表示刀的长度，单位：厘米
3. 首先检查是否有车票，如果有，才允许进行 **安检**
4. 安检时，需要检查刀的长度，判断是否超过 20 厘米
    - 如果超过 20 厘米，提示刀的长度，不允许上车
    - 如果不超过 20 厘米，安检通过
5. 如果没有车票，不允许进门

```python

# 定义布尔型变量 has_ticket 表示是否有车票

has_ticket = True

# 定义整数型变量 knife_length 表示刀的长度，单位：厘米

knife_length = 20

# 首先检查是否有车票，如果有，才允许进行 安检

if has_ticket: print("有车票，可以开始安检...")

# 安检时，需要检查刀的长度，判断是否超过 20 厘米
# 如果超过 20 厘米，提示刀的长度，不允许上车
if knife_length >= 20:
    print("不允许携带 %d 厘米长的刀上车" % knife_length)
# 如果不超过 20 厘米，安检通过
else:
    print("安检通过，祝您旅途愉快……")

# 如果没有车票，不允许进门

else: print("大哥，您要先买票啊")

```

05. 综合应用 —— 石头剪刀布

目标

1. 强化 **多个条件** 的 **逻辑运算**
2. 体会 `import` 导入模块（“工具包”）的使用

需求

1. 从控制台输入要出的拳 —— 石头（1） / 剪刀（2） / 布（3）
2. 电脑 **随机** 出拳 —— 先假定电脑只会出石头，完成整体代码功能
3. 比较胜负

| 序号 | 规则 | | :---: | :---: | | 1 | 石头 胜 剪刀 | | 2 | 剪刀 胜 布 | | 3 | 布 胜 石头 |

5.1 基础代码实现

- 先 **假定电脑就只会出石头**，完成整体代码功能

```python
# 从控制台输入要出的拳 —— 石头（1） / 剪刀（2） / 布（3）
player = int(input("请出拳 石头（1） / 剪刀（2） / 布（3）："))

# 电脑 随机 出拳 - 假定电脑永远出石头
computer = 1

# 比较胜负

# 如果条件判断的内容太长，可以在最外侧的条件增加一对大括号

# 再在每一个条件之间，使用回车，PyCharm 可以自动增加 8 个空格
if ((player == 1 and computer == 2) or (player == 2 and computer == 3) or (player == 3 and computer == 1)):

    print("噢耶！！！电脑弱爆了！！！")

elif player == computer: print("心有灵犀，再来一盘！") else: print("不行，我要和你决战到天亮！")
```

5.2 随机数的处理

- 在 `Python` 中，要使用随机数，首先需要导入 **随机数** 的 **模块** —— “工具包”

```
python import random
```

- 导入模块后，可以直接在 **模块名称** 后面敲一个 `.` 然后按 `Tab` 键，会提示该模块中包含的所有函数
- `random.randint(a, b)` ，返回 `[a, b]` 之间的整数，包含 `a` 和 `b`
- 例如：

```
python random.randint(12, 20) # 生成的随机数n: 12 <= n <= 20 random.randint(20, 20) # 结果永远是 20 random.randint(20, 10) # 该语句是错误的，下限必须小于上限
```

运算符

目标

- 算数运算符
- 比较（关系）运算符
- 逻辑运算符
- 赋值运算符
- 运算符的优先级

数学符号表链接：https://zh.wikipedia.org/wiki/数学符号表

01. 算数运算符

- 是完成基本的算术运算使用的符号，用来处理四则运算

| 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 |

- 在 Python 中 `*` 运算符还可以用于字符串，计算结果就是字符串重复指定次数的结果

```
python In [1]: "-" * 50 Out[1]: '--------------------------------------------------'
```

02. 比较（关系）运算符

| 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 |

> Python 2.x 中判断 **不等于** 还可以使用 `<>` 运算符
>
> `!=` 在 Python 2.x 中同样可以用来判断 **不等于**

03. 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 |

04. 赋值运算符

- 在 Python 中，使用 `=` 可以给变量赋值
- 在算术运算时，为了简化代码的编写，`Python` 还提供了一系列的 与 **算术运算符** 对应的 **赋值运算符**
- 注意：**赋值运算符中间不能使用空格**

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

05. 运算符的优先级

- 以下表格的算数优先级由高到最低顺序排列

| 运算符 | 描述 | | --- | --- | | ** | 幂 (最高优先级) | | * / % // | 乘、除、取余数、取整除 | | + - | 加法、减法 | | <= < > >= | 比较运算符 | | == != | 等于运算符 | | = %= /= //= -= += *= **= | 赋值运算符 | | not or and | 逻辑运算符 |

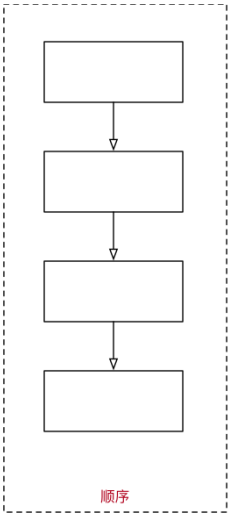
循环

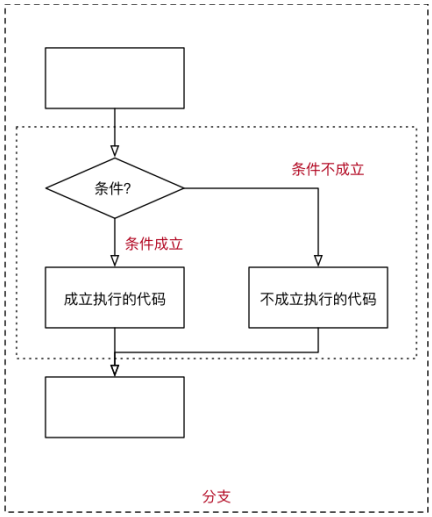
目标

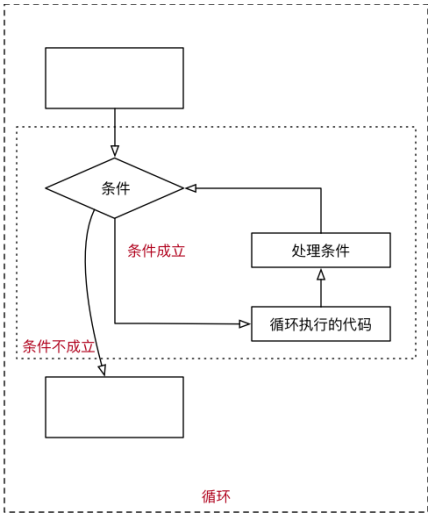
- 程序的三大流程
- while 循环基本使用
- break 和 continue
- while 循环嵌套

01. 程序的三大流程

- 在程序开发中，一共有三种流程方式：
  - **顺序** —— **从上向下**，顺序执行代码
  - **分支** —— 根据条件判断，决定执行代码的 **分支**
  - **循环** —— 让 **特定代码 重复** 执行







02. `while` 循环基本使用

- 循环的作用就是让 **指定的代码** 重复的执行
- `while` 循环最常用的应用场景就是 **让执行的代码** 按照 **指定的次数** **重复** 执行

- 需求 —— 打印 5 遍 `Hello Python`

- 思考 —— 如果要求打印 100 遍怎么办？

2.1 `while` 语句基本语法

```python 初始条件设置 —— 通常是重复执行的 计数器

while 条件(判断 计数器 是否达到 目标次数): 条件满足时，做的事情1 条件满足时，做的事情2 条件满足时，做的事情3 ...(省略)...

```
处理条件(计数器 + 1)
```

```

注意：

- `while` 语句以及缩进部分是一个 **完整的代码块**

第一个 while 循环

需求

- 打印 5 遍 Hello Python

```while

# 1. 定义重复次数计数器

i = 1

# 2. 使用 while 判断条件

while i <= 5: # 要重复执行的代码 print("Hello Python")

```
# 处理计数器 i
i = i + 1
```

print("循环结束后的 i = %d" % i) ```

> 注意：循环结束后，之前定义的计数器条件的数值是依旧存在的

#### 死循环

> 由于程序员的原因，**忘记** 在循环内部 **修改循环的判断条件**，导致循环持续执行，程序无法终止！

### 2.2 赋值运算符

- 在 Python 中，使用 `=` 可以给变量赋值
- 在算术运算时，为了简化代码的编写，`Python` 还提供了一系列的 与 **算术运算符** 对应的 **赋值运算符**
- 注意：**赋值运算符中间不能使用空格**

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

### 2.3 Python 中的计数方法

常见的计数方法有两种，可以分别称为：

- **自然计数法**（从 `1` 开始）—— 更符合人类的习惯
- **程序计数法**（从 `0` 开始）—— 几乎所有的程序语言都选择从 0 开始计数

因此，大家在编写程序时，应该尽量养成习惯：**除非需求的特殊要求，否则 循环 的计数都从 0 开始**

### 2.4 循环计算

> 在程序开发中，通常会遇到 **利用循环 重复计算** 的需求

遇到这种需求，可以：

1. 在 `while` 上方定义一个变量，用于 **存放最终计算结果**
```

2. 在循环体内部，每次循环都用 **最新的计算结果，更新** 之前定义的变量

需求

- 计算 0 ~ 100 之间所有数字的累计求和结果

```python
# 计算 0 ~ 100 之间所有数字的累计求和结果

# 0. 定义最终结果的变量
result = 0

# 1. 定义一个整数的变量记录循环的次数
i = 0

# 2. 开始循环
while i <= 100: print(i)

    # 每一次循环，都让 result 这个变量和 i 这个计数器相加
    result += i

    # 处理计数器
    i += 1

print("0~100之间的数字求和结果 = %d" % result)
```

需求进阶

- 计算 0 ~ 100 之间 所有 **偶数** 的累计求和结果

开发步骤

1. 编写循环 **确认 要计算的数字**
2. 添加 **结果** 变量，在循环内部 **处理计算结果**

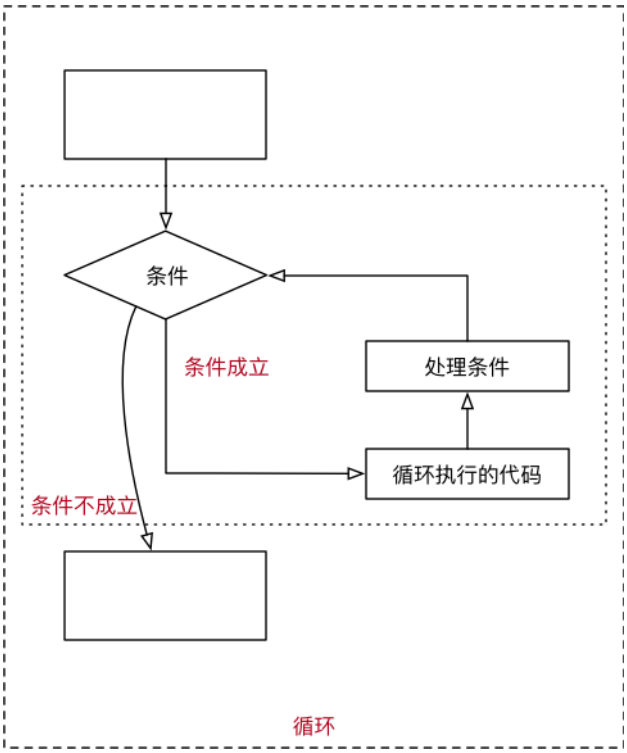
```python
# 0. 最终结果
result = 0

# 1. 计数器
i = 0

# 2. 开始循环
while i <= 100:

    # 判断偶数
    if i % 2 == 0:
        print(i)
        result += i

    # 处理计数器
    i += 1

print("0~100之间偶数求和结果 = %d" % result)
```

03. break 和 continue

> `break` 和 `continue` 是专门在循环中使用的关键字

- `break` **某一条件满足时**，退出循环，不再执行后续重复的代码
- `continue` **某一条件满足时**，不执行后续重复的代码

> `break` 和 `continue` 只针对 **当前所在循环** 有效



3.1 break

- **在循环过程中**，如果 **某一个条件满足后，不** 再希望 **循环继续执行**，可以使用 `break` 退出循环

```python i = 0

while i < 10:

    # break 某一条件满足时，退出循环，不再执行后续重复的代码
    # i == 3
    if i == 3:
        break

    print(i)

    i += 1

print("over") ```
```

> `break` 只针对当前所在循环有效

3.2 continue

- **在循环过程中**，如果 **某一个条件满足后，不** 希望 **执行循环代码，但是又不希望退出循环**，可以使用 `continue`
- 也就是：在整个循环中，**只有某些条件**，不需要执行循环代码，而其他条件都需要执行

```python i = 0

while i < 10:

    # 当 i == 7 时，不希望执行需要重复执行的代码
    if i == 7:
        # 在使用 continue 之前，同样应该修改计数器
        # 否则会出现死循环
        i += 1

        continue

    # 重复执行的代码
    print(i)

    i += 1
```

```

- 需要注意：使用 `continue` 时，**条件处理部分的代码，需要特别注意**，不小心会出现 **死循环**

> `continue` 只针对当前所在循环有效

## 04. `while` 循环嵌套

### 4.1 循环嵌套

- `while` 嵌套就是：`while` 里面还有 `while`

```python while 条件 1: 条件满足时，做的事情1 条件满足时，做的事情2 条件满足时，做的事情3 ...(省略)...

```
while 条件 2:
    条件满足时，做的事情1
    条件满足时，做的事情2
    条件满足时，做的事情3
    ...(省略)...

    处理条件 2

处理条件 1
```

```

### 4.2 循环嵌套演练 —— 九九乘法表

#### 第 1 步：用嵌套打印小星星

**需求**

- 在控制台连续输出五行 `*`，每一行星号的数量依次递增

``` * **

```

```

- 使用字符串 * 打印

```python

# 1. 定义一个计数器变量，从数字1开始，循环会比较方便

row = 1

while row <= 5:

```
print("*" * row)

row += 1
```

```

第 2 步：使用循环嵌套打印小星星

知识点 对 `print` 函数的使用做一个增强

- 在默认情况下，`print` 函数输出内容之后，会自动在内容末尾增加换行
- 如果不希望末尾增加换行，可以在 `print` 函数输出内容的后面增加 `, end=""`
- 其中 `""` 中间可以指定 `print` 函数输出内容之后，继续希望显示的内容

- 语法格式如下：

```python

# 向控制台输出内容结束之后，不会换行

print("*", end="")
```

单纯的换行

print("") ```

> `end=""` 表示向控制台输出内容结束之后，不会换行

假设 `Python` 没有提供 字符串的 `*` 操作 **拼接字符串**

需求

- 在控制台连续输出五行 `*`，每一行星号的数量依次递增

``` * **

---

---

---

```

开发步骤

- 1> 完成 5 行内容的简单输出
- 2> 分析每行内部的 `*` 应该如何处理？
  - 每行显示的星星和当前所在的行数是一致的
  - 嵌套一个小的循环，专门处理每一行中 `列` 的星星显示

```python row = 1

while row <= 5:

```
# 假设 python 没有提供字符串 * 操作
# 在循环内部，再增加一个循环，实现每一行的 星星 打印
col = 1

while col <= row:
    print("*", end="")

    col += 1

# 每一行星号输出完成后，再增加一个换行
print("")

row += 1
```

```

第 3 步： 九九乘法表

需求 输出 九九乘法表，格式如下：

``` 1 * 1 = 1 1 * 2 = 2 2 * 2 = 4 1 * 3 = 3 2 * 3 = 6 3 * 3 = 9 1 * 4 = 4 2 * 4 = 8 3 * 4 = 12 4 * 4 = 16 1 * 5 = 5 2 * 5 = 10 3 * 5 = 15 4 * 5 = 20 5 * 5 = 25 1 * 6 = 6 2 * 6 = 12 3 * 6 = 18 4 * 6 = 24 5 * 6 = 30 6 * 6 = 36 1 * 7 = 7 2 * 7 = 14 3 * 7 = 21 4 * 7 = 28 5 * 7 = 35 6 * 7 = 42 7 * 7 = 49 1 * 8 = 8 2 * 8 = 16 3 * 8 = 24 4 * 8 = 32 5 * 8 = 40 6 * 8 = 48 7 * 8 = 56 8 * 8 = 64 1 * 9 = 9 2 * 9 = 18 3 * 9 = 27 4 * 9 = 36 5 * 9 = 45 6 * 9 = 54 7 * 9 = 63 8 * 9 = 72 9 * 9 = 81

```

开发步骤

- 1. 打印 9 行小星星

``` * **

---

---

---

---

---

```

- 2. 将每一个 `*` 替换成对应的行与列相乘

```python

# 定义起始行

row = 1
```

最大打印 9 行

while row <= 9: # 定义起始列 col = 1

```
# 最大打印 row 列
while col <= row:

    # end = "", 表示输出结束后，不换行
    # "\t" 可以在控制台输出一个制表符，协助在输出文本时对齐
    print("%d * %d = %d" % (col, row, row * col), end="\t")

    # 列数 + 1
    col += 1

# 一行打印完成的换行
print("")

# 行数 + 1
row += 1
```

```

**字符串中的转义字符**

- `\t` 在控制台输出一个 **制表符**，协助在输出文本时 **垂直方向** 保持对齐
- `\n` 在控制台输出一个 **换行符**

> **制表符** 的功能是在不使用表格的情况下在 **垂直方向** 按列对齐文本

| 转义字符 | 描述 | | --- | --- | | \\ | 反斜杠符号 | | \' | 单引号 | | \" | 双引号 | | \n | 换行 | | \t | 横向制表符 | | \r | 回车 |
```