

判断与循环

复习

H3 review1 基本操作符

- 双目运算符 `+`, `-`, `*`, `/`, `&`, `|`, `^`, `~`, `<<`, `>>` 以及 `()` 与C中的使用方法完全一致:

```
1  4>>> 1 + 1
2  2
3  >>> 1 - 1
4  0
5  >>> 2 * 3
6  6
7  >>> 3 / 2
8  1.5
9  >>> 1 & 1    # 按位与
10 1
11 >>> 1 | 1    # 按位或
12 1
13 >>> 1 ^ 1    # 按位异或
14 0
15 >>> ~ 1      # 按位取反
16 -2
17 >>> 1 >> 1   # 右移
18 0
19 >>> 1 << 1   # 左移
20 2
```

需要注意的是，Python的 `/` 会返回一个 `float` 类型的变量，而不是像C那样根据除数与被除数对结果强制类型转换。

- Python中的求模运算 `%` 比较灵活：

```
1 >>> 17 % 3
2 2
3 >>> 17 % 3.2
4 0.9999999999999991
5 >>> -18.2 % 2.55
6 2.1999999999999993
```

- 与C不同的双目运算符有 `//`，`**`，其中 `//` 是整数除，返回除法结果的下取整

```
1 >>> 123.2 // 21
2 5.0
3 >>> 123.12 // 213.2
4 0.0
5 >>> 123.2 // 3.1
6 39.0
```

- `**` 是求幂操作，功能与C中的 `power()` 函数一致，非常方便：

```
1 >>> 3 ** 2
2 9
3 >>> 2 ** 0.5
4 1.4142135623730951
```

- Python中的赋值符号 `=` 与C中基本一致：

```
1 >>> a = 1
2 >>> b = 2
3 >>> a, b
4 (1, 2)
```

- 不过它神奇地支持多对多的赋值，从此你再也不需要为交换变量创建临时变量或者异或半天了：

```
1 >>> a, b = 3, 4 # 多对多赋值
2 >>> a, b
3 (3, 4)
4 >>> a, b = b, a # 交换变量
5 >>> a, b
6 (4, 3)
```

- 本地(增强)操作运算符 `+=` , `-=` , `*=` 等等和C中一样：

```
1 >>> a += 1
2 >>> a
3 2
4 >>> a -= 2
5 >>> a
6 0
```

其余新增的操作符对应的增强操作符（如 `**=` , `//=` ）都存在。需要注意的是，你们在C中最爱的 `++` 和 `--` 在Python中不存在。

Python3.8版本后，新增了 `:=`，也就是海象运算符，可在表达式内部为变量赋值。感兴趣的同学可以去了解一下（估计你们也没有兴趣`==`）

- 判断实数的大小的 `==`，`≠`，`<`，`>`，`≤`，`≥` 与C中用法基本一样的：

```
1 >>> a, b, c = 1, 1, 2
2 >>> a == b
3 True
4 >>> a == c
5 False
6 >>> a > b
7 False
8 >>> a >= b
9 True
10 >>> a < c
11 True
```

- 不同的是，Python中的大于小于支持数学家喜欢的写法。比如在描述变量 `a` 大于1，小于2这件事上，C中我们会写成 `a > 1 && a < 2`，但是在Python中，你可以一气呵成：

```
1 >>> a = 1.3
2 >>> 1 < a < 2
3 True
```

- 关于逻辑操作符，Python中比较贴合自然语言，Python中不存在 `&&`，`||`，`!`，取而代之的是 `and`，`or`，`not`：

```
1 >>> a, b = True, False
2 >>> a and b
3 False
4 >>> a or b
5 True
6 >>> not a
7 False
```

H3 review2 字符串

Python中的字符串是一个功能异常强大的类，这个类的名字叫做 `str`：

```
1 >>> a = "hello"
2 >>> type(a)
3 <class 'str'>
```

关于 `str` 能讲的太多了，这边只讲一些基础的，想要玩得花里胡哨，请移步官网、CSDN、知乎等处

直接用一对双引号或者一对单引号就可以创建一个字符串，可以直接赋值给变量：

```
1 >>> a = "我是双引号"
2 >>> b = '我是单引号'
3 >>> a, b
4 ('我是双引号', '我是单引号')
```

可以看到，无论是使用单引号还是双引号，对最终创建的 `str` 都没有区别。那么Python设计两种引号有什么必要吗？

首当其冲，方便了呀，相信没几个人愿意多按几下Shift。其次如果你在一个字符串中想要打印双引号怎么办？如果你直接写。。。

```
1 >>> a = "我小时候常听马也大佬说: "champion, yyds!" "  
2 File "<stdin>", line 1  
3     a = "我小时候常听马也大佬说: "champion, yyds!" "  
4                                     ^  
5 SyntaxError: invalid syntax
```

这样写不行，因为会引起解释器的歧义。Python取消单引号和双引号就让我们打印双引号容易了，我们只需要把最外面的双引号换成单引号就bingo了：

```
1 >>> a = '我小时候常听马也大佬说: "champion, yyds!" '  
2 >>> print(a)  
3 我小时候常听马也大佬说: "champion, yyds!"
```

当然，你也可以使用转义符来打印双引号，这与C中一样：

```
1 >>> a = "我小时候常听马也大佬说: \"champion, yyds!\" "  
2 >>> print(a)  
3 我小时候常听马也大佬说: "champion, yyds!"
```

SOMETHING NEW

还是接着字符串讲讲，这个很好用，很好玩。同往常一样，我们可以创建两个字符串：

```
1 >>> a = "hello"
2 >>> b = "world"
3 >>> print(a, b)      # print函数用逗号隔开变量会自动在打印时生成变量之间的空格
4 hello world
```

Python常用的标准输入函数 `input()` 读入的值全都是 `str` :

```
1 >>> txt = input("请随便输入点什么吧: ")
2 请随便输入点什么吧: hello world
3 >>> type(txt)
4 <class 'str'>
```

所以对 `str` 的基本操作非常重要，否则你连 `input()` 读入的数值怎么处理都不知道。

使用 `+` 可以连接两个字符串：

```
1 >>> a + b
2 'helloworld'
```

不大好看，我们可以再连一个空格：

```
1 >>> a + " " + b
2 'hello world'
```

通过 `*`，我们可以让字符串自我复制，一般来说，你可以使用这个方法快速打印定长的分割线：

```
1 >>> a = "复读机"
2 >>> a * 3
3 '复读机复读机复读机'
4 >>> a = "-"
5 >>> a * 20
6 '-----'
```

记住，`str` 的 `*` 操作符后面只能跟整数，如果 `*` 后面跟的是`0`或者负数，则返回空字符串：

```
1 >>> b = a * (-1)
2 >>> c = a * 0
3 >>> b, c
4 ('', '')
5 >>> len(b), len(c)      # 通过len方法获取字符串的长度
6 (0, 0)
```

如果跟小数，直接报错。

`str` 中定义了比较大小的运算符，也就是支持 `>`，`<`，`≥` 和 `≤`，它的用法和C中字符串的比较几乎一致。对于单个字符，比较两个字符也就是比较两个字符的Unicode编码：

```
1 >>> 'a' > 'b'
2 False
3 >>> ord('a'), ord('b')  # ord函数获取字符的Unicode编码
4 (97, 98)
```

Python中的默认编码格式是Unicode，通过 `ord()` 函数可以获取字符的Unicode编码，`chr()` 函数根据输入的Unicode码返回对应的Unicode字符。

对应字符串的比较，逻辑与C中的 `strcmp()` 函数逻辑一致，按位比较两个字符串的字符，直到比较到两个不同的字符，这两个字符的比较结果就是整体比较的结果：

```
1 >>> "abcde" > "abcfg"
2 False
```

空字符小于任何字符：

```
1 >>> "sadaW" > ""
2 True
```

`str` 的强大之处在于它有很多方法可以调用，熟悉这些方法后，字符串操作就会变得超级方便，比如我们现在有一个字符串 `a`：

```
1 >>> a = " many space on two sides "
```

我们发现它旁边有一些空格，出于某种需求，我们想要把旁边的空格去除，我们可以使用 `strip()` 方法：

```
1 >>> a.strip(" ")    # 默认参数为空格
2 'many space on two sides'
```

我们想要单独提取每个单词，返回一个列表：

```
1 >>> a.split(" ")    # 默认参数为空格
2 ['many', 'space', 'on', 'two', 'sides']
```

我们突然发现，`space`是不可数名词，不应该用`many`描述，应该换成`much`，我们可以使用 `replace()` 方法来替换字符串中的子串：

```
1 >>> a.replace("many", "much")
2 ' much space on two sides '
```

你甚至可以一气呵成，把上面的方法一行写完：

```
1 >>> a.strip().replace("many", "much").split()
2 ['much', 'space', 'on', 'two', 'sides']
```

`str` 对象有很多实用的方法，你不需要全部记住，在不记得的时候，通过 `dir()` 函数直接查看当前对象的所有属性和方法就可以偷窥到你或许要的东西了：

```
1 >>> a = "I'm a str object"
2 >>> dir(a)
3 ['__add__', '__class__', '__contains__', '__delattr__', '__dir__',
  '__doc__', '__eq__', '__format__', '__ge__', '__getattr__',
  '__getitem__', '__getnewargs__', '__gt__', '__hash__', '__init__',
  '__init_subclass__', '__iter__', '__le__', '__len__', '__lt__',
  '__mod__', '__mul__', '__ne__', '__new__', '__reduce__',
  '__reduce_ex__', '__repr__', '__rmod__', '__rmul__', '__setattr__',
  '__sizeof__', '__str__', '__subclasshook__', 'capitalize', 'casefold',
  'center', 'count', 'encode', 'endswith', 'expandtabs', 'find',
  'format', 'format_map', 'index', 'isalnum', 'isalpha', 'isascii',
  'isdecimal', 'isdigit', 'isidentifier', 'islower', 'isnumeric',
  'isprintable', 'isspace', 'istitle', 'isupper', 'join', 'ljust',
  'lower', 'lstrip', 'maketrans', 'partition', 'replace', 'rfind',
  'rindex', 'rjust', 'rpartition', 'rsplit', 'rstrip', 'split',
  'splitlines', 'startswith', 'strip', 'swapcase', 'title', 'translate',
  'upper', 'zfill']
```

除了前后都有 `__` 的方法，我们称之为magic method（魔法方法），那是一类神奇的方法。其余的方法都是一些对字符串的实用操作，具体请移步官方文档。

再随便举几个例子，先让你们知道有什么。

```
1 >>> a
2 "I'm a str object"
3 >>> a.lower()    # 全部小写
4 "i'm a str object"
5 >>> a.upper()    # 全部大写
6 "I'M A STR OBJECT"
7 >>> a.title()    # 标题化
8 "I'M A Str Object"
9 >>> a.count('t')  # 统计子串的个数
```

```
10 2
11 >>> a.find("str")    # 在a中寻找子串的位置，找到子串首字母的位置
12 6
13 >>> a.find("**")     # 找不到返回-1
14 -1
```

来一个难一点的，下面这个demo请记住，因为很好用，也用得很多。比如我们现在有一堆不规整的数据，我们需要为深度学习的前馈网络训练制作loader，那么我们可能需要将这些不规整的数据整合成 csv 文件格式，比如我们现在的一条数据为： 11.0 123 2.00 122.0，我们希望得到的数据为 11.0,123,2.00,122.0。在Python中，为了完成上述需求，一行代码足矣~~~

```
1 >>> data = "11.0 123 2.00 122.0"
2 >>> ",".join(data.split())
3 '11.0,123,2.00,122.0'
```

除了功能丰富的方法外，Python的 str 还提供了一些特性。

H3 成员运算符 in

其实 in 是Python中通用的成员运算符，只要你是可迭代对象，就可以通过 in 来判断我们想找的某个元素在不在其中。我们的 str 对象是可迭代对象，其迭代逻辑在魔法方法 __iter__ 中已经指明，感兴趣的同学可以去看一下source。

既然是可迭代对象，那么我们就可以通过 in 来判断某个子串是不是在目标字符串中，相信这会比调用 find() 方法更加优雅：

```
1 >>> 'str' in a
2 True
3 >>> 'int' in a
4 False
```

你也可以加入 `not` 来取反，达到判断某个子串不在目标字符串中。

```
1 >>> not('int' in a)
2 True
```

这么写其实不怎么优雅，Python中的 `not` 可以用得如同在写英文文章：

```
1 >>> 'int' not in a
2 True
```

H3 索引与切片

索引(index)，顾名思义。这个用法与C++中的 `string` 如出一辙：

```
1 >>> a = "0123456789"
2 >>> a[0]
3 '0'
4 >>> a[9]
5 '9'
6 >>> a[-1]    # -m代表倒数第m个
7 '9'
```

索引编号从0开始，你完全可以把这个字符串类比成C中的一维 `char` 数组。在C的数组中，我们可以通过 `[]` 操作符完成对数组元素的访问与修改。不过在Python中，`str` 是不可变对象，我们可以通过 `[]` 对 `str` 完成某个字符的访问，但是我们无法修改，修改就会报错：

```

1 >>> a = "don't adjust me!"
2 >>> a[2] = "1"          # 无法修改成单个字符
3 Traceback (most recent call last):
4   File "<stdin>", line 1, in <module>
5   TypeError: 'str' object does not support item assignment
6 >>> a[2] = "casual words"    # 也无法修改
7 Traceback (most recent call last):
8   File "<stdin>", line 1, in <module>
9   TypeError: 'str' object does not support item assignment

```

由于我们的索引只能取得一个字符，但是 `str` 对象存储的信息不分单个字符或是字符串，因此Python提供了切片(slice)操作，来使我们获取可迭代对象中连续一片元素。由于 `str` 是典型的可迭代对象，那么自然而然地，我们可以使用切片来操作 `str`。

对于字符串 `a`，`a[M:N]` 会返回 `a[M]` 到 `a[N-1]` 所有单个字符构成的子串：

```

1 >>> a = "0123456789"
2 >>> a[1:3]
3 '12'
4 >>> a[0:7]
5 '0123456'
6 >>> a[:7]
7 '0123456'
8 >>> a[3:-1]
9 '345678'
10 >>> a[3:]
11 '3456789'
12 >>> a[:]
13 '0123456789'

```

若参数 `M` 和 `N` 的参数没有给出，那么那一端的切片会达到那一端尽头。

其实，除了 `M,N` 两个参数，切片还有一个步长参数，步长参数必须是整数，代表每隔几步取一个元素：

```
1 >>> a[:]
2 '0123456789'
3 >>> a[::2]
4 '02468'
5 >>> a[1:9:2]
6 '1357'
7 >>> a[::-1]      # 逆置，且步长为1
8 '9876543210'
9 >>> a[::-2]      # 逆置，且步长为2
10 '97531'
```

若切片步长为负数，则会先将字符串逆置，然后再按步长绝对值切片。

需要说明的是，切片操作和索引操作在所有的可迭代对象中用法一致，这不是 `str` 的特性。不过对于更加高维的情况，索引和切片的用法会有所扩展（比如 `ndarray` 对象中的语法糖 `...`）感兴趣的同学自己去了解。

条件语句

Python的条件语句通过 `if`、`elif` 和 `else` 来控制。对于一个简单的二分支条件语句，Python中的基本结构为：

```
1 if <条件>:
2     语句1
3 else:
4     语句2
```

举两个例子。demo1，奇偶性判断：

```
1 >>> a = 3
2 >>> if a % 2 == 1:
3 ...     print("a is an odd")
4 ... else:
5 ...     print("a is an even")
6 ...
7 a is an odd
```

demo2，判断一个字符是否为空：

```
1 >>> txt = "hello"
2 >>> if len(txt) != 0:
3 ...     print("not empty")
4 ... else:
5 ...     print("empty")
6 ...
7 not empty
```

对于多分支，基本结构如下：

```
1 if 条件1:
2     语法1
3 elif 条件2:
4     语法2
5 elif 条件3:
6     语法3
7 .....
8 else:
9     语法n
```

举个例子。


```
1 >>> a = 123
2 >>> if 0 <= a <= 9:
3 ...     print("a 是一个一位数")
4 ... elif 10 <= a <= 99:
5 ...     print("a 是一个二位数")
6 ... elif 100 <= a <= 999:
7 ...     print("a 是一个三位数")
8 ... else:
9 ...     print("a 大于999")
10 ...
11 a 是一个三位数
```

Python的条件判断很简单，但是需要注意两点：

- 条件可以有括号，可以没有括号
- `if` , `elif` 和 `else` 后面必须要跟英文冒号
- 通过一个Tab键的缩进代表下面的语句属于该判断，Python中，缩进是语法的一部分。

H3 更为简洁的if-else

还记得C中的三元运算符 `?:` 吗？这个语句让我们能够一行完成一个简单的判断，并执行相应的语句，比如一个简单的 `max` 在C中就可以写成这样：

```
1 #include <stdio.h>
2 void main()
3 {
4     int a=1, b=2, c;
5     c = (a > b) ? a : b;    // c是a和b之间的最大值
6     printf(c);
7 }
```

这种简单的判断在Python中也可以实现，Python中支持把 `if-else` 写在一行，而且写法很贴近自然语言：

```
1 >>> a, b = 1, 2
2 >>> c = a if a > b else b # c是a和b之间的最小值
3 >>> c
4 2
```

将单行 `if-else` 与之后会讲到的列表生成式结合，可以把许多构造性的循环语句只用一行完成。

除了上面的常规写法，还有两种写法

```
1 >>> c = [b, a][a > b] # 改编1
2 >>> c
3 2
4 >>> c = (a > b and a or b) # 改编2
5 >>> c
6 2
```

改编1比较好理解，如果 `a>b` 成立，则返回前面那个列表中的第一个元素；不成立，则返回第二个。

改编2比较骚，没事情干别这么写。限于课时，此处不细讲，感兴趣的同学可以去看看这篇博客：

[python 中if-else的多种简洁的写法](#)

作为实用的用法，大家只要记住第一种就可以了。

```
1 c = a if a > b else b
```

循环语句

H3 while循环

Python中的循环只有 `for` 和 `while`，其中 `for` 循环的用法和C区别较大，所以先讲讲 `while`。

`while` 循环的基本格式为：

```
1 while 条件:
2     循环体语句
```

这个和C中没有区别，举个简单的例子：

```
1 >>> a = 3
2 >>> while a > 0:
3 ...     print(a)
4 ...     a -= 1
5 ...
6 3
7 2
8 1
```

需要注意的和前面的 `if-else` 一样：

- `while` 的后面要加冒号
- 循环语句要缩进一个Tab键

`while` 也可以配合 `else` 使用当 `while` 首次不满足而跳出循环时，`else` 下的语句会被执行一次：

```
1 >>> import random
2 >>> while random.random() < 0.2:
3 ...     print("20%的概率你都没抽到新六星，你也太非了=_=")
4 ... else:
5 ...     print("恭喜，你抽到了新干员！")
6 ...
7 恭喜，你抽到了新干员！
```

Python的循环语句中，也有 `break` 和 `continue`，用法和C中完全一样。`break` 用来打断最内层的循环：

```
1 >>> a = 5
2 >>> while True:
3 ...     a -= 1
4 ...     if a == 3:
5 ...         print("a is 3 and I break the loop!")
6 ...         break
7 ...     print(a)
8 ...
9 4
10 a is 3 and I break the loop!
```

`continue` 用来跳过最内层的该次循环：

```
1  >>> a = 5
2  >>> while a > 0:
3  ...     a -= 1
4  ...     if a == 3:
5  ...         print("a is 3 and I skip the loop!")
6  ...         continue
7  ...     print(a)
8  ...
9  4
10 a is 3 and I skip the loop!
11 2
12 1
13 0
```

`break` 和 `continue` 一般被称为循环控制语句，除了这两个外，Python中还有一个 `pass` 关键字。这个关键字就是`pass`，啥也不做。那么为什么Python要设计这样一个废柴关键字呢？

思考一个场景，你想写一个死循环，但是你还没想到怎么写，在C中你会这么写：

```
1  #include <stdio.h>
2
3  int main()
4  {
5      while true
6      {
7
8      }
9      return 0;
10 }
```

那在Python中呢？会报错：

```
1 >>> while True:
2 ...
3 File "<stdin>", line 2
4
5 ^
6 IndentationError: expected an indented block
```

因为格式是Python语法中的一部分，一旦有了冒号后，后面一定得跟上语句，因为不加语句解释器就无法解释 `while` 块下面应该执行什么，Python中的空语句不像C中那样能够灵活地被编译器解释。Python不存在空语句这一说法。为了满足我们这个要求，我们需要往 `while` 下面扔点什么东西，当你不知道要扔什么的时候，试试 `pass` 吧！它没有什么卵用，但是可以让你的程序被解释器成功执行。

```
1 >>> while True:      # 你可以通过Ctrl C来中断死循环
2     pass
```

H3 for循环

Python中的 `for` 循环与C中有很大的不同，如果要打印1-9的所有数字，我们在C中会这么写：

```
1 #include <stdio.h>
2 void main()
3 {
4     for (int i = 1; i ≤ 9; ++ i)
5         printf("%d ", i);
6 }
```

在Python中，如果我们要打印1-9的所有数字，我们会这么做：

```
1 >>> for i in range(1, 10):
2     ...     print(i)
3     ...
4     1
5     2
6     3
7     4
8     5
9     6
10    7
11    8
12    9
```

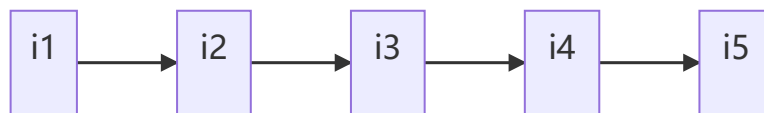
从中可以看出 `for i in range(M, N)` 语句会生成 `M` 到 `N-1` 的数字。其实这与切片很像：`a[M,N]` 切片是返回 `a[M]` 到 `a[N-1]` 这么多数字。类似于切片，`range()` 函数也存在步长，比如我们现在要打印1-9所有奇数，就可以这么打印：

```
1 >>> for i in range(1, 10, 2):
2     ...     print(i)
3     ...
4     1
5     3
6     5
7     7
8     9
```

`range()` 函数也可以只接受一个参数 `n`，那么循环中就会遍历0到`n-1`所有的整数：

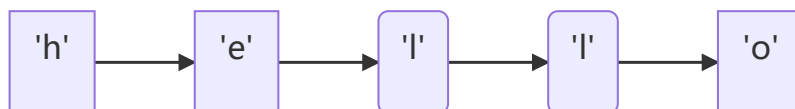
```
1 >>> for i in range(3):
2     ...     print(i)
3     ...
4     0
5     1
6     2
```

那么for循环这么写的内在逻辑是什么呢？我们首先得了解一个概念：迭代器。我说不出严格的定义，大家可以先有一个感性的认知。一个内部存储元素可以被逐步访问的东西被我们称为迭代器。



如图上图所示的链式结构，我们可以通过某种方式逐个访问上述 i1 到 i5 这五个元素（线性表通过索引值和内存地址的线性映射得到下一个元素的地址，链表通过指针获取下一个元素的地址，总之就是可以访问这个结构内的所有元素啦）。由于我们可以顺序访问这个结构中所有的元素，因此，我们称之为迭代器；我也称该对象是可迭代的。

我们之前讲过的 str 对象就是一个迭代器，这个迭代器中的每个元素就是一个字符。比如现在有一个字符串 a="hello"，那么这个字符串可以看成是一个如下图的迭代器：



后续我们就可以通过for循环来访问这个迭代器，拭目以待。

为什么要讲迭代器呢？因为Python中的for循环实际上是在遍历迭代器。比如刚才的字符串，我们可以通过for循环来顺序访问字符串这个迭代器中的每个元素：


```
1 >>> a = "hello"
2 >>> for i in a:
3 ...     print(i)
4 ...
5 h
6 e
7 l
8 l
9 o
```

上述过程可以简单理解为：循环开始，`i` 这个变量赋为迭代器 `a` 中的首个元素，在本例中，迭代器 `a` 是字符串，字符串的首元素是 `'h'`，所以循环第一轮，`i='h'`；接着 `i` 指向下一个元素，并赋为该元素的值，进行第二轮循环，以此类推，直到迭代器的最后一个元素访问结束，那么做完该轮循环后，`for`循环结束。

因此Python中循环的基本结构为：

```
1 for 迭代元素 in 迭代器:
2     循环体语句
```

这个语法其实和自然语言很贴近，也就是我们访问迭代器中的每个元素，每个元素要做什么操作，`for`循环不就在说这么一件事情吗？

你现在已经知道了这么一件事：`for i in range(10)` 会在 `i=0, ..., 9` 执行10次循环。其实 `range()` 函数就是返回一个 `range` 类型的迭代器，我们试试：

```
1 >>> iter = range(10)
2 >>> type(iter)      # 查看迭代器的类型
3 <class 'range'>
4 >>> len(iter)       # range对象也定义了__len__魔法方法，所以可以使用len()方法查看
                        迭代器长度
5 10
6 >>> iter[2]         # 这个迭代器也支持索引，由于每个元素都是整数，所以返回整数
7 2
8 >>> iter[::-2]      # 也支持切片操作，由于iter本身是range对象，所以它的子集也是
                        range对象
9 range(0, 10, 2)
```

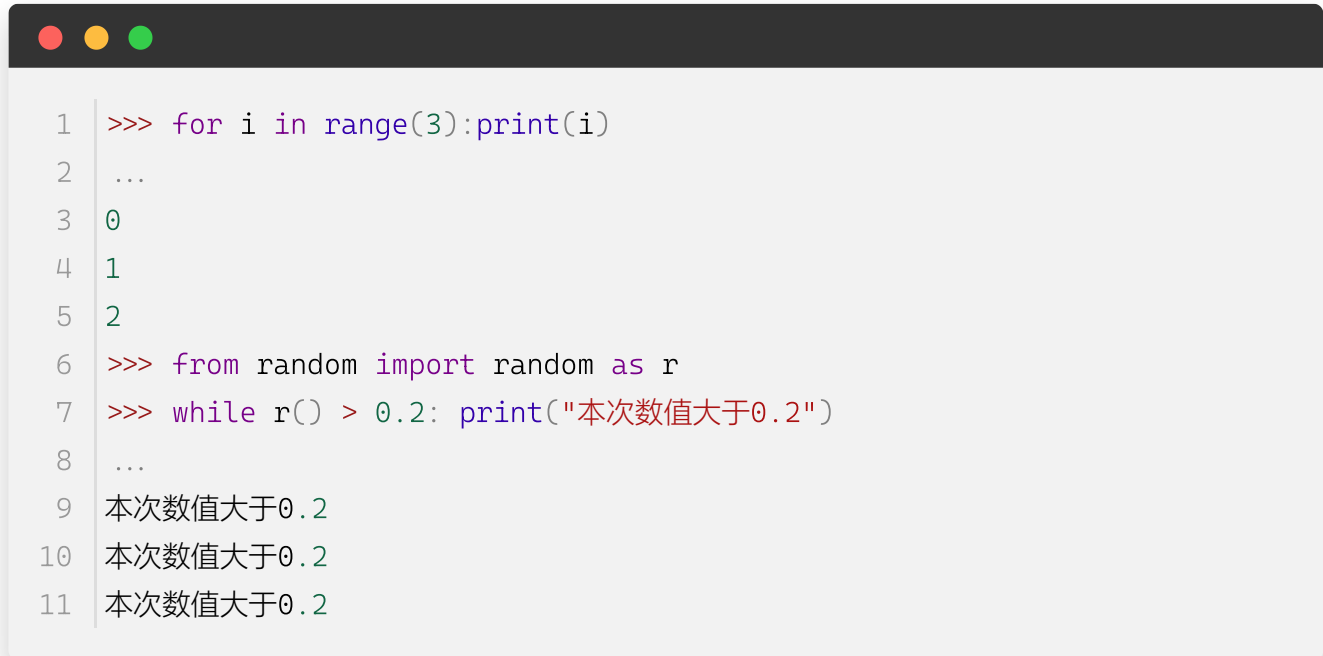
由此，以下的代码你也应该看得懂：

```
1 >>> for i in range(10)[::-1]:
2 ...     print(i)
3 ...
4 9
5 8
6 7
7 6
8 5
9 4
10 3
11 2
12 1
13 0
```

需要说明的是，`range()` 函数的步长不支持负数，所以如果你想要达到C中 `for (int i = 9; i ≥ 1; -- i)` 这样的效果，上面的写法是Python中不可多得的写法。

H3 更加简洁的循环语句

如果你想要执行的循环体语句可以写成一行，那么你的 `while` 循环或者 `for` 循环的循环体语句可以和循环关键字写在一行，这在后面的列表生成式中用得很多很多很多：



```
1 >>> for i in range(3):print(i)
2 ...
3 0
4 1
5 2
6 >>> from random import random as r
7 >>> while r() > 0.2: print("本次数值大于0.2")
8 ...
9 本次数值大于0.2
10 本次数值大于0.2
11 本次数值大于0.2
```

个人建议：除非是列表生成式，否则别把循环写在一行上，原因很简单：破坏了Python的业界码风，而且这样写真的很丑，很不优雅。。。

SIMPLE EXERCISE

下面开始愉快的刷题环节，我们将更新编程装备，将命令行式的编程变为IDE式编程，也就是回到你们喜欢的那种写完一堆代码，再编译执行的dev时代。而不是我们现在这种写一行，执行一行的命令行。

你可以使用Python解释器自带的IDLE，而且它有语法高亮很不错。当然，喜欢花里胡哨的、想要成为优秀的业余平面设计的你，不妨试试万能编辑器 `vscode` 或者自动补全最丝滑的编辑器 `pycharm`，不过请确保你的python解释器的路径已经扔进了环境变量中。

vscode与pycharm的成功执行需要配置一些简单的选项，本节课不讲。

H3 EX1 字符串

01. 判断 name 变量对应的值是否以 "X" 结尾，并输出结果。

```
1 name = "aleX"
2
3 if name[-1] == "X":
4     print("Yes")
5 else:
6     print("No")
```

02. 判断 name 变量对应的值是否以 "al" 开头，并输出结果。

```
1 name = "aleX"
2
3 if name.find("al") == 0:
4     print("Yes")
5 else:
6     print("No")
```

03. 将 name 变量对应的值中的 "l" 替换为 "p"，并输出结果。

```
1 name = "aLeX"
2
3 print(name.replace("l", "p"))
```

04. 我们现在有一个存有整形数的变量 `a`，输出 `a` 的位数：

```
1 a = 1234
2 print(len(str(a)))
```

05. 我们现在有一个存有浮点数的变量 `a`，输出 `a` 的整数部分的位数和小数部分的位数。

```
1 a = 12.08
2 b, c = str(a).split(".")
3 print(len(b))    # 整数部分的位数
4 print(len(c))    # 小数部分的位数
```

06. 假设我们的标记语言为 `LaTeX`，请将分别打印大写和小写后的值。

```
1 lag = "LaTeX"
2
3 print(lag.upper())
4 print(lag.lower())
```

o7. 还是o4的要求，但是不允许使用 `str` 对象自己的方法。

```
1 lag = "LaTeX"
2
3 upper = ""
4 for i in lag:
5     if "A" <= i <= "Z":      # 这个字符已经是大写，直接加入结果中
6         upper += i
7     else:                   # 这个字符是小写，需要先转成大写，再加入结果中
8         res = ord("a") - ord("A")  # 我们得知道在Unicode编码中a与A差
多少
9         up_ch = chr(ord(i) - res)  # 通过ord()与chr()进行转换
10        upper += up_ch
11
12 print(upper)
```

转小写留做作业

o8. 倒序输出"hello world"。

```
1 print("hello world"[::-1])
```

o9. 对于字符串 `---split---`，分别去除左边所有的 `-`、右边所有的 `-` 和两边所有的 `-`，并打印去除后的字符串。

```
1 string = "---strip---"
2 print(string.lstrip("-"))    # 去除左边的-
3 print(string.rstrip("-"))    # 去除右边的-
4 print(string.strip("-"))     # 去两边的-
```

10. 统计字符串 `asdfghjklasdfghjkl` 中子串 `as` 的个数。

```
1 s = "asdfghjklasdfghjkl"
2 print(s.count("as"))
```

H3 EX2 条件&循环语句

01. 输出1-99之间所有不含因子5的整数之和。

```
1 count = 0
2 for i in range(1, 100):
3     if i % 5 == 0:
4         continue
5     count += i
6 print(count)
```

这道题还可以通过 `result = sum([0 if i % 5 == 0 else i for i in range(1, 100)])` 一行解决，这涉及到下节课会讲到的列表生成式。感兴趣的同学可以自己回去研究一下。

02. 现在有三个数字 `a=1, b=1, c=1.1`，请输出其中的最大值。

```
1 a, b, c = 1, 1, 1.1
2
3 if a >= b and a >= c:
4     print(a)
5 elif b >= a and b >= c:
6     print(b)
7 else:
8     print(c)
```

03. 输出1-100内所有的质数。

```
1 for i in range(1, 101):
2     if i == 1: # 1不是质数
3         continue
4     is_prime = True
5     for j in range(2, i):
6         if i % j == 0:
7             is_prime = False
8     if is_prime:
9         print(i)
```

04. 打印99乘法表。

```
1 for i in range(1, 10):
2     for j in range(1, i + 1):
3         print("{}x{}={}".format(i, j, i * j), end=" ")
4     print() # 为了输出回车
```


05. 输出字符串 "asd123 12 123mmk1 ?A!#As" 中字母，数字，空格，其他字符的个数。

```
1 ch = 0
2 num = 0
3 space = 0
4 other = 0
5 for i in "asd123 12 123mmk1 ?A!#As":
6     if i.isalpha():
7         ch += 1
8     elif i.isdigit():
9         num += 1
10    elif i == " ":
11        space += 1
12    else:
13        other += 1
14
15 print(ch, num, space, other)
```