流程控制

在相对深入了解了值的基本操作之后，我们需要再返回来对流程控制做更深入的了解。

之前我们看过这个寻找质数的程序：

```
for n in range(2, 100):
    if n == 2:
        print(n)
        continue
    for i in range(2, n):
        if (n % i) == 0:
            break
    else:
        print(n)
```

这其中，包含了_分支_与_循环_ —— 无论多复杂的流程控制用这两个东西就够了，就好像无论多复杂的电路最终都是由通路和开路仅仅两个状态构成的一样。

> 今天的人们觉得这是 “天经地义” 的事情，可实际上并非如此。这是 1966 年的一篇论文所带来的巨大改变 —— *Communications of the ACM* by Böhm and Jacopini (1966)。实际上，直到上个世纪末，`GOTO` 语句才从各种语言里近乎 “灭绝”.....
>
> 任何进步，无论大小，其实都相当不容易，都非常耗时费力 —— 在哪儿都一样。有兴趣、有时间，可以去浏览 Wikipedia 上的简要说明 —— Wikipedia: Minimal structured control flow。

if 语句

`if` 语句的最简单构成是这样 —— 注意第 1 行末尾的冒号：

```
if expression:
    statements
```

如果表达式 `expression` 返回值为真，执行 `if` 语句块内部的 `statements`，否则，什么都不做，执行 `if` 之后的下一个语句。

```
import random
r = random.randrange(1, 1000)

if r % 2 == 0:
    print(f'{r} is even.')
```

如果，表达式 `expression` 返回值无论真假，我们都需要做一点相应的事情，那么我们这么写：

```
if expression:
    statements_for_True
else:
    statements_for_False
```

如果表达式 expression 返回值为真，执行 if 语句块内部的 statements_for_True，否则，就执行 else 语句块内部的 statements_for_False

```
import random
r = random.randrange(1, 1000)

if r % 2 == 0:
    print(f'{r} is even.')
else:
    print(f'{r} is odd.')
```

```
126 is even.
```

有时，表达式 <expression> 返回的值有多种情况，并且针对不同的情况我们都要做相应的事情，那么可以这么写：

```
if expression_1:
    statements_for_expression_1_True

elif expression_2:
    statements_for_expression_2_True

elif expression_3:
    statements_for_expression_3_True

elif expression_...:
    statements_for_expression_..._True
```

Python 用 elif 处理这种多情况分支，相当于其它编程语言中使用 switch 或者 case.....

elif 是 else if 的缩写，作用相同。

以下程序模拟投两个骰子的结果 —— 两个骰子数字加起来，等于 7 算平，大于 7 算大，小于 7 算小：

```
import random
r = random.randrange(2, 13)

if r == 7:
    print('Draw!')
```

```
elif r < 7:
    print('Small!')
elif r > 7:
    print('Big!')
```

```
Big!
```

当然你还可以模拟投飞了的情况，即，最终的骰子数是 0 或者 1，即，< 2：

```
import random
r = random.randrange(0, 13) # 生成的随机数应该从 0 开始了；

if r == 7:
    print('Draw!')
elif r >= 2 and r < 7:      # 如果这里直接写 elif r < 7: ，那么，else: 那一部分永远不会被执行……
    print('Small!')
elif r > 7:
    print('Big!')
else:
    print('Not valid!')
```

```
Small!
```

for 循环

Python 语言中，for 循环不使用其它语言中那样的计数器，取而代之的是 range() 这个我称其为“整数等差数列生成器”的函数。

用 C 语言写循环是这样的：

```
for( a = 0; a < 10; a = a + 1 ){
    printf("value of a: %d\n", a);
}
```

用 Python 写同样的东西，是这样的：

```
for a in range(10):
    print(f'value of a: {a}') #每次 a 的值都不同，从 0 递增至 9
```

```
value of a: 0
value of a: 1
value of a: 2
value of a: 3
value of a: 4
value of a: 5
value of a: 6
value of a: 7
value of a: 8
value of a: 9
```

range() 函数

`range()` 是个内建函数，它的文档是这样写的：

> **range**(*stop*)
>
> **range**(*start, stop[, step]*)

只有一个参数的时候，这个参数被理解为 `stop`，生成一个从 `0` 开始，到 `stop - 1` 的整数数列。

这就解释了为什么有的时候我们会在 `for ... in range(...):` 这种循环内的语句块里进行计算的时候，经常会在变量之后写上 `+ 1`，因为我们 range(n) 的返回数列中不包含 `n`，但我们有时候却需要 `n`。点击这里返回看看第一章里提到的例子：所谓算法那一小节。

```
from IPython.core.interactiveshell import InteractiveShell
InteractiveShell.ast_node_interactivity = "all"

range(10)
list(range(10)) # 将 range(10) 转换成 list，以便清楚看到其内容。
```

```
range(0, 10)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

`start` 参数的默认值是 `0`。如需指定起点，那么得给 `range()` 传递两个参数，比如，`range(2, 13)`.....

```
list(range(2, 13))
```

```
[2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12]
```

第三个参数可选； `step`，步长，就相当于是“等差数列”当中的“差”，默认值是 `1`。例如，`range(1, 10, 2)` 生成的是这样一个数列 `[1, 3, 5, 7, 9]`。所以，打印 `0 ~ 10` 之间的所有奇数，可以这样写：

```
for i in range(1, 10, 2):
    print(i)
```

```
1
3
5
7
9
```

我们也可以生成负数的数列：

```
list(range(0, -10, -1))
```

```
[0, -1, -2, -3, -4, -5, -6, -7, -8, -9]
```

Continue、Break 和 Pass

在循环的过程中，还可以用 `continue` 和 `break` 控制流程走向，通常是在某条件判断发生的情况下 —— 正如你早就见过的那样：

```
for n in range(2, 100):
    if n == 2:
        print(n)
        continue
    for i in range(2, n):
        if (n % i) == 0:
            break
    else:
        print(n)
```

`continue` 语句将忽略其后的语句开始下次循环，而 `break` 语句将从此结束当前循环，开始执行循环之后的语句：

`for` 语句块还可以附加一个 `else` —— 这是 Python 的一个比较有个性的地方。附加在 `for` 结尾的 `else` 语句块，*在没有 `break` 发生的情况下会运行*。

```
for n in range(2, 100):
    if n == 2:
        print(n)
        continue
    for i in range(2, n):
        if (n % i) == 0:
            break
    else:               # 下一行的 print(n) 事实上属于语句块 for i in range(2, n):
        print(n)        # 整个循环结束，都没有发生 break 的情况下，才执行一次 print(n)
```

```
2
3
5
7
11
13
17
19
23
29
31
37
41
43
47
53
59
61
67
71
73
79
83
89
97
```

试比较以下两段代码：

```
for n in range(2, 100):
    if n == 2:
        print(n)
        continue
    for i in range(2, n):
        if (n % i) == 0:
```

```
            break
    print(n)            # 事实上相当于针对 range(2, 100) 中每个 n 都执行了一次
print(n)
                        # 这个 print(n) 属于语句块 for n in range(2, 100):
```

```
for n in range(2, 100):
    if n == 2:
        print(n)
        continue
    for i in range(2, n):
        if (n % i) == 0:
            break
        print(n)            # 事实上相当于针对 range(2, n) 中每个 n 都执行了一次
print(n)
```

```
2
3
5
5
5
7
7
7
7
7
9
11
11
11
11
...
97
97
97
97
97
97
99
```

pass 语句什么都不干：

再比如，

```
def someFunction():
    pass
```

又或者：

```
for i in range(100):
    pass
    if i % 2 == 0:
        pass
```

换个角度去理解的话可能更清楚：`pass` 这个语句更多是给写程序的人用的。当你写程序的时候，你可以用 `pass` 占位，而后先写别的部分，过后再回来补充本来应该写在 `pass` 所在位置的那一段代码。

写嵌套的判断语句或循环语句的时候，最常用 `pass`，因为写嵌套挺费脑子的，一不小心就弄乱了。所以，经常需要先用 `pass` 占位，而后逐一突破。

while 循环

今天，在绝大多数编程语言中，都提供两种循环结构：

> - Collection-controlled loops（以集合为基础的循环）
> - Condition-controlled loops（以条件为基础的循环）

之前的 `for ... in ...` 就是 Collection-controlled loops；而在 Python 中提供的 Condition-controlled loops 是 `while` 循环。

`while` 循环的格式如下：

```
while expression:
    statements
```

输出 1000 以内的斐波那契数列的程序如下：

```
n = 1000
a, b = 0, 1
while a < n:
    print(a, end=' ')
    a, b = b, a+b
print()
```

```
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987
```

`for` 和 `while` 的区别在哪里？什么时候应该用哪个？

`for` 更适合处理序列类型的数据（Sequence Type）的迭代，比如处理字符串中的每一个字符，比如把 `range()` 返回的数列当作某种序列类型的索引。

`while` 更为灵活，因为它后面只需要接上一个逻辑表达式即可。

一个投骰子赌大小的游戏

虽然还不可能随心所欲写程序，但是，你现在具备了起码的“阅读能力”。有了以上大概的介绍，你也许可以读懂一些代码了 —— 它们在你眼里再也不是天书了……

以下是一个让用户和程序玩掷骰子赌大小的程序。规则如下：

> - 每次计算机随机生成一个 `2... 12` 之间的整数，用来模拟机器人投两个骰子的情况；

- 机器人和用户的起始资金都是 10 个硬币
- 要求用户猜大小：
  - 用户输入 `b` 代表“大”；
  - 用户输入 `s` 代表“小”；
  - 用户输入 `q` 代表“退出”；
- 用户的输入和随机产生的数字比较有以下几种情况：
  - 随机数小于 `7`，用户猜小，用户赢；
  - 随机数小于 `7`，用户猜大，用户输；
  - 随机数等于 `7`，用户无论猜大还是猜小，结局平，不输不赢；
  - 随机数大于 `7`，用户猜小，用户输；
  - 随机数大于 `7`，用户猜大，用户赢；
- 游戏结束条件：
  - 机器人和用户，若任意一方硬币数量为 `0`，则游戏结束；
  - 用户输入了 `q` 主动终止游戏。

```
from random import randrange

coin_user, coin_bot = 10, 10 # 可以用一个赋值符号分别为多个变量赋值
rounds_of_game = 0

def bet(dice, wager):    # 接收两个参数，一个是骰子点数，另一个用户的输入
    if dice == 7:
        print(f'The dice is {dice};\nDRAW!\n') # \n 是换行符号
        return 0
    elif dice < 7:
        if wager == 's':
            print(f'The dice is {dice};\nYou WIN!\n')
            return 1
        else:
            print(f'The dice is {dice};\nYou LOST!\n')
            return -1
    elif dice > 7:
        if wager == 's':
            print(f'The dice is {dice};\nYou LOST!\n')
            return -1
        else:
            print(f'The dice is {dice};\nYou WIN!\n')
            return 1
```

```
while True:         #  除 for 之外的另外一个循环语句
    print(f'You: {coin_user}\t Bot: {coin_bot}')
    dice = randrange(2, 13)   # 生成一个 2 到 12 的随机数
    wager = input("What's your bet? ")
    if wager == 'q':
        break
    elif wager in 'bs':  # 只有当用户输入的是 b 或者 s 得时候，才“掷骰子”……
        result = bet(dice, wager)
        coin_user += result    # coin_user += result 相当于 coin_user = coin_user
+ result
        coin_bot -= result
        rounds_of_game += 1
    if coin_user == 0:
        print("Woops, you've LOST ALL, and game over!")
        break
    elif coin_bot == 0:
        print("Woops, the robot's LOST ALL, and game over!")
        break

print(f"You've played {rounds_of_game} rounds.\n")
print(f"You have {coin_user} coins now.\nBye!")
```

总结

有控制流，才能算得上是程序。

- 只处理一种情况，用 `if ...`
- 处理 `True`/`False` 两种情况，用 `if ... else ...`
- 处理多种情况，用 `if ... elif ... elif ... else ...`
- 迭代有序数据类型，用 `for ... in ...`，如果需要处理没有 `break` 发生的情况，用 `for ... else ...`
- 其它循环，用 `while ...`
- 与循环相关的语句还有 `continue`、`break`、`pass`
- 函数从控制流角度去看其实就是子程序