



首都师范大学

為學為師 求實求新

高级程序设计

---Python与深度学习

5. 赋值、条件和循环

5. Assignment if and loop

李冰

副研究员

交叉科学研究院



课程内容

- 赋值
 - 序列赋值
 - 序列引用
 - 序列解包
 - 多目标赋值
 - 增强赋值语句
- 条件
 - if/else
- 循环
 - while/for循环
- 序列生成函数
 - rang()
 - zip()
 - enumerate()

赋值语句

- 赋值语句将对象赋给一个变量，其基本形式是在等号左边写赋值语句的目标，而要赋值的对象位于右侧。

运算	解释
<code>spam='Spam'</code>	基本形式
<code>spam, ham = 'yum', 'YUM'</code>	元组赋值运算
<code>[spam, ham] = ['yum', 'YUM']</code>	列表赋值运算
<code>a, b, c, d = 'spam'</code>	序列赋值运算
<code>a, *b = 'spam'</code>	序列解包
<code>spam = ham = 'lunch'</code>	多目标赋值
<code>spams += 42</code>	增强赋值运算 (相当于 <code>spams = spams + 42</code>)

赋值语句

- 赋值语句将对象赋给一个变量，其基本形式是在等号左边写赋值语句的目标，而要赋值的对象位于右侧。

运算	解释
<code>spam='Spam'</code>	基本形式
<code>spam, ham = 'yum', 'YUM'</code>	元组赋值运算
<code>[spam, ham] = ['yum', 'YUM']</code>	列表赋值运算
<code>a, b, c, d = 'spam'</code>	序列赋值运算
<code>a, *b = 'spam'</code>	序列解包
<code>spam = ham = 'lunch'</code>	多目标赋值
<code>spams += 42</code>	增强赋值运算 (相当于 <code>spams = spams + 42</code>)

序列赋值

```
theta = 1  
delta = 2  
[C, D] = [theta, delta] # List assignment  
C, D
```

(1, 2)

```
A, B = theta, delta # Tuple assignment  
A, B
```

(1, 2)

列表赋值语句

元组赋值语句

序列赋值

```
theta = 1  
delta = 2  
[C, D] = [theta, delta] # List assignment  
C, D
```

(1, 2)

```
A, B = theta, delta # Tuple assignment  
A, B
```

(1, 2)

列表赋值语句

元组赋值语句是 Python 中一个常用的编写代码技巧

序列赋值

```
theta = 1  
delta = 2  
[C, D] = [theta, delta] # List assignment  
C, D
```

(1, 2)

```
A, B = B, A # Swap values  
A, B
```

(2, 1)

```
[a, b, c] = (1, 2, 3)  
a, c
```

(1, 3)

```
(a, b, c) = 'ABC'  
a, c
```

('A', 'C')

序列赋值

- Python 支持右侧是任何类型的序列（可迭代的对象），只要左右两侧长度相等即可。

```
for (a, b, c) in [(1, 2, 3), (4, 5, 6)]:  
    print(a, c)
```

```
1 3  
4 6
```

```
for ((a, b), c) in [((1, 2), 3), ((4, 5), 6)]:  
    print(a, c)
```

```
1 3  
4 6
```


序列赋值

- Python 支持右侧是任何类型的序列（可迭代的对象），只要左右两侧长度相等即可。

```
L = [1, 2, 3, 4]
while L:
    front, L = L[0], L[1:]
    print(front, L)
```

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

序列解包

- 序列赋值

- 要求左边的目标名称数量与右边的主体中的数量完全相同。

```
#a, b, c = [1, 2, 3, 4] # ValueError: too many values to unpack
```

序列解包

- 序列赋值

- 要求左边的目标名称数量与右边的主体中的数量完全相同。

```
#a, b, c = [1, 2, 3, 4] # ValueError: too many values to unpack
```

- 序列解包

- 在目标中使用带单个星号的名称来更通用的匹配。

```
a, *b = [1, 2, 3, 4]  
a
```

1

```
b
```

[2, 3, 4]

```
*a, b = [1, 2, 3, 4]  
a, b
```

([1, 2, 3], 4)

```
a, *b, c = [1, 2, 3, 4]  
a, b, c
```

(1, [2, 3], 4)

带星号的名称可以出现在目标中的任何地方。

序列解包

- 序列解包

- 在目标中使用带单个星号的名称来更通用的匹配。

```
a, *b = 'spam'  
a, b  
( 's', [ 'p', 'a', 'm' ] )
```

对于任何序列类型都有效

序列解包

```
L = [1, 2, 3, 4]
while L:
    front, *L = L
    print(front, L)
```

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

序列解包总是返回多个匹配项的一个列表

```
for (a, *b, c) in [(1, 2, 3, 4), (5, 6, 7, 8)]:
    print(a, b, c)
```

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

对于for 循环有效

解包边界情况

- 首先，带星号的名称可能只匹配单个的项，但总是会向其赋值一个列表：

```
a, b, c, *d = [1, 2, 3, 4]
print(a, b, c, d)
```

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

如果没有剩下的内容可以匹配带星号的名称，它会赋值一个空的列表

```
a, b, c, d, *e = [1, 2, 3, 4]
print(a, b, c, d, e)
```

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

```
a, b, *c, d, e = [1, 2, 3, 4]
print(a, b, c, d, e)
```

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

如果有多个带星号的名称，以及带星号的名称没有编写到一个列表或元组中，都会引发错误。

```
#a, *b, *c, d = [1, 2, 3, 4] # SyntaxError: two starred expressions in assignment
```

```
#*a = [1, 2, 3, 4] # SyntaxError: starred assignment target must be in a l
```

多目标赋值语句

- 直接把所有提供的变量名都赋值给右侧的对象。

```
a = b = c = 'spame'  
print(a, b, c)
```

spame spame spame

```
a = b = []  
b.append(42)  
a, b
```

([42], [42])

多目标赋值语句

- 直接把所有提供的变量名都赋值给右侧的对象。

```
a = b = c = 'spame'  
print(a, b, c)
```

```
spame spame spame
```

```
a = b = []  
b.append(42)  
a, b
```

```
([42], [42])
```

因为 a 和 b 引用相同的对象，通过 b 在原处添加元素上去，通过 a 也会看见修改的结果。

多目标赋值语句

- 直接把所有提供的变量名都赋值给右侧的对象。

```
a = b = c = 'spame'  
print(a, b, c)
```

```
spame spame spame
```

```
a = b = []  
b.append(42)  
a, b
```

```
((42), [42])
```

因为 a 和 b 引用相同的对象，通过 b 在原处添加元素上去，通过 a 也会看见修改的结果。

```
a = []  
b = []  
b.append(42)  
a, b
```

```
(([]), [42])
```

为了避免这个问题，要在单独的语句中初始化可变对象，以便分别执行独立的常量表达式来创建独立的空对象：

增强赋值语句

- 传统形式: $x = x + y$
- 增强赋值: $x += y$

```
x = 1
x += 2
x
```

3

```
S = 'spam'
S += 'SPAM'           # Implied concatenation
S
'spamSPAM'
```

每个 Python 二元表达式的运算符都有对应的增强赋值形式:

$x += y$ $x -= y$ $x *= y$ $x /= y$

$x \% = y$ $x ** = y$ $x //= y$ $x \& = y$

$x |= y$ $x \wedge = y$ $x << = y$ $x >> = y$

增强赋值语句

- 推荐使用
 - 自动优化、计算次数减少一次

```
L = [1, 2]  
L = L + [3]
```

Slower concatenate

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

```
L += [4]
```

Faster

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

- 隐含了原处修改的意思

```
L = [1, 2]
```

```
M = L
```

```
L = L + [3, 4]
```

Concatenation makes a new object

```
L, M
```

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

```
L = [1, 2]
```

```
M = L
```

```
L += [3, 4]
```

+= really means in-place extend

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

课程内容

- 赋值
 - 序列赋值
 - 序列引用
 - 序列解包
 - 多目标赋值
 - 增强赋值语句
- 条件
 - if/else
- 循环
 - while/for循环
- 序列生成函数
 - rang()
 - zip()
 - enumerate()

条件语句

- if 语句

- 复合语句 = 首行 + ":" + 缩进语句

```
if test1:                # if test
    statements1           # Associated block
elif test2:              # Optional elifs
    statements2
else:                    # Optional else
    statements3
```

```
x = 'killer rabbit'
if x == 'roger':
    print("shave and a haircut")
elif x == 'bugs':
    print("what's up doc?")
else:
    print('Run away! Run away!')
```

Run away! Run away!

if / else三元表达式

```
if X:  
    A = Y  
else:  
    A = Z
```

`A = Y if X else Z`

```
A = 't' if 'spam' else 'f'  
A
```

For strings, nonempty means true

't'

```
A = 't' if '' else 'f'  
A
```

'f'

真值测试

- Python的三种表达式运算符为 **'and' 'or' 'not'**
 - 任何非零数字或非空对象都为真
 - 数字零、空对象以及特殊对象 `None` 都被认作是假
 - 布尔 **and** 和 **or** 运算符会返回真或假的操作对象

真值测试

- Python 会从左向右求算操作对象，然后返回第一个为真的操作对象
- 如果左边操作数为假，则计算右边的操作数并将其返回。

```
2 or 3, 0 or 3
```

```
(2, 3)
```

```
2 and 3, 0 and 2
```

```
(3, 0)
```

```
[] or 3
```

```
3
```

```
3 and []
```

```
[]
```


课程内容

- 赋值
 - 序列赋值
 - 序列引用
 - 序列解包
 - 多目标赋值
 - 增强赋值语句
- 条件
 - if/else
- 循环
 - while/for循环
- 序列生成函数
 - rang()
 - zip()
 - enumerate()

循环

- while循环和for 循环

- while 语句

- 只要顶端测试一直为真值，就会重复执行一个语句块（通常有缩进）。当测试为假时，控制权会传给 while 块后的语句。

```
while <test>:
    <statements1>
else:
    <statements2>
```

Loop test
Loop body
Optional else
Run if didn't exit loop with break

while 循环

```
x = 'spam'|
while x:
    print(x, end=' ')
    x = x[1:]
# While x is not empty
# In 2.X use print x,
# Strip first character off x
```

spam pam am m

•无限循环

```
#while True:
#    print('Type Ctrl-C to stop me!')
```

while 循环

• break、continue、pass和循环else

```
while <test1>:
    <statements1>
    if <test2>: break           # Exit loop now, skip else if present
    if <test3>: continue       # Go to top of loop now, to test1
else:
    <statements2>             # Run if we didn't hit a 'break'
```

• break

- 跳出整个循环语句

• continue

- 跳到最近所在循环的开头处

• 循环else块

- 只有当循环正常离开（没有碰到 break 语句）才会执行

while循环

- continue

```
x = 10
while x:
    x = x - 1
    if x % 2 != 0:
        continue
    print(x, end=' ')
```

Or, x -= 1
Odd? -- skip print

8 6 4 2 0

while循环

- continue

```
x = 10
while x:
    x = x - 1
    if x % 2 != 0:
        continue
    print(x, end=' ')
```

Or, x -= 1
Odd? -- skip print

8 6 4 2 0

```
x = 10
while x:
    x = x - 1
    if x % 2 == 0:
        print(x, end=' ')
```

Even? -- print

8 6 4 2 0

推荐这个

while 循环

- break

```
while True:
    name = input('Enter name:')          # Use raw_input() in 2.X
    if name == 'stop':
        break
    age = input('Enter age: ')
    print('Hello', name, '=>', int(age) * 2)
```

```
Enter name:Bob
Enter age: 20
Hello Bob => 400
Enter name:stop
```

while循环

- 循环else:
 - Python特有

```
y = 12
x = y // 2                # For some y > 1
while x > 1:
    if y % x == 0: # Remainder
        print(y, 'has factor', x)
        break      # Skip else
    x -= 1
else:                  # Normal exit
    print(y, 'is prime')
```

12 has factor 6

while循环

- 例如，假设需要写个循环搜索列表的值，而且需要知道离开循环后该值是否已找到，可能会用这种方式编写该任务：

```
found = False
while x and not found:
    if match(x[0]):           # Value at front?
        print('Ni')
        found = True
    else:
        x = x[1:]            # Slice off front and repeat
if not found:
    print('not found')
```

while循环

- 例如，假设需要写个循环搜索列表的值，而且需要知道离开循环后该值是否已找到，可能会用这种方式编写该任务：
 - 用循环else

```
while x:                                # Exit when x empty
    if match(x[0]):
        print('Ni')
        break                            # Exit, go around else
    x = x[1:]
else:
    print('Not found')                   # Only here if exhausted x
```

for 循环

- for 循环在 Python 中是一个通用的序列迭代器：
 - 可以遍历任何有序的序列对象元素。
 - for 语句可用于字符串、列表、元组、其它内置可迭代对象以及自己创建的一类对象。

```
for target in object:           # Assign object items to target
    statements                  # Repeated loop body: use target
else:                           # Optional else part
    statements                  # If we didn't hit a 'break'
```

- 当运行 for 循环时，会逐个将序列对象中的元素赋值给目标，然后为每个元素执行循环主体。
- 循环主体一般使用赋值的目标来引用序列中当前的元素，因此目标就像遍历序列的游标。
- for 循环也支持一个可选的 else 块

for 循环

- 如果循环离开时没有碰到 `break` 语句，就会执行（也就是所有元素都访问过了）。

```
for target in object:           # Assign object items to target
    statements
    if test:
        break                   # Exit loop now, skip else
    if test:
        continue               # Go to top of loop now
    else:
        statements              # If we didn't hit a 'break'
```

for 循环

- 基本应用

- for 循环遍历任何一种序列对象，包括列表、字符串和元组。

```
for x in ["spam", "eggs", "ham"]:  
    print(x, end='')
```

spam eggs ham

```
S = "lumberjack"  
for x in S:  
    print(x, end=' ')           # Iterate over a string
```

l u m b e r j a c k

```
T = ("and", "I'm", "okay")  
for x in T:  
    print(x, end=' ')           # Iterate over a tuple
```

and I'm okay

for 循环

- 在 for 循环中的元组赋值

```
T = [(1, 2), (3, 4), (5, 6)]  
for (a, b) in T:                # Tuple assignment at work  
    print(a, b)
```

```
1 2  
3 4  
5 6
```

for 循环

- 在 for 循环中的元组赋值

```
T = [(1, 2), (3, 4), (5, 6)]  
for (a, b) in T:                # Tuple assignment at work  
    print(a, b)
```

```
1 2  
3 4  
5 6
```

- 在循环中解包

```
for both in T:  
    a, b = both                  # Manual assignment equivalent  
    print(a, b)
```

```
1 2  
3 4  
5 6
```

for 循环

- 在遍历字典中键值对

```
D = {'a': 1, 'b': 2, 'c': 3}
for key in D:
    # Use dict keys iterator and index
    print(key, '=>', D[key])
```

```
a => 1
b => 2
c => 3
```

```
for (key, value) in D.items():
    # Iterate over both keys and values
    print(key, '=>', value)
```

```
a => 1
b => 2
c => 3
```


嵌套 for 循环

```
items = ["aaa", 111, (4, 5), 2.01]           # A set of objects
tests = [(4, 5), 3.14]                       # Keys to search for

for key in tests:                             # For all keys
    for item in items:                       # For all items
        if item == key:                     # Check for match
            print(key, "was found")
            break
        else:
            print(key, "not found!")
```

```
(4, 5) was found
3.14 not found!
```

嵌套 for 循环

```
items = ["aaa", 111, (4, 5), 2.01]           # A set of objects
tests = [(4, 5), 3.14]                       # Keys to search for

for key in tests:                             # For all keys
    for item in items:                       # For all items
        if item == key:                     # Check for match
            print(key, "was found")
            break
    else:
        print(key, "not found!")
```

(4, 5) was found
3.14 not found!

```
for key in tests:                             # For all keys
    if key in items:                         # Let Python check for a match
        print(key, "was found")
    else:
        print(key, "not found!")
```

(4, 5) was found
3.14 not found!

关于循环else

- 不推荐使用

```
for i in range(3):  
    print('Loop {}'.format(i))  
    if i == 1:  
        break  
else:  
    print('Else block!')
```

Loop 0
Loop 1

```
while False:  
    print('Never runs')  
else:  
    print('While Else block!')
```

While Else block!

```
for x in []:  
    print('Never runs')  
else:  
    print('For Else block!')
```

For Else block!

在循环里用 break 语句提前跳出，会导致程序不执行 else 块。

初始循环条件为 False 的 while 循环，如果后面跟着 else 块，那它会立刻执行。

for 循环要遍历的序列是空的，那么也会立刻执行 else 块。

循环序列相关函数

- Python 提供了三个内置函数，可以在 for 循环内定制迭代。
 - 内置 range() 函数返回一系列连续增加的整数，可作为 for 中的索引
 - 内置 zip() 函数返回并行元素的元组的列表，可用于在 for 中遍历多个序列
 - 内置 enumerate() 函数同时返回迭代对象的索引和数值

rang()

- 通常用于产生序列索引

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

```
list(range(-5, 5))
```

```
[-5, -4, -3, -2, -1, 0, 1, 2, 3, 4]
```

```
list(range(5, -5, -1))
```

```
[5, 4, 3, 2, 1, 0, -1, -2, -3, -4]
```

range()

```
for i in range(3):  
    print(i, 'Pythons')
```

要打印3行时，使用range() 产生适当的整数数字

```
0 Pythons  
1 Pythons  
2 Pythons
```

```
x = 'spam'  
for i in range(len(x)):  
    print(x[i], end=' ')
```

用 range() 产生用于迭代索引的列表。

```
s p a m
```

rang()

```
for i in range(3):  
    print(i, 'Pythons')
```

要打印3行时，使用range() 产生适当的整数数字

```
0 Pythons  
1 Pythons  
2 Pythons
```

```
x = 'spam'  
for i in range(len(x)):  
    print(x[i], end=' ')
```

但是，它的运行速度会较慢。

```
s p a m
```

```
X = 'spam'  
for item in X:  
    print(item, end=' ') # Simple iteration
```

```
s p a m
```

zip() 函数

- 内置的 **zip()** 函数允许我们使用 for 循环来并行遍历多个序列。zip 会取得一个或多个序列作为参数，然后返回元组的列表，将这些序列中的并排元素配成对

```
L1 = [1, 2, 3, 4]  
L2 = [5, 6, 7, 8]  
list(zip(L1, L2))
```

```
[(1, 5), (2, 6), (3, 7), (4, 8)]
```


zip() 函数

- 内置的 **zip()** 函数允许我们使用 for 循环来并行遍历多个序列。zip 会取得一个或多个序列作为参数，然后返回元组的列表，将这些序列中的并排元素配成对

```
L1 = [1, 2, 3, 4]
L2 = [5, 6, 7, 8]
list(zip(L1, L2))
```

```
[(1, 5), (2, 6), (3, 7), (4, 8)]
```

- for 循环使用元组赋值运算以解包 zip 结果中的每个元组。
- 第一次迭代时，就类似执行了赋值语句
 - $(x, y) = (1, 5)$

```
S1 = 'abc'
S2 = 'xyz123'
list(zip(S1, S2))
```

```
[('a', 'x'), ('b', 'y'), ('c', 'z')]
```

当参数长度不同时，zip 会以最短序列的长度为准来截断所得到的元组

zip()函数

- 构造字典

```
keys = ['spam', 'eggs', 'toast']  
vals = [1, 3, 5]  
D3 = dict(zip(keys, vals))  
D3
```

```
{'spam': 1, 'eggs': 3, 'toast': 5}
```

enumerate() 函数

- 返回元素以及这个元素的偏移值。

```
S = 'spam'
offset = 0
for item in S:
    print(item, 'appears at offset', offset)
    offset += 1
```

```
s appears at offset 0
p appears at offset 1
a appears at offset 2
m appears at offset 3
```

enumerate() 函数

- 返回元素以及这个元素的偏移值。

```
S = 'spam'
offset = 0
for item in S:
    print(item, 'appears at offset', offset)
    offset += 1
```

```
s appears at offset 0
p appears at offset 1
a appears at offset 2
m appears at offset 3
```

```
S = 'spam'
for (offset, item) in enumerate(S):
    print(item, 'appears at offset', offset)
```

```
s appears at offset 0
p appears at offset 1
a appears at offset 2
m appears at offset 3
```

enumerate()函数

- 返回一个对象
- 这个对象有很多的方法
 - next(E)

```
E = enumerate(S)
```

enumerate(S) 返回一个生成器对象

```
type(E)
```

```
enumerate
```

```
next(E)
```

```
(0, 's')
```

```
next(E)
```

```
(1, 'p')
```

练习

- 实现从1到100求和
- 从一个包含多个字符串的列表中找出长度最长的字符串及其位置
 - `names = ['Cecilia', 'Lise', 'Marie', 'Jennifer']`
- 输出九九乘法口诀