Python 语言程序设计: 列表表达式等与函数

李宽
likuan@dgut.edu.cn

东莞理工学院

2019.11

目录

1. 序列相关
   - 一切皆对象
   - 变量与对象
   - 深浅拷贝
   - 可变数据与不可变数据
   - 列表推导式
   - 生成器表达式
   - 序列解包
2. 函数
   - 函数概述
   - 定义函数
   - 调用函数

目录

1. 序列相关
   - 一切皆对象
   - 变量与对象
   - 深浅拷贝
   - 可变数据与不可变数据
   - 列表推导式
   - 生成器表达式
   - 序列解包
2. 函数
   - 函数概述
   - 定义函数
   - 调用函数

问题：谈谈你现在对 Python 的理解

keywords:

- 变量
- 缩进
- 流程
- 数据存放结构

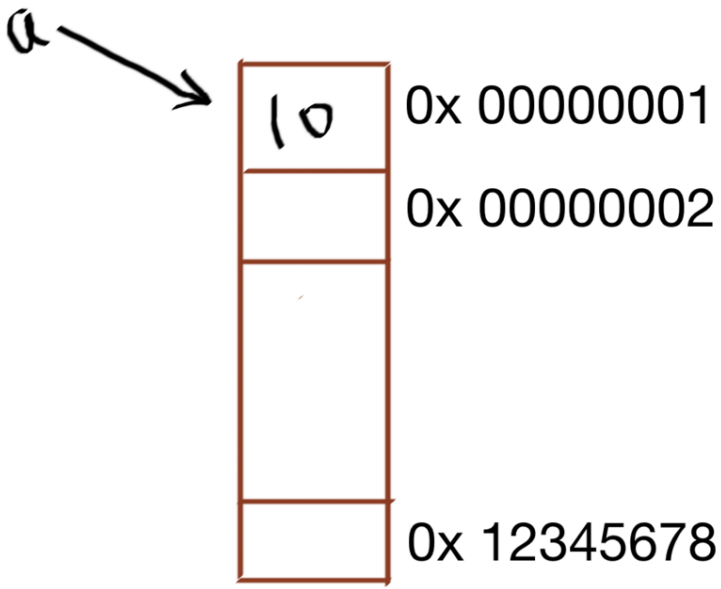
20% - 80% 原则

变量

近似意义上:

变量存在内存中，内存可类比一排排抽屉，每个抽屉上面有个编号. 定义一个变量，类似于把想存放的东西 (信息) 放到对应编号的抽屉里。

变量 a=10，如下图所示 (简略示意图，未考虑数据类型占用的长度) ：



程序的本质: 存储 + 运行，信息（变量承载）的流动

Python: 一切皆对象 1/2

python 中的对象都会有三个特征

- 身份，存储地址，可通过 id() 方法来查询
- 类型，即对象所属的类型，可用 type() 方法来查询
  进而决定有哪些属性和方法，可进行哪些操作，遵循怎样的规则
- 值，对应的数据

```
1 #id-type-value.py，对象的三个特征：身份，类型和值
2
3 n1 = 1# 整数类型
4 str1 = "hello"# 字符串类型
5 # 依次输出身份，类型和值
6 print ( "n1, id = {}, type = {}, value = {}"
7         .format(id(n1), type(n1), n1))
8 print ( "str1, id = {}, type = {}, value = {}"
9         .format(id(str1), type(str1), str1))
```

Python: 一切皆对象 2/2

“身份”、”类型”和”值”在对象创建时被赋值。

如果对象支持更新操作，则它的值是可变的;

否则为只读（数字、字符串、元组等均不可变）。

问题：数字是只读的？如何理解?

目录

1. 序列相关
   - 一切皆对象
   - 变量与对象
   - 深浅拷贝
   - 可变数据与不可变数据
   - 列表推导式
   - 生成器表达式
   - 序列解包
2. 函数
   - 函数概述
   - 定义函数
   - 调用函数

变量与对象 1/6

```
1    a = 3
```

上述代码执行了以下操作：

1. 创建一个对象来代表数字 3 (Python 缓存策略: 先检查 3 是否存在)
2. 如果变量 a 不存在，创建一个新的变量 a
   问题：变量存储在哪？标签
3. 将变量 a 和数字 3 进行连接，即 a 成为对象 3 的一个引用
   - 内部来看，变量是到对象的内存空间的一个指针
   - 变量总是连接到对象，而不会连接到其他变量

变量与对象 2/6

理解：id(变量)返回变量连接的对象的地址.
强调：变量是对象的引用，不会连接到其他变量

```
#var.py, 变量是到对象的引用

a = 3
print("a: id = {}, type = {}, value = {}".format(id(a),type(a),a) )
print("3: id = {}, type = {}, value = {}".format(id(3),type(3),3) )

b = a # 多个变量都引用了相同的对象，成为共享引用
print("b: id = {}, type = {}, value = {}".format(id(b),type(b),b) )

print("修改后")
# 对变量 a 修改，因为数字是不可修改类型
# 等价于重新" 建立" 对象 2，然后将 a 指向它
a = 2
print("a: id = {}, type = {}, value = {}".format(id(a),type(a),a) )
print("2: id = {}, type = {}, value = {}".format(id(2),type(2),2) )
print("b: id = {}, type = {}, value = {}".format(id(b),type(b),b) )
```

变量与对象 3/6

反例：

```
1 #list-ex.py, 列表类型共享引用
2
3 a = [1, 2, 3]
4 b = a
5 a[0] = 0 # 并未改变 a 的引用，只是改变了被引用对象的内部元素
6 print(a, "id(a) is", id(a))
7 #b 引用的对象发生了变化，因此 b 的值也发生了改变
8 print(b, "id(b) is", id(b))
```

变量与对象 4/6

如何解决: 拷贝, 指向新的对象

```
1 #list-copy, 列表类型共享引用
2
3 a = [1, 2, 3]
4 b = a[:] # 单独创建了新的对象, 并拷贝列表中的元素到新对象中
5 print("id(a) is", id(a), "id(b) is", id(b))
6
7 a[0] = 0 # 修改 a 引用对象的内部元素
8 print(a, "id(a) is", id(a))
9 #b 引用的对象并未发生变化, 故 b 的值不变
10 print(b, "id(b) is", id(b))
```

两点说明：

- 对列表类型：同样的值在内存中可能有多份不同地址
- 对数字和字符串只有 1 份,hashable/unhashable, 可变/不可变

对于字典和集合等没有分片概念的类型来说: copy() 方法

变量与对象 5/6

```
#dict-set-ex
a = {1, 2, 3}
b = a.copy() # 集合的复制，浅复制
print("id(a) is", id(a), ",id(b) is", id(b))

c = {1:"a", 2:"b", 3:"c"}
d = c.copy() # 字典的复制，浅复制
print("id(c) is", id(c), ",id(d) is", id(d))
```

变量与对象 6/6

对象相等

- == 操作符用于测试两个被引用的对象的值是否相等
- is 用于比较两个被引用的对象是否是同一个对象

```
1  #is.py, 相等和 is 的区别
2  a = [1, 2, 3]
3  b = a#b 和 a 是一个实体
4  print ( "id(a) is ", id(a), ",id(b) is ",
5          id(b), ", a is b:", a is b)
6
7  c= [1, 2, 3]#c 和 a 不同
8  print ( "id(a) is ", id(a), ",id(c) is ",
9          id(c), ", a is c:", a is c)
10
11 d= 7# 不可变类型, 缓存机制, 同样实体
12 e= 7
13 print ( "id(d) is ", id(d), ",id(e) is ",
14         id(e), ", d is e:", d is e)
```

目录

1. 序列相关
   - 一切皆对象
   - 变量与对象
   - 深浅拷贝
   - 可变数据与不可变数据
   - 列表推导式
   - 生成器表达式
   - 序列解包
2. 函数
   - 函数概述
   - 定义函数
   - 调用函数

深浅拷贝 1/3

浅拷贝和深拷贝的不同仅仅是对组合对象来说:

- 组合对象: 包含其它对象的对象，如列表，字典，类实例等
- 对数字、字符串以及其它“原子”类型，都是原对象的引用。

“浅拷贝”：创建一个新的对象，其内容是原对象中元素的引用 (拷贝组合对象，不拷贝子对象)
切片操作和 copy() 方法均属于浅拷贝:

```
1 #list-shallow-copy.py, 浅复制
2 a = [1, [4, 5, 6]]
3 b = a[:] # 单独创建了新的对象，并拷贝列表中的元素到新对象中
4
5 # 浅拷贝，虽然 a 和 b 指向不同对象，
6 # 但其中的元素（尤其是二级列表）指向相同的对象
7 print("id(a) is", id(a), "id(b) is", id(b))
8 print("id(a[0]) is", id(a[0]), "id(b[0]) is", id(b[0]))
9 print("id(a[1]) is", id(a[1]), "id(b[1]) is", id(b[1]))
```

深浅拷贝 2/3

深拷贝: copy 模块的 deepcopy 函数

```
1  #list-deep-copy.py 深拷贝
2  import copy
3
4  a = [1, [4, 5, 6]]
5  b = copy.deepcopy(a)
6
7  # 深拷贝, a 和 b 指向不同对象,
8  # 其中的元素 (尤其是二级列表) 也指向相同的对象
9  print("id(a) is", id(a), "id(b) is", id(b))
10 print("id(a[0]) is", id(a[0]), "id(b[0]) is", id(b[0]))
11 print("id(a[1]) is", id(a[1]), "id(b[1]) is", id(b[1]))
```

重申：为什么使用了深拷贝，a[0] 和 b[0] 中元素的 id 还是一样呢？
对于不可变对象，当需要一个新的对象时，python 可能会返回已经存在的某个类型和值都一致的对象的引用 (缓存)。
这种机制并不会影响 a 和 b 的相互独立性，当两个元素指向同一个不可变对象时，对其中一个赋值不会影响另外一个。

深浅拷贝 3/3

总结：

- 赋值：简单地拷贝对象的引用，两个对象的 id 相同。
- 浅拷贝：创建一个新的组合对象，这个新对象与原对象共享内存中的子对象。
- 深拷贝：创建一个新的组合对象，同时递归地拷贝所有子对象，新的组合对象与原对象没有任何关联。
  虽然实际上会共享不可变的子对象，但不影响它们的相互独立性。

目录

1. 序列相关
   - 一切皆对象
   - 变量与对象
   - 深浅拷贝
   - 可变数据与不可变数据
   - 列表推导式
   - 生成器表达式
   - 序列解包
2. 函数
   - 函数概述
   - 定义函数
   - 调用函数

回忆：现在已经学了哪些数据类型，简要解释

可变数据与不可变数据 2/3

复习：现在已经学习了哪些数据类型?

- 数字 1234, 3.14, 1.2e9, 3+4j
- 字符串'hello', "world"
- 列表 [1,2,3,4]
- 元组 (1, "hello", 2)
- 字典 {1 : "hello", 2 : "world"}
- 集合 {1,2,3,4}
- ...

可变数据与不可变数据 3/3

- 不可变数据类型: 运行过程不可以改变值的数据类型
  - tuple(元组)
  - num(数字, 包括整型与浮点型),str() 字符串
- 可变数据类型: (改变不需要新创建对象,原地址)
  在运行过程中可以更改其值的数据类型, 常见的有列表, 字典和集合

不可变数据类型与哈希 1/3

不可变数据类型：变量引用的地址处的值是不可变的，例如 int 类型，改变该类型变量的值，实际是改变该变量引用的地址值，即改变了该变量引用的对象

- 优点，不管内存中有多少个引用，相同的对象只占用一块内存；
- 缺点：当对变量进行运算从而改变变量的值时，需要创建新对象，不断的改变会不断的创建新对象[1]。

可变数据类型：内存中有可能存在多个同样值的对象，彼此地址值不同

- 当值发生改变时，并不会创建新的对象，只是改变了原地址的值
- 注意： 可变数据类型的赋值操作会改变该变量的地址值

[1]不再使用的变量会被 Python 垃圾回收机制回收

不可变数据类型与哈希 2/3

问题: 什么是哈希 (Hash)
场景: 海量电话号码 (姓名: 号码对), 如何高效存储?

哈希表, Hash Table

根据关键字（Key value）而直接访问在内存存储位置的数据结构。

通过把键值 (姓名) 通过一个函数的计算，映射到表中一个位置来访问记录，以加快查找速度。
映射函数称做散列函数，存放记录的数组称做散列表。

不可变数据类型与哈希 3/3

Python 中的字典和集合都是根据 Hash 函数来组织的:
集合中的元素或字典中的键, 根据其 id() 进行 hash
可哈希要求列集合中的元素或字典中的键必须是不可变类型
why? (值与 id() 要一一对应才能 hash!)
可变数据类型在改变值的同时却没有改变 id, 无法由地址定位值的唯一性, 因而无法哈希。

- 不可变数据类型：hashable, 如字符串
- 可变数据类型: unhashable, 如列表、字典、集合

目录

1. 序列相关
   - 一切皆对象
   - 变量与对象
   - 深浅拷贝
   - 可变数据与不可变数据
   - 列表推导式
   - 生成器表达式
   - 序列解包
2. 函数
   - 函数概述
   - 定义函数
   - 调用函数

列表推导式 1/2

列表推导式（又称列表解析式）
结构:

- 中括号里包含一个表达式
- 表达式：至少一个 for 语句，后面可跟 if 条件语句或嵌套 for 语句
- 返回结果: 一个新的列表

最简单的形式: [exprssion for item in iterable]

```
1 x_list = [x for x in range(1,10)]
```

加上条件表达式的形式: [exprsssion for item in iterable if condition]

```
1 y_list = [x for x in range(1,7) if x %2 ==1]
```

列表推导式 2/2

多个 for 的嵌套表达式 (生成一个 x,y 的元组的列表)：

```
z = [(x,y) for x in x_list for y in y_list]
```

```
#list-expression.py, 列表推导式

multiples = [i for i in range(30) if i % 3 is 0]
print(multiples)

squared1 = []
for x in range(10):
    squared1.append(x**2) # 依次添加到列表中
print(squared1)

squared2 = [x**2 for x in range(10)]# 和上面依次添加等价
print(squared2)
```

字典/集合推导式

最简单的形式：

{key_expression:value_expression for expression in iterable}

```
1 word = 'python'
2 let_dict = {let:word.count(let) for let in word}
```

集合推导式 {expression for expression in iterable}
元组没有推导式，圆括号用来坐生成器表达式

目录

1. 序列相关
   - 一切皆对象
   - 变量与对象
   - 深浅拷贝
   - 可变数据与不可变数据
   - 列表推导式
   - 生成器表达式
   - 序列解包
2. 函数
   - 函数概述
   - 定义函数
   - 调用函数

生成器表达式 1/2

```
my_generator = ( f(x) for x in sequence if cond(x) )
```

生成器可以转化成列表推导式

```
t_list = list(t_generotor)
```

生成器只能运行一次:

```
#generator-once.py 生成器只能执行 1 次

range_ = (x ** 3 for x in range(5))
ls1 = list(range_)
print(ls1)
ls2 = list(range_) # 再次执行时, ls2 为空
print(ls2)
```

生成器表达式 2/2

惰性求值

```
1  #generator.py 生成器惰性求值
2
3  range_ = (x ** 3 for x in range(5))
4
5  print(range_)# 惰性求值
6  print(range_.__next__())# 惰性求值，调用 __next__ 函数才触发
7  print(range_.__next__())
8  print(range_.__next__())
9
10 print("for 能触发 __next__")
11 #for 循环可触发 __next__
12 for n in (x ** 3 for x in range(5)):
13     print(n)
```

目录

1. 序列相关
   - 一切皆对象
   - 变量与对象
   - 深浅拷贝
   - 可变数据与不可变数据
   - 列表推导式
   - 生成器表达式
   - 序列解包
2. 函数
   - 函数概述
   - 定义函数
   - 调用函数

序列解包 1/3

解包：

list/tuple/set/dict 是整体，把其中每个元素当成很多个体剥离出来的过程

1. 对等赋值:

```
1 #depack.py 序列解包
2 a, b, c = (1, 2, 3) # 参数个数与元组长度一致
3 print(a,c)
4 [a, b, c] = [1, 2, 3] # 参数个数与列表长度一致
5 print(a,c)
6 a, b, c = 'SUN' # 刚好三个字符的字符串
7 print(a,c)
8 a, b, c = 'PYTHON' # 字符串长度较长，会报错
9 print(a,c)
```

序列解包 2/3

2. 非对等赋值

```
1  #depack-star.py 序列解包
2  a, *b, c = 'PYTHON' # 字符串长度较长，注意 * 号的效果
3  print(a,c)
4  print(b)
5
6  s = 'ABCDEFGH'
7  while s:
8      # 等价于 front,s = s[0], list(s[1:])
9      front, *s = s # 解包，除第 1 个元素外，其余都赋值给 s
10     print(front, s)
```

序列解包 3/3

字典的解包:

```
#depack-dict.py 字典解包
dict1 = {'a':1,'b':2,'c':3,'d':4}
a,*b,c = dict1 # 默认情况下，解包的是字典键值
print(a,b,c)
a,*b,c = dict1.values()#values() 函数，解包的是值
print(a,b,c)
a,*b,c = dict1.items()# 键值对
print(a,b,c)
```

目录

1. 序列相关
   - 一切皆对象
   - 变量与对象
   - 深浅拷贝
   - 可变数据与不可变数据
   - 列表推导式
   - 生成器表达式
   - 序列解包
2. 函数
   - 函数概述
   - 定义函数
   - 调用函数

为什么需要函数：代码复用 + 抽象

从两个例子谈起：画圆，计算圆的面积

计算圆的面积

```
1  #client-area.py
2
3  import math
4  r1 = 12.34 # 指定不同的半径
5  r2 = 9.08
6  r3 = 73.1
7
8  s1 = math.pi * r1 * r1 # 面积公式 s=pi * r~2
9  s2 = math.pi * r2 * r2 # 相同的写法要调用多次
10 s3 = math.pi * r3 * r3
11
12 print("s1 is {:.2f}, s2 is {:.2f}, s3 is {:.2f}".format(s1,s2,
```

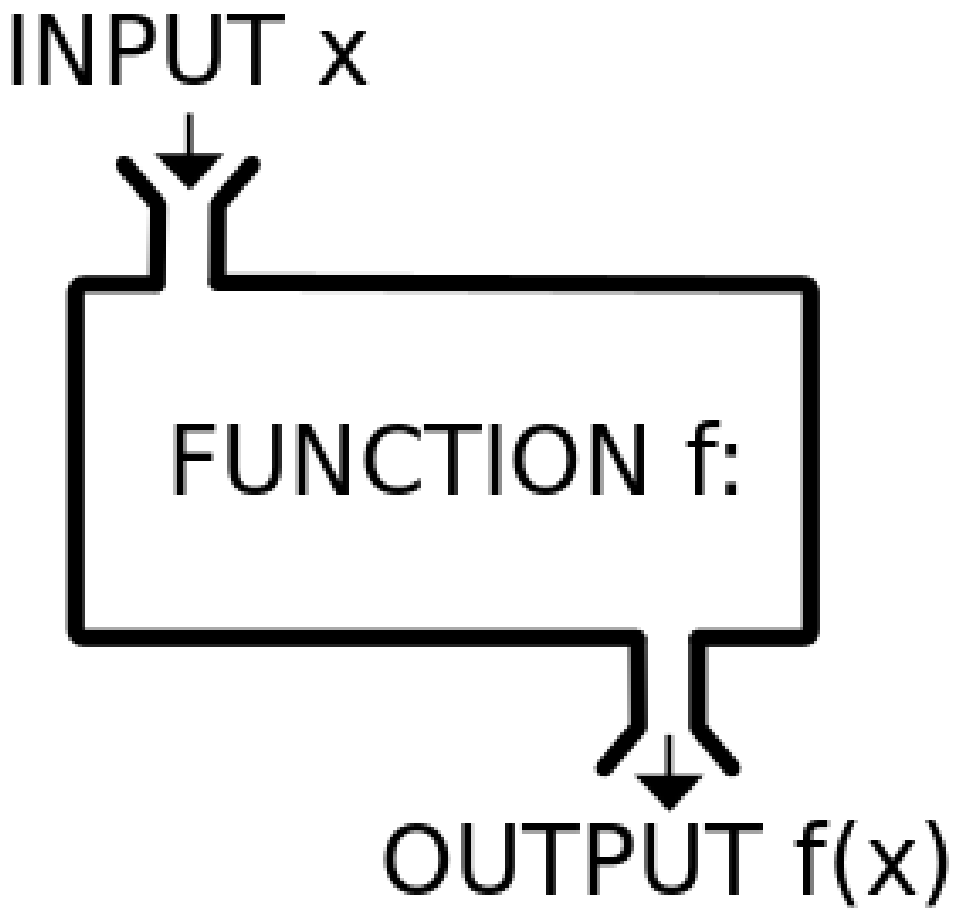
问题, 如果要修改计算方式?! How?

目录

1. 序列相关
   - 一切皆对象
   - 变量与对象
   - 深浅拷贝
   - 可变数据与不可变数据
   - 列表推导式
   - 生成器表达式
   - 序列解包
2. 函数
   - 函数概述
   - 定义函数
   - 调用函数

定义函数 1/5

函数的数学定义: 输入到输出的映射



定义函数 2/5

定义函数:

```
        def 函数名(参数):
            函数体/函数块
            return 值
```

```
#func.py    将求圆面积公式抽象为函数
import math

# 定义函数 circle_area
def circle_area(r):
    s = math.pi * r * r# 面积公式 s=pi * r^2
    return s

r1 = 12.34
r2 = 9.08
r3 = 73.1

s1 = circle_area(r1) # 调用 circle_area 函数即可, 参数是半径
s2 = circle_area(r2)
s3 = circle_area(r3)

print("s1 is {:.2f}, s2 is {:.2f}, s3 is {:.2f}".format(s1,s2,s3))
```

定义函数 3/5

内置函数与自定义函数的比较：

```
1  #func-abs.py
2  # 对比内置函数 abs 和自定义函数 my_abs
3  def my_abs(x):
4      if x >= 0:
5          return x
6      else:
7          return -x
8
9  print("my_abs(-2) is ", my_abs(-2))
10 print("abs(-2) is ", abs(-2))
```

定义函数 4/5

- 参数列表可以为空
- 遇到 return 语句就返回（可以有多个 return），表示执行完毕
- 如果没有 return 语句，函数执行完毕后也会返回结果，只是结果为 None, return None 可简写为 return
- return 可以返回多个值

定义函数 5/5

空函数:

```
1 def nop_func():
2     pass
```

pass 用作占位符，框架

目录

1. 序列相关
   - 一切皆对象
   - 变量与对象
   - 深浅拷贝
   - 可变数据与不可变数据
   - 列表推导式
   - 生成器表达式
   - 序列解包
2. 函数
   - 函数概述
   - 定义函数
   - 调用函数

调用函数 1/3

参数检查: 调用用函数时，如果参数个数不对，Python 解释器会报错并抛出异常：

```
#func-para-err.py, 演示参数个数有误的情况
# 对比 abs 和自定义 my_abs

def my_abs(x):
    if x >= 0:
        return x
    else:
        return -x

#Python 解释器会报错
print("my_abs(-2) is ", my_abs(-2,-3))
```

解释器会 " 帮忙 " 检查参数个数

问题：如果参数类型出错，如何处理？

调用函数 2/3

isinstance 自行检查类型错误：

```
#func-para-type.py 演示参数类型错误的情况
# 对比 abs 和自定义 my_abs

def my_abs(x):
    # 参数类型错误必须自己判断处理
    if not isinstance(x, (int, float)):
        # 抛出异常
        raise TypeError('自定义:my_abs 要求输入 int 或 float 类型')
    if x >= 0:
        return x
    else:
        return -x

print("my_abs(\"A\") is ", my_abs("A"))
```

错误和异常处理将在后续讲到。

调用函数 3/3

返回多个值:

```
1  #func-move.py 演示函数可返回多个值
2  # 导入 math 包，并允许后续代码引用 math 包里的 sin,cos 等函数
3  import math
4
5  def move(x, y, step, angle):
6      nx = x + step * math.cos(angle)
7      ny = y - step * math.sin(angle)
8      return nx, ny # 实际返回的是1个元组 (nx,ny)
9
10 a,b = move(100, 100, 60, math.pi / 6) # 隐含着元组的解包
11 print(a, b)
```

Python 的函数返回多值实际是返回一个 tuple，但书写更方便

小结

- 定义函数时，需要确定函数名和参数个数
- 如果有必要，可以先对参数的数据类型做检查
- 函数体内部可以用 return 随时返回函数结果
- 函数执行完毕也没有 return 语句时，自动 return None
- 函数可以同时返回多个值，但其实是一个 tuple