

Python - P1

I/O - Python 中输出使用函数 `print()`, 用于将传入的参数以字符串的形式输出至终端
`print()` 可以传入任意数量的参数, 参数间以逗号分隔, 转输出时, '显示为白符' (output)
传入的参数不必要是字符串类型, 只要可以以字符串形式显示。(类似 Haskell 中的 Show)

Python 中输入使用函数 `input()`, 交互式命令行中执行后会等待输入。

将回车(换行符之前的内容)保存为一个字符串(注意与 C++ 不同, 不为空白行为中断)

输入的字符串中如果存在需要转义的字符, 会自动在字符串末尾加反斜杠 \ (backslash)

python 中字符串默认以单引号 '' (single quote) 表示, 所以输入中的双引号 "" (double quote) 不会转义

comment <以 '#' 开头, '#' 后至行结束的部分将被视为 comment, interpreter 将忽略 comment 部分>

缩进 <若上一语句以冒号 ':' 结尾, 则从下一语句开始具有相同缩进的语句将被视为同一代代码块 (block)>
注意: 复制代码可能导致缩进错误, 所以尽管使同一 block 代码简短.

大小写 Python 是大小写敏感的, 所以应特别注意大小写可能导致错误

整型值 (Integer), Python 中可以处理任意大小的整型值, 其类型类似 Haskell 中的 Integer

另外 Python 中可以在立即数前加 0x, 以表示十六进制整型值

浮点数 (Float), Python 中浮点数可用科学计数法表示, 形如 significance e ± magnitude, 如 6.02×10^{23}

在交互式命令行中, 浮点数的显示精度为 16 位有效数字, 且自动截去尾数, 立即数直接显示

当使用科学计数法时, 保留 16 位小数, 不论立即数还是变量都采用四舍五入

另外注意 Python 中整数除法与浮点数除法的存储方式不同, 可能导致计算结果不同

整数除法是精确的, 即 $d = 1 / 99 \dots 9 \rightarrow 1e-26$, $d * 99 \dots 9 \rightarrow 1.0$, 其结果是精确的

浮点数除法会进行四舍五入, 即 $d = 1.0 / 99 \dots 9 \rightarrow 9.99 \dots 9e-27$, $d * 99 \dots 9 \rightarrow 0.99 \dots 9$, 出现误差

可能在 Python 解释器中, 整数除法采用的是以 fraction 形式存储表达式, 采用惰性求值,

字符串 str (String) Python 中, 以单引号 '' 或双引号 "" 表示字符串, 但参考 `input()` 函数, 认为单引号是默认的

同样有转义字符 \, backlash 会转义其之后的一个字符, 如 \n 换行, \t tab, \' 单引号

在交互式命令中, 采用 """ 的格式输入多行文本, 即在 3 个单引号引导的字符串中, 换行出现...

在 `print()` 函数中, 在字符串前加 r, 表示忽略字符串中的转义字符.

如 `print('\\\"') \rightarrow \\\", 而 print(r'\\\"') \rightarrow \\\"`

布尔值 (Bool), Python 中布尔值表示 True 和 False, 支持 and, or, not 运算, 具有非 False 即 True 规则

Python - P2

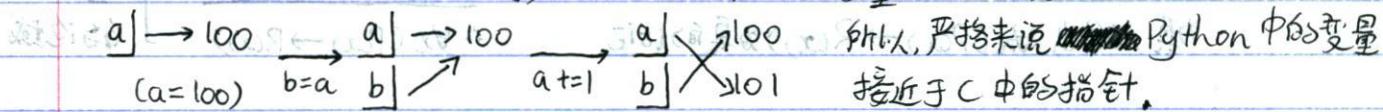
空值

None, Python 中存在一个特殊的空值, 与 0 不同, 在进行布尔计算时视为 False

变量

在 Python 中, 变量原则上可以是任何数据类型, 用 `name = value / expression` 的形式赋值
变量名必须是大小写英文, 数字, 下划线的组合, 且不允许以数字开头, 注意 Python 对大小写敏感
等号是赋值语句, 可以将任何数据类型赋值给变量, 同一变量可以反复赋值
注意: 对同一变量的赋值可以是不同类型的, 编译中不会报出错误, 但可能导致程序错误

动态语言: 变量本身类型不固定的语言, 如 Python, 静态语言变量定义必须指定变量类型, 赋值类型不匹配即报错
变量赋值时, 解释器首先在内存中创建一个 `value`, 然后在内存中创建一个变量 `name` 并指向 `value`
在将变量 `a` 赋值给变量 `b` 时 (`b = a`), 实际上是 ~~使~~ 变量 `b` 指向 变量 `a` 指向的 `value`
之后对 `a` 进行的赋值不会影响 `b` 的值, 因为 变量 `a` 已指向其他 `value`



常量

Python 中, 常量通常用全部大写的变量名表示, 但也是一个习惯性用法, 如 `PI = 3.14`

注意: Python 中没有任何机制保证常量不被修改 (实际上是没有常量)
Python 把任何数据视为一个“对象”(object), 变量是程序用以指向对象的, 变量赋值即关联变量和对象

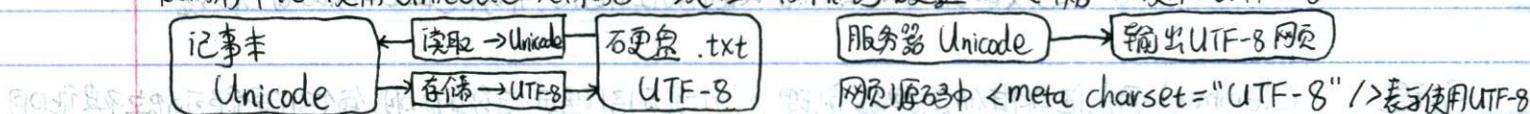
Python 中超过一定范围的浮点值会被记为 `inf`, 如 `single precision` 中 ~~0x1.ffffp+0~~ 和 ~~0x1.ffffp+1~~
~~0x1.ffffp+0~~ 表示 ~~1.111111111111111e+0~~, ~~0000 0000 0000 0000 0000 0000 0000 0000~~, $= 0x7F80 / 0xFF80$
~~sign exponent~~ 表示 ~~Significant~~ 即 `exponent` 达到上限的值

字符串

美国 \rightarrow ASCII, 中国 \rightarrow GB2312, 日本 \rightarrow Shift-JIS, 韩国 \rightarrow Euc-kr, 统一 \rightarrow Unicode

Unicode \rightarrow 可变长编码 UTF-8, ~~编码为 1-6 字节, 1 开头为 ASCII, 汉字通常为 3 字节,~~

在内存中统一使用 Unicode 以保持一致性, 存储到硬盘或传输时使用 UTF-8



ord()

获取字符的整数表示, 如 `ord('i')` \rightarrow 105, 注意 `ord()` 接收的实际上是长度为 1 的字符串

chr()

将整数编码转为字符, 如 `chr(65)` \rightarrow 'A', 注意 `chr()` 输出的实际上是长度为 1 的字符串

字符串中的 \u 会将紧跟其后的 4 个字符当作十六进制数使用, 转为对应字符, 要求有 '\uXXXX' 的格式

bytes

在网络上传输或 ~~存储~~ 到硬盘, Python 会使用以字节为单位的 type bytes, 以带 `b` 前缀的字符串表示与 str 的区别在于 bytes 中 ASCII 字符直连表示, 而其他字符以字节表示, 即形成 '`\x##`' 的格式

Python - P3

section

9 - ~~字符串~~

$\times e6 \times 96 \times 87'$

encode() encode() 方法可以将 str 转为 bytes, 如 '中文'.encode('utf-8') \rightarrow b'\xe4\xb8\xad'

decode() decode() 方法用于将 bytes 转为 str, 如 b'\xe4\xb8\xad'.decode('utf-8') \rightarrow '中'

另外可以加入参数 errors='ignore' 来忽略 decode() 中的无效文字节

len(bytes) 注意 bytes 的长度是以字节数计算的, 即形如 '\x##' 视为 1 个字节

格式化 Python 中的字符串可以用形如 '%...%?...%' % arg 的形式进行格式化, 从而快速生成格式化 string
如果有多个占位符, 则可用 '%...%?...%' % (arg1, arg2, ...)

常用的占位符有 %d 整数, %f 浮点数, %s 字符串(通用), %x 十六进制整数(用于 bytes 类型)

另外对于 %d 可以用 %kd 表示长度至少为 k 个字符, 不足的以空格填充, %0kd 则是以 '0' 填充

而对于 %s, 可以用 %knf 表示长度至少为 k 个字符(包含小数点), 且小数部分为 n 位(不足以 '0' 填充), 整数

注意格式化字符串中如有 '%' 需写作 '%%' 以显示单个 '%', 在没有格式化字符串中直接用 '%'

format

字符串中的 format() 方法, 用于将一连串的参数依序传入字符串中的占位符, 此时占位符以 {} 表示
即有形如 '...{}...{}'.format(arg0, ..., argn); 另外占位符可用 {k: M.nf} 的形式定义特别格式

format() 方法适用于需要传入多个占位符而可能存在类型不定的情况, 或是顺序比较重要(序号)

list

Python 中的 list 的语法与表示与 Haskell 大致一样, 即有 $L = [a_1, a_2, \dots, a_n]$ 的形式

但是 Python 的 list 的最大区别是可以在一个 list 中加入不同类型的值, 即 $[1, 1.0, 'a']$ 是合法列表

可能是因为 Python 列表的实质是变量的 list, 即 list 中的元素分别指向不同内存数据层的变量

需要特别注意在用类似 map 和 fold 等高级函数操作 list 时, 注意存储的值是否类型有效

另一个区别是 Python 支持以负数索引查找元素, 即有 $[0, \dots, len() - 1]$, i-len() 指向同一元素

list.append() 添加元素至列表末尾, 如 $[1, 2].append('a') \rightarrow [1, 2, 'a']$, 与 C++ 的 pushback() 类似, 注意不要类型检查

list.insert() 插入元素至指定位置, 如 $[1, 2].insert(1, 'a') \rightarrow [1, 'a', 2]$, 与 C++ 的 insert() 类似, 但同样不做类型检查

list.pop() 删除元素并返回删除的元素, 如 $[1, 'a', 2].pop() \rightarrow 'a'$, 不插入索引时默认 pop 最后一个元素

len() 用于返回列表的元素个数, 如 $len([1, 2]) \rightarrow 2$, 同样可作用于元组 tuple

tuple

tuple 在初始化后就不能修改, 在 Python 中 tuple 的语法与 Haskell 大致一样, 也可以通过下标调用元素

元组 tuple 在定义时就必须确定 tuple 包含的元素, 与 list 一样, 也是指向不同内存数据的变量

与 Haskell 不同的是 Python 支持单元素 tuple, 即可以定义 tuple 如 (1,) 表示只包含一个元素的 tuple

注意 tuple 的不变指的是作为变量的元素指向的内存数据指向不变, 但数据本身可以改变

$t \rightarrow [1, 2]$ 允许执行 $t[1][1] = 'b'$ $t \rightarrow [1, 'b']$

$t \rightarrow \text{tuple } 0 \rightarrow 'a'$ 因为并没有改变 t[1] 指向

而是改变指向的数据

Python - P4

if

在 Python 中，if 语句结构与 Haskell 相似：`if <condition> : <body>` if `<cond1>` then
需要注意有两点：

1. Python 中对 else if 有特殊关键字 elif

`elif <condition2> : <body2>` else if `<cond2>` then

2. 注意格式，判断后的冒号和 block 的缩进

`<body2>` true else : `<body3>` else `<body3>`

另外非零数值(包含负数), 非空串, 非空list 等都视为 True

for

在 Python 中，for 循环的形式为：`for <ids> in <list/tuple>` 像 C++ 中的 `for (type x : vector)` 和 VBA 中的 `for each rng in range`

而 Python 中没有显式的类似 C++ 中的 `for (int i=0; i<n; ++i) {}` 的循环。

而是隐式地通过定义循环用的 list/tuple 来实现。如 `for x in range(n)[0::] / range(0, n, 1)`

range()

用于生成一个整数序列，完整形式为 `range(start, end, step)`，生成范围为 [start, end)，相邻元素间隔有两种形式可用：`range(end)`，则生成 [0, end) 间隔为 1 的序列，等价于 `range(0, end, 1)`

`range(start, end, step)`，则生成 [start, end) 间隔为 step 的序列。

注意，如果无法到达 end，即 `(end-start)/step < 0`，则返回 list 为 []

另外传入参数必须是整型值 integer 且 step ≠ 0。

while

Python 中 while 循环有结构如 `while <condition> :` 像 C++ 中的 `while (<expr>) {`

但是注意 Python 没有 do-while 循环。

break

Python 中 break 语句与 C++ 中的 break; 语句类似，用于从当前循环结构中退出，注意多层嵌套时只退出最内层。

continue

Python 中 continue 语句用于提前结束当前 loop 而直接进行下一个 loop，注意多层嵌套时只针对当前循环。

另外注意在 while 循环中如果使用 continue，必须注意对 while 条件判断的影响，可能出现死循环。

如 `n=1, while (n<10):`

注意此处 continue 执行后 n 在这次循环中并未改变，即进

`if (n%2==0): continue` 轮上并进入下一个 loop，从而陷入死循环。

陷入死循环时，在交互式命令中 `Ctrl+C` 以终止运行

注意：C++ 中也有 continue; 语句，其使用方法与遍历与 Python 中一致，注意只针对当前循环体有效。

dict

全称 dictionary，在其他语言（如 Haskell）中也称 map，使用键值对存储数据 (key-value)

字典

dict 的搜索速度快于 list，因为 dict 可以直接计算 key 值得到存放 value 的内存地址 (hash 算法)

可以

通过 `dict[key] = value` 语句向 dict 中添加新的 key-value 对，或修改已有 key 的 value

可以

通过 `dict[key]` 语句查询对应 key 的 value，如果 key 不存在，throw Key Error

Python - P5

in 可以用 `key in dict` 语句查询 key 值是否存在在 dict 中，返回 Bool 值
get() 通过 `dict.get(key)` 方法，如果 key 存在返回 value，否则返回 None，可以通过 `dict.get(key, def)` 来指定

pop() 通过 `dict.pop(key)` 方法，删除指定的 key-value 对，函数返回 value，找不到 key 抛 Key Error

与 list 比较 dict 内部存放顺序与 key 放入顺序无关，所以如果需要遍历，最好假设以随机顺序遍历
dict 查找与 list 速度不随 key 值增加而变慢，list 随元素数增加而变慢
dict 需要占用大量内存，造成浪费，list 占用空间少

key 不可度对键 dict 的 key 值必须是不可变对象（如整数、浮点数、字符串），但不要求同一字典 key 为同一类型 (unhashable)

如果 key 值可变，则 hash 算法可能得到不同结果，从而指向错误地址，如用 [] 时返回 TypeError

注意：这里的不可变对象是指当对象 value 发生改变时，变量是否被重新定向

生成 dict

可以通过 `dict()` 方法包裹 dict，调用形式为 `dict([(key, value)])`

如 `dict(map(lambda x,y:[x,y], 'abc', range(3)))` → {'a': 0, 'b': 1, 'c': 2}

也可以通过字典推导式，调用形式为 `{key: value for key, value in [(key, value)]}`

如 `{x:y for x,y in map(lambda x,y:[x,y], 'abc', range(3))}` → {'a': 0, 'b': 1, 'c': 2}

特别注意在组装前先组装成形，如 `[(key, value)]` 的结构，否则可能产生错误结果

如 `{x:y for x in 'abc' for y in range(3)}` → {'a': 2, 'b': 2, 'c': 2}

实际上运行了 `dict['a']=0, dict['b']=0, ..., dict['c']=0, dict['f']=1, dict['c']=2`

替代 switch 可以用 `dict.get()` 方法代替 switch 语句，尤其是在 switch 中需考虑 case 过多的情况

if arg == 0:
 dict = {0:"zero", 1:"One", 2:"Two", 3:"Three", 4:"Four", 5:"Five", 6:"Six", 7:"Seven", 8:"Eight", 9:"Nine"}
 switch(n):

return "zero"
 Case 0:
 Case 1:
 Case 2:
 Case 3:
 Case 4:
 Case 5:
 Case 6:
 Case 7:
 Case 8:
 Case 9:

elif arg == 1:
 return "One"; break;

return "One"
Case 1:
Case 2:
Case 3:
Case 4:
Case 5:
Case 6:
Case 7:
Case 8:
Case 9:

else:
 dict.get(key, "nothing")
 return "one"; break;

return 'nothing'
Case 0:
Case 1:
Case 2:
Case 3:
Case 4:
Case 5:
Case 6:
Case 7:
Case 8:
Case 9:

特别注意：Python 中没有 switch-case 语句，可以通过 if-elif-else 和 dict 实现

items() `dict.items()` 方法用于返回一个迭代器对象，可以用于如 `for k, v in dict.items():` 的形式使用

由于迭代器对象具有惰性加载的特性，只有需要时才生成值，所以生成过程中不需要另外内存装载体数据

Python - P6

在 Python 中 set 与 dict 类似，但是只存 key 值，由于 key 不重复，所以 set 中元素不重复
与 Haskell 中的 Set 类似，可以 $\text{Set}(\text{list})$ 生成，如 $\text{Set}([1, 1, 2, 3, 5, 8, 1, 3]) \rightarrow \{1, 2, 3, 5, 8\}$
注意 set 虽然显示时自动排序，但不能通过 index 的方式调用
另外由于 set 会删除重复元素，所以列表装入 set 后会丢失顺序信息

`add()` set.add(key) 用于添加元素，如 $\{1, 2\}.add(3) \rightarrow \{1, 2, 3\}$ ，添加已有元素不会产生任何效果
另外 set 可以装入不同类型的 value，如 $\{1, 2, 'a'\}$ ，但不可装入可变对象，如 list
`remove()` set.remove(key) 用于从 set 中删除元素，当尝试删除的元素不在 set 中时，throw KeyError
`update()` set.update(list) 用于向 set 中装入多个元素，如 $\{\}.update((2, 3, 4)) \rightarrow \{1, 2, 3, 4\}$
其逻辑与 $\text{for i in list/tuple: set.add(i)}$ 是等价的

集合运算 Python 中支持集合运算符，如 $s \cup t$ (并集)， $s \cap t$ (交集)， $s - t$ (差集)， $s \hat{\wedge} t = (s \cup t) - (s \cap t)$ (对称差集)

集合比较 Python 中支持集合间比较，如 $s.issubset(t)$ 用于 $(s \subseteq t?)$ ， $s.issuperset(t)$ 用于 $(s \supseteq t?)$

str 不可变 特别注意：尽管 str 拥有与 list 相似的属性与方法，但 str 是不可变对象，而 list 是可变对象
即参考 Haskell 的观点，可变对象的内部是可变的，或者说并未定义 (Eq) 的支持方法
参考 C++ 的逻辑，区别在于修改时是否生成了一个新的变量，可变对象返回其本身
即不可变对象遵循类似 `Type f(c const Type&val) { Type temp; ... return temp; }`
而可变对象遵循类似 `Type& f(Type &val) { ... return val; }`

由于 Python 中变量都是 pointer，所以对可变对象的修改会影响所有指向它的变量

如 $\ggg a = 'abc'$ $a \rightarrow 'abc'$ $b = a.replace('a', 'A')$ $b \rightarrow 'Abc'$ (生成)

传入参数 在 Python 中，传入数量不对或类型不匹配的参数，会报出 TypeError。考虑 `def my_abs(n): abs(n)`
但注意两个错误是在不同时间触发的，数量不对会在调用函数时报错，即 `my_abs()` 时
而类型不匹配则会在实际使用参数时报错，即 `my_abs('a')`，显示报错为其中的 `abs()`。
特别注意：由于 Python 是逐句执行的，所以在报错前语句已执行，所以不一定是传入时的类型错误

`int()` 将 float 或 str 转为 int 类型，float 会直接截去小数部分，str 按表示转化，但无法实现如 `int('1123.58')`
`float()` 将 int 或 str 转为 float 类型，str 按小数表示转化，即 `float('1123.58') \rightarrow 1123.58`，无法实现 `float('11/7')`
`str()` 将其他类型转化为 str 类型，类似于 Haskell 中的 `show` 函数。
`bool()` 将其他类型转化为 bool 类型，规则为 non-false value 皆视为 True (非 0 数，非空 tuple，非空 set，非 None)
`hex()` 将 int 转化为十六进制表示的 str，如 `hex(1123) \rightarrow '0x463'`，但注意 `int(hex(1123))` 会失败

Python - P7

函数定义

在 Python 中，函数的定义形式为 `def name(parameters):`] def 语句，函数名(传入参数)，冒号
注意 Python 中的缩进要求 body] 函数体

在 Python 中，函数将结果通过 `return` 语句返回。与 C++ 类似，`return` 与 `return None` 等价
但与 C++ 不同的是，Python 函数无需指定返回值类型，所以必须注意返回值的使用

导入函数

在交互式命令下，用语句 `from module import name` 来导入函数，与 Haskell 的 `import <module> (name, ...)` 类似

pass

空语句，与 C++ 中的 ; 类似，可以用于占位符，使得程序在语法结构上完整，如 `if cond: pass`

> `isinstance()` 用于判断变量是否属于一个 type 元组中的某一个。即如 `isinstance('a', (int, float))` → False

返回多个值

Python 中函数的 `return` 语句可以接受多个参数传入，组装成 tuple 后作为函数结果返回
即 `def name(): return x, y → (x, y)` 也有理解为返回 tuple 时可省略括号

与 C++ 中返回 struct 类似，相当于 `return` 语句创造了 1 struct 将多个变量值打包返回

多变量赋值

Python 中可以对多个变量同时赋值，语句形式为 `val1, val2, ..., valn = (elem1, ..., elemn)`

注意等式右边必须是 tuple，运行结果为 `val1 = elem1, ..., valn = elemn`

与函数返回多个值结合使用可得到 `val1, ..., valn = t → val1 = r1, ..., valn = rn`

类似于 C++ 中函数返回特定的 struct 由调用者分配给不同变量

默认参数

在 Python 中，可用 `def name(param=default1, ...):` body 的形式定义函数参数的默认值

注意，默认参数必须在必选参数之后，否则在传入参数时编译器无法区分是否默认参数

另外在调用参数时，可以通过 `parameter=default` 来明确参数传递(令哪个参数)

注意，在这种情形下调用可以不用遵循参数声明顺序，但是注意声明之后必须都是声明

使用默认参数时，越少改动的参数在声明时越靠后

特别注意：默认参数的贝武值必须是不可变对象，考虑 `def add_end(l=[]): l.append('end')`

则 `add_end() → ['end']`, `add_end() → ['end', 'end']`, `add_end() → ['end', 'end', 'end']`

考虑其原因，在程序编译时，编译器生成空列表 []，并将变量 l 指向 []，则在函数调用中，如果

使用默认参数值，则改动会实现在唯一的列表上，所以多次调用的改变会累加在默认参数中

但是对比 Racket 的 lexically scoped language (词法作用域性质)，考虑利用 Python 的这个特点实现相同的锁

即使函数可以保存自身运行过的结果，并加入一个控制位来输出结果，

`def a(s='', t=False, l=[]):` `a('a') → 'a'` | 如果要实现一个默认参数是可变对象

`if t: return l` `a('xyz') → 'xyz'` | 可以先赋默认值 None,

`else: l.append(s)` `a('', True) → ['a', 'xyz']` | so `def a(l=None): l=[] ...`

`return s`

Python - P8

可变参数

在 Python 中，可以向函数接收任意数量的位置参数 (Positional argument)，然后组装成 tuple。

定义函数时有形如 `def name(*args): body` 的声明，则传入的参数皆装入 `args`

注意：所有在可变参数之前的参数全部视为位置参数，且不能使用 key word argument 传入。在传入参数时，Python 允许将 list 和 tuple 整体作为位置参数传入，变量前加上 *。

即 $l = [1, 2, 3]$, 则可以使用形如 `name(*l)` 的用法, 作为位置参数传入

注意：对于定义可变参数或参数数量匹配的情况下，允许接收任意数量list或tuple且列表、元组和立即数可以以任意顺序组合。考虑 def name (m u, v, w, x, y, z, *args)

name c5,*l,4,*[,*] — | 5 | [1 2 3] | 4 | [1 2 3] | [1 2 3] |) = 20101

传入位置参数会依序传入 [u|v|w|x|y|z] [2|3|1|2|3] → args

注意：位置参数传入仅与 value 在参数序列中的位置有关。

$$950 \text{ name}(5, *l, 4, l, *l, l) = 5 [1 \ 2 \ 3] 4 [1 \ 2 \ 3] [1 \ 2 \ 3] [1 \ 2 \ 3]$$

需要特別注意川流序和位置 $u \ u \ v \ w \ x \ y \ z$ $(1, 2, 3) [1 2 3]) \rightarrow args$

关键字参数

●(keyword argument)与可变参数类似,传入任意数量含参数的参数,并自动组装成dict

定义函数时有形如 def name(*args, **kw): body，则传入的 keyword argument 比传入 kw

注意：所有的 positional argument 必须出现在所有 keyword argument 之前，否则 Syntax Error

传入参数时，Python 允许将 dict 整体作为 keyword argument 传入，在 dict 前加**

注意：在参数名不冲突的情况下，允许接受任意数量的 dict 作为关键字参数

考虑如下情形：
def func(x, y, z, *args, name, **kw):

body | args=(2,3,{‘a’:1}))| kw={‘b’:2}

~~d = {'a': 1}~~ → 5 [1, 2, 3] [1, 2, 3] {'a': 1} {'b': 2, 'name': 3}

e = {'b': 2, 'name': 3} | x | y | z | 2 | 3 | 'a' | {'b': 23} | name=3

`func(5,l,*l,d,**e) --> args=(2,3,'a') ; kw={'b':2}`

func (c5, l, *l, *d, **e) -> 5 [1 2 3] [1 2 3] {'a': 1} {'b': 2, 'name': 3}

func (5, l, *l, **d, **e) - ~~return~~ x | y | z | 2 | 3 | {'a': 1} | {'b': 2} | name = 3

注意如果不期待传入可变参数，可用 * 代替 *args | args = (2, 3) | kw = {'a': 1, 'b': 2}

```
def name(a,b,*name,**kw): body
```

注意 Python 中参数传入参数的特殊模式 (在函数定义时) 有形如

`def name(pos1, ..., posn, *args, name1, ..., namen, **kw): body`

必选参数 可选参数 命名关键字参数 关键字参数

位置参数 (positional arguments) 和 关键字参数 (keyword arguments)

dereference
(unpackag e)