

Python - Pg

print(*arg*)

print(*arg*)

recursive
function

Python 中的递归函数与 Haskell 等语言类似。
`def fact(n):` 注意 Python 中没有直接
`so fact:: Int → Int` 的类型提示，不检查类型，
`fact 0 = 1` return 1
`fact n = n * cfact(n-1)` return n * fact(n-1)
注意 可变对象，如 list，作为参数传入递归函数时，其改变会积累下来。
`so def acl(l, n):` l = [] , acl(l, 3) → l = [0, 0, 0, 1, 1, 0, 1, 1, 0, 0, 1, 1, 0, 1]
`if n == 0:` l = [] acl, 3
`return l`
`for i in range(2):` l.append(ci)
`acl, n-1`
`return l`

Stack overflow

在 Python 中需要特别注意递归造成 堆栈溢出 (stack overflow) 堆栈溢出
特别注意：Python 中没有尾递归优化 (tail recursion optimization)，无法通过尾递归解决
尾递归实际上和循环是等价的，没有循环语句的编程语言通过尾递归实现循环 (如 Haskell)

切片 (slice) 在 Python 中提供对 list 和 tuple 的切片方法，用于快速生成新 list 或 tuple

调用形式如 `name[start=0] : [end = len(name)] [: step = 1]]`

即有类似 `for ci = start ; ci < end ; ci += step { ret.push_back(name[ci]) } return ret;`

注意 切片的截止到索引为 end-1 的元素，如 `[: 3] → [0, 0, 0]`

注意 可以使用负数为索引，即索引范围为 `-len(name) (0) 到 -1 (len(name)-1)`

注意 正负数索引混用，python 会自动统一，需要注意索引的相对位置

注意 slice 同样适用于字符串。如 "abcdefg" [1:5:2] → "bd"

迭代 (Iteration) 在 Python 中，迭代是通过 `for ... in ...` 来完成的，类似于 C++ 中的 `for (int i : list) { ... }`

Python 的 `for` 循环抽象程度要高于以 `for (int i=0; i < list.size() - 1; ++i) { list[i] ... }` 的形式

即可以应用于任何可迭代 (iterable) 的对象上，如 list, tuple, dict, str

如对于 dict，可以用 `for key in dict` 迭代所有 key，也可以用 `for k, v in dict.items()` 迭代 value

Iterable 用于查看对象是否是可迭代的对像，在模块 collections 中

即 `from collections import Iterable, instance([1, 2, 3], Iterable) → True`

注意：对于 set 和 dict，由于存储顺序不同于 list 和 tuple，迭代顺序可能与显示顺序不同

Python - P10

enumerate 用于将 list 转换为 index-value 对 (生成的可迭代对象) (理解)

如 enumerate([{'a': 'b', 'c': 'd'}]) → [(0, {'a': 'b'}), (1, {'c': 'd'})] print 时是嵌套的

列表生成式 (List Comprehensions) 用于以类似集合构造器记法 (set builder notation) 的方式创建 list

如 [x * x for x in range(10)] → [0, 1, 4, 9, 16, 25, 36, 49, 64, 81]

与 {x * x | x ∈ N, x ∈ [0, 10]} 类似, 与 Haskell foldr (\x → (x * x) :) [] [0..9] 类似

也可以加入 if 条件判断, 如 [x * x for x in range(10) if x % 2 == 0] → [0, 4, 16, 36, 64]

与 {x * x | x ∈ N, x ∈ [0, 10], 2 | x} 类似

也可以用多层循环生成, 如 [x * y for x in range(1, 4) for y in range(1, 4)]

与 {x * y | x, y ∈ N, x, y ∈ [1, 4]} 类似 → [1, 2, 3, 2, 4, 6, 3, 6, 9]

也可以在一个循环中同时迭代多个值, 如 [key + ' = ' + value for key, value in dict.items()]

列表生成式的形式可描述为 [f(x₁, ..., x_n) | Iter(x₁), ..., Iter(x_n), cond*]

注意 f(x₁, ..., x_n) 表达式 唯一的提供 x₁, ..., x_n 的迭代; cond 提供 x_i 的筛选条件

注意对于任意 x_i, 提供 x_i 的迭代必须早于限制 x_i 的 cond 出现

另外, 由于列表生成式实质上源于集合构造器记法, 所以可以用列表生成式生成集合

如 {x * x for x in range(-10, 11)} → {64, 1, 0, 100, 36, 4, 9, 16, 81, 49, 25}

注意生成集合过程中会自动删除重复的元素, 而且元素并不按输入顺序排序

生成器 (generator), 用于在运算过程中不断推算出后续的元素, 不必创建完整的 list, 节省大量空间

创建生成器的方法之一是用 () 替代列表生成器中的 [], 形如 (f(x₁, ..., x_n) | Iter(x_i) cond*)

可以通过 next() 函数获得生成器的下一个元素, 没有下一个元素时抛出 StopIteration

如: g = (x * x for x in range(3)) 注意可以直接用 for 循环迭代生成器

next(g) → 0, next(g) → 1, next(g) → 4 or for i in g: 一旦出现 StopIteration

next(g) → StopIteration. 遍历结束后 print(i) → 0, 1, 4

创建生成器的方法之二是通过 def 函数实现, 在函数中包含 yield 关键字 (类似于 return)

如: def fib(): 注意形如 n, a, b = n+1, b, a+b 在 Python 是合法的

n, a, b = 0, 0, 1 while True: 其原理为先将表达式右边装入 tuple 再分别赋值

while True: yield (n, a, b) 即 t = (n+1, b, a+b), 然后 n, a, b = t

n, a, b = n+1, b, a+b

g = fib(), for i in range(5): 生意由此处是一个无限的 generator

print(next(g)) A = 0, 单步运行 如果直接使用生成器作为 for 循环的迭代

→ 0, 1, 1, 2, 3 A = 1, 2, 3 会陷入一个无限循环

需要循环来确定停止条件

Python - P11

StopIteration 当对已经达到末尾的 generator 使用 next() 函数时会抛出的错误

注意：在用作 for 循环的迭代时，Python 会自动处理 StopIteration 的情况（结束 for 循环）

如果在一个循环中用 next() 得到 generator 的下一个值，则需要注意处理 StopIteration

方案是 try : / except StopIteration as e : 来处理，类似于 C++ 中的 exception & e { ... }

类似于 try { ... } catch(exception & e) { ... }，C++ 中的处理

如果有原有的 generator 中有 return 值，会存在 e.value 中。

迭代器 (Iterator)，在 Python 中表示一个数据流，可以被 next() 函数调用并不断返回下一个数据

可以将其看作一个无法提前知道长度的有序序列，采用惰性求值，只在被 next() 函数调用时计算下一值

而当其没有数据时，即抛出 StopIteration 错误

可迭代的 (Iterable) 可以直接用于 for 循环的对象（统称为可迭代对象）

一类是集合数据类型，如 list, tuple, set, dict, str

一类是 generator，如生成器和带 yield 语句的 generator function

注意第一类中 list, tuple, set, dict, str 虽然是 Iterable，但不是 Iterator

但是可以用 iter() 函数转换为 Iterator

如 isinstance([1, 2], Iterator) → False, isinstance(iter([1, 2]), Iterator) → True

注意 Iterator 可以是一个无限的数据流，但集合数据类型 (如 list, set) 必须是有限的

特别注意：由于 Haskell 本身是惰性求值的，所以如 [1..] 的列表生成式实际上是一个迭代器

另外 Iterator 与 Racket 中的 Stream 类似，即可以认为其本身是一个 procedure

每次调用产生一个 pair，包括一个返回值和产生后续 stream 的 procedure，即形如 (procedure) → (value . procedure)

注意：Python 中对多变量同时赋值是通过 Iterator 实现的，而非以 tuple 的形式实现

即实际的过程是 list, tuple (Iterable) $\xrightarrow{\text{iter()}}$ Iterator (或直接写) $\xrightarrow{\text{unpack}}$ multi-var assignment

特别注意 Python 期望在用 next() 获得足够的 value 后即出现 StopIteration

如定义 generator：gen = (x for x in range(3))

a, b, c = gen → a=0, b=1, c=2 (正常情形)

a, b, c = iter([1, 2, 3]) → a=1, b=2, c=3 (由 list 转化为 Iterator)

a, b, c = [1, 2, 3] → a=1, b=2, c=3 (直接使用 Iterable 的对象)

a, b, c, d = gen → not enough values to unpack

a, b = gen → too many values to unpack

注意在这种情况下，generator 已被调用了 3 次 next() (a, b, Stop)

但是并没有完成对 a, b 的赋值，所以要特别注意这个语句对不同元素的影响

特别注意，单独的 Iterator 对象每次 next() 都是不可逆的，而不论与之相关的语句是否正确执行

Python - P12

3月9日 - 周五

函数式编程(functional programming)是一种编程范式(programming paradigm)

将计算机中的运算(computation)视为数学上的函数计算(evaluation of mathematical function)

避免使用可变状态(changing-state)和可变对象(mutable data)

基于λ演算(lambda calculus, λ-calculus)是一套数学逻辑中的形式系统(formal system in mathematical logic)

由入项(lambda term)构成的语言，其语法和定义有：

x 变量(Variable) 表示参数或数学/逻辑值(mathematical/logical value)的字符/字符串(string)

$(\lambda x.M)$ 抽象(Abstraction)：如果 x 是变量， M 是入项(function definition)， x 被定在表达式中，则 $(\lambda x.M)$ 是有效的入项

$(M N)$ 调用(Application)或称应用：applying a function to an argument. 如果 M, N 是入项，则 $(M N)$ 是有效入项

$(\lambda x.M[x]) \rightarrow$ 入演算有归约的操作：

$(\lambda y.M[y]) \rightarrow$ (λ-conversion) α-转换：重命名表达式中绑定(形式)变量(bound variable)以避免名称冲突(name collision)

$((\lambda x.M) E) \rightarrow$ β-归约(β-reduction)：以参数表达式(argument expression)代替抽象中的绑定变量(bound variable in body of abstraction)

入演算中关于自由变量(free variable)的定义：

x 中的自由变量只有 x . $S = \{x\}$

$S = T - \{x\}$

$(\lambda x.t)$ 的自由变量集合 S ，为 t 的自由变量集合 T 去掉元素 x ，此时 x 是整个入项的约束变量

$(t S)$ 的自由变量集合 V ，为 t 与 S 的自由变量集合 T 与 S 的并集， $V = T \cup S$

入演算中的避免捕获的替换记法(capture-avoiding Substitution)

如果 t, s, r 为入项， x, y 为变量，则 $[t[x:=r]]$ 表示在入项 t 中，用 r 来替换 x 变量。

$x[x:=r] = r$ ，由于 x 中的自由变量只有 x

$y[x:=r] = y$, 如果 $x \neq y$

$(t s)[x:=r] = ((t[x:=r])(s[x:=r]))$

$(\lambda x.t)[x:=r] = (\lambda x.t)$, 在 $(\lambda x.t)$ 中 x 是约束变量

$(\lambda y.t)[x:=r] = (\lambda y.(t[x:=r]))$, 如果 $x \neq y$ 且 y 不在 r 的自由变量集合中

对于入项 t , 变量 y 称为 fresh.

$(\lambda x.t)s \rightarrow$ 入演算中的 β -归约(β-reduction)有形式如：(application form)

$t[x:=s]$ 可以简化为 $t[x:=s]$, 即：将一个调用(应用)形式 归约为一个入项 term

可以看出入演算可以接受函数作为输入(argument)或输出(return value)

λ -calculus 可以看出是 Haskell 的理想化版本, Haskell 是纯函数式语言

如 $((\lambda x \rightarrow (y \rightarrow (x^2 + y^2))) (5)) (2)$ 等价于 $(\lambda x \rightarrow (\lambda y \rightarrow x^2 + y^2)) 5 2$

Syntax $= (y \rightarrow (5^2 + y^2)) (2) = 5^2 + 2^2 = (\lambda y \rightarrow 5^2 + y^2) 2 = 5^2 + 2^2$

x (variable) : a character or string representing a parameter or mathematical/logical value

$(\lambda x.M)$ (abstraction) : function definition (M is λ term), variable x becomes bound in the expression

$(M N)$ (application) : applying a function to an argument, M and N are λ terms

anonymous form 特别注意：匿名形式与 Haskell 和 Python 中的 lambda 函数有同样的形式

$(\lambda x \rightarrow (y \rightarrow (x^2 + y^2)))$, $(\lambda x \rightarrow (\lambda y \rightarrow x^2 + y^2))$, lambda $x, y : x^2 + y^2$ 是等价的

Python - P13

8.9 - 1. Haskell

高阶函数 (Higher-order function) 利用入演算中函数是有效输入项 (valid input term) 的规则

即函数本身可以作为输入 (argument) 或输出 (return value), 且可被变量 (variable) 指向

在 Python 中可以使变量指向函数, 且通过变量调用函数.

如 $f = \text{abs} \rightarrow f(-1) \rightarrow 1$, 即变量 f 指向函数 $\text{abs}()$

在 Python 中, built-in 的函数名也属于变量, 可以重新定义指向

如 $\text{abs} = 10$, 则此时 abs 不再指向函数 $\text{abs}()$, 而是整型值 10,

特别注意: Python 中没有机制来保证这些内建函数名不被修改,

所以编程时要特别注意变量命名

在 Python 中, 同样可以使用函数作为另一个函数的传入参数, 与 Haskell 类似

如 $\text{def add}(x, y, f): \quad \text{等价于 } \text{add} :: (\text{Num} a) \Rightarrow a \rightarrow a \rightarrow (a \rightarrow a) \rightarrow a$

$\text{return } f(x) + f(y)$ $\text{即 } \text{add } x \ y \ f = (f \ x) + (f \ y)$

如 $\text{add}(-2, 3, \text{abs}) \rightarrow 5$ $\text{即 } \text{add } -2 \ 3 \ \text{abs} \rightarrow 5$

map Python 中的 map 函数与 Haskell 中的 map 函数有相同的抽象运算规则, 但实现细节不同

在 Python 中, map 函数接收一个 Iterable 的参数, 并返回一个 Iterator

而在 Haskell 中, map 函数接收一个 list, 并返回一个 list,

如 $\text{myMap} :: (a \rightarrow b) \rightarrow [a] \rightarrow [b]$ $\text{等价于 } \text{def myMap}(f, l):$

$\text{myMap} [] = []$ $\text{for } x \text{ in } l:$

$\text{myMap } f (x:xs) = (f x) : (\text{myMap } f xs)$ $\text{yield } f(x)$

注意: 与 Haskell 本身采用惰性求值而 map 直接输出 list 作为结果不同

Python 实际上是通过 map 返回的 Iterator 实现了惰性求值 (lazy evaluation)

如果需要一个 list 结果, 需要再利用 list (map(f, l)) 的调用方式

从运算逻辑上看, reduce 从右向左依次将前一个结果作为参数传入下一个函数 (Iterable) -> point

reduce Python 中 functools 模块中的 reduce 函数用于将一个函数在序列的元素上做累积计算

即从运算逻辑上看 $\text{reduce}(f, [x_1, x_2, x_3, x_4]) = f(f(f(x_1, x_2), x_3), x_4)$

当用 Haskell 实现此函数时, 可见传入的函数要求传入两个参数且传出结果与参数又有相同类型

考虑两种实现方法;

$\text{myReduce} :: (a \rightarrow a \rightarrow a) \rightarrow [a] \rightarrow a$ $\text{myReduce} [] = \text{error } "empty \ list"$ (1)

$\text{myReduce} [] = \text{error } "empty \ list"$ 基础实现

$\text{myReduce } f (x:xs) = \text{foldl } f x xs$ 归纳实现

$\text{myReduce } f (x:y:xs) = \text{myReduce } f$

可用于构造复杂函数如 $\text{String} \rightarrow \text{Int}$: 如 $\text{DIGITS} = \{'0': 0, '1': 1, \dots, '9': 9\}$

$\text{def char2num}(s): \quad \text{def str2int}(s):$

$\text{return DIGITS}[s]$ $\text{return reduce(lambda x, y: x*10 + y, map(char2num, s))}$

通过 key 查找 associated value

匿名函数作为参数传入

Iterable 对象

Python - P14

filter

Python 中的 filter 函数传入一个 unary predicate (单目谓词) 和一个 Iterable 对象，返回一个 Iterator。对其中每个元素应用传入函数，根据返回 True / False 决定是否保留该元素。

运算符与 Haskell 中的 filter 函数类似，差别在于 Haskell 的传入返回 list。

注意 filter 与 map 一样，返回的是一个 Iterator。如需列举需要类似 list() 函数。

可以用 filter 编写素数生成器。

基于埃托尼拉斯筛法。

def oddIter(): 返回一个奇数迭代器。

n = 3 从 3 开始的

while True: 奇数迭代器

yield n

n += 2

while True:

n = next(it) 取下一个素数 n

yield n 从 it 中筛选可被 n 整除的数

it = filter(lambda x: x % n != 0, it)

由于 Iterator 采用惰性求值，所以可以用类似方法生成具有特定性质的无限序列。

注意 Python 与 Haskell 的不同实现方式

myFilter :: (a → Bool) → [a] → [a]

myFilter [] = []

myFilter f (x:xs)

| f x = x : (myFilter f xs)

| otherwise = myFilter f xs

Sorted

在 Python 中，sorted 函数用于对一个 Iterable 对象进行排序，默認為 comparison function 是 <，且对于 Python 的 sorted 函数，不能修改使用的 comparison function。

如 sorted([1, -1, 2, -3, 5, -8]) → [-8, -3, -1, 1, 2, 5]

但是 sorted 允许将一个函数传入 key 参数，sorted 会使用每个元素传入 key() 的返回值进行排序。

如 sorted([1, -1, 2, -3, 5, -8], key=abs) → [1, -1, 2, -3, 5, -8]

特别注意：key 函数的返回值必须可以作为 operator < 的操作数。

如 class a: sorted([a, a, a]) →

pass TypeError: '<' not supported between instances of 'a' and 'a'

虽然 sorted 实际上只是使用了每个元素传入 key() 的返回值，但并未改动元素本身。

如实现忽略字符串中大小写字母的字典排序。

sorted(['abc', 'XYZ', 'ABC']) → ['ABC', 'XYZ', 'abc']

而 sorted(['abc', 'XYZ', 'ABC'], key=str.lower) → ['abc', 'ABC', 'XYZ']

如 sorted() 还允许设置一个判断量 reverse，当 reverse=True 时，返回前会先 reverse 操作。

如 sorted([1, -1, 2, -3, 5, -8], reverse=True) → [5, 2, 1, -1, -3, -8]

注意 sorted 接受任何 Iterable 对象用作排序，但返回的是 list。

如 sorted((1, -1, 2, -3, 5, -8)) → [-8, -3, -1, 1, 2, 5]

Python - P15

89 - Haskell

返回函数

在 Python 中，可以将函数作为另一个函数的返回值使用，即入算所支持的另一种用法。

返回的函数可以赋值给变量，并经由()函数调用运算符进行调用。

注意 Python 中同样使用在 Racket 中使用的 Stream 定义形式。

如生成一个正整数的 stream，每次调用返回一个当前值 x 及生成后续 stream 的 pair。

```
def nats(x=1): (define nats (letrec ([f (lambda (x) :+ +
```

```
f = lambda : nats(x+1))
```

```
(cons x (lambda () (f (+ x)))))))
```

```
def kItem(stream, n): (define (k-item stream n) :+ +
```

```
f, l = stream, [] (letrec ([f (lambda (st acc n) :+ +
```

```
(x for i in range(10): :+ + if i == n) (reverse acc))
```

```
(let ([pr (st)])
```

```
k, f = f() (let ([pr (f cdr pr) (cons (car pr) acc))
```

```
(x: x.l.append(k)) (l, f) (f (lambda () (f (- n 1)))))))
```

```
def kItem(x): :+ + return (f stream 'c) n))
```

```
kItem(nats, 10) → [1, 2, 3, 4, 5, 6, 7, 8, 9, 10] (k-item nats 10) → '(1 2 3 4 5 6 7 8 9 10)
```

注意：根据 Python 和 Stream 的实现，`(define (nats [x 1]))`

Racket 中 `hasStream` 可优化为

```
(cons x (lambda () (nats (+ x))))))
```

可以如此解释 Python 函数的计算环境：传入参数 覆盖 定义时环境(不断闭包) 覆盖 调用时环境

在 Python 需要特别注意如果返回函数中有自由变量，则自由变量的来源可能是不同环境。

如 `def count():`

```
def count():
```

```
fs = []
```

注意这两段中定义的区别

```
fs = []
```

for i in range(1, 4): 在于对 return i*i 中

```
def f(j):
```

```
def f(j):
```

return i*i i 的取值确认位置不同

```
def g():
```

```
def g():
```

return j*j j 的取值确认位置不同

```
return g()
```

```
fs.append(f)
```

```
for i in range(1, 4): fs.append(f(i))
```

```
return fs ← f1, f2, f3 = count()
```

```
return fs
```

```
j = 3 * i
```

注意这两段中定义的区别

```
j = 3 * i
```

```
j = 3 * i
```

for i in range(1, 4): j = 3 * i + 3

```
j = 3 * i + 3
```

```
j = 3 * i
```

f1() → 6

```
f1() → 6
```

```
j = 3 * i
```

f2() → 9

```
f2() → 9
```

```
j = 3 * i
```

f3() → 12

```
f3() → 12
```

所以在 Python 中使用闭包时需特别注意对循环变量或其他参数改动对自由变量的使用。

可以理解为，Python 并不是在函数定义时完成闭包，而是在退出定义环境时不断完成的。

def clos():

+1, +2, +3 = clos()

所以，如 t+ 表达式的值被捕获 [] 和

def f(j):

j = 3 * i

Racket 的 lexer-parser-eval 都是

def g():

j = 3 * i + 3

在函数定义时对环境进行闭包

return j+i

f1() → 6

Haskell 不允许修改变量，所以自由变量

for i in range(1, 4):

j = 3 * i + 3

在定义环境和调用环境之间

fs.append(f(3*i))

f2() → 9

Python 在函数离开定义环境时进行闭包

return fs

i=3

i=4

插入语句

Python - P16

`lambda` 在 Python 中可使用 `lambda` 语法定义匿名函数，其语法为 `lambda [args*]: expr`

(由于不能显示) 500 (lambda x : x*x)(9) → 81 <function <lambda> at address>

特别注意： Python 中 lambda 函数允许在定义时使用仍未定义的变量，

只要在函数调用时当前计算环境中有变量定义即可。

所以需要特别注意 lambda 函数中 使用全局变量的情形

装饰器 (decorator), 在 Python 中用于为函数定义添加额外功能。

基本语法结构为：

def name(*args, **kw): 定义装饰器的name与需要使用的参数，

def decorator(func): 传入装饰器作用的函数是func

`def wrapper(*func_args, **func_kw):` 定义函数传入位置参数和关键字参数

block (before the func) 定义在函数 func 之前 执行的语句

ret = func(*func_args, **func_kw) # 调用函数 func(*func_args, **func_kw)

如果在函数 func 之后添加了语句 block (after the func), 它将定义在函数 func 之后执行的语句

返回 func(*func_args, **func_kw) 的结果

return wrapper

return decorator([soft_unwind])。向函数调用 = smartify ret 为 None，并不会出现错误

注意：虽然经过了decorator的返回，但是因为在wrapper完成的包装，所以 `now._name_` → wrapper

可以通过调用 `functools`, 在 `decorator` 中使用 `@functools.wraps(func)` 使得 `wrapper` 拥有函数名。

④ def decorator(func): @functools.wraps(func)
 def wrapper(*args, **kwargs):
 pass

```
def now():  
    print('%s %s %s' % ('text', func_name_))
```

```
print('end %sc)' % func_name)
```

return wrapper now() → exec now():

return `hello world`

另外decorator 可用于程序运行计时:

RIP 有 `start = time.time()`, call func, `end = time.time()`, `print(end - start, 's')`

面向对象(Object-oriented programming, OOP)的设计模式中, decorator称为装饰模式

通常用继承和组合实现。在Python中从语法层次支持decorator，可以~~通过函数/类实现~~：

Epimerization - $\text{CH}_3\text{CH}(\text{OH})\text{CH}_2\text{CH}_3 \rightarrow \text{CH}_3\text{CH}(\text{OEt})\text{CH}_2\text{CH}_3$ (Et = CH_2CH_3)

Partially applied function，在Python中通过functools中的functools.partial()实现。

即有固定参数如 `partial(func, *args, **kw)` 如 `f = functools.partial(max, 10)`

返回一个函数与func相同，但已添加 *args, **kwargs 到 func 中。f(2, 3, 5) → 10 等价于 max(10, 2)

部分应用函数的原理与 Haskell 中类似，但实现更为灵活。