4. 编程基础 IV：程序的结构.

一. 代码是用来读的（实践经验）.

1. 团队的需要.

2. 维护的需要.

二. 降低复杂度：分解与抽象（方法学）.

1. 分解：分而治之（divide-and-conquer）.

关键点：(1) 分解之后，每一部分的复杂度要变小.

(2) 相互之间关联度要小，相互独立.

Example：CPU结构的设计.

2. 抽象：复杂系统 → 系统接口 + 系统实现.

关键点：(1) 抽象之后，接口的复杂度变小.

(2) 接口和实现之间达成一种契约.

Example：门电路.

三. 编程 = 数据结构 + 算法.

算法是计划、过程、步骤，数据结构是操作的目标、对象.

四. 算法建模.

1. 三种机制：基本表达式 + 分解 + 抽象.

(1) 基本表达式：数字运算，逻辑运算.

前缀表达式：树形表示法.

(2) 分解.

(3) 抽象.

2. 两种思路：迭代，递归.

Example：计算阶乘.

递归：

```
(define (factorial n)
 (if ( = n 1)                    ;=  代表是否等于
  1                              ;n等于1，则取值1
  (* n (facorial (- n 1))))))   ;否则，则取值n乘以n-1的阶乘
```

迭代：

```
(define ( factorial n)
 (fact-iter 1 1 n))

(define (fact-iter product counter max-count)
 (if (> counter max-count)
     product
     (fact-iter ( * counter product)  (+ counter 1)  max-count)))
```

Example：Fibonacci数列

递归：

```
//递归实现斐波那契数列
 long fab_recursion(int index)
 {
  if(index == 1 || index == 2)
  {
    return 1;
  }
  else
  {
    //递归求值
    return fab_recursion(index-1)+fab_recursion(index-2);
  }
}
```

迭代：

```
#include <iostream>
using namespace std;

//迭代实现斐波那契数列
long fab_iteration(int index)
{
  if(index == 1 || index == 2)
  {
    return 1;
  }
  else
  {
    long f1 = 1L;
    long f2 = 1L;
    long f3 = 0;
    for(int i = 0; i < index-2; i++)
    {
      f3 = f1 + f2; //利用变量的原值推算出变量的
一个新值
      f1 = f2;
      f2 = f3;
    }
     return f3;
  }
}
```

对比：

```
(factorial 6)
(* 6 (factorial 5))
(* 6 (* 5 (factorial 4)))
(* 6 (* 5 (* 4 (factorial 3))))
(* 6 (* 5 (* 4 (* 3 (factorial 2)))))
(* 6 (* 5 (* 4 (* 3 (* 2 (factorial 1))))))
(* 6 (* 5 (* 4 (* 3 (* 2 1)))))
(* 6 (* 5 (* 4 (* 3 2))))
(* 6 (* 5 (* 4 6)))
(* 6 (* 5 24))
(* 6 120)
720
```

```
(factorial 6)
(fact-iter 1 1 6)
(fact-iter 1 2 6)
(fact-iter 2 3 6)
(fact-iter 6 4 6)
(fact-iter 24 5 6)
(fact-iter 120 6 6)
(fact-iter 720 7 6)
720
```

递归：先展开再规约。

迭代：在循环中改变某个值，达到某个条件后结束。

五、数据建模。

基础数据

整数，浮点数，布尔值。

数据的组合

数组　相同类型的数据

结构体　不同类型的数据

对象　数据与行为的组合

数据的抽象

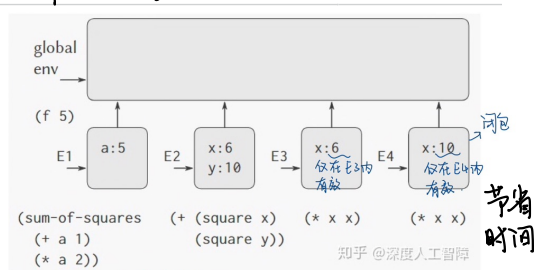
有序对

六、模块化。

1. 函数实参的传入无赋值过程，所有的值都不发生变化。

```
(define (square x)
  (* x x))
(define (sum-of-squares x y)
  (+ (square x) (square y)))
(define (f a)
  (sum-of-squares (+ a 1) (* a 2)))

(f 5)
```



节省时间。

数值以参数和返回值的形式流动。

环境之间无约束，完全通过参数传递数值

无赋值过程

2. 函数内部存在赋值，值存在变化。

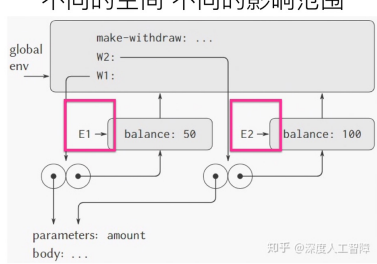
E1 E2 不同的环境

不同的空间 不同的影响范围

```
(define (make-withdraw balance)
  (lambda (amount)
    (if (>= balance amount)
        (begin (set! balance (- balance amount))
               balance)
        "Insufficient funds")))

(define W1 (make-withdraw 100))
(W1 50)

(define W2 (make-withdraw 100))
(W2 50)
```



节省空间（自带占位）。

环境之间有约束，新环境维护局部状态变量，this指针

七. 编程的典型场景：数据处理.

1. 输入.

2. 处理.

3. 输出.

八. 用日志记录数据.

你想要执行的任务	此任务最好的工具
对于命令行或程序的应用，结果显示在控制台。	`print()`
在对程序的普通操作发生时提交事件报告(比如：状态监控和错误调查)	`logging.info()` 函数(当有诊断目的需要详细输出信息时使用 `logging.debug()` 函数)
提出一个警告信息基于一个特殊的运行时事件	`warnings.warn()` 位于代码库中，该事件是可以避免的，需要修改客户端应用以消除告警 `logging.warning()` 不需要修改客户端应用，但是该事件还是需要引起关注
对一个特殊的运行时事件报告错误	引发异常
报告错误而不引发异常(如在长时间运行中的服务端进程的错误处理)	`logging.error()`, `logging.exception()` 或 `logging.critical()` 分别适用于特定的错误及应用领域

级别	何时使用
`DEBUG`	细节信息，仅当诊断问题时适用。
`INFO`	确认程序按预期运行
`WARNING`	表明有已经或即将发生的意外（例如：磁盘空间不足）。程序仍按预期进行
`ERROR`	由于严重的问题，程序的某些功能已经不能正常执行
`CRITICAL`	严重的错误，表明程序已不能继续执行

```
import logging
logging.warning('Watch out!')  # will print a message to the console
logging.info('I told you so')  # will not print anything
```

如果你在命令行中输入这些代码并运行，你将会看到：

```
WARNING:root:Watch out!
```

如果你的程序包含多个模块，这里有一个如何组织日志记录的示例:

```
# myapp.py
import logging
import mylib

def main():
    logging.basicConfig(filename='myapp.log', level=logging.INFO)
    logging.info('Started')
    mylib.do_something()
    logging.info('Finished')

if __name__ == '__main__':
    main()
# mylib.py
import logging

def do_something():
    logging.info('Doing something')
```

如果你运行 myapp.py ，你应该在 myapp.log 中看到:

```
INFO:root:Started
INFO:root:Doing something
INFO:root:Finished
```