

Week 01

cook book

特色：

- 每個步驟是一個 function
- main 負責呼叫每個步驟

限制：

- No goto
- 使用 procedures 把問題切割成小問題
 - procedures: 不限制回傳與參數
- 使用全域變數共享狀態
- 修改 global 變數

分析 Commentary:

- 容易有副作用(side effect)
- 非 idempotent, 跑多次會有不一樣的結果
 - idempotent: 多執行一次結果不變

應用：

- 適合使用於 隨時間累積的外部資料(全域變數) 且需要對外部資料做操作的時候
- 全域變數難以追蹤, 不建議使用於大型專案

Principle 原則

- Names
- Bindings : name 與實做的關聯
- scope

1. Binding Time

- Binding 發生的時間點
 - 例如 int a = 1; a = 2; compile time...

2. Life time & storage management

- Life time
- scope
- garbage : 沒有人指到的 address
- Gangling reference: 還有變數指向 address, 但已經被釋放了
- Storage management
 - static: compile time
 - stack: 函數參數 本地變數 暫存
 - heap: dynamic allocation
 - malloc/free
 - new/delete

3. static scope vs dynamic scope

- static scope: compile time 跟序語結構
 - 例如: C/C++
- dynamic scope: run time 根據呼叫順序
 - 例如: Python

4. Aliasing

- 兩個變數指向同一個記憶體位置
- 例如：C/C++的 reference
- 會造成副作用，難以追蹤
- 妨礙最佳化：對 compiler 優化造成影響

5. polymorphism(多型)

- 函數名稱相同但實作不同
- overriding (Subtype polymorphism)
- overloading(多載) (Ad hoc polymorphism)
 - 函數名稱相同但參數不同 -> void func(int a) vs void func(double a)
 - 例如：C++的 operator overloading
- Generic/Template (泛形) (Parametric polymorphism)
 - 函數名稱相同但參數不同 -> void func(T a) vs void func(T b)
 - 例如：C++的 template
- duck typing (結構多型/動態多型)
 - go lang 的 interface

6. Reference Environment 如何找到變數 以及呼叫

- Association list 堆疊
- central table 集中表

Week 02

pipeline style

Constraints 限制:

- 沒有副作用
- 不使用全域變數
- $f(g(x))$ function 串接

評價 COmmentary:

- pure function 無副作用 多次呼叫結果相同(idempotent)
- 容易測試(Testable)
- 容易並行(Concurrent)

Currying :

```
def add(x):  
    def add_y(y):  
        return x + y  
    return add_y  
  
print(add(1)(2)) # 3
```

Runtime Environment - Runtime memory layout

- stack frame
 - local variables
 - return address
 - parameters
 - 遞迴 recursion -> stack
 - stack overflow

- heap
 - new/malloc
 - memory leak
- Data
 - const
 - global
- Text(code)
 - code segment
 - read only

Week 03

- Language Design time: 設計者決定語法與語意
- Language Implementation time: int is a 64 bit integer
- Program Writing time: int x=1
- compile time: 呼叫哪個版本的 function ex. fmt.Println
- Link time: go build 會 link package
- Load time: 載入執行檔到記憶體
- Runtime: interface runtime 動態決定

Thinks style(物件導向風格)

Constraints 限制:

- 大問題拆成 things
- 每個物件都是一個封裝 capsule(class)
- 可重用其他 capsule(繼承)

分析 Commentary:

- 封裝 + private data + Public method

特徵:

- 資料私有 方法公開
- 可抽換(interface)
- 可繼承

Letterbox style (信箱風格)

Constraints 限制:

- 同 Things
- dispatch 接收訊息(message) 在依據訊息執行對應 function
- message 也可以被轉發(forward)
- delegation 委派 讓 B 做 A 的事情但不需要繼承 A

```
class Controller(){
    def dispatch(self, message):
        if message[0] == "init"
            return self.someMethod;
}

// main
c = Controller
c.dispatch(['inti', ...])
```

Week 04

沒有*就不能更改 c

```
type Circle struct {
    radius float64
}

func (c *Circle) Area() float64 {
    c.radius = 2
}

func (c Circle) Perimeter() float64 {
    return 2 * math.Pi * c.radius
}
```

Week 05

Go OOP Language Features & Analysis

- struct embedding 結構體嵌入

```
type A struct {
    a int
}

type B struct {
    A
    b int
}
```

- Template method Pattern

同樣的步驟但不同的實作

```
type Template interface {
    Step1()
    Step2()
    Step3()
}

type ConcreteTemplate struct {
    Template
}

func (c *ConcreteTemplate) Step1() {
    // do something
}

func (c *ConcreteTemplate) Step2() {
    // do something
}

func (c *ConcreteTemplate) Step3() {
    // do something
}

type ConcreteTemplate2 struct {
    Template
}

func (c *ConcreteTemplate2) Step1() {
    // do something
}

func (c *ConcreteTemplate2) Step2() {
    // do something
}

func (c *ConcreteTemplate2) Step3() {
```

```

    // do something
}
func main() {
    var t Template
    t = &ConcreteTemplate{}
    t.Step1()
    t.Step2()
    t.Step3()

    t = &ConcreteTemplate2{}
    t.Step1()
    t.Step2()
    t.Step3()
}

```

Week 06 Overloading and Generic in c++ and go and java

- overriding (Subtype polymorphism)
- overloading(多載) (Ad hoc polymorphism)
 - C++

```

int Add(int a, int b) ...
int Add(int a) ...
// C++ can operation overloading
int operator+ (int a, int b)

```

- java

```

static int Add(int a, int b) ...
static int Add (int a) ...
// use Universal Type interface{}
public static Object add(Object a, Object b){
    if (a instanceof Integer && b instanceof Integer){
        ...
    }
}

```

- Go (no support overloading)

```

func AddInt(a, b int) int { return a+b}
func AddFloat(a, b float) float {return a+b}
// use Universal Type interface{}
func a(val interface{}){
    switch v:=val.(type){
        case int : ...
        case string : ...
    }
}

```

- Generic/Template (泛形) (Parametric polymorphism)
 - C++

```

template <typename T>
T Add(T a, T b){ return a+b}

```

- java

```

public static <T extends Number> Number add(T a, T b){
    if (a instanceof Integer) { ... }
}

```

```
// ? super T
//
class Arbiter<T extends Number> { }
Arbiter<? extends Number> h =...;

// if T = integer
// 只可接受integer
class Arbiter<T> { }

class Arbiter<T> {
    T bestSoFar;
    // 可定義method
    // Method可接受number, object, integer...
    Chooser<? super T> comp;
}
```

► go

```
func Add[T int | float64](a, b T){ }
// Addable is a interface
func Add[T Addable](a, b T){ }
// Any = interface{}
func Add[T, U Any ](a T, b U){ }
func Add[T Any](a, b T){ }
```

Week 07

演講

Week 08

Bulletin Board Style 公告欄風格

Constraints 限制:

- 事件導向 不直接呼叫彼此 method
- subscribe 訂閱
- publish 發布
- 統一由 Bulletin Board

特性:

- 完全解耦合 不知道對方存在
- 易擴充 模組化良好
- 去中心化

缺點:

- subscribe 沒有阻止同個 event_type 不同 handler, handler 間不可有相依性 不然容易出錯

用途:

- 分散式系統

```
class EventManager:
    def __inti__(self):
        self._subscriptions = {}

    def subscribe(self, event_type, handler):
        if event_type in self._subscriptions:
            self._subscriptions[event_type].append(handler)
```

```

        else
            self._subscriptions[event_type] = (handler)

    def publish(self, event):
        event_type = event[0]
        if event_type in self._subscriptions:
            for h in self._subscriptions:
                h(event)

class A:
    def __inti__ (self, event_manager):
        self._event_manager = event_manager
        self._event_manager.subscribe('load', self.load)
    def load(self, event):
        some_data = event[1] ...

em = EventManager()
A(em)
em.publish(('load', sys.argv[1]))

```

I don't know why but tuple

Tuple 性質:

- Ordered: 有序的 (有 index 0,1,2,3 O(1) get element)
- Immutable: 不可修改
- Heterogeneous : 可以包不同資料型態
- Indexable 可用 t[0]
- Iterable 可用 for

Introspective style 自省風格

Constraints 限制:

- 不可修改 locals 的東西 但可以修改自己的
- Reflection 的第一步

特性:

- Introspective(自省): 程式可以存取自己或其他程式的部份結構資訊
- C/C++/go/ 不支援 可以當只有 python ruby 支援
- Introspection 讀取變數型別資訊...debug 用途 限制行為

```

import inspect

# 回傳這個scope當前的List<FrameInfo>
inspect.stack()
# get the local variables by locals()
def frequencies(word_list):
    a = 0
    locals()['word_list']
    globals()['...']
    a = 1

```

Reflection style

Constraints 限制:

- Introspection 內省
- reflection 反射: 程式可於執行時修改自身
- 可動態新增 function

用 `exec()` 含意： 高度彈性 難以除錯 安全性低 惡意注入

用途： 動態插件系統 腳本引擎 為經典用途

- python

```
sort_func = "lambda word_freq: sorted(word_freq.items(), key=operator.itemgetter(1),
reverse=True)"
exec('sort = ' + sort_func)
locals()['sort'](arg)
```

- Go

```
import reflect
type User struct {
    Name string `json:"name" db:"user_name"`
    Age  int    `json:"age" db:"user_age"`
}

func main() {
    u := User{"Alice", 20}
    t := reflect.TypeOf(u)

    for i := 0; i < t.NumField(); i++ {
        field := t.Field(i)
        fmt.Printf("Field: %s, json: %s, db: %s\n",
            field.Name, // Name Age
            field.Tag.Get("json"), // name age
            field.Tag.Get("db")) // user_name user_age
    }
}
```

Week 09

Prolog

```
:- encoding(utf8).
# Facts
parent(你媽, 你). # 你媽是你的父母之一
parent(你爸, 你). # 你爸是你的父母之一
female(你媽)

# Rules
mather(X, Y) :- parent(你媽, 你), female(你媽).
sister(X, Y) :- parent(你媽, X), parent(你媽, Y), X \= Y.

# Queries
? - parent(X, 你).
X = 你媽 ;
X = 你爸 ;

? - mather(你媽, 你).
true.
```

Prolog

```
# Facts
sum(1, 1).
# Rules sum(N, Total) :-
    N > 1,
    N1 is N - 1,
```



```
sum(N1, Temp),
Total is Temp + N.
```

```
# Queries
?- sum(3, Total).
Total = 6.
```

Prolog data structure

- atom 原子 -> tom, [], 'Hello' 最小單位 不可再分
- number 數字 -> 42, 3.13, -5
- variable 變數 -> X, Y, Z, _x
- structure 結構 -> parent(你媽, 你)

```
? - functor(parent(你媽, 你), F, Y).
F = parent,
Y = 2 ;
```

list 列表 -> [1, 2, 3], [a, b, c]

```
[apple, banana, orange] = [H | T].
# H = apple, T = [banana, orange] ;

list_length([], 0).
list_length([_|T], N) :-
    list_length(T, N1),
    N is N1 + 1.

list_length([1, 2, 3], N).
N = 3 ;
```

- 內建 predicate

```
member(X, L) # X在L裡面
append(L1, L2, L3) # L3 = L1 + L2
length(L, N) # L的長度是N
reverse(L, R) # L的反轉是R
```

Operator notation

```
# directives
# the precedence優先度 of 'is' is 500
# xfx is a infix operator 中序關係
:- op(500, xfx, 'is').
```