

$$(X, O) = e^{-\frac{x^2}{2\sigma^2}}$$

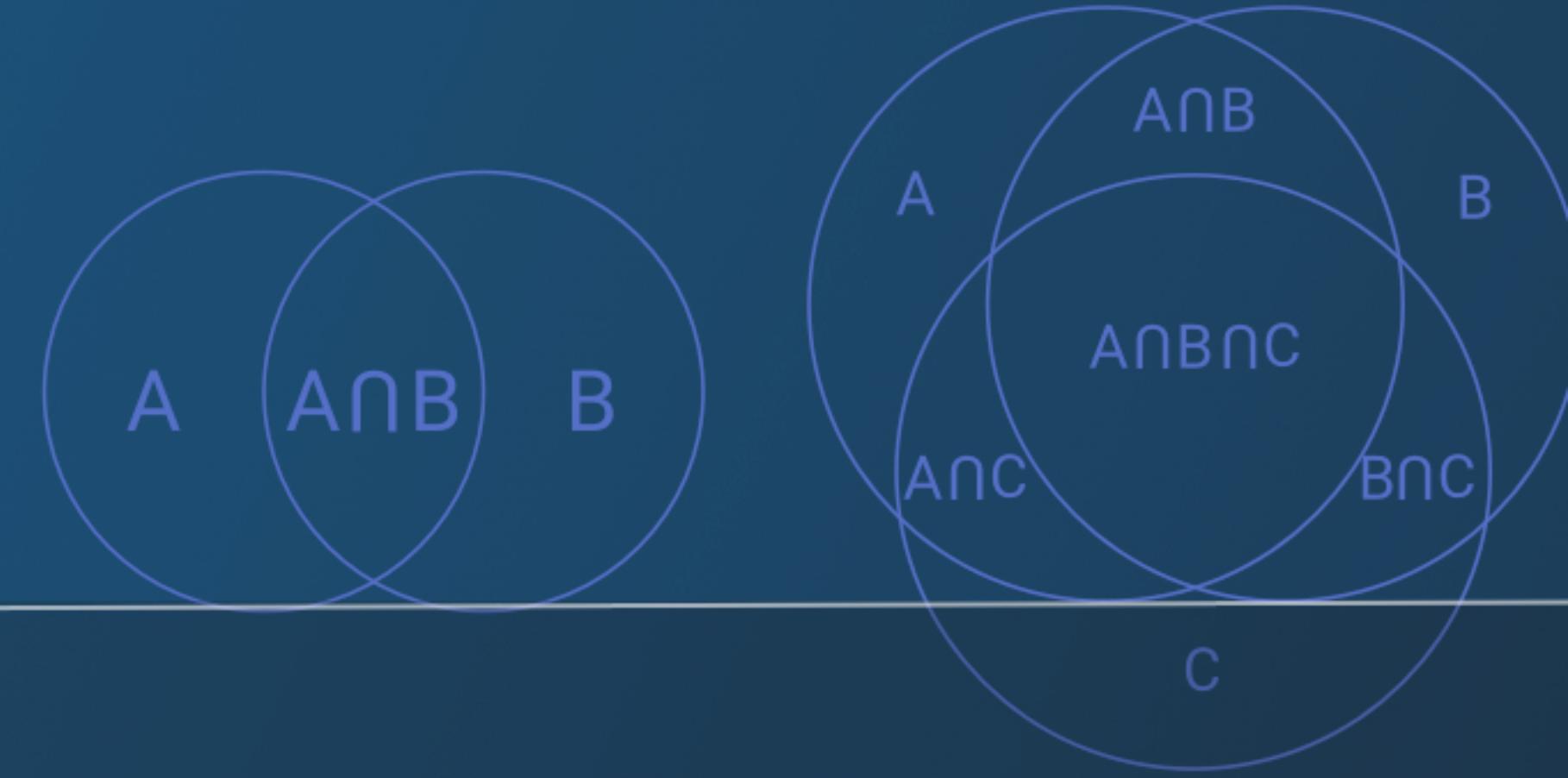
$$x(X, O) = -\frac{x}{\sigma^2} G(X, O) = -\frac{x}{\sigma^2} e^{-\frac{x^2}{2\sigma^2}}$$

$$xx(X, O) = \frac{x^2 - \sigma^2}{\sigma^4} G(X, O) = \frac{x^2 - \sigma^2}{\sigma^4} e^{-\frac{x^2}{2\sigma^2}}$$

$$xxx(X, O) = -\frac{x^3 - x\sigma^2}{\sigma^6} G(X, O) = -\frac{x^3 - x\sigma^2}{\sigma^6} e^{-\frac{x^2}{2\sigma^2}}$$

Julia 程式語言學習馬拉松

Day 15



$$\ln(x + \sqrt{1+x^2}) + x - \frac{1}{x + \sqrt{1+x^2}} \left(1 + \frac{x}{\sqrt{1+x^2}} \right)$$



cupay 陪跑專家 : James Huang

日誌與例外處理

(Logging & Exception Handling)





重要知識點



- Logging 日誌模組提供了在 Julia 程式中記錄事件過程與歷史的功能。
- Exception Handling 例外處理則是為了避免不可控制的情況發生時產生系統 Error，造成程式執行中斷，這時候我們就需要在適當的位置中，加入例外處理機制，讓我們可以捕捉到這些錯誤，並進行相對應的處理，以避免程式中斷而失去控制。



日誌 (Logging) 的建立



- 日誌的記錄是透過巨集加上要記錄的訊息字串組成，例如：
 `@debug "this is a debug message"`。
- 日誌的輸出方式可以透過設定 logger 來指定，logger 有下列幾種：
 - ConsoleLogger，預設值，輸出到螢幕。
 - SimpleLogger，輸出到純文字檔案。
 - NullLogger，不輸出。
- Logger 可以分為 `global_logger` 與 `task-local logger`。



日誌 (Logging) 的建立

- 日誌是否要顯示或記錄是由 Log level (等級) 來控制，主要的 Log level 等級依照由小到大如下所列：

Log Level	說明
Debug	用來記錄 debug 所需要的訊息，預設不顯示/記錄。
Info	正常運作的訊息。
Warn	警告訊息，記錄可能發生的情況。
Error	在可控制的情況下產生的錯誤訊息。不可控制的錯誤可能會產生 Exception，會在下面章節中介紹。



將日誌 (Logging) 寫入文字檔案



- 將日誌 (Logging) 寫入文字檔案的流程如下：
 - 開啟文字檔 IO，並照上兩頁的流程建立日誌。
 - 記錄日誌訊息。
 - 呼叫 flush() 函式，會將所有暫存的日誌訊息寫入到文字檔案中。
 - 最後記得要呼叫 close() 函式關閉檔案。





例外處理 (Exception Handling)



- 在程式的控制流程中，如果是在可控制的情況下，也就是說程式不會因為發生異常而停止執行，我們可以用日誌來記錄我們所定義的訊息，包含錯誤訊息，例如加入 @error、@warn 日誌訊息。
- 但是如果不可控制的情況發生，可能就會產生系統 Error，造成程式執行中斷，這時候我們就需要在適當的位置中，加入例外處理機制，讓我們可以捕捉到這些錯誤，並進行相對應的處理，以避免程式中斷而失去控制。



例外處理 (Exception Handling)



- Julia 內建的 Error 如下表，另外也可以自行定義 Error。

ArgumentError	InitError	MethodError
BoundsError	InterruptException	OverflowError
CompositeException	InvalidStateException	Meta.ParseError
DivideError	KeyError	SystemError
DomainError	LoadError	TypeError
EOFError	OutOfMemoryError	UndefRefError
ErrorException	ReadOnlyMemoryError	UndefVarError
InexactError	RemoteException	StringIndexError



try-catch 區塊



- 使用 try-catch 例外處理。例如：
 - 開啟檔案時可能產生出錯，我們將有可能需要對例外進行測試的表達式放在 try 區塊中。
 - 若發生錯誤時，由 catch 區塊來處理捕捉到的錯誤，避免程式因為錯誤而異常中斷。

```
1 try
2     f = open("thisfiledoesnotexist.txt")
3 except ex
4     println("檔案開啟錯誤，請檢查檔案是否存在。錯誤訊息：")
5     println("Exception: ", ex)
6 end
```

檔案開啟錯誤，請檢查檔案是否存在。錯誤訊息：

Exception: SystemError("opening file \\"thisfiledoesnotexist.txt\\\"", 2, nothing)



finally 區塊

- 使用 try-catch-finally 例外處理。例如：
 - try-catch 的例外處理同前頁所述。
 - finally 區塊是在處理過後，不論正常或有例外產生，一定會進入的區塊。可以將處理過後的訊息，或後續要執行的動作，在 finally 區塊中定義。

```
1  try
2      log(-10)
3  catch ex
4      println("捕捉到例外情況，請檢查")
5      println("Exception: ", ex, "\n")
6  finally
7      println("This is finally.")
8 end
```



throw() 函式

- 在控制流程當中，如果要很明確地產生例外情況，可以使用 `throw()` 函式。下面範例是，在 `square root` 時，若輸入負數的話，拋出 `DomainError`。

```
1 | foo(x) = x>=0 ? sqrt(x) : throw(DomainError(x, "必須為正數"))
```

```
foo (generic function with 1 method)
```

```
1 | foo(-4)
```

```
DomainError with -4:  
必須為正數
```

Stacktrace:

```
[1] foo(::Int64) at ./In[72]:1  
[2] top-level scope at In[73]:1
```



stacktrace() 與 backtrace()



- 如果想要了解程式執行該時間點的執行堆疊，可以呼叫 stacktrace() 函式，尤其是在處理例外情況發生時，例如可以依此堆疊追蹤錯誤發生時的程式、函式呼叫流程與順序，並判斷錯錯可能發生的原因。
- backtrace() 函式也是回傳堆疊追蹤的資訊，與 stacktrace() 不同的地方在於，backtrace() 會包含低階呼叫 C 語言的訊息。

知識點 回顧

- 在今天的內容中介紹了日誌、例外處理，以及如果查看堆疊資訊，讓我們在開發程式時，對於事件進行記錄，並在可控制的範圍下依據訊息進行錯誤的追蹤，避免因為不可控制的情況發生而造成程式異常中斷無法完成運作。
- 日誌的架構與使用方法，也跟其他的程式語言非常類似，只要能掌握其架構，相信對於使用日誌就能得心應手。
- 除了內建的 Error，在 Julia 內也可以自訂 Error，以符合客製化的需求。



推薦閱讀

- [Logging 官方文件](#)
- [Exception Handling 官方文件](#)





解題時間

請跳出 PDF 至官網 Sample Code
& 作業開始解題