Goal: use playground to write a small script

Swift Playground Introduction

- review of basic layout of XCode
- how to open playground

Basic Swift Syntax (following the [swift programming guide](https://developer.apple.com/swift/))

OOP

- variable and constants
- print statement
- formatted string
- array and dictionary
  - create literal
  - type notation
    - [<type>]
    - [<key_type>:<value_type>]
  - type inference
    - [:]/[]
- control flow
  - if else elseif
  - for ... in ...
  - while {} / do {} while
  - range 0...<4, 0...5
  - for loop

- Class
  - constructor of the class: init((<param_name: param_type>))
Finalizer: `deinit()`

- override keyword is required

- Getter and Setter can be defined while the variable is defined: var
  
  `<var_name>` : `<var_type>` { get { `<getter body>` } set { `<setter body using` `newValue as the name for the input>` `willSet { `<method for synchronization` `using newValue as the name for the input>` `didSet { `<method for` `synchronization using new value as the name for input>` } } }

### Protocols

- classes/enum/structs

- protocol `<Name>` { `<var declaration>` | `<fun declaration>` }

  - In the declaration: mutating shows that the method can mutate the struct

- extension `<Old class>` : `<Protocol>` { `<implementation of the protocol>` }

  - The protocol type will have only the protocol method available

### Generics

- same syntax as java `< (T :<Protocol>)?* >`

- can be used in enum as well as class

- where keyword: `<T, U,... where T:<Protocol>, U:<Protocol>,...>`

### Functional Programming

- Function

  - definition

    - signature: `<decoration> fund <name>(<inner_para>:<type>[,...]) (-> <return_type>) {body}`

    - Local variable, parameters, return values

    - Closure/Function:
      - definition:
      - type definition (- >)
      - `{ (<param_name>:<param_type>[,...]) -> <return_type> in`
Given that the type is already known: { <var_name> in <return expression>} }

If really really short, can use number to refer as the variable ($i$ for the $i^{th}$ variable)

- Block/function as input and output
  - return (1,2,'2','String')
  - func <name>(<[param>]) - ( [<ret_name>] : [<ret_type>]) { <body goes here> }
  - list of parameters as input: func <fun_name>(<param_nem> : <param_type> … ) (ret_type)* { <body> }
  - nested function (function is a value)
  - return a function : func <fun_name>( (<param_name> : <param_time>)*) - ( <OCAML TYPE DEFINITION> ) { <body> }
  - function as input: (just need to change the input type)

- Tuple:
  - define tuples
  - named tuples

- Option Types
  - if let
  - switch
    - .<name>
    - case let x where x.hasSuffix("paper") : example will be on page 10

- Enum
  - define: enum <Capitalized first letter name> : <enum basic type> { (case <Capitalized case name>)+ ( func <funcName>()-<return_type> {} )* }*
  - can use init?(rawValue:) as the initializer
  - each enum function can use self to refer to the self value
  - Like the OCaml Variant, enum case can provide associate value: enum <Cap.Name> { case <CaseName>( (<assoc.val.type>)* ) }