Programación Funcional

INFO175

Cristian Rojas Pérez
crrojasperez@gmail.com

¿Qué es la programación funcional?

- “Estilo de programación que enfatiza la evaluación de expresiones, antes que la ejecución de comandos”
- Un programa funcional está compuesto enteramente por funciones.
  - A simple vista un programa en lenguaje C se ajustaría a la definición de programa funcional, pero no es así, ya que la programación convencional (Imperativa) posee varios aspectos no incluidos en la programación funcional pura.
- En un lenguaje funcional los únicos elementos contructores son la definición y aplicación de funciones. No hay variables, asignaciones, ciclos, etc.
- Puede parecer restrictivo el no poder utilizar variables, asignaciones ni ciclos iterativos. Pero se ha demostrado matemáticamente que la definición y aplicaicón de funciones es suficiente para construir cualquier función computable .

¿Qué es la programación funcional?

- Cualquiera que haya programado una hoja de cálculo conoce la experiencia de la programación funcional.
- En una hoja de cálculo, se especifica cada celda en términos de los valores de otras celdas. **El objetivo es que debe ser calculado y no en como debe calcularse.**
- Por ejemplo:
  - No especificamos el orden en el que las celdas serán calculadas, en cambio obtenemos el orden que garantiza que la hoja de cálculo puede calcular las celdas respetando las dependencias.
  - No indicamos a la hoja de cálculo como manejar la memoria, en cambio esperamos que nos presente un plano de celdas, aparentemente infinito, pero que solo utiliza la memoria de las celdas que están actualmente en uso.
  - Lo más importante, especificamos el valor de una celda por una expresión (cuyas partes pueden ser evaluadas en cualquier orden), en vez de una secuencia de comandos que calculan los valores.

Historia

- La programación funcional apareció como un paradigma independiente a principio de los sesenta.
- Su creación es debida a las necesidades de los investigadores en el campo de la inteligencia artificial y en sus campos secundarios del cálculo simbólico, prueba de teoremas, sistemas basados en reglas y procesamiento del lenguaje natural.
- Estas necesidades no estaban cubiertas por los lenguajes imperativos de le época.

Efecto secundario

- Se dice que una función o expresión tiene efecto colateral o efecto secundario si esta, además de retornar un valor, modifica el estado de su entorno. Por ejemplo, una función puede modificar una variable global o estática, modificar uno de sus argumentos, escribir datos a la pantalla o a un archivo, o leer datos de otras funciones que tienen efecto secundario.
- Los efectos secundarios frecuentemente hacen que el comportamiento de un programa sea más difícil de predecir.
- En programas muy grandes es difícil conocer el estado completo del sistema en un momento específico de ejecución.
  - Hay herramientas de depuración de código (debug) que nos permiten concer el estado de cada variable
- La programación imperativa y POO generan efectos secundarios.

Características

- La característica principal de la programación funcional es que los cálculos se ven como una función matemática que hacen corresponder entradas y salidas.
- No hay noción de posición de memoria y por tanto, necesidad de una instrucción de asignación.
- Los bucles se modelan a través de la recursividad ya que no hay manera de incrementar o disminuir el valor de una variable.
- Como aspecto práctico casi todos los lenguajes funcionales soportan el concepto de variable, asignación y bucle. Estos elementos no forman parte del modelo funcional “puro”.
- Basado en el cálculo lambda.
  - Revisar: http://es.wikipedia.org/wiki/C%C3%A1lculo_lambda

Lenguajes

- Lenguajes funcionales
  - CALM
  - SCHEME
  - HASKELL
  - LISP
  - ERLANG
  - …
- Lenguajes con soporte de algunas características del lenguaje funcional
  - Python
  - Ruby
  - Java (En su última versión)

Ejemplo

- “Construir una función ***f*** que reciba como argumento un natural ***n*** y retorne la suma de los naturales desde 1 hasta ***n***, es decir:”

$$f(n)=\sum_{i=1}^{n} i$$

- En un lenguaje como C haríamos lo siguiente:

```
int f(int n)
 {
   int i;
   int suma=0;
   for ( i=1; i<=n ; i++ )
     suma=suma+ i;
   return suma;
 }
```

Ejemplo

- La implementación de esta función en un lenguaje funcional exigiría otra estrategia ya que no tenemos variables, asignaciones o ciclos. Entonces, se utilizará una definición recursiva equivalente.

$$f(n) = \begin{cases} 0 & \text{si } n = 0 \\ f(n-1) + n & \text{en otro caso} \end{cases}$$

- Solución en CAML

```
let rec f= fun  0-> 0
              | n-> f (n -1)+ n;;
```

- Solución en SCHEME

```
(define f (lambda (n)(if (n=0)0 (+(f(-n 1))n))))
```

- Solución en HASKELL

```
f n= sum [1..n]
```

Programación funcional en Python

Python no es un lenguaje enteramente funcional pero da soporte a varias características de este paradigma

Funciones de orden superior

- En python las funciones son objetos, por lo que pueden tener atributos y pueden ser referenciadas y asignadas en variables.
- La definición de función de orden superior se refiere al uso de las funciones con si de un valor cualquiera se tratara.
  - Se pueden pasar funciones como parámetros de otras funciones
  - Se pueden devolver funciones como valor de retorno

```
# Función como argumento de otra función
def evaluar(f, *args):
    return f(*args)


def suma(x, y):
    return x + y


print(evaluar(suma, 3, 4))
```

```
# Las funciones son objetos
def funcion_original(valor):
    return valor


una_variable_con_la_funcion = funcion_original
print(una_variable_con_la_funcion(5))


def saludar(lang):
    def saludar_es():
        return "Hola"

    def saludar_en():
        return "Hi"

    def saludar_fr():
        return "Salut"

    lang_func = {"es": saludar_es,
                 "en": saludar_en,
                 "fr": saludar_fr}
    return lang_func[lang]

#Esto retorna una función!!
print(saludar("es"))
#Se puede pasar la función retornada
#a una variable e invocarla
f = saludar("es")
print(f())
#Para saltarse la asignación.
print(saludar("fr")())
```

Funciones anónimas

- Las funciones anónimas no poseen nombre y no pueden ser referenciadas posteriormente.
- En Python se pueden crear funciones anónimas en una línea con la sentencia ***lambda***.
- Comúnmente utilizadas cuando se pasa una función simple como argumento de otra función.

```
# Función como argumento de otra función
def evaluar(f, *args):
    return f(*args)


def suma(x, y):
    return x + y


print(evaluar(suma, 3, 4))
```

```
# función anónima
print(evaluar(lambda x, y: x + y, 3, 4))
```

Iteraciones de orden superior sobre listas

- **map**(*funcion*, *iterable*)
  - Aplica ***funcion*** a cada elemento de ***iterable*** y retorna una lista con los resultados

```
# Uso de MAP
def cuadrado(n):
    return n ** 2

l = [1, 2, 3]
l2 = map(cuadrado, l)
print(l2)
```

- **filter**(function, iterable)
  - Construye una lista con aquellos elementos de *iterable* para los cuales ***function*** retorna True**.**

```
# Uso de FILTER
def es_par(n):
    return (n % 2.0 == 0)

l = [1, 2, 3]
l2 = filter(es_par, l)
print(l2)
```

- **reduce**(function, iterable)
  - Aplica ***function*** a pares de elementos de ***iterable*** hasta dejarlo en un solo valor

```
# Uso de REDUCE
def sumar(x, y):
    return x + y

l = [1, 2, 3]
l2 = reduce(sumar, l)
print(l2)
```

Ejercicio

- Reemplaze las funciones cuadrado, es_par y sumar por lambdas para pasarlas directamente como parámetro a las funciones map, filter y reduce.

```python
# Uso de MAP
def cuadrado(n):
    return n ** 2

l = [1, 2, 3]
l2 = map(cuadrado, l)
print(l2)


# Uso de FILTER
def es_par(n):
    return (n % 2.0 == 0)

l = [1, 2, 3]
l2 = filter(es_par, l)
print(l2)


# Uso de REDUCE
def sumar(x, y):
    return x + y

l = [1, 2, 3]
l2 = reduce(sumar, l)
print(l2)
```

Comprensión de listas

- Este concepto permite crear listas en una forma fácil y natural, parecido a definiciones matemáticas.

```
S = {x² : x in {0 ... 9}}
V = (1, 2, 4, 8, ..., 2¹²)
M = {x | x in S and x even}
```

```
S = [x**2 for x in range(10)]
V = [2**i for i in range(13)]
M = [x for x in S if x % 2 == 0]
```

- ¿Cómo generar una lista con todos los dígitos dentro de la cadena de texto “hello 12345 world”? Tip. x.isdigit()

```
numbers = [x for x in "Hello 12345 World" if x.isdigit()]
print numbers
```

Comprensión de listas

- “Si buscas resultados distintos no hagas siempre lo mismo”
- Generar una lista de listas que contenga todas las palabras de la frase anterior en mayúsculas (upper), minúsculas (lower) y el largo de la palabra.
- Ej: [[“SI”, “si”, 2], [“BUSCAS”, “buscas”, 6]...]

```
stuff = [[w.upper(), w.lower(), len(w)] for w in \
    'Si buscas resultados distintos no hagas siempre lo mismo'.split()]
```

Generadores

- Suponga que posee una máquina con muy poca memoria (100 bytes) y que cada número cuesta 10 bytes de almacenamiento en memoria.
- Crear una lista de más de 10 números consumiría todos los recursos.
  - mi_lista = [i for i in range(10)]
- Para solucionar este inconveniente existe el concepto de generador, el cual permite

```
1 # Build and return a list
2 def firstn(n):
3     num, nums = 0, []
4     while num < n:
5         nums.append(num)
6         num += 1
7     return nums
8
9 sum_of_first_n = sum(firstn(1000000))
```

```
1 # a generator that yields items instead of returning a list
2 def firstn(n):
3     num = 0
4     while num < n:
5         yield num
6         num += 1
7
8 sum_of_first_n = sum(firstn(1000000))
```

La palabra reservada yield se diferencia de return en que cede momentáneamente el control del programa pero luego la función “firstn” lo retomará. Es una especie de streaming de datos.

Generadores

- En python 2.x la función range es una lista y la función xrange es un generador

.

```
# Note: Python 2.x only
# using a non-generator
sum_of_first_n = sum(range(1000000))

# using a generator
sum_of_first_n = sum(xrange(1000000))
```

- La comprensión de lista puede ser transformada a generadores simplemente cambiando los corchetes [ ] por paréntesis ( )

```
# list comprehension
doubles = [2 * n for n in range(50)]

# same as the list comprehension above
doubles = list(2 * n for n in range(50))
```

Decoradores

Investigar sobre decoradores en Python!!!!!

Recursividad

```
def exp(x, n):
    """
    Computes the result of x raised to the power of n.

        >>> exp(2, 3)
        8
        >>> exp(3, 2)
        9
    """
    if n == 0:
        return 1
    else:
        return x * exp(x, n-1)
```

```
exp(2, 4)
+-- 2 * exp(2, 3)
|       +-- 2 * exp(2, 2)
|       |       +-- 2 * exp(2, 1)
|       |       |       +-- 2 * exp(2, 0)
|       |       |       |       +-- 1
|       |       |       +-- 2 * 1
|       |       |       +-- 2
|       |       +-- 2 * 2
|       |       +-- 4
|       +-- 2 * 4
|       +-- 8
+-- 2 * 8
+-- 16
```

Ejercicio

- Crear una función recursiva que calcule el n-ésimo elemento de la suseción de fibonacci

$$F_n = \begin{cases} 0 & \text{if } n = 0; \\ 1 & \text{if } n = 1; \\ F_{n-1} + F_{n-2} & \text{if } n > 1. \end{cases}$$

```
def F(n):
    if n == 0: return 0
    elif n == 1: return 1
    else: return F(n-1) + F(n-2)
```

¿Cómo es el rendimiento al calcular F(30) y F(100)?

¿Cómo se puede hacer de forma eficiente?