Lezione 04: Programmazione funzionale, Corto Circuitazione, Monadi

Programmazione funzionale

Si rimanda agli appunti di Linguaggi di Programmazione: Lezione 01 di OCaml

In breve:

- funzioni come first class citizen
- ricorsione come control flow principale
- focus principale su manipolazione di liste
- niente side effects, immutabilità

Funzioni anonime: `lambda`

Le funzioni anonime sono definite utilizzando `lambda <args>: <expr>`.

```
lambda x: x + 1 # incrementa di 1
lambda: 5 # restituisce sempre 5
```

Assegnare una lambda ad una variabile è esattamente come una funzione normale.

```
def inc1(x):
    return x + 1

inc2 = lambda x: x + 1

print(inc1) # <function inc1 at 0x00007f2daab0ee80>
print(inc2) # <function <lambda> at 0x00007f2daab0ef20>

inc2.__qualname__ = 'inc2'

print(inc1) # <function inc1 at 0x00007f2daab0ee80>
print(inc2) # <function inc2 at 0x00007f2daab0ef20>
```

Manipolazione di liste

Utilizzo massiccio di `map`, `filter` (che restituiscono un *iteratore*) e `reduce` (`functools.reduce`, che restituisce una *oggetto*, del tipo restituito dalla funzione *accumulatrice* passata).

```
import functools
numbers = [1, 2, 3, 4, 5]

list(map(lambda x: x**2, numbers)) # [1, 4, 9, 16, 25]
list(filter(lambda x: x % 2 == 0, numbers)) # [2, 4]

# senza valore di partenza:
functools.reduce(lambda acc, x: acc + x, numbers) # 15
# con valore di partenza:
functools.reduce(lambda acc, x: acc + x, numbers, 10) # 25
```

Gli stessi risultati si possono ottenere attraverso le comprehensions, che sono anche considerate più *idiomatiche* in Python.

Control flow: corto circuitazione al posto di `if`

È possibile sfruttare la **corto circuitazione** per evitare di utilizzare `if` e `else`. Quando una parte di espressione non ha bisogno di essere valutata, viene restituita **così com'è** (senza essere appunto valutata).

```
(True and "secondo operando") # "secondo operando"
(False and "secondo operando") # False
```

```
(True or "secondo operando") # True
(False or "secondo operando") # "secondo operando"
```

Concatenando le espressioni, si può ottenere un comportamento simile a `if` e `else`.

```
def cond(x):
    if x == 1: return "uno"
    elif x == 2: return "due"
    else: return "altro"

# si può trasformare in:
def cond(x):
    return (x == 1 and "uno") or (x == 2 and "due") or "altro"

cond(1) # "uno"
cond(2) # "due"
cond(3) # "altro"
```

In caso sia necessario effettuare delle operazioni che non si possono scrivere "inline", si può astrarre in una funzione ausiliaria:

```
def f(x):
    a, b = x[0], x[1]
    return b + a

def cond(x):
    return (x == 1 and f("uno")) or (x == 2 and f("due")) or f("altro")

cond(1) # "nu"
cond(2) # "ud"
cond(3) # "la"
```

Control flow: sequenza

Al posto di eseguire una serie di operazioni (chiamate di funzione) in sequenza, si possono eseguire applicandole da una lista. Questo è vantaggioso ad esempio per eseguire una serie di operazioni in ordine diverso in base a come è stata costruita la lista.

```
def a(): pass
def b(): pass
def c(): pass

# imperativo:
a()
b()
c()

# funzionale:
actions = [a, b, c] # si può costruire anche in modo dinamico

[a() for a in actions]
# oppure
map(lambda f: f(), actions)
```

Control flow: ricorsione al posto di iterazione

Al posto di iterare utilizzando cicli `for` o `while` si utilizza la ricorsione. Ad esempio, per trasformare la funzione `echo`, definita:

```
def echo():
    while True:
        inp = input(">> ")
```

```
        if inp == "exit": break
        print(inp)
```

Andiamo a sfruttare una *print identitaria*, che oltre a stampare **restituisce** anche il valore stampato, la **ricorsione** e la **corto circuitazione**:

```
def monadic_print(x):
    print(x)
    return x

def echo():
    return monadic_print(input(">> ")) == "quit" or echo()
```

Fino a quando la prima espressione (`monadic_print(...) == "quit"`) rimane falsa, allora il risultato dell'or dipende dalla seconda espressione (`echo()`), che viene quindi valutata ricorsivamente.
Quando l'input diventa `"quit"`, la prima espressione diventa vera e il risultato dell'or è vero, che quindi non ha più bisogno di valutare la seconda espressione, interrompendo la ricorsione.

Monadi (Monads)

Una **monade** è un pattern che permette di gestire operazioni che hanno *effetti collaterali* in modo *funzionale* e *componibile*.

Si può considerare la funzione `monadic_print` vista sopra una monade, in quanto restituisce l'identità con un side-effect (la print).

Formalmente, una monade è un *type constructor* con due operazioni principali:

- `return`: prende un valore e lo incapsula in un contesto *monadico*
- `bind`: prende un valore incapsulato e una funzione, *applica* la funzione e *restituisce* il risultato *incapsulato*

In particolare, l'operazione `bind` è quella che offre:

- *incapsulamento*: racchiude un valore in un "contenitore" con contesto
- *composizione*: permette di **concatenare** operazioni senza gestire manualmente il contesto
- *controllo*: gestisce **automaticamente** aspetti come errori, stato, I/O

Un esempio classico di monade è la *Maybe monad*, che nella maggior parte dei linguaggi è implementata come il tipo `Optional` o `Maybe`:

```
class Maybe:
    def __init__(self, value):
        self.value = value

    def bind(self, func):
        if self.value is None:
            return Maybe(None)
        return func(self.value)

def safe_divide(x, y):
    return Maybe(x / y if y != 0 else None)

def safe_sqrt(x):
    return Maybe(x ** 0.5 if x >= 0 else None)

result = (Maybe(32)
            .bind(lambda x: safe_divide(x, 2))  # 32/2 = 16
            .bind(lambda x: safe_sqrt(x)))      # sqrt(16) = 4
print(result.value)  # 4

result = (Maybe(32)
            .bind(lambda x: safe_divide(x, 0))  # 32/0 = None
```

```
            .bind(lambda x: safe_sqrt(x)))      # Non viene eseguito
print(result.value)  # None
```

Se una qualsiasi operazione fallisce, il risultato finale sarà `None` senza propagare errori.