Programming Tutorial

08 Lambda and Functional Programming

Lambda in Python

Basice Functions adn Operator:

- Function：
  - range(): Enumeration
  - map(): Map
  - reduce(): Reduce
  - filter(): Filter
- Operator：
  - lambda: Function is First-Class.

```
#range(start, stop[, step])
range(1,100)
range(1,100,)
```

```
#map(func, iterable)
double_func = lambda s : s * 2
list1 =map(double_func, [1,2,3,4,5])
```

```
from functools import reduce
#reduce(func, iterable[, initializer])

plus = lambda x, y : x + y
sum1 = reduce(plus, [1,2,3,4,5])
sum2 = reduce(plus, [1,2,3,4,5], 10)
```

```
#filter(func, iterable)

mode2 = lambda x : x % 2
list1 = filter(mode2, [1,2,3,4,5,6,7,8,9,10])
```

```
#lambda
func1 = lambda : print('hello')
func3 = lambda x,y : print(str(x+y))

func1()
func3(2,3)
```

Fuction Chain in Python

There is a interesting libary to enable Python to be functionally chain-styled.

https://github.com/Riparo/fc-python

> $ pip install fc

```
from fc import Fc

l = (
  Fc([1, 2, 3, 4, 5])
    .map(lambda x: x + 1)
    .filter(lambda x: x > 4)
    .print()  # [5, 6]
    .map(lambda x: x + 1)
    .done()
)
assert l == [6, 7]
```

Sequence, Branch, and Loop in Functional Language

Functional language can utilize **funcation-chain expression**, **Equivalent "short circuit" expression**, **Recursion** to implements any imperative programming statements.

Sequence

> ** sequence statement**
>
> func1()
>
> func2()
>
> func3()

> ** function-chain sytle expression**
>
> func3(func2(func1()))

```
# sequence statement
x = 1
x = x*2
print(x)

#FP-style expression
func1 = lambda : 1
func2 = lambda x : x*2
print(func2(func1()))
```

Branch

> **Flow control statement** if : func1() elif : func2() else: func3()

> **Equivalent "short circuit" expression** (and func1()) or (and func2()) or (func3())

```
pr = lambda s:s

print_num = lambda x: (x==1 and pr("one")) \
    or (x==2 and pr("two")) \
    or (pr("other"))

print(print_num(1))
print(print_num(2))
print(print_num(3))
```

Loop

> **Statement-based for loop** for e in lst: func(e)

> **Equivalent map()-based loop** map(func, lst)

```
square = lambda x : x * x
for x in [1,2,3,4,5]:
    square(x)
list1 = map(square, [1,2,3,4,5])
print([a for a in list1])
```

> **Statement-based while loop** while : if <break_condition>: break else:

> **Equivalent FP-style recursive while loop** def while_block(): if <break_condition>: return 1 else: return 0
>
> while_FP = lambda: and (while_block() or while_FP()) while_FP()

```
# imperative version of "echo()"
def echo_IMP():
    while 1:
        x = raw_input("IMP -- ")
```

```python
        print x
        if x == 'quit':
            break

echo_IMP()

def monadic_print(x):
    print x
    return x

# FP version of "echo()"
echo_FP = lambda: monadic_print(raw_input("FP -- "))=='quit' or echo_FP()
echo_FP()
```