BY: Zhu_Ming

<CS1010S>

Tutorial 4

Higher-Order Function

Lecture Recap

1. Higher-Order Function
2. Abstract Data Type (ADT) and Data Type

1. Higher-Order Function

- Return function references as variables
- Create function from another function

```python
def compose(f, g):
    return lambda x: f(g(x))

def thrice(f):
    return compose(compose(f, f), f)

thrice(thrice)(lambda x: 1 + x)(3) # 30
thrice(thrice(lambda x: 1 + x))(3) # 12
```

Code Tracing tips

From inner to outer
From left to right

Simplify the code
Identify the scope of variable
Rename the variable

1. Higher-Order Function

```
def fold_1(op, f, n):
    if n == 0:
        return f(0)
    else:
        return op(fold_1(op, f, n-1), f(n))

def fold_2(op, f, n):
    if n == 0:
        return f(0)
    else:
        return op(f(n), fold_2(op, f, n-1)
```

ARE THEY THE SAME??

```
fold_1(lambda x, y: x+y, lambda x: x**2, 4)
fold_2(lambda x, y: x+y, lambda x: x**2, 4)


fold_1(lambda x, y: x-y, lambda x: x**2, 4) #-30
fold_2(lambda x, y: x-y, lambda x: x**2, 4) # 10
```

Important

Code trace to identify the operation behavior!!

Especially with NOT associative/commutative operator

2. Functional Abstraction

- Use **previously defined function** to achieve **lower-level** tasks (**Don't Repeat Yourself** (DRY) principal)
- Once Functional Abstraction is established, **EVERYONE** should follow the abstraction barrier

"Let the function do its job"

Lecture_Recap.tut04

3. Data Abstraction & Data Encapsulation

- Internal Implementation of Abstract Data Type is hidden from user
- **Tell, Don’t Ask Principle**
- Use **Constructor**, **Getter**, **Setters** to maintain the abstraction barrier

YOUR CLIENT

```
point_1 = make_point(1, 0)
get_x(point_1) # 1
get_y(point_2) # 0
```

YOUR IMPLEMENTATION_1

```
make_point = (lambda x, y: (x, y))
get_x = (lambda point: point[0])
get_y = (lambda point: point[1])
```

YOUR IMPLEMENTATION_2

```
make_point = (lambda x, y: “x” + “y”)
get_x = (lambda point: int(point[0]))
get_y = (lambda point: int(point[1]))
```

Important!!!

Don’t break the abstraction barrier
Don’t access/manipulate them directly

```
p = make_point(1, 0)
x = p[0] # wrong
x = get_x(p) # correct
```

BY: Zhu_Ming

Any Questions?

Question_1.tut04

Define a function that takes as argument f, a, b, and n and returns the value of the integral, computed using the Composite Simpson's Rule

Iterative Version

```
def calc_integral(f, a, b, n):
    h = (b - a) / n
    total = 0
    for i in range(n + 1):
        term = f(a + i * h)
        if i == 0 or i == n:
            total += term
        elif i % 2 == 0:
            total += 2 * term
        else:
            total += 4 * term
    return (total * h) / 3.0
```

// $h = \frac{b-a}{n}$

// $y_k = f(a + kh)$

$\frac{h}{3}[y_0 + 4y_1 + 2y_2 + 4y_3 + 2y_4 + \cdots + y_n]$

Question_1.tut04

Define a function that takes as argument f, a, b, and n and returns the value of the integral, computed using the Composite Simpson's Rule

HOF Version

```python
def calc_integral(f, a, b, n):
    h = (b - a) / n
    total = 0
    def term(i):
        yi = f(a + i * h)
        if i == 0 or i == n:
            return yi
        elif i % 2 == 0:
            return 2 * yi
        else:
            return 4 * yi
    add1 = lambda x: x + 1
    return sum(term, 0, add1, n) * h / 3
```

// $h = \frac{b-a}{n}$

// $y_k = f(a + kh)$

$\frac{h}{3}[y_0 + 4y_1 + 2y_2 + 4y_3 + 2y_4 + \cdots + y_n]$

Question_2.tut04

$$g(k) = \prod_{x=0}^{k} (x - (x+1)^2)$$

Write a function g(k) that solves the following product using the higher order function **fold**.

```python
def fold(op, f, n):
    if n == 0:
        return f(0)
    else:
        return op(f(n), fold(op, f, n-1))



def g(k):
    return fold(lambda x, y: x * y, lambda x: x - (x + 1) ** 2, k)
```

Question_3.tut04

$$a_1 = a, a_n \leq b$$

$$accumulate(\oplus, base, f, a, next, b): (f(a_1) \oplus f(a_2) \oplus (\ldots \oplus (f(a_n) \oplus base) \ldots)))$$

Write the **accumulate** function and show how **sum** can be defined as a simple call to **accumulate**

```python
def accumulate(combiner, base, term, a, next, b):
    if a > b:
        return base
    else:
        return combiner(term(a), accumulate(combiner, base, term, next(a), next, b))
```

Question_4.tut04

$$a_1 = a,\ a_n \leq b$$
$$accumulate(\oplus, base, f, a, next, b): (f(a_1) \oplus f(a_2) \oplus (\ldots \oplus (f(a_n) \oplus base) \ldots)))$$

Write the **accumulate** function and show how **sum** can be defined as a simple call to **accumulate**

```
def accumulate(combiner, base, term, a, next, b):
    if a > b:
        return base
    else:
        return combiner(term(a), accumulate(combiner, base, term, next(a), next, b))


def sum(term, a, next, b):
    return accumulate(lambda x, y: x + y, 0, term, a, next, b)
```

$\underbrace{\texttt{lambda x, y: x + y}}_{\texttt{combiner}}$

Question_5.tut04

$$a_1 = a,\, a_n \le b$$
$$accumulate(\oplus, base, f, a, next, b): (f(a_1) \oplus f(a_2) \oplus (\ldots \oplus (f(a_n) \oplus base) \ldots)))$$

Write the **accumulate** function and show how **sum** can be defined as a simple call to **accumulate**

```
def accumulate_iter(combiner, base, term, a, next, b):
    terms = ()                                  # initialization the terms
    while a <= b:
        terms = (term(a),) + terms              # collect all the terms first
        a = next(a)
    result = base
    for term in terms:
        result = combiner(term, result)         # combine all the terms
    return result
```

Question_6.tut04

Point ADT: make_point, x_point, y_point

Constructor

```
def make_point(x, y):
    return (x, y)
```

Getter

```
def x_point(point):
    return point[0]

def y_point(point):
    return point[1]
```

Constructor

```
def make_point(x, y):
    def point(axis):
        if axis == 0:
            return x
        else:
            return y
    return point
```

Getter

```
def x_point(point):
    return point(0)

def y_point(point):
    return point(1)
```

What is the difference???
Which one is better?? Why?

Question_7.tut04

Line Segment ADT: make_segment, start_segment, end_segment

Constructor

```
def make_segment(start_point, end_point):
    def segment(s):
        if s == 0:
            return start_point
        else:
            return end_point
    return segment
```

Getter

```
def start_point(segment):
    return segment(0)

def end_point(segment):
    return segment(1)
```

Question_8.tut04

Define a function **midpoint_segment** that takes a line segment as argument and returns its **midpoint.**

```
def midpoint_segment(segment):
    start_point = segment(0)
    end_point = segment(1)
    mid_x = 0.5 * (start_point(0) +end_point(0))
    mid_y = 0.5 * (start_point(1) +end_point(1))
    return (mid_x, mid_y)
```

```
def midpoint_segment(segment):
    start_point = start_segment(segment)
    end_point = end_segment(segment)
    mid_x = 0.5 * (x_point(start_point) + x_point(end_point))
    mid_y = 0.5 * (y_point(start_point) + y_point(end_point))
    return make_point(mid_x, mid_y)
```

Question_9.tut04

Rectangle ADT

What Attributes would you choose to define a rectangle??

What Methods (functions) does my rectangle need to be able to do??

Rectangle Attributes	**Rectangle Methods**
Length	area
Width (What is the data types?)	perimeter

Question_9.tut04

Rectangle ADT

Constructor

```
def make_rectangle(length_segment, width_segment):
    def segment(s):
        if s == 0:
            return length_segment
        else:
            return width_segment
    return segment
```

Getter

```
def get_rect_length_seg(rect):
    return rect(0)

def get_rect_width_seg(rect):
    return rect(1)
```

Question_9.tut04

Rectangle ADT

Methods

```
def segment_length(segment):
    start_p = start_segment(segment)
    end_p = end_segment(segment)

    dx = (x_point(start_p) - x_point(end_p))
    dy = (y_point(start_p) - y_point(end_p))
    return math.sqrt(dx ** 2 + dy ** 2)


def rect_length(rect):
    return segment_length(get_rect_length_seg(rect))

def rect_width(rect):
    return segment_length(get_rect_width_seg(rect))

def perimeter(rect):
    return 2 * rect_length(rect) + 2 * rect_width(rect)
```

Question_9.tut04

Rectangle ADT

Alternative implementations??

Use ADT **Point**!!

Can I still use the **perimeter** and **area** in the same way??

What should I modify??

Question_9.tut04

Rectangle ADT

Constructor

```python
def make_rectangle(length_segment, width_segment):
    def segment(s):
        if s == 0:
            return length_segment
        else:
            return width_segment
    return segment
```

Getter

```python
def get_rect_length_seg(rect):
    return rect(0)

def get_rect_width_seg(rect):
    return rect(1)
```

Constructor

```python
def make_rectangle(p0, p1, p2):
    def point(p):
        if p == 0:
            return p0
        elif p == 1:
            return p1
        else:
            return p2
    return point
```

Getter

```python
def get_rect_length_seg(rect):
    return make_segment(rect(0), rect(1))

def get_rect_width_seg(rect):
    return make_segment(rect(1), rect(2))
```

Question_9.tut04

Rectangle ADT

If there is changes in the implementation of **rectangle,**
There will be **some** changes in the rectangle method

But for our case we just need to change the **CONSTRUCTOR & GETTER**
, just the low-level method

Because our implementation have **GOOD DATA & FUNCTIONAL ABSTRACTION**

Question_9.tut04

BAD Rectangle ADT

Methods

```
def segment_length(segment):
    start_p = segment[0]
    end_p = segment[1]

    dx = (start_p[0] - x_point[0])
    dy = (start_p[1] - end_p[1])
    return math.sqrt(dx ** 2 + dy ** 2)


def rect_length(rect):
    return segment_length(rect[0])

def rect_width(rect):
    return segment_length(rect[1])

def perimeter(rect):
    return 2 * rect_length(rect) + 2 * rect_width(rect)
```

Bad Segment ADT

Constructor

```
def make_segment(start_point, end_point):
    return (start_point, end_point)
```

Getter

```
def start_point(segment):
    return segment(0)

def end_point(segment):
    return segment(1)
```

Question_9.tut04

Functional Abstraction & Data Abstraction
will be heavily emphasise

DO NOT BREAK THEM!!

IN YOUR EXAM, YOU WILL BE PINALIZE EVERY SINGLE TIME YOU BREAK THEM

Thank You!!

The End

See you next lesson

Extra_Question.tut04

EXTRA Practices

EXTRA_Question.tut04

(EXTRA)

Nested Lambda Function

Question 1

```
def f(x):
    return lambda y: (x, y(x))

def g(y):
    return lambda x: x(y)

print(g(2)(f)(lambda x: x + 1))
```

EXTRA_Question.tut04

OUTPUT:

(2, 3)

EXTRA_Question.tut04

(EXTRA)

Lovely Lambda

QUESTION 2

```
def bar(f, g):
      return lambda x: (lambda y: f(x))(g(x))

print(bar(lambda x: x+1, lambda x:x**2)(5))
```

EXTRA_Question.tut04

6

EXTRA_Question.tut04

(EXTRA)

What is block again?

QUESTION 3

```
m, p = “mutton”, “python”
mp = p[:4] + m[-2:]
if p not in mp:
    print(“yum”)
else:
    print(“hum”)
if “on” in p:
    print(“bum”)
else:
    print(“tum”)
```

EXTRA_Question.tut04

```
hum
bum
```

```
“hum”
“bum”
```

EXTRA_Question.tut04

(EXTRA)

Wait... what??

QUESTION 3

```
s = “python is easy”

while not s == “”:
    s = s[1:]
    out = s[0]
    if s == “ ”:
        break
print(out)
```

EXTRA_Question.tut04

```
IndexError: string index
out of range
```

BY: Zhu_Ming

Any Questions?