

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

```
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

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"

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
```

Important!!!

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

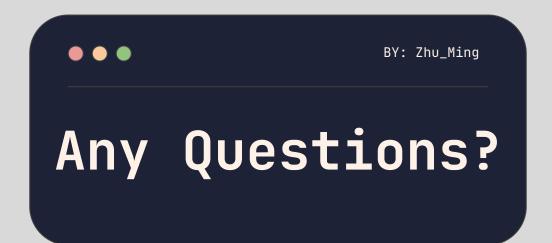
```
p = make_point(1, 0)
x = p[0] # wrong
x = get_x(p) # correct
```

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]))
```





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 \leftarrow
     total = 0
    for i in range(n + 1):
          term = f(a + i * h) \leftarrow
                                             // y_k = f(a + kh)
          if i == 0 or i == n:
               total += term
          elif i % 2 == 0:
                                             \frac{h}{3}[y_0 + 4y_1 + 2y_2 + 4y_3 + 2y_4 + \dots + y_n]
               total += 2 * term
          else:
               total += 4 * term
    return (total * h) / 3.0
```



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



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

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

```
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)
```



```
a_1 = a, a_n \le b accumulate(\bigoplus, base, f, a, next, b): (f(a_1) \bigoplus f(a_2) \bigoplus (... \bigoplus (f(a_n) \bigoplus base) ...)))
```

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))
```



```
a_1 = a, a_n \le b accumulate(\bigoplus, base, f, a, next, b): (f(a_1) \bigoplus f(a_2) \bigoplus (... \bigoplus (f(a_n) \bigoplus base) ...)))
```

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



```
a_1 = a, a_n \le b accumulate(\bigoplus, base, f, a, next, b): (f(a_1) \bigoplus f(a_2) \bigoplus (... \bigoplus (f(a_n) \bigoplus base) ...)))
```

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



Point ADT: make_point, x_point, y_point

<u>Constructor</u>

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

<u>Getter</u>

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

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

<u>Constructor</u>

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

<u>Getter</u>

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

def y_point(point):
```

return point(1)

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



```
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)



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)
```



What Attributes would you choose to define a rectangle??

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

Rectangle Attributes

Length Width (What is the data types?) perimeter

Rectangle Methods

area



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)
```



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)
```



Alternative implementations??

Use ADT Point!!

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

What should I modify??



Constructor

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

<u>Getter</u>

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

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

Constructor

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

<u>Getter</u>

```
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))
```



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



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])
                                                                  def end_point(segment):
    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

return segment(1)

Constructor

```
def make_segment(start_point, end_point):
    return (start_point, end_point)
Getter
def start_point(segment):
    return segment(0)
```



DO NOT BREAK THEM!!

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







(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))
```

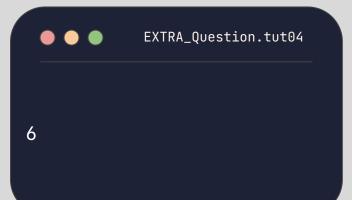




Lovely Lambda

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))
```

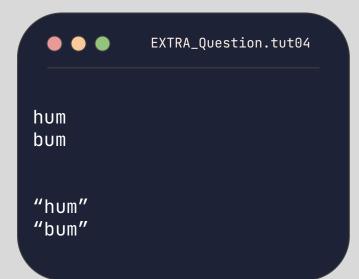


(EXTRA)

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")
```




```
Wait... what??
```

QUESTION 3

```
s = "python is easy"
while not s == "":
    s = s[1:]
    out = s[0]
    if s == " ":
        break
print(out)
```



IndexError: string index
out of range

