BY: Zhu_Ming

<CS1010S>

Tutorial 2

Functional Abstraction
Recursion & Iteration

Code Quality

1. Readable
2. Understandable
3. Easy to maintain & modify
4. Do well what is intended to do

How to write a “GOOD” code

1. Think before you code

2. Adopt good practices and follow the PEP8 styling
   a. Comments #
   b. Name variable meaningfully
   c. Avoid bad habit, (spacing, indentation, etc)
   d. Don’t hard-code everything

3. Review your code, optimize it
   a. Observe similar pattern
   b. Think of alternative solution

4. Iterative process

PEP8 Python Style Guide!!

https://peps.python.org/pep-0008/

Lecture_Recap.tut02

```
def use(m ,n):
    if (m - n) <0:
        return “Not enough, net = “+str(m-n)
    return ”net = “ + str(m-n)
    pass



def buy(wallet_money, expenses):

    # compute remaining money
    remaining = wallet_money - expenses

    if (remaining < 0): # if not enough money
        return f”(Not enough, net = {remaining})”
    else: # enough money
        return f”(net = {remaining})”
```

Python_Tips!!

f-string a special way to write a string

```
f”(... {argument} ...)”


name = Zhu_ming
print(f”(I’m {name}.)”
>>> “I’m Zhu_ming.”
```

Lecture_Recap.tut02

Lecture Recap

1. Variable Scope
   Global Scope & Local Scope

2. Functional Abstraction

3. Wishful Thinking

4. Divide & Conquer

5. Function
   lambda function

6. Recursion

```python
def factorial(n):
    if n == 0:
        return 1
    else:
        return n * factorial(n-1)
```

7. Iteration

```python
def factorial(n):
    f = 1
    for i in range(2, n+1):
        f *= i
    return f
```

1. Variable Scope

Local Variable	Global Variable
Created inside a function	Created in the main body
Can only be access within the function	Can be access throughout the code

```
x = 1                     GLOBAL

def local(x):             LOCAL
        x = 2
        return x + 2

print(local(5)) # 4
print(local(x)) # 4
print(x) # 1
```

Python_Tips!!
Local variable always override the Global variable

Lecture_Recap.tut02

2. Function

```
def <name>(<parameters>):
    <body>
    return <statements>


def emtyp_function(x, y, z):
    # not sure what to write
    pass
    # return None
```

BAD PRACTICES :(

```
def bad(x):
    # bla bla bla
    return ...
    pass
```

DON'T DO THIS…
It wouldn't affect the code but is unnecessary

3. Anonymous Function (Lambda function)

```
lambda_func = lambda <parameters>: <expression>
```

```
add_one = lambda x: x + 1
addition = lambda x, y: x + y
positive_add_one = lambda x: x + 1 if x > 0 else “negative value”
nothing = lambda : 0

add_one(1) # 2
addition(1,1) # 2
positive_add_one(1) # 2
positive_add_one(-1) # “negative value”
nothing() # 0
```

<u>What will happen?</u>

```
add
>>> <function <lambda>
at 0x0000011492BB8820>

addition(1)
>>> Error
```

4. Indentation

**** Python use indentation to indicate* ***block of code***

```
def print_hello(n):
    while n > 0:
        n -= 1
        print("hello")
```

```
"hello"
"hello"
.
.
.
```

```
def print_hello(n):
    while n > 0:
        n -= 1
    print("hello")
```

```
"hello"
```

5. Recursion

- Function that usually call **itself** (special cases, is other function)
- Base case (Terminating Condition)

```
def recursive(<parameters>):
     if <base case>:
            return <base_case_value>
        else:
            return <statements> + recursive(<next_parameters>)
```

```
def factorial(n):
    if n == 0:
       return 1
    else:
       return n * factorial(n-1)
```

```
factorial(4)
>>> 4 * factorial(3)
>>> 4 * 3 * factorial(2)
>>> 4 * 3 * 2 * factorial(1)
>>> 4 * 3 * 2 * 1 * factorial(0)
>>> 4 * 3 * 2 * 1 * 1
>>> 4 * 3 * 2 * 1
>>> 4 * 3 * 2
>>> ...
>>> 24
```

6. Iteration (For Loop)

- range(start,stop,steps)
- start is included; stop is not included

```
result = 0
for i in range(start,stop,steps):
    <body
    result += ...>
```

```
def sum_all(n):
    result = 0
    for i in range(1,n+1):
      result += i
    return result
```

```
sum_all(4)
result = 0

---------For Loop Starts---------
i = range(1,4+1)          result += i

i = 0                     result = 0
i = 1                     result = 1
i = 2                     result = 3
i = 3                     result = 6
i = 4                     result = 10
----------For Loop End----------

return result
```

7. Iteration (While Loop)

- Initialization of counter

```
result = 0
counter = 0
while <condition>:
     <body
     result += ...
     counter += ...>
```

```
def add_one(n):
    result = 0
    counter = 0
    while counter < n:
      result += 1
      counter += 1
    return result
```

```
add_one(4)
result = 0
counter = 0

---------For Loop Starts---------
result = 1          counter = 1
result = 2          counter = 2
result = 3          counter = 3
result = 4          counter = 4
----------For Loop End----------

return result
```

6. break & continue

- If break is call, quite the loop
- If continue is call, skip the current turn of loop

```
def add_even(n):
    result = 0
    for i in range(1,n+1):
        if i % 2: # is_odd
            continue
        elif i > 5:
            break
        else:
            result += i
    return result
```

BY: Zhu_Ming

Any Questions?

Question_1.tut02

Define a function magnitude that takes in the coordinates of TWO POINTS on a plane: **(x1, y1)** and **(x2, y2)** as arguments and returns the **magnitude of the vector** between them.

```
from math import sqrt

def magnitude(x1, x2, y1, y2):
    return sqrt((x1 - x2) ** 2
                + (y1 - y2) ** 2)
```

Question_1.tut02

How to improve this code??

Question_1.tut02

Functional abstraction!!

```
from math import sqrt

def sqr(x):
    return x ** 2

def sqr_diff(x1, x2):
    return sqr(x1 - x2)

def magnitude(x1, x2, y1, y2):
    return sqrt(sqr_diff(x1, x2) +
                sqr_diff(y1, y2))
```

python_tip01.tut02

How to import??

1. Only sqrt function

```
from math import sqrt
>>> sqrt(x)
```

2. All math function

```
from math import *
>>> sqrt(x)
```

3. Imported math packages

```
import math
>>> math.sqrt(x)
```

4. Packages Aliasing

```
import math as m
>>> m.sqrt(x)
```

Question_2.tut02

Area of a triangle = 1/2 * base * height.

Define a function area that calculates and returns the area of any given triangle.

Decide what arguments it requires as input and what its return value should be.

Question_2.tut02

```
def area(base, height):
    return 0.5 * base * height
```

Question_3.tut02

Area of a triangle = 1/2 * A * B * sin(AB)

Define a function area2 that calculates and returns the area of any given triangle in this formula.

Method_01

```
def area2(A, B, angle_AB):
     return 0.5 * A * B * sin(angle_AB)
```

Question_3.tut02

What is wrong with the Method_01???

Method_02

```
import math   # wrong

def area2(A, B, AB):
     return 0.5 * A * B * sin(AB)
```

Question_3.tut02

Wrong

```
import math
```

Correct

```
from math import *
```

Question_3.tut02

Are they the same?? Why??

```
def area(base, height):
    return 0.5 * base * height

def area2(A, B, angle_AB):
    return 0.5 * A * B * sin(angle_AB)
```

NO, because the parameters of function refer to different thing/ have different meaning.

Common_Mistake.tut02

Are they the same??

```
def area(A, B):
    return 0.5 * A * B

def area2(A, B, AB):
    return 0.5 * A * B
            * sin(AB)
```

NO! the A & B is just a variable.

```
f(x) = x        g(x) = x
f(a) = a        h(x) = x
```

Both function f is same
But g & h is different

Question_4.tut02

Given a function **herons_formula** that takes **3 arguments a, b, c** and returns the area of a triangle with sides of length a, b, c.

Define a function area3 that uses Heron's formula to calculate and return the **area of the given triangle** given the **x,y coordinates** of the 3 points of the triangle.

```
def area3(x1, y1, x2, y2, x3, y3):
    a = magnitude (x1, y1, x2, y2)
    b = magnitude (x2, y2, x3, y3)
    c = magnitude (x3, y3, x1, y1)
    return herons_formula(a, b, c)
```

Once again

FUNCTIONAL ABSTRACTION

We don't even discuss anything related to the herons_formula

Question_4.tut02

Imagine we don't have functional abstraction

Heron's Formula

math.sqrt(s(s - a)(s - b)(s - c)), where s = (a+b+c)/2.

```
def area3(ax, ay, bx, by, cx, cy):
    a = sqrt((ax - bx) ** 2 + (ay - by) ** 2)
    b = sqrt((bx - cx) ** 2 + (by - cy) ** 2)
    c = sqrt((ax - cx) ** 2 + (ay - cy) ** 2)
    s = (a + b + c) / 2
    return sqrt(s * (s - a) * (s - b) * (s - c))
```

What if one day we want to edit the code, is a painful process without functional abstraction to simplify the code

Question_5.tut02

```python
def foo1():
    i = 0 # initializing i
    result = 0 # initializing result
    while i < 10:
        result += i
        i += 1
    return result

print(foo1())
```

Question_5.tut02

```
result : 45
i : 10
```

Question_5.tut02

```
i = 0                   result = 0

------While Loop Starts-------
i += 1                  result += i

i = 1                   result = 0
i = 2                   result = 1
i = 3                   result = 3
  .                             .
  .                             .
  .                             .
  .                             .
i = 9                   result = 45
i = 10
-------While Loop End----------

return result
```

Question_5.tut02

```python
def foo2():
    i = 0 # initializing i
    result = 0 # initializing result
    while i < 10:
        if i == 3:
            break
        result += i
        i += 1
    return result

print(foo2())
```

Question_5.tut02

```
result : 3
i : 3
```

Question_5.tut02

```
i = 0                result = 0

------While Loop Starts-------
i += 1               result += i

i = 1                result = 0
i = 2                result = 1
i = 3                result = 3
break
-------While Loop End----------

return result
```

Question_5.tut02

```python
def bar1():
    result = 0 # initializing result
    for i in range(10):
        result += i
    return result

print(bar1())
```

Question_5.tut02

```
result : 45
i : 9
```

Question_5.tut02

```
result = 0

------For Loop Starts-------
i = range(10)     result += i

i = 0             result = 0
i = 1             result = 1
i = 2             result = 3
  .                     .
  .                     .
  .                     .
  .                     .
i = 9             result = 45

-------For Loop End---------


return result
```

Question_5.tut02

```python
def bar2():
    result = 0 # initializing result
    for i in range(10):
        if i % 3 == 1:
            continue
        result += i
    return result

print(bar2())
```

Question_5.tut02

```
result : 33
i : 9
```

Question_5.tut02

```
result = 0

------For Loop Starts-------
i = range(10)     result += i

i = 0             result = 0
i = 1             continue
i = 2             result = 2
i = 3             result = 5
i = 4             continue
i = 5             result = 10
  .                 .
i = 9             result = 33

-------For Loop End----------


return result
```

Question_6.tut02

Write a function **sum_even_factorials** that finds the sum of the factorials of the **non-negative even numbers** that are **less than or equal to n.**

Wishful Thinking Method

```
def sum_even_factorials(n):
    if n == 0 :
        return factorial(1)
    if n % 2 == 1:
        return sum_even_factorials(n-1)
    else:
        return factorial(n) +
            sum_even_factorials(n-2)
```

Question_6.tut02

```
def factorial(n):
    result = 1
    for i in range(2, n+1):
        result *= i
    return result
```

Question_6.tut02

Bottom-Up Approach

```
def factorial(n):
    result = 1
    for i in range(2, n+1):
        result *= i
    return result

def sum_even_factorials(n):
    if n == 0 :
        return factorial(1)
    if n % 2:
        return sum_even_factorials(n-1)
    else:
        return factorial(n) +
            sum_even_factorials(n-2)
```

Question_6.tut02

Without D&C and Functional Abstraction

```
def sum_even_factorials(n):
    result = 0
    for i in range(0, n+1):
        f = 1 # setting the factorial
        if i % 2 != 0: # if odd
            continue # next i
        else:
            for j in range(2, i+1): # compute factorial
                f *= j
        result += f
    return result
```

How complicated is that. . .

Question_7.tut02

```
def f(g):
    return g(2)

def square(x):
    return x ** 2

f(square)
>>> 4

f(lambda z: z * (z + 1))
>>> 6

f(f)
>>> OUTPUT_03
```

Question_7.tut02

```
f(square)
>>> square(2)
>>> 2 ** 2
>>> 4

f(lambda z: z * (z + 1))
>>> lambda z: z * (z + 1)(2)
>>> 2 * (2 + 1)

What happened to f(f)??
f(f)
>>> f(2)
>>> 2(2)
>>> TypeError
```

Extra_Question.tut02

EXTRA Practices

EXTRA_Question.tut02

(EXTRA)

Recursion

QUESTION 1

```
def harr(n):
    if n == 0:
        return 0
    else:
        return n + harr(n-2)

print(harr(4))
print(harr(5))
print(harr(2))
```

EXTRA_Question.tut02

OUTPUT:

```
6
???? RecursionError
```

EXTRA_Question.tut02

(EXTRA)

Infinite Loop

QUESTION 2

```
def infinity(n):
    counter = 0
    while counter < 0:
        print(n)
    return n

print(infinity(-1))
```

EXTRA_Question.tut02

OUTPUT:

RecursionError??

In fact, NO Python Error
But **infinite loop** happens

Common_Mistake
Infinite loop occurs because **while loop condition** is never False.

Possible reason is counter is not updated, for example, missing
counter += 1

If infinite loop happens use ctrl + C
To stop the shell running the code

EXTRA_Question.tut02

(EXTRA)

Nested Lambda Function

Question 3

```
f = lambda z: (z + 1)

x = lambda z: (f(1))

y = lambda z: (f(z))


print(x(2))
print(y(2))
print(x(f))
print(f(f))
```

EXTRA_Question.tut02

OUTPUT:

```
2
3
2
TypeError
```

EXTRA_Question.tut02

(EXTRA)

print VS return VS pass

```python
def return_only(x):
    return x

def print_only(x):
    print(x)

def pass_only(x):
    pass

print(return_only(1))
print(print_only(2))
print(pass_only(3))
```

EXTRA_Question.tut02

OUTPUT:

```
1
2
None
None
```

BY: Zhu_Ming

Any Questions?

Thank You!!

The End

See you next lesson