CS1302 - Lecture 6 More Functions – (a) Functional Programming
(Dr Helena WONG)

==

[Contents]

[Helena's supplements - Exercises]

[Helena's supplements for **Functional_Programming.ipynb]**

1. Recursion

2. Global Variables and Closures

==

[Helena's supplements - Exercises]

Starter Exercise 1. Greatest Common Divisor

Exercise: What is the GCD of 15, 35?
Exercise: What is the GCD of 80, 64?

Challenge: try Exercise 1 in Functional_programming.ipynb

Starter Exercise 2a. Can the following swap x and y?

```
x = 10
y = 20
x = y
y = x
print('x is:', x)
print('y is:', y)
```

Starter Exercise 2b. Can the following swap x and y?

```
x = 10
y = 20
z = x    # a variable for temporary use

x = y
y = z

print('x is:', x)
print('y is:', y)
```

Starter Exercise 2c. Can the following swap x and y?

```
x = 10
y = 20

original_x = x
original_y = y
x = original_y
y = original_x

print('x is:', x)
print('y is:', y)
```

Starter Exercise 2d. Can the following swap x and y?

```
x = 10
y = 20

x, y = y, x # no temporary variable?
print('x is:', x)
print('y is:', y)
```

first change x using y's value, then change y using x's new value? No
first change y using x's value, then change x using y's new value? No
Fact: It works like **Exercise 2c**!!

==

[Helena's supplements for **Functional_Programming.ipynb]**

1. Recursion

What is recursion?

Recursion vs Iteration

Fibonacci number: 0, 1, 1, 2, 3, 5, 8, 13, 21, ...

GCD (Euclidean Algorithm)

2. Global Variables and Closures

Local variables vs Global variable

Name collision

Avoid gobal variables unless needed

Nested function and Nonlocal variable

Python Closures

==

1. Recursion – A function calls itself

Fibonacci number: 0, 1, 1, 2, 3, 5, 8, 13, 21, ...

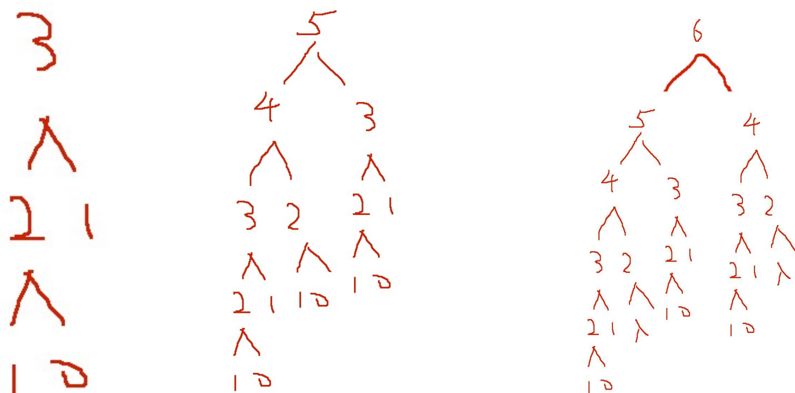
F0	F1	F2	F3	F4	F5	F6	F7	F8
0	1							
		0+1 = 1						
			1+1 = 2					
				1+2 = 3				
					2+3 = 5			
						3+5 = 8		
							13	
								21

```
def fibonacci(n):
    if n > 1:
        return fibonacci(n - 1) + fibonacci(n - 2)
    elif n == 1:
        return 1
    else:
        return 0

fibonacci(3)
```

```
def fibonacci_iteration(n):
    if n > 1:
        x, y = 0, 1    # F0 and F1
        while n > 1:
            x, y, n = y, x + y, n - 1
        return y
    elif n == 1:
        return 1
    else:
        return 0

fibonacci_iteration(3)
```



GCD (Euclidean Algorithm)

https://www.mathsisfun.com/greatest-common-factor.html

https://www.mathsisfun.com/definitions/euclidean-algorithm.html

=====================================

GCD (Euclidean Algorithm)

Example 1 : 112 and 42

divide 112 by 42. We get 2 with a remainder of 28

divide 42 by 28. We get 1 with a remainder of 14

divide 28 by 14. We get 2 with a remainder of 0

divide 14 by 0?   No need.   The GCD is 14

Example 2 : 42 and 112

divide 42 by 112. We get 0 with a remainder of 42

divide 112 by 42. We get 2 with a remainder of 28

divide 42 by 28. We get 1 with a remainder of 14

divide 28 by 14. We get 2 with a remainder of 0

divide 14 by 0?   No need.   The GCD is 14

Example 3 : 14 and 0

divide 14 by 0?   No need.   The GCD is 14

Example 4 : 0 and 14

divide 0 by 14. We get 0 with a remainder of 0

divide 14 by 0?   No need.   The GCD is 14

=====================================

```
def gcd(a, b):
    if b==0:
        return a
    else:
        return gcd(b, a%b)

gcd(15, 35)
```

```
def gcd_iteration(a, b):
    while b>0:
        a, b =b, a % b
    return a

gcd_iteration(15, 35)
```

Tail-recursion: the recursive call is the very last operation in the function

Recursion -- benefits

- Recursion is often shorter and easier to understand. It can provide *elegant* solutions to complex problems.
- Recursion can be easier to write code by *wishful thinking* or *declarative programming* as supposed to *imperative programming*.

Recursion -- efficiency issue

```
%%timeit
fibonacci(10)

%%timeit -n 1 -r 1
fibonacci(36)

# try fibonacci_iteration(300000)!!    # x = fibonacci_iteration(300000)
```

Atto a	10^{-18}	0.000 000 000 000 000 001
Femto f	10^{-15}	0.000 000 000 000 001
Pico p	10^{-12}	0.000 000 000 001
Nano n	10^{-9}	0.000 000 001
Micro μ	10^{-6}	0.000 001
Milli m	10^{-3}	0.001
Centi c	10^{-2}	0.01
Deci d	10^{-1}	0.1
	10^{0}	1
Deca da	10^{1}	10
Hecto h	10^{2}	100
Kilo k	10^{3}	1 000
Mega M	10^{6}	1 000 000
Giga G	10^{9}	1 000 000 000
Tera T	10^{12}	1 000 000 000 000

https://www.quora.com/What-is-the-difference-between-nanoseconds-microseconds-and-milliseconds

```
%%timeit -n 1 -r 1
100+200+300
```
Last executed at 2026-02-24 06:00:07 in 4ms

```
676 ns ± 0 ns per loop (mean ± std. dev. of 1 run, 1 loop each)
```

```
%%timeit -n 1 -r 1
gcd(2**100, 2**100+1)
```
Last executed at 2026-02-24 05:57:27 in 4ms

```
5.05 µs ± 0 ns per loop (mean ± std. dev. of 1 run, 1 loop each)
```

```
%%timeit -n 1 -r 1
gcd_iteration(2**100, 2**100+1)
```
Last executed at 2026-02-24 05:58:25 in 7ms

```
5.32 µs ± 0 ns per loop (mean ± std. dev. of 1 run, 1 loop each)
```

```
%%timeit -n 1 -r 1
fibonacci(37)
```
Last executed at 2026-02-24 05:59:14 in 2.80s

```
2.8 s ± 0 ns per loop (mean ± std. dev. of 1 run, 1 loop each)
```

```
%%timeit -n 1 -r 1
fibonacci_iteration(37)
```
Last executed at 2026-02-24 05:59:20 in 6ms

```
2.97 µs ± 0 ns per loop (mean ± std. dev. of 1 run, 1 loop each)
```

Other contents in [Functional_Programming.ipynb]

- Gcd: need to modify the while-loop version: “abs”, otherwise “assert” failed
- divide-and-conquer: reducing problems to smaller subproblems

- converting recursion to iteration is always possible; speed improved especially for non tail-recursion

Issues	**Tail-recursion**	**Non-tail recursion**
Converting recursion to iteration	can be converted to iteration easily	can be converted to iteration
Speed of Iteration version	Iteration version is not slower than Tail-recursion (at least as efficient)	Iteration version is much faster than recursion

==

2. Global Variables and Closures

Local variables vs Global variable
Name collision
Avoid gobal variables unless needed
Nested function and Nonlocal variable
Python Closures

==

(Global variables) Introductory examples:

```
x = 5

def add_1():
    result = x + 1
    return result

print(add_1())    # 6
print(x)    # 5
```

*** What if we need to update global variable?**

```
x = 5

def add_1():
    global x       # the global keyword
    x = x + 1

add_1()
print(x) # 6
```

"global" is a must if we need to change the value of the global

*** What if we omit "global x"?**

```
x = 5

def add_1():
    # let's omit global x
    x = x+1 # "x = " means creating a local variable!!

add_1()
print(x)    # 5? 6? (Ans: Neither. Because program stopped at add_1())
```

Therefore the x in x+1 is such a new local variable (that has no value yet!!) So, it causes a run-time error: "UnboundLocalError"

*** Mind naming collision (e.g. a local variable hides the like-named global variable)**

```
x = 5

def do_something():
    if 5 > 4 and 1 < 2:
        #... a lot of code
        x = 15
        #... a lot of code

    print(x) # suppose we want to print the global variable

do_something()    # output ____
```

*** With global variables**

codes are less predictable, more difficult to reuse/extend, and tests cannot be isolated, making debugging difficult.

*** it's always a good habit to use global to decare global variables to make your code easy to read**

*** Nested function and Nonlocal variable**

```
s="CS1302" #this is a global variable

def print_msg(msg):
    s1=msg

    def printer():
        nonlocal s1 # optional (unless we need to change s1)
        global s # optional (unless we need to change s)
        s2=s1
        print(s2)
        print(s)

    printer()

print_msg("Hello")
```

*** Python Closures**

We create a closure in Python if

- We have a nested function, i.e. function within a function

- The nested function refers to a variable of the outer function

- The enclosing function returns the enclosed function

Example 1:

```
s="CS1302" #this is a global variable

def print_msg(msg):
    s1=msg

    def printer():
        s2=s1
        print(s2)
        print(s)

    return printer

saved_printer = print_msg("Hello")
saved_printer()
saved_printer()

del print_msg
saved_printer() # still works!
```

Example 2:

```
def start_bonus(start):
    b = start

    def bonus():
        nonlocal b
        b+=1
        print("bonus is now", b)

    return bonus

amy_EarnBonus = start_bonus(100)
amy_EarnBonus() # bonus is now 101
amy_EarnBonus() # bonus is now 102

ada_EarnBonus = start_bonus(1000)
ada_EarnBonus() # bonus is now 1001
ada_EarnBonus() # bonus is now 1002
```

*** Function as data**

```
# We can also use funtion name as input parameters
# We can return functions (see closure)
def add(x, y):
    return x + y

def multiply(x, y):
    return x * y

def evaluate(f, x, y):
    return f(x,y)

print(add(2, 3))
print(multiply(2, 3))
print(evaluate(add, 2, 3))
print(evaluate(multiply, 2, 3))
```

*** Learn more in Functional_Programming.ipynb !!**

- **Fibonacci: Avoid redundant computation** [apply **Global Variables** for Fn, Fnn, n]
- **A While-loop version** – dangers concerning global variable!!

  => _single_leading_underscore notation

- **Fibonacci: applying Closure (without global variables ^_^)**

  Encapsulating Fn, Fnn, n

  ```
  usual_fib = create_fibonacci(0, 1)
  cs1302_fib = create_fibonacci("cs", "1302")
  ```

- **Lexical / static scoping**

  Allows local functions to capture and remember the scope in which they were created (not called)

- **Object-Oriented Programming Approach** with **Closure**

  Adding attribute (some objects, including functions) to a nested function, return the nested function

  => Closures enable an OOP approach by allowing the creation of objects that share methods but maintain possibly distinct attribute values.