AI전문가과정 – Python Review

# Function

Jinwook Seo, Ph.D.
Professor, Department of Computer Science and Engineering
Seoul National University

# Table of Contents

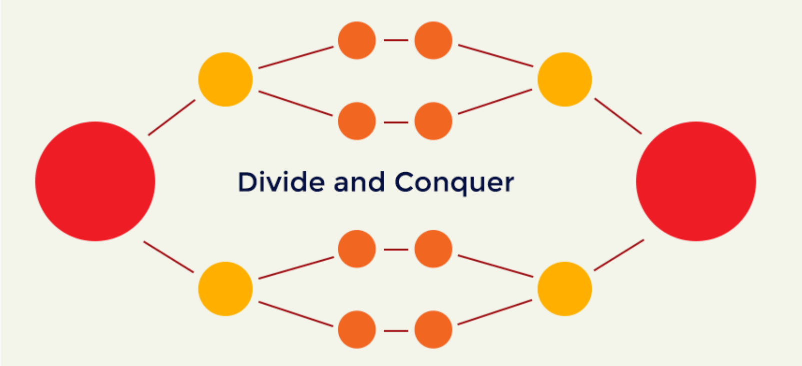
AI전문가과정 - Python Review

# Designing with Functions

# Introduction to Functions

- Function: group of statements within a program that perform a specific task
  - Usually one task of a large program
  - Known as divide and conquer approach (decomposition)

- Divide and Conquer Paradigm
  - Divide step
  - Conquer step
  - Combine step




source: [Divide and Conquer Paradigm in Algorithms, Gaurav Mishra]

# Introduction to Functions

- Modular programming: a software design technique that emphasizes separating the functionality of a program into independent, interchangeable modules
  - each module contains everything necessary to execute only one aspect of the desired functionality
  - high-level `decomposition` of the code of an entire program into pieces: package(folder) -> module(file) -> function
- Abstraction: it allows the users of a function to use a piece of code as if it were a black box
  - interior implementation details: users cannot see, don't need to see, and shouldn't even want to see
  - users of a function just have to know `assumptions` (about input) and `guarantees` (about output)
- Top-down design: technique for breaking an algorithm into functions

# Benefits of Modularizing a Program with Functions

- Simpler code
- Code reuse
  - write the code once and call it multiple times
- Better testing and debugging
  - can test and debug each function individually
- Faster development
- Easier facilitation of teamwork
  - different team members can write different functions

AI전문가과정 – Python Review

# Defining a Function

hci lab
SEOUL NATIONAL UNIVERSITY

# Function Definitions

- In Python, each fuinction definition is of the form:

  def *name of function* (list of *formal parameters*):

  *body of function*

- Example:

```
def maxVal(x, y):   # function header
    """Return maximum of x and y."""
    if x > y:
        return x
    else:
        return y
```

# Indentation in Python

- Each block must be indented
  - Lines in a block must begin with the same number of spaces
  - Use tabs or spaces to indent lines in a block, but not both as this can confuse the Python interpreter
- IDLE automatically indents the lines in a block
- Blank lines that appear in a block are ignored

hci lab
SEOUL NATIONAL UNIVERSITY

# Function Definitions

- A function is also an object.
- We can call a function after definition:

```
>>> maxVal
<function maxVal at 0x7fb2c65751f0>
>>> maxVal(3, 4)
4
>>> mV = maxVal
>>> mV(3, 4)
4
>>>
```

- At the time of function invocation (or function call)
  - Formal parameters (`x` and `y`) are bound to actual parameters (`3` and `4`)
    -> binding
  - actual parameters are also referred to as arguments

SEOUL NATIONAL UNIVERSITY

# Passing Arguments to Functions

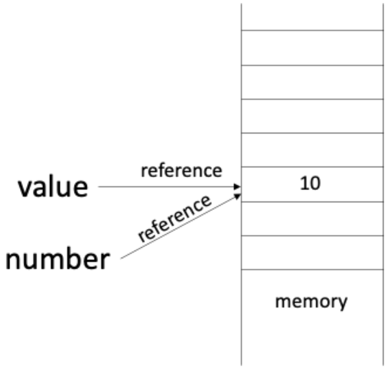
- Argument: piece of data that is sent into a function
  - Function can use arguments in calculations
  - When calling the function, the argument is placed in parentheses following the function name

- Formal Parameter (Parameter variable): variable that is assigned the value of an argument when the function is called
  - The parameter and the argument reference the same value

- Scope of a parameter: within the function in which the parameter is used
  - Scope: the part of a program in which a variable may be accessed

```
def show_triple(number):
    result = number * 3
    print(result)

def main():
    value = 10
    show_triple(value)
```



# Local Variables

- Local variable: variable that is assigned a value inside a function
  - Belongs to the function in which it was created
  - Only statements inside that function can access it, error will occur if another function tries to access the variable

- Scope: the part of a program in which a variable may be accessed
  - For local variable: its scope is the function in which it is created
  - Local variable cannot be accessed by statements inside its function which precede its creation

```
def f(args):
    print(args, x)
    x = 3
```

- Different functions may have local variables with the same name
  - Each function does not see the other function's local variables, so no confusion

AI전문가과정 - Python Review

# Calling a Function

# When a function is called: e.g., `maxVal(1+2, z)`

```
z = 6
result = maxVal(1 + 2, z)
```

- The expressions that make up the actual parameters are evaluated.
- The formal parameters are bound to the evaluation results.
  - `x -> 3`, `y ->` whatever value of `z` at the time of call
- The code in the body is executed.
- A return statement determins the value of invocation (or call).
  - If there is no return statement to execute, returns the value `None`.
- The value of the invocation (i.e., the function call) is the returned value.
  - main effect

# Function returns a value

```
    z = 6
    result = maxVal(1 + 2, z)  # maxVal returns a value, 6
    print(result)
```

- Even void functions (without a `return` statement) do return a value, i.e., `None`

```
def naturalNumbers(n):
    i=1
    while i <= n:
        print(i)
        i += 1

naturalNumbers(10)       # prints natural numbers from 1 to 10
naturalNumbers(0)        # prints nothing
print(naturalNumbers(0)) # prints "None"
```

Run Code Visualize

# Returning Multiple Objects

```
def sum_and_mul(a, b):
    return a + b, a * b   # returns a tuple

sum, mul = sum_and_mul(1, 2)
print(sum) # prints 3
print(mul) # prints 2

result = sum_and_mul(1, 2)
print(result)  # prints (3, 2)
```

Run Code Visualize

# Positional Arguments vs. Keyword Arguments

- Two ways that formal parameters get bound to actual parameters
  - positional arguments (bound by position of argument)
  - keyword arguments (bound by the name of the formal parameter)

```python
def printName(firstName, lastName, reverse):
    if reverse:
        print(lastName + ', ' + firstName)
    else:
        print(firstName, lastName)

printName('Jason', 'Mraz', False)  # positional
printName('Jason', 'Mraz', True)   # positional
printName('Jason', 'Mraz', reverse = False) # positional and keyword
```

# Positional Arguments vs. Keyword Arguments

```python
def printName(firstName, lastName, reverse):
    if reverse:
        print(lastName + ', ' + firstName)
    else:
        print(firstName, lastName)

printName('Jason', 'Mraz', False)  # positional
printName('Jason', 'Mraz', True)   # positional
printName('Jason', 'Mraz', reverse = False) # positional and keyword

printName(reverse=False, lastName = 'Mraz', firstName = 'Jason') # keyword
printName('Jason', firstName = 'Jason', False) # error
```

- keyword arguments can appear in any order
- keyword arguments must follow positional arguments

hci lab
SEOUL NATIONAL UNIVERSITY

# Keyword Arguments and Default (Argument) Values

- We can specify a default value for one or more arguments/parameters.
- Keyword arguments are commonly used with default argument values

```python
def printName(firstName, lastName='Mraz', reverse=False):
    if reverse:
        print(lastName + ', ' + firstName)
    else:
        print(firstName, lastName)

printName('Jason')
printName('Jason', 'Bourne')
printName('Jackson', 'Brown', True)
printName(firstName='Jackson')
```

Run Code Visualize

# More about Default Values

- The default values are evaluated at the point of function definition

```
i = 5

def f(arg = i):
    print(arg)

i = 6
f()
```

Run Code Visualize

# More about Default Values

- The default values are evaluated only once.

```
def append2List(a, L = []):
    L.append(a)
    return L

print(append2List(1))   # prints [1]
print(append2List(2))   # prints [1, 2]
print(append2List(3))   # prints [1, 2, 3]
```

Run Code Visualize

- A reference to the default value (i.e., object) is saved in the function definition
- Don't use a mutable object as a default value!

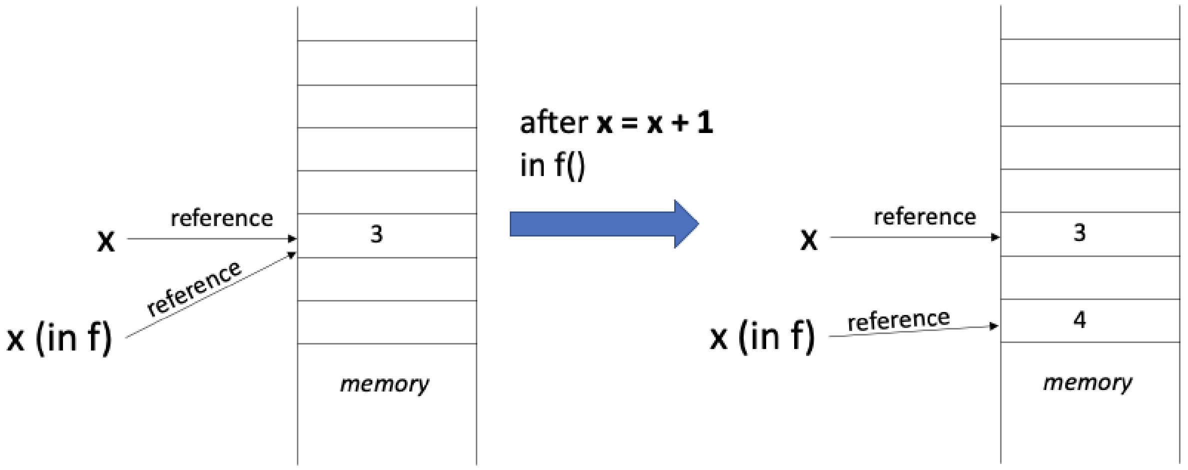
AI전문가과정 – Python Review

# Scope

# Scoping

```
def f(x):
    x = x + 1
    print(x)
    # prints 4

x = 3
f(x)
print(x)
# prints 3
```




- Even though the actual and formal parameters have the same name, they are not the same variable
- Each function defines a new name space, also called a scope
- The formal parameter `x` exists (can be accessed) only within `f`
- The assignment statement `x = x + 1` binds the local name `x` to the object `4`
- This assignment have no effect on the bindings of the name `x` outside `f`

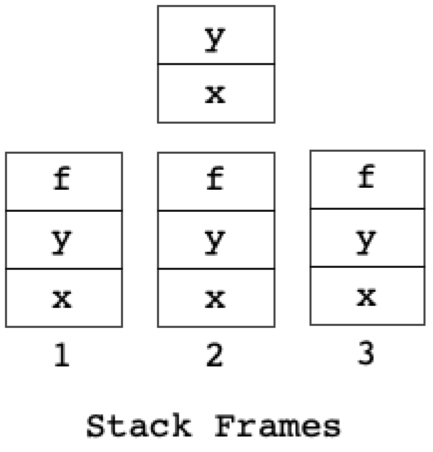
# Symbol Table and Stack Frame

- At top level, i.e., the level of the shell, a symbol table (often called a stack frame) keeps track of all names defined at that level and their current bindings
- When a function is called, a new symbol table is created for the function
  - It keeps track of all names defined within the function and their current bindings
  - If a function is called from within the function, another stack frame is created
  - Functions can reference variables from the containing scope
- When the function completes, its stack frame goes away (is popped off)

```
def f(x):
    y = 1
    x = x + 1
    print(x)   # prints 4

x = 3
y = 2
f(x)
print(x) # prints 3
print(y) # prints 2
```

Run Code Visualize



hci lab
SEOUL NATIONAL UNIVERSITY

# Static (or Lexical) Scoping

- We can determine the scope of a name by just looking at the program text
  - what matters is the context of where it is defined
  - without considering the run-time context, i.e., call stack (dynamic scoping)
- The order in which references to names occur is not relevant
  - If an object is bound to a name anywhere in the function body, it is treated as local to that function

```
def f():
   print(x)  # x references the x outside f

def g():
   print(x)  # x references the local x
   x = 1

x = 3
f()    # prints 3
x = 3
g()    # unboundLocalError: local variable 'x' referenced before assignment
```

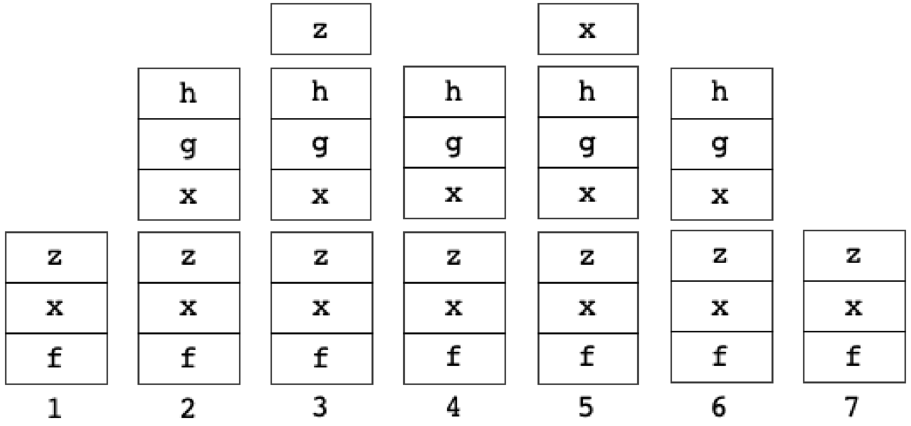
Run Code Visualize

# Nested Scopes

```python
def f(x):
    def g():
        x = 'abc'
        print('x =', x)
    def h():
        z = x
        print('z =', z)
    x = x + 1
    print('x =', x)
    h()
    g()
    print('x =', x)
    return g

x = 3
z = f(x)
print('x =', x)
print('z =', z)
z()
```

Run Code Visualize




Stack Frames

Result:

```
x = 4
z = 4
x = abc
x = 4
x = 3
z = <function f.<locals>.g at 0x7fd7552ec1f0>
x = abc
```

AI전문가과정 - Python Review

# Global Variable

# Global Variable

- Global variable is created by assignment statement written outside all the functions
- Can be accessed by any statements in the program file, including from within a function
- If a function needs to assign a value to a global variable, the global variable must be redeclared within the function `global variable_name`
- In Python, variables that are only `referenced` inside a function are implicitly global.
    - don't have to redeclare it with `global variable_name`
- If a variable is `assigned a value` anywhere within the function's body, it's assumed to be a local unless explicitly declared as `global`.

hci lab
SEOUL NATIONAL UNIVERSITY

# Referencing a Global Variable

- In Python, variables that are only `referenced` inside a function are implicitly global.

```
# Create a global variable.
my_value = 10

# The show_value function prints
# the value of the global variable.

def show_value():
    print(my_value)

# Call the show_value function.
show_value()
```

Run Code Visualize

# To Change a Global Variable

- If a function needs to assign a new value to a global variable, the global variable must be redeclared with the keyword `global` within the function
  - gernal format: `global variable_name`

```python
# Create a global variable.
number = 0

def main():
    global number
    number = int(input('Enter a number: '))
    show_number()

def show_number():
    print('The number you entered is', number)

main()
```

Run Code Visualize

hci lab
SEOUL NATIONAL UNIVERSITY

# Hide/Shadow a Global Variable

- If you do not redeclare a global variable with the `global` keyword inside a function,
  - you cannot change the variable's value inside that function
  - you are creating a new local variable with the same name
  - --> hiding/shadowing the global variable

```
# Create a global variable.
number = 0

def main():
    global number   # redeclare the variable, number
    number = int(input('Enter a number: '))
    show_number()

def show_number():
    number = 100    # shadow the global variable
    print('The number you entered is', number)

main()
```

Run Code Visualize

# Global Constants

- Named Constants in Global Scope
  - You should use named constants instead of magic numbers.

- If you do not redeclare a global variable with the `global` keyword in side a function, you cannot change the variable's value inside that function.
  - global constant: global name that references a value that cannot be changed.
  - `CONTRIBUTION_RATE = 0.05` (next page)

hci lab
SEOUL NATIONAL UNIVERSITY

# Gobal Constant – Example

```
# The following is used as a global constant
# the contribution rate.
CONTRIBUTION_RATE = 0.05

def main():
    gross_pay = float(input('Enter the gross pay: '))
    bonus = float(input('Enter the amount of bonuses: '))
    show_pay_contrib(gross_pay)
    show_bonus_contrib(bonus)

def show_pay_contrib(gross):
    contrib = gross * CONTRIBUTION_RATE
    print('Contribution for gross pay: $', format(contrib, ',.2f'), sep='')

def show_bonus_contrib(bonus):
    contrib = bonus * CONTRIBUTION_RATE
    print('Contribution for gross pay: $', format(contrib, ',.2f'), sep='')

main()
```

# Avoid Using Global Variables

- Global variables make debugging difficult
  - Many locations have to be checked to track down a bug
- Functions that use global variables are usually dependent on those variables
  - Makes such functions hard to transfer to another programs
- Global variables make a program hard to understand
  - The key to making programs readable is locality
  - Global variables can be modified or read in a wide variety of places
- There are times when global variables are just what is needed
  - for debugging or analyzing purpose
  - e.g., count how many times a function has been called

AI전문가과정 - Python Review

# Arbitrary Argument Lists

hci lab
SEOUL NATIONAL UNIVERSITY

# Arbitrary Argument Lists

- a function can be called with an arbitrary number of arguments.

```
def sum_many(*args):
    """
    args: passed as a tuple
    """

    print(args)
    sum = 0
    for i in args:
        sum = sum + i
    return sum

result = sum_many(1, 2, 3)
print(result)  # prints 6
result = sum_many(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)
print(result)  # prints 55
```

Run Code Visualize

hci lab
SEOUL NATIONAL UNIVERSITY

# Arbitrary Argument Lists

```
def sum_mul(ops, *args):
    if ops == 'sum':
        result = 0
        for i in args:
            result = result + i
    elif ops == 'mul':
        result = 1
        for i in args:
            result = result * i
    else:
        result = None
    return result

result = sum_mul('sum', 1, 2, 3, 4, 5)
print(result)  # prints 15
result = sum_mul('mul', 1, 2, 3, 4, 5)
print(result)  # prints 120
```

Run Code Visualize

hci lab
SEOUL NATIONAL UNIVERSITY

# Arbitrary Argument Lists

```
def sum(*values, **options):
    """
    values: passed as a tuple
    options: passed as a dictionary
    """

    sum = 0
    answer = ''

    for i in values:
        sum = sum + i

    if 'neg' in options:
        if options['neg']:
            sum = -sum

    if 'explain' in options:
        if options['explain']:
            answer = "The answer is "

    return answer + str(sum)
```

- `**options` expects keyword arguments
    - of the form `parameter = value` pairs
    - passed as a `dictionary` to the function

```
>>> sum(1, 2, 3)
'6'
>>> sum(1, 2, 3, neg = True)
'-6'
>>> sum(1, 2, 3, neg = False)
'6'
>>> sum(1, 2, 3, explain = True)
'The answer is 6'
>>> sum(1, 2, 3, neg = True, explain = True)
'The answer is -6'
```

AI전문가과정 - Python Review

# Anonymous Functions

# Anonymous Functions with Lambda Expression

- Small anonymous functions can be created with the `lambda` keyword.

```
>>> (lambda a, b: a+b)(1, 2)
3
>>> f = lambda a, b: a+b   # define a lambda function
>>> f
<function <lambda> at 0x7fc9657ac5e0>
>>> f(1, 2)
3
```

This lambda function is equivalent to:

```
def f(a, b):
    return a + b
```

hci lab SEOUL NATIONAL UNIVERSITY

# Anonymous Functions with Lambda Expression

- Lambda functions can be used wherever function objects are required.
- They are syntactically restricted to a single expression.
- Semantically, they are just `syntactic sugar` for a normal function definition.
- Like nested function definitions, lambda functions can reference variables from the containing scope.

```
>>> def make_incrementor(n):
...     return lambda x: x + n
...     # lambda function can reference the formal parameter (local variable, n)
...
>>> f = make_incrementor(42)
>>> f(0)
42
>>> f(1)
43
```

# Anonymous Functions with Lambda Expression

- Let's translate the following code using lambda expression

```
def is_even(x):
    if x % 2 == 0:
        return True
    else:
        return False

source_list = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

filtered_list = list(filter(is_even, source_list))

print(filtered_list)   # prints [2, 4, 6, 8, 10]
```

- Built-in function: `filter(function, iterable)`
    - Construct an iterator from those elements of `iterable` for which `function` returns `True`.

# Anonymous Functions with Lambda Expression

- Another use of Lambda expression is to pass a small function as an argument:

```
# Program to filter out only the even items from a list

source_list = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

filtered_list = list(filter(lambda x: (x % 2 == 0) , source_list))

print(filtered_list)   # prints [2, 4, 6, 8, 10]
```

Run Code Visualize

AI전문가과정 - Python Review

# Q&A

hci lab
SEOUL NATIONAL UNIVERSITY

# Acknowledgement


- The Python Tutorial, https://docs.python.org/3/tutorial/index.html
- Lecture Notes, Professor Hyungjoo Kim
- Starting out with Python, Professor Tony Gaddis and Pearson Education, Ltd.
- Introduction to Computation and Programming Using Python, John V. Guttag