Functions

Python Programming

Byeongjoon Noh

powernoh@sch.ac.kr

SOON CHUN HYANG UNIVERSITY

Most of the slides are available on Senseable AI Lab homepage: https://sailab.space/courses/

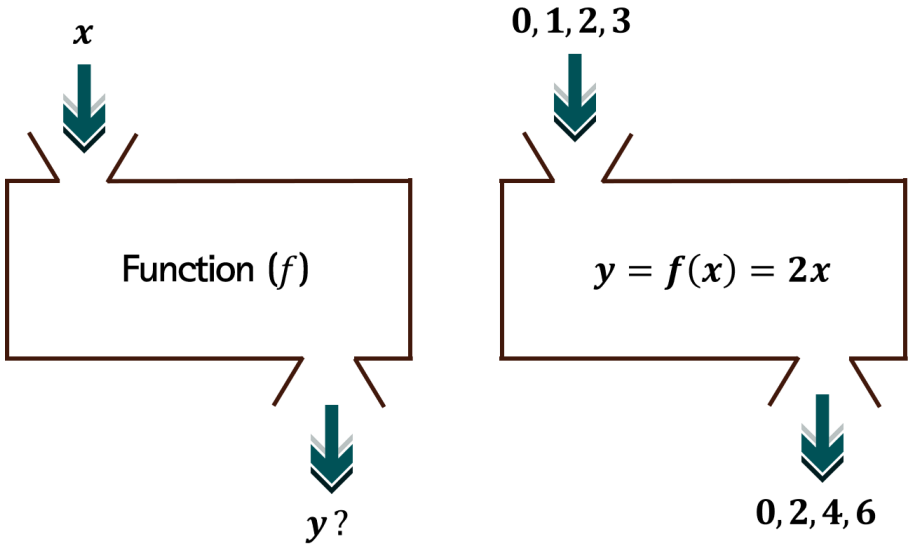
Contents

Textbook: Chapter 11, Chapter 13, Chapter 29

1. Functions in Python

What are functions?

- Function learned from your mathematic class



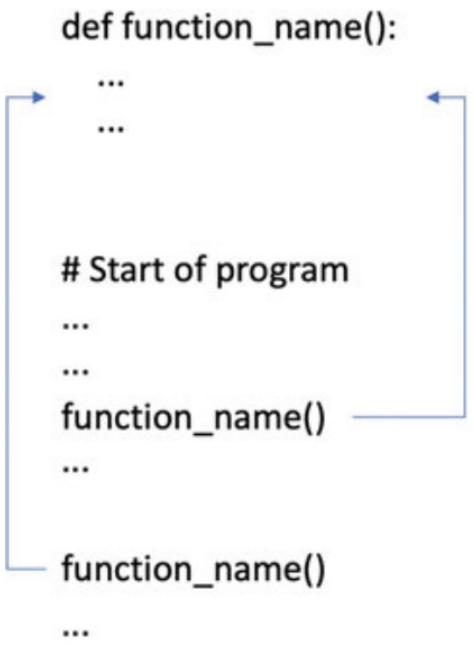
What are functions?

- Function in computer language
  - a block of organized, reusable code that performs a specific task (functionality, modularity)
    - provide better modularity and facilitate code reusability

- Why use function?
  - the same function can be called multiple times or in multiple locations (reusability)
    - defined in one place and called or invoked in another
  - easier to update and debug (maintainability)
    - help to make code more modular and easier to understand

How functions work

- When a function is **called** (or invoked) the flow of control a program jumps from where the function was called to the point where the function was **defined**



  - the body of the function is then executed before control returns back to where it was called from
  - each time the call is made to `function_name()` the program flow jumps to the body of function and executes the statements there

Types of functions

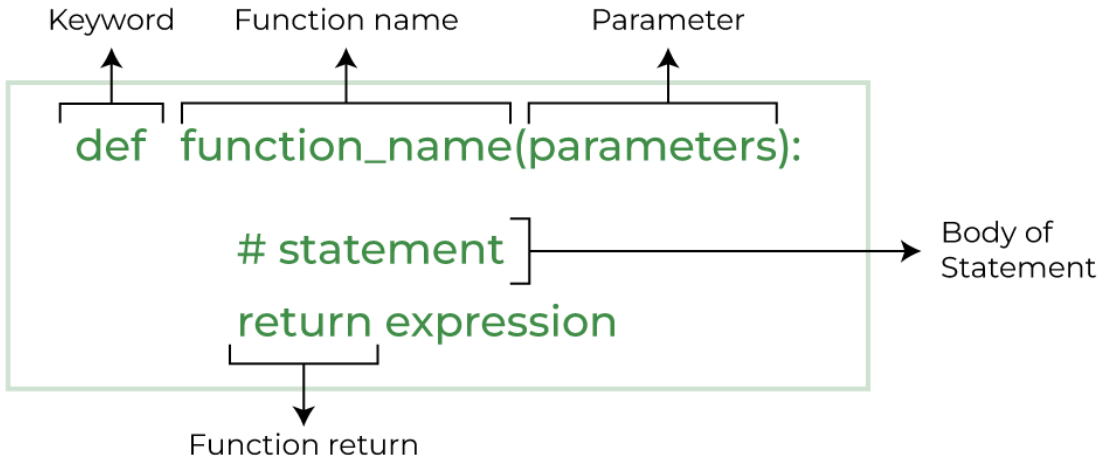
- Built-in functions
  - provided by the language and we have seen several of these already
  - e.g. `print()` and `input()`
  - did not need to define them ourselves as they are provided by Python

- User-defined function
  - written by developers

Defining functions

- Basic syntax of a function

```
def function_name(parameter list):
    """docstring"""
    statement
    statement(s)
```



- all functions are defined using *keyword* `def`; indicates the start of a function definition
- function can have a name which uniquely identifiers
- naming conventions; adopting for variables' name rules
- (optional) function has a list of parameters which allow data to be passed into the function
- colon is used to mark the end of function header; start of the function body
- (optional) docstring describes what the function does
- one or more Python statement make up the function body

Example functions

```
def print_msg():
    print('Hello World')

'''
'''
print_msg()
```

```
Hello World
```

- print_msg() function takes no parameters and has only a single statement that print out the message

Example functions

```
def print_my_msg(msg):
    print(msg)
'''
'''
print_my_msg('Hello World')
print_my_msg('Good day')
print_my_msg('Welcome')
print_my_msg('Ola')
```

```
Hello World
Good day
Welcome
Ola
```

- `print_my_msg()` function takes a single parameter and this parameter becomes a variable which is available within the body of the function
  - variable `msg`; not available outside of the function

Returning values from functions

- use return statement to return a value from a function
  - whenever a return statement is encountered within a function, function will terminate and return any values following return keyword
    - the returned value can be used at the point that the function was invoked

```
def square(n):
    return n * n
'''
'''
result = square(4)
print(result)
print(square(5))

if square(3) < 15:
    print('Still less than 15')
```

```
16
25
Still less than 15
```

Returning values from functions

- possible to return multiple values from a function

```
def swap(a, b):
    return b, a
'''
'''
a = 2
b = 3
x, y = swap(a, b)
print(x, ',', y)
```

```
3 , 2
```

Quiz

- What is the output of the following code?

```
def func(x, y):
    if x == 0:
        return y
    else:
        return func(x-1, x*y)
print(func(4, 2))
```

  - a) 48
  - b) 96
  - c) 16
  - d) 32

Quiz

- What is the output of the following code?

```
def func(num):
    if num > 10:
        return num - 10
    return func(func(num + 11))

print(func(5))
```

  - a) 10
  - b) 1
  - c) 9
  - d) 0

Returning values from functions

- Return value annotation

```
def function() -> int:
    x = 100
    return x**2
```

  - `->`
    - arrow notation guide to specify what each function will return
    - just guidance for developers, not mandatory

Docstring

- allows the function to provide some guidance on what is expected in terms of the data passed into the parameters, potentially what will happen if the data is incorrect, as well as what the purpose of the function is in the first place

```
def get_integer_input(message):
    """
    This function will display the message to the user
    and request that they input an integer.
    If the user enters something that is not a number
    then the input will be rejected
    and an error message will be displayed.
    The user will then be asked to try again.
    """

    value_as_string = input(message)
    while not value_as_string.isnumeric():
        print('The input must be an integer')
        value_as_string = input(message)
    return int(value_as_string)
```

Docstring

- can be read directly from the code but is also available to the programmer via a very special property of the function called __doc__
  - double underbars front and read of ‘doc’ letters

```
print(get_integer_input.__doc__)
```

```
    This function will display the message to the user
    and request that they input an integer.
    If the user enters something that is not a number
    then the input will be rejected
    and an error message will be displayed.
    The user will then be asked to try again.
```

Function parameters

- Parameter
  - a **variable** defined as part of the function header and is used to make data available within the function itself

- Argument
  - the actual **value** or data passed into the function when it is called
  - the data will be held within the parameters

```
def swap(a, b):
    return b, a
'''
'''
x, y = swap(2, 3)
```

Function parameters

- Multiple parameter functions
  - parameter list contains a list of parameter names separated by a comma

```
def greeter(name, message):
    print('Welcome', name, '-', message)

greeter('Eloise', 'Hope you like Rugby')
```

  - two parameters; name and message
    - used within only the body of function
    - current version allows any number of parameters defined in a function

Function parameters

- Default parameter values
  - once you have one or more parameters you may want to provide default values for some or all of those parameters; particular for ones which might not be used in most cases

```
def greeter(name, message = 'Live Long and Prosper'):
    print('Welcome', name, '-', message)

greeter('Eloise')
greeter('Eloise', 'Hope you like Python')
```

```
Welcome Eloise - Live Long and Prosper
Welcome Eloise - Hope you like Python
```

  - the default value must be declared in the function header along with the parameter name
  - if a value is supplied for the parameter, it will override the default
  - if no value is supplied when the function is called, the default will be used

Function parameters

- Default parameter values

```
def greeter(name, message = 'Live Long and Prosper'):
    print('Welcome', name, '-', message)
```

  - name; a mandatory field/parameter
  - message; an optional field/parameter (as it has a default value)

- a default value all remaining parameters to the right of that parameter must also have default value
  - can not defined the function as:

```
def greeter(message = 'Live Long and Prosper', name):
    print('Welcome', name, '-', message)
```

  - name must have a default value as it comes after (to the right) of a parameter with a default value

Function parameters

- Named arguments
  - why use
    - if a function has several parameters, it may become impossible to rely on using the position of a value to ensure it is given to the correct parameter
    - ➔ provide the name of each argument (with parameter)
  - how to use
    - write the names in each parameter as follows:

```
def greeter(name, title = 'Dr', prompt = 'Welcome', message = 'Live Long and Prosper'):
    print(prompt, title, name, '-', message)

'''
'''
greeter(message = 'We like Python', name = 'Lloyd')
greeter('Lloyd', message = 'We like Python')
```

```
Welcome Dr Lloyd - We like Python
Welcome Dr Lloyd - We like Python
```

Function parameters

- Named arguments
  - also, cannot place positional arguments after a named argument as follows
    - similar with the rule of ‘default parameter values’

```
greeter(name='John', 'We like Python')
```

Function parameters

- Arbitrary arguments
  - why use
    - we don’t know how many arguments will be supplied when a function is called
    - allows to pass an arbitrary number of arguments into a function and then process those arguments inside the function
  - how to use
    - a parameter is marked with an asterisk (*);
      - to define a parameter list as being of arbitrary length

Function parameters

- Arbitrary arguments

```
def greeter(*args):
    for name in args:
        print('Welcome', name)
'''
'''
greeter('John', 'Denise', 'Phoebe', 'Adam', 'Gryff', 'Jasmine')
```

```
Welcome John
Welcome Denise
Welcome Phoebe
Welcome Adam
Welcome Gryff
Welcome Jasmine
```

  - in this case, another use of the `for` loop; but this time it is a sequence of strings rather than a sequence of integers that is being used

Quiz

- What is the output of the following code?

```
def func(a, args, s = '!'):
    print(a, s)
    for i in args:
       print(i, s)

func(100)
```

  - a) Error
  - b) 100 !
  - c) ! 100
  - d) 100

Quiz

- What is the output of the following code?

```
def func(a, *args, s = '!'):
    print(a, s)
    for i in args:
       print(i, s)

func(100)
```

  - a) Error
  - b) 100 !
  - c) ! 100
  - d) 100

Quiz

- What is the output of the following code?

```
def func(a, *args, s = '!'):
    print(a, s)
    for i in args:
       print(i, s)

func(100, 'a', 'b', 'c')
```

  - a) Error
  - b) 100 abc !
  - c) 100 a ! b ! c !
  - d) 100

    a !

    b !

    c !

Quiz

- What is the output of the following code?

```
def func(s, *args):
    print("First_letters: ", s)
    for item in args:
        print("Next_letter: ", item)

func('H', 'E', 'L', 'L', 'O')
```

Function parameters

- Positional and keyword arguments
  - some functions are defined such that the arguments to the methods can either be provided using a variable number of positional or keyword arguments
  - two arguments `*args` and `**kwargs`
    - for positional arguments and keyword arguments (=named arguments), respectively
  - useful if you do not know exactly how many of either position or keyword arguments are going to be provided

Function parameters

- Positional and keyword arguments

```
def my_function(*args, **kwargs):
    for arg in args:
        print('arg:', arg)
    for key in kwargs.keys():
        print('key:', key, 'has value: ', kwargs[key])

'''
'''
my_function('John', 'Denise', daughter='Phoebe', son='Adam')
print('-' * 50)
my_function('Paul', 'Fiona', son_number_one='Andrew', son_number_two='James', daughter='Joselyn')
```

Function parameters

- Positional and keyword arguments

```
arg: John
arg: Denise
key: daughter has value:  Phoebe
key: son has value:  Adam
-----------------------------------------------
arg: Paul
arg: Fiona
key: son_number_one has value:  Andrew
key: son_number_two has value:  James
key: daughter has value:  Joselyn
```

Function parameters

- Positional and keyword arguments
  - keywords used for the arguments are not fixed
  - can only use one of the *args and **kwargs depending on requirements

```
def named(**kwargs):
    for key in kwargs.keys():
        print('arg:', key, 'has value:', kwargs[key])
'''
'''
named(a=1, b=2, c=3)
```

```
arg: a has value: 1
arg: b has value: 2
arg: c has value: 3
```

Function parameters

- Anonymous functions
  - why use
    - in some cases, we want to create a function and use it **only once**
    - giving it a name for this one time can *pollute* the program
    - also someone might call it when we don't expect them to
  - how to use
    - `lambda` function
      - possible to define an anonymous function; does not have a name and can only be used at the point that it is defined
    - general syntax

```
lambda arguments: expression
```

Function parameters

- Anonymous functions
  - `lambda` function (cont'd)
    - can have any number of arguments but only one expression
      - this expression is a statement that returns a value
    - the expression is executed, and the value generated from it is returned as the result of func.
  - example as follows:

```
res = lambda i : i * i
print(res(10))
```

```
100
```

    - one parameter (`i`) to the anonymous function
    - body of the function is defined after the colon (:)
    - value of returned in this function; which multiples `i * i`

Function parameters

- Anonymous functions
  - other examples using `lambda` function

```
func0 = lambda: print('no args')
func1 = lambda x: x * x
func2 = lambda x, y: x * y
func3 = lambda x, y, z: x + y + z
'''
'''
func0()
print(func1(4))
print(func2(3, 4))
print(func3(2, 3, 4))
```

```
no args
16
12
9
```

Quiz

- What is the output of the following code?

```
add_ten = lambda x: x + 10
print(add_ten(5))
```

  - a) 5
  - b) 15
  - c) Error
  - d) 25

Main function

- a special construct that check whether a Python script is being run as the main program or it is being imported into another script as a module

```
def my_function():
    print("Function was called")

if __name__ == "__main__":
    # This code block will only execute if the script is the main program.
    my_function()  # This will call the function only when running the script directly.
```

  - ‘__name__ ’
    - a special variable when a Python script is executed, Python sets the ‘__name__ ’
  - "__main__"
    - block will execute first in Python script
    - if the file is imported as a module in another script,

      the block under ‘__name__ == "__main__"’ will not execute

In class practice

- P04-01 사용자로부터 숫자 하나를 입력받고, 그 숫자가 홀수인지 짝수인지 출력하는 함수를 작성해보아라
  - input: one number by user input
  - output: print out odd or even
  - requirements
    - return type is boolean

```
def odd_even(num) -> bool:
    ''' CODE HERE '''

if __name__ == '__main__':
    num = int(input("Enter one integer: "))
    rs = odd_even(num)

    ''' CODE HERE '''
```

```
Enter one integer: 1923
1923 is odd
```

In class practice

- P04-02 정수 두 개를 입력받고, 두 수 사이의 모든 정수의 합을 출력하는 함수를 작성해보아라
  - input: three integer variables; each start, end and step by user inputs
  - output: sum between start and end numbers by step
  - requirements: define a function with these three parameters for calculation, and return result
    - if start number (first number) is less than end number (second number), take input again

```
def num_sum(start, end, step):
    ''' CODE HERE '''

if __name__ == '__main__':
    start = int(input("Enter three numbers: "))
    end = int(input("Enter end number: "))
    step = int(input("Enter step: "))
    print(f'Result is {num_sum(start, end, step)}')
```

```
Enter start numbers: 5
Enter end number: 10
Enter step: 2
Result is 21
```

In class practice

- P04-03 사용자로부터 숫자 하나를 입력받고, 해당 숫자의 구구단 표를 출력하는 함수를 작성해보아라
  - input: range of the input number; 2 to 9
    - take the input again if out of the range (assume that input is an integer)
  - output: multiplication table for the given input number
  - requirements
    - define and use a function printing out multiplication table
      - parameter: one number
      - no return value; just print it out

```
Please enter the number: 6
6 x 1 = 6
6 x 2 = 12
6 x 3 = 18
6 x 4 = 24
6 x 5 = 30
6 x 6 = 36
6 x 7 = 42
6 x 8 = 48
6 x 9 = 54
```

```
Please enter the number: -3
Please enter the correct value.
```

2. Scope and lifetime of variables

Introduction

- Background
  - so for, we defined the various variables; known as global variables
    - global variables: (potentially) accessible anywhere (or globally) in our programs
    - ➔ can result in unexpected behaviors
      - the cause of many, many bugs in all sort of programs over the years
  - make your program to be more modular code which has been proven to be easier to maintain and in fact develop and test

- Scope
  - location in the program where variables are **accessible**
- Lifetime
  - location in the program where variables **exist**

Local variables

- Declared within a function or block and is only accessible within that function or block
  - cannot be accessed or modified outside its local scope
  - example of local variable

```
def my_function():
    a_variable = 100
    print(a_variable)
'''
'''
my_function()
```

Local variables

```
def my_function():
    a_variable = 100
    print(a_variable)
'''
'''
my_function()
```

- a_variable
  - called as local variable in function
  - created and initialized to hold the value 100;
  - and be printed out to the console 100 successfully

Local variables

```
# outside my_function()
my_function()
print(a_variable)
```

- but, if we attempt to access `a_variable` outside the function
  - error will be generated (`NameError: name 'a_variable' is not defined`)
    - undefined at the top level; `a_variable` not globally defined
- `a_variable` as a new variable each time the function is called
  - the value in `a_variable` is not even seen from one invocation of the function to another

Local variables

```
def my_function():
    a_variable = 100

a_variable = 25
my_function()
print(a_variable)
```

- What is the result of the program above? 100? or 25?
    - local variable `a_variable` in function is separated with the global global `a_variable`

- the lifetime of local variable `a_variable` in `my_function()`
    - from function call to function terminated

Global keyword

- How can we refer the global variable within a function?
  - in fact, the following program works well as global variables

```
max = 100
def print_max():
    print(max)
print_max()
```

  - but this program generates error

```
def print_max():
    max = max + 1
print(max)
print_max()
```

```
Traceback (most recent call last):
  File "C:\Users\user\#2 파이썬프로그래밍\src\hello_world.py", line 4, in <module>
    print_max()
  File "C:\Users\user\#2 파이썬프로그래밍\src\hello_world.py", line 2, in print_max
    max = max + 1
UnboundLocalError: local variable 'max' referenced before assignment
```

Global keyword

- Use `global` keyword

```
max = 100

def print_max():
    global max
    max = max + 1
    print(max)

print_max()
print(max)
```

```
101
101
```

Global keyword

- What is the difference?

```
max = 100

def print_max():
    max = 100
    max = max + 1
    print(max)

print_max()
print(max)
```

```
101
100
```

Nonlocal variables

```
def outer():
    title = 'original title’

    def inner():
        title = 'another title'
        print('inner:', title)

    inner()
    print('outer:', title)

outer()
```

```
inner: another title
outer: original title
```

- Python supports to define function inside other functions
  - very useful when we are working with collections of data and operations; such as `map()`

Nonlocal variables

```
def outer():
    title = 'original title’

    def inner():
        title = 'another title'
        print('inner:', title)

    inner()
    print('outer:', title)

outer()
```

```
inner: another title
outer: original title
```

- Local variables are local to a specific function
  - even functions defined within another function cannot modify the outer functions local variables
    - as the inner function is a separate function

Global keyword

- how can `title` variable be modified in `inner()` and `outer()` as the same variable works?
  - Use `nonlocal` keyword

```
def outer():
    title = 'original title'

    def inner():
        nonlocal title
        title = 'another title'
        print('inner:', title)

    inner()
    print('outer:', title)

outer()
```

```
inner: another title
outer: another title
```

Quiz

- What is the output of the following code?

```
x = 5
def add():
    x = 3
    x = x + 5
    print(x)

add()
print(x)
```

  - a) 8 8
  - b) 5 5
  - c) 8 5
  - d) 5 8

3. Calculator implementation

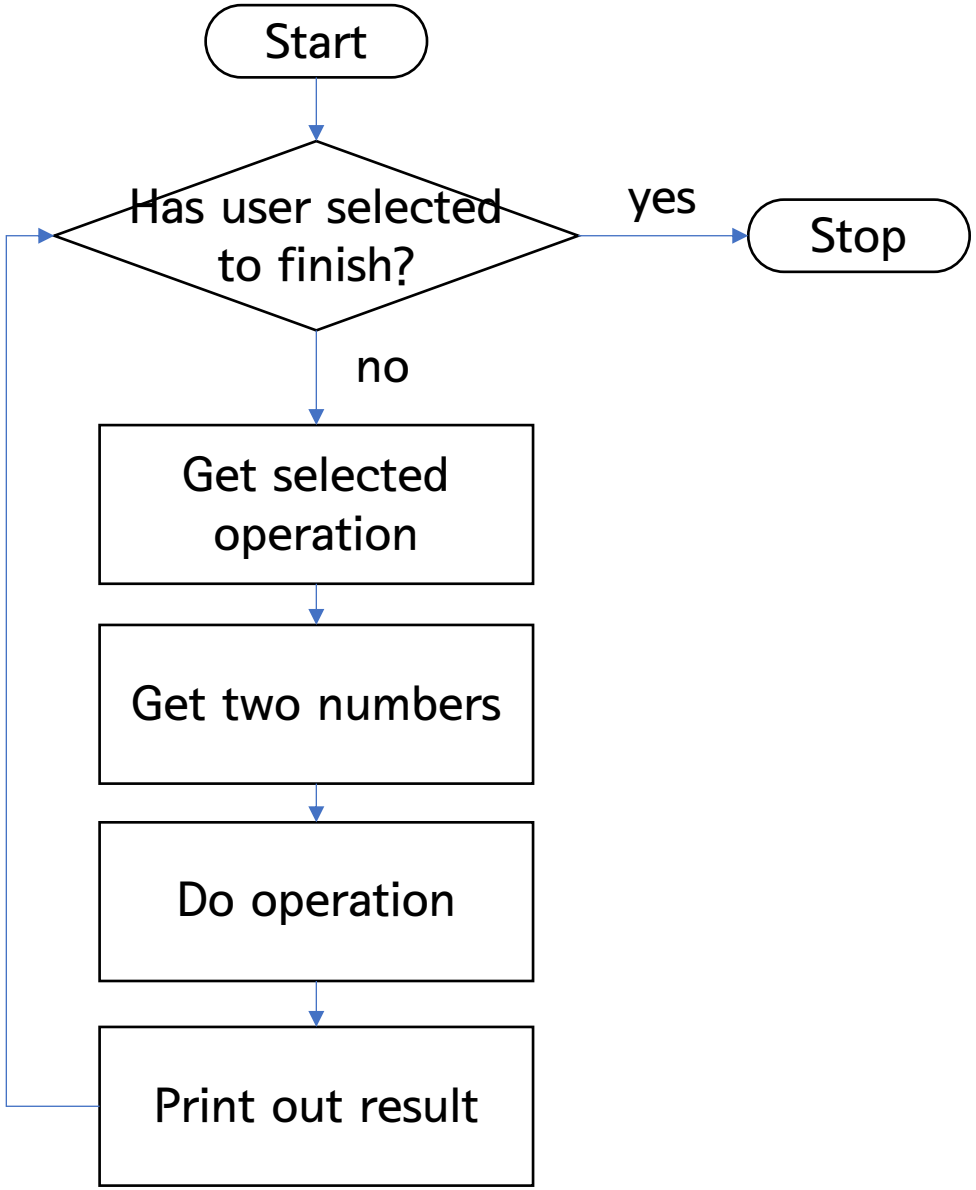
Goal

- To implement “Calculator” in Python with function to help modularize the code

- What the Calculator will do
  - to operation to perform and
  - to two numbers to use with that operation

- When the program starts up it can use a loop to keep processing operations until the user indicates that they wish to terminate the application
  - ex) termination option: ENTER ‘end’

Behavior of the calculator



Behavior of the calculator

- Skeleton code

```
if __name__ == '__main__’:

    finished = False
    while not finished:
        result = 0

        # Get the operation from the user

        # Get the numbers from the user

        # Do operation

        print('Result:', result)
        print('==================')

        # Determine if the user has finished

    print('Bye')
```

Identifying whether the user has finished

- How can we finish the program?
  - by user input? or other conditions?
  - general termination option: ENTER ‘y’ or ‘n’
  - other options
    - TYPE ‘end’
    - after 5 times executions
    - etc.

Identifying whether the user has finished

- Function for check if user has finished

```
def check_if_user_has_finished():
    """Checks that user wants to finish or not. Performs verification of the input."""

    ok_to_finish = True
    user_input_accepted = False

    while not user_input_accepted:
        user_input = input('Do you want to finish (y/n): ')

        if user_input == 'y':
            user_input_accepted = True
        elif user_input == 'n':
            ok_to_finish = False
            user_input_accepted = True
        else:
            print('Response must be (y/n), please try again')

    return ok_to_finish
```

Selecting the operation

- Skeleton code updates

```
if __name__ == '__main__’:

    finished = False
    while not finished:
        result = 0
        # Get the operation from the user

        # Get the numbers from the user

        # Do operation

        print('Result:', result)
        print('================')

        # Determine if the user has finished
        finished = check_if_user_has_finished()

    print('Bye')
```

Selecting the operation

- There are four options available to user: Add, Subtract, Multiply and Divide
  - How can input these operations from user?
    - numbers; 1, to 4
    - strings; ‘abb’, ‘subtract’, ⋯ or ‘+’, ‘-’, ⋯

Selecting the operation

- Function for operation choice

```
def get_operation_choice():
    input_ok = False

    while not input_ok:
        print('Menu Options are:')
        print('\t1. Add')
        print('\t2. Subtract')
        print('\t3. Multiply')
        print('\t4. Divide')
        print('-----------------’)

        user_selection = input('Please make a selection: ')

        if user_selection in ('1', '2', '3', '4'):
            input_ok = True
        else:
            print('Invalid Input (must be 1 - 4)')

    print('-----------------')
    return user_selection
```

Selecting the operation

- Skeleton code updates v2

```
if __name__ == '__main__':

    finished = False
    while not finished:
        result = 0
        # Get the operation from the user
        menu_choice = get_operation_choice()

        # Get the numbers from the user

        # Do operation

        print('Result:', result)
        print('================‘)

        # Determine if the user has finished
        finished = check_if_user_has_finished()

    print('Bye')
```

Obtaining the input numbers

- We need to obtain two numbers from user to use with the selected operation
  - need to ask the user for two numbers
  - and take input from user and convert it (safely) into an integer or a float type
    - if the user entered a non-number then we would prompt them to enter an actual number

Obtaining the input numbers

- Function for obtaining the two numbers from user

```
def get_numbers_from_user():
    num1 = get_integer_input('Input the first number: ')
    num2 = get_integer_input('Input the second number: ')

    return num1, num2
```

- Function for getting one integer from user

```
def get_integer_input(message):
    value_as_string = input(message)

    while not value_as_string.isnumeric():
        print('The input must be an integer')
        value_as_string = input(message)

    return int(value_as_string)
```

Selecting the operation

- Skeleton code updates v3

```
if __name__ == '__main__':
    finished = False
    while not finished:
        result = 0
        # Get the operation from the user
        menu_choice = get_operation_choice()

        # Get the numbers from the user
        n1, n2 = get_numbers_from_user()

        # Do operation

        print('Result:', result)
        print('=================')

        # Determine if the user has finished
        finished = check_if_user_has_finished()

    print('Bye')
```

Do operation

- Determining the operation to execute

```
if menu_choice == '1':
    result = n1 + n2
elif menu_choice == '2':
    result = n1 - n2
elif menu_choice == '3':
    result - n1 * n2
elif menu_choice == '4':
    result = n1 / n2
```

Do operation

- Skeleton code updates v4

```
if __name__ == '__main__':
    finished = False
    while not finished:
        result = 0
        menu_choice = get_operation_choice()

        n1, n2 = get_numbers_from_user()

        if menu_choice == '1': result = n1 + n2
        elif menu_choice == '2': result = n1 - n2
        elif menu_choice == '3': result - n1 * n2
        elif menu_choice == '4':result = n1 / n2

        print('Result:', result)
        print('================')

        finished = check_if_user_has_finished()

    print('Bye')
```

Running the calculator

```
Menu Options are:
  1. Add
  2. Subtract
  3. Multiply
  4. Divide
-----------------
Please make a selection: 5
Invalid Input (must be 1 - 4)
Menu Options are:
  1. Add
  2. Subtract
  3. Multiply
  4. Divide
-----------------
Please make a selection: 1
-----------------
Input the first number: 5
Input the second number: 4
Result: 9
=================
Do you want to finish (y/n): y
Bye
```

In class practice

- P04 Implement the calculator
  - the same input and output
  - more requirements
    - print the error message when to be divided by zero in “divide” operation
      - not terminated; just print error and get the number again

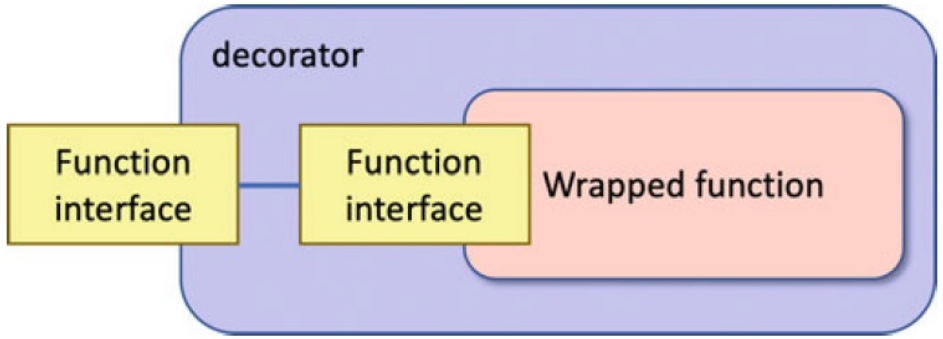
4. Decorators

What is a decorator?

- A function that takes another function as an argument
  - can extend or enhance the behavior of that function without changing its source code
  - to mark a callable object
    - function, method, class, or object
  - *decorates* the original behavior

What is a decorator?

- Basic idea as following diagram:



  - a decorator wrapping a callable object
    - in this case a function
  - presents exactly the same interface as the original function would present
    - takes the same parameters and either return noting (None) or something

Defining a decorator

- To define a callable object such as a function
  - takes another function as a parameter and returns a new function
  - a very simple example

```
def logger(func):
    def inner():
        print('calling ', func.__name__)
        func()
        print('called ', func.__name__)

    return inner
```

    - `logger` decorator wraps the original function within a new function; call `inner`
      - when this function is executed, a statement is logged before and after the original function is executed

Using decorators

- Define a target function

```
def target():
    print('In target function')

'''
'''
t1 = logger(target)
t1()
```

```
calling target
In target function
called target
```

  - can explicitly apply the logger decorator to this function by passing the reference to the target
    - we actually execute the `inner()` function returned by the decorator

Using decorators

- More typical format of decorator's usage

```
@logger
def target():
    print('In target function')

target()
```

```
calling target
In target function
called target
```

  - decorator is declared by using '@' syntax (more general form)

Functions with parameters

- Can be applied with parameters

```
def logger(func):
    def inner(x, y):
        print('calling ', func.__name__, 'with', x, 'and', y)
        func(x, y)
        print('returned from ', func.__name__)
    return inner

'''
'''
@logger
def my_func(x, y):
    print(x, y)
my_func(4, 5)
```

```
calling  my_func with 4 and 5
4 5
returned from  my_func
```

Functions with parameters

- Useful to check a processing time without modifying the code (improving readability)

```
import time

def timing_decorator(func):
    def wrapper(*args, **kwargs):
        start_time = time.time()
        result = func(*args, **kwargs)
        end_time = time.time()
        print(f"Execution time: {end_time - start_time} seconds")
        return result
    return wrapper

@timing_decorator
def slow_function():
    for i in range(0, 100000000):
        pass # your code for iteration

slow_function()
```

```
Execution time: 1.4814832210540771 seconds
```

End of slide