CA6000 Applied AI Programming

NANYANG TECHNOLOGICAL UNIVERSITY SINGAPORE

College of Computing and Data Science (CCDS)

Associate Professor Nicholas Vun

aschvun@ntu.edu.sg

Functions

Function

A function is a block of code which only runs when it is called

- make the code reusable

We have been using some built-in functions, e.g. `print()`, `input()`, `len()`, `sorted()`, `enumerate()` etc.

You can pass data (known as arguments) into a function.

Example: the `print()` function:

```
name = ”Nick”
print(“Hello”, name)    #this has two arguments
print(“Hello ” + name) #this has one argument
```

print() variations (formatting)

print() with formatting enables more readability:

```
   print(“Hello”, name, “, how are you?”)
vs
   print(f“Hello {name}, how are you?”)
```

We can also specify additional argument to modify the print() behaviour:

E.g. Changing the end of line:

```
print(“Hello”, end=“, ”)    #instead of default \n
print(“how are you today?”)
```

Changing the separator in a sentence:

```
print("Brand", "Honda", sep=":")
```

Creating a Customized Function

But it will be more useful if we can create our own customised function

- which can be done by defining new function

Function is defined by using the keyword `def`

```
def my_function():
    print("Hello from a function")
```

Indentation to define the scope of the function

You can then call the function in your program (i.e., the caller) by its name

```
def my_function():
    print("Hello from a function")

my_function() #this is the ‘caller’ to the function
```

Naming Convention

You can use letters, the digits 0 to 9, and the underscore (_) for a function name.

Best practice: use descriptive names for function

- names should describe the actions being performed by the function clearly and concisely whenever possible

Guidelines for naming function using snake_case convention

- all lowercase letters and with an underscore separating words
  E.g. `get_input()` (or `getInput()` which is the camel_case)

Aside: a single leading underscore, such as `_calculate_sum()` indicates that the function is meant only for internal use within a module or a class (see later).

Passing data into the function

You can pass data to the function as argument

- specified inside the parentheses

parameter

```
def greeting(to):
    print(f"Hello {to} ! ")

name = input("What is your name?\n")
greeting(name)
```

argument

Argument: data that you send to a function (the variable `name` when calling the function)

Parameter: data that is received by the function (`to`)

Passing data into the function

You can send any data type as argument to a function

```
def type_of_food(food):
    for x in food
        print(x)

fruits = ["Apple", "Banana", "Durian"] #passing a list
type_of_food(fruits)
```

Note: A function that does not return any value is also called a Procedure

Default and Multiple Parameters

You can specify default parameter(s) to be used:

```
def greeting(to = "World"):
    print(f"Hello {to} ! ")
greeting()
greeting("Nick")
```

Function can take multiple parameters

```
def greeting(msg = "Hello", to = "world"):
    print(f"{msg}, {to} ! ")


greeting()
name = input("What is your name?\n")
greeting("Good day", name)
```

Variable number of Parameters

If you do not know beforehand the number of parameters that will be received

- use *parameter

```
def greeting(*names):
    for name in names:
        print(f"Good day {name} !")

greeting("Nick", "Jess", "Alex")
```

Sharing a variable: `global`

Instead of passing data through parameter(s)

- you can share a global variable among multiple functions, using the `global` keyword

```
 RUN = 0
 STOP  = 1
 game = RUN

 def check():
    global game  # ‘game’ is a global variable
       :
    game = STOP


main():
   while(game == RUN): # still in play
       :
     check()
```

Aside: Variable Scope

Depending on where a variable is defined in the code

- it can have different values with the same name
- this is known as the scope of the variable:

  Local, Enclosing(nonlocal), Global (and Build-in).

Try the following code and execute the function `outer()`

```
x = 'global'
def outer():
    print(x)
```

```
x = 'global'
def outer():
    x = 'enclosing'
    def inner():
        print(x)
    inner()
```

```
x = 'global'
def outer():
    x = 'enclosing'
    def inner():
        x = 'local'
        print(x)
    inner()
```

Order of definitions

We usually want to abstract (i.e. hide) the detail of the functions from the main program logic when design program

```
def main():
    greeting_1("Nick")
    greeting_2("Alex")

def greeting_1(name):
    print(f"Hello, {name} !")

def greeting_2(name):
    print(f"Good Day, {name} !")

main()
```

Returning data from the function

It is very common for function to return data (i.e. result) to the caller after performing the computation

- use the `return` statement

```
def main():
    answer = multiply(5, 10)
    print(answer)

def multiply(para1, para2):
    value = para1*para2
    return value

main()
```

Returning with formatting

We can also return a formatted string

- similar to print with formatting

```
def introduce(name, age):
    return f"Hello, I am {name}, and I am {age} \
                 years old."

message = introduce(‘Nick’, 30)
print(message)

# Output: Hello, I am Nick, and I am 30 years old
```

Type Annotations/Hints

To increase the readability of your code, you can also (optionally) indicate in a function definition

- the intended data type of the function parameters
- the data type of the return value

Example:

```
def my_funct(a:str, b:int) -> float:
     :
```

- the annotation `:str` and `:int` indicate that the caller should pass a `str` value and an `int` value as arguments
- the annotation `-> float` indicates that the function will return a `float` result

Docstring with `help()` function

Python built-in `help()` function prints out the documentation of a function describe using docstring

```
def calculate_area(length:float, width:float) -> float:
    """
    Calculates the area of a rectangle.
    Args:  length (float): The length of the rectangle.
           width (float): The width of the rectangle.
    Returns: float: The calculated area of the rectangle.
    """
    return length * width

help(calculate_area) # accessing its docstring using help()

calculate_area(7.5, 3.5) # call the function
```

Generator function using `yield`

Generators provide an elegant way to work with big data set or time sequence

- functions use the yield keyword which return values one at a time without terminating the function
  - compute and return values on-the-fly as they are requested
- improve code efficiency by reducing the memory needed
  - especially useful for processing large sequences

```
import time  #import a python library/module
def main():
    for value in generator():
        print(value)
        time.sleep(1)     #pause for 1 second

def generator():
    yield “Alex”
    yield “Brian”
    yield “Cindy”

main()
```

Lambda Function

Lambda function allow us to implement a function without having to separately define it first

- i.e. an anonymous function
- come in handy to write concise code
  - a function that will only be used once
  - a function that contains simple statements

Lambda functions consist of only three parts:

1. the keyword: `lambda`
2. placeholder to hold the value(s) to be passed to the expression
3. the expression

Example:

```
def mac(x, y):
    return (x * y + x)

print(mac(3,4))
```

can be implemented using lambda :

```
mac =(lambda x, y: (x*y)+x)
print(mac(3,4))
```

Aside: Lambda for List filtering

List has a `filter()` function that can be used for filtering a list to select elements that satisfy certain criteria

- it takes two arguments
  - a function
  - an iterable
- returns an iterator with the elements for which the function returns True.

We can pass the lambda as the argument to the filter() function:

```
number_list = [1, 12, 13, 24, 35, 38, 47]
odd_numbers = filter(lambda x: x % 2 != 0, number_list)
print(list(odd_numbers)) #convert to list type
```

Decorator Function

The symbol @ is known as decorator

- a function that takes another existing function as its argument
- to extend the functionality of the existing function.

Example:

```
def divide(x, y):
    return x/y
```

But what happen if y = 0?

- use a decorator function: e.g., guard_zero()

```
def guard_zero(func):
    def wrapper(x, y): #process the parameters of the function
        if y == 0:
            print("Cannot divide by 0.")
            return
        return func(x, y) #execute the function
    return wrapper

@guard_zero  #run this first when divide() is called
def divide(x, y): #function divide()
    return x/y
```

Summary

Functions allow us to reuse code in various parts of the program

- it also makes the code development process easier to manage

There are many built-in functions available in Python

- but we can define our own function which is more useful in practice

There are certain special type of functions that we may use in AI applications

- Generator function
- Lambda function
- Decorator function