KIT100 PROGRAMMING PREPARATION

Lecture Six:

Defining and calling functions

Lecture Objectives

- Introduction to Functions
- Defining and Calling a *Void* Function
- Passing arguments / parameters / inputs to Functions
- Writing your own Value-Returning Functions

Creating and using functions

- To manage data properly, we need to organise the tools used to perform the required task.
- Each line of code we create performs a ***specific*** task, and we combine one or more lines of code together to achieve the solution to our problem.
- Why we have to use functions?
  - Sometimes we will need to repeat the sequence of instructions (with **different data**), and in some cases our code can become very long – then it's hard to keep track of what each part does.
    - For example, you have used the 'print' / 'input' function (you may not see the function code behind the python interface.)

Creating order out of mess!

- If our program has many lines of repetitive code, it can make it hard to read (for someone looking at the source code), and it increases the likelihood of errors (more code = more errors).
- **Functions** take sequences of logically-grouped code that achieves a certain task and packages/groups it in a single unit to:
  - make it easier to see and understand what the code does
  - make the code more efficient
  - make the code less error-prone, and more reusable

Understanding code reusability

- Functions are **re-usable**.
  - No one wants to keep repeating the same task, how boring!

- When we create a function, we define a section of code that we can re-use over and over to perform the same task.

- All we need to do is tell the computer to perform a specific task by telling it ("calling") what function to use.
  - "calling": the interpreter remembers where the calling code was, goes and executes the function's code, and then comes back to resume execution after the calling code.

Why reuse code?

- Reduce development time
- Reduce programmer error
- Increase application reliability
- Make code easier to understand
- Improve application efficiency
- Stop reinventing the wheel over and over – <u>solve it once and reuse</u>!

Working with functions

- The code that **needs** services from the function is named the **caller**, and it **calls** upon the function to perform tasks for it.
- The caller must supply information (data) to the function, and the function gives back (**returns**) information to the caller.

the call…

the data

e.g.

Caller: "*hey add function, here are two numbers I'd like you to use*"

"add" Function: "I've added the two numbers, here's the result.."

the return value

Void functions and value-returning functions

- A **void function**:
  - Simply executes the statements it contains and then terminates. No data is returned to the caller.
- A **value-returning function**:
  - Executes the statements it contains, and then it returns a value back to the statement (the caller) that called it.
    - The **`input`**, **`int`**, and **`float`** functions are examples of value-returning functions. What ***type*** of data do they return?

```
x = int("3")
```

data given to the int function, a string. This data is called a parameter (or argument) to the function.

data returned by the int function, an integer, here stored in variable x

'print' function?

– void function (no calculation)

How about 'print' function?

Defining and Calling a Function

- Functions are given names
  - **Function naming rules**:
    - **Cannot** use keywords as a function name
    - **Cannot** contain spaces
    - The first character **must** be a letter or underscore
    - All other characters **must** be letters, numbers or underscores
    - Uppercase and lowercase characters are distinct

Good programming style – function names should **start** with a **lowercase letter.**

```
goodName()  BadName()
```

Defining and Calling a Function

- A function name should be *descriptive* of the task carried out by the function i.e. *what does the function do?*
  - Often includes a verb
- **Function definition**: specifies the code contained in the function:

definition

```
def function_name():        def functionName():
    statement                   statement
    statement                   statement
```

indent

both naming styles are ok..

- Function definitions generally should occur near the top of your source code.
- The interpreter when it sees the definitions doesn't execute them, it merely notes (and verifies syntax) where the code is if you happen to call it later.

Defining and Calling a Function

- **Function header**: the first line of function
  - Includes the keyword `def` and the function name, followed by parentheses/brackets and colon

- **Block**: a set of indented statements that belong together as a group

```
def function_name():
  indent → statement
  indent → statement
```

Defining a function

```
def firstFunction():
```

This tells Python to define a function named `firstFunction`.

The brackets are **required** as they define any parameters (data) for using the function (in this example there is no data).

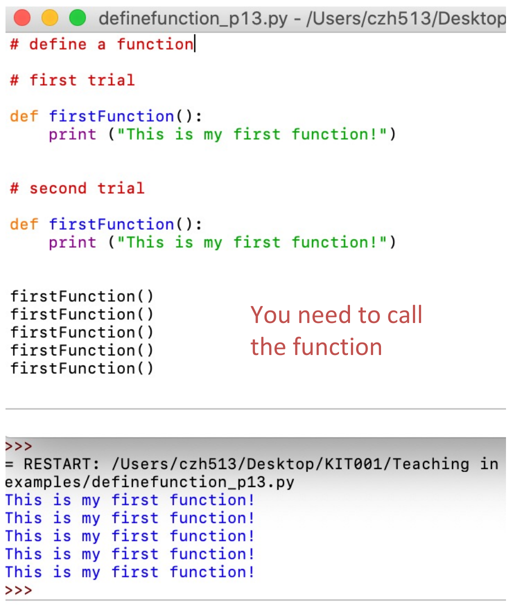
The prompt/colon at the end is **required** and it is important, as it tells Python we have finished defining the way people will ***access*** the function, i.e. what the function's identifier (name) is, and what data it expects to be given.

Defining a function

```
def firstFunction():
    print ("This is my first function!")
```

Try typing this code into a new source code document and then run it.

Does anything happen?



definefunction_p13.py - /Users/czh513/Desktop

```
# define a function

# first trial

def firstFunction():
    print ("This is my first function!")


# second trial

def firstFunction():
    print ("This is my first function!")


firstFunction()
firstFunction()
firstFunction()
firstFunction()
firstFunction()
```

You need to call the function

```
>>>
= RESTART: /Users/czh513/Desktop/KIT001/Teaching in
examples/definefunction_p13.py
This is my first function!
This is my first function!
This is my first function!
This is my first function!
This is my first function!
>>>
```

All we have done is specify what the function does. It doesn't require any data to be given to it, nor does it return any data. Python doesn't actually run the code until we ask it to (i.e. call the function)

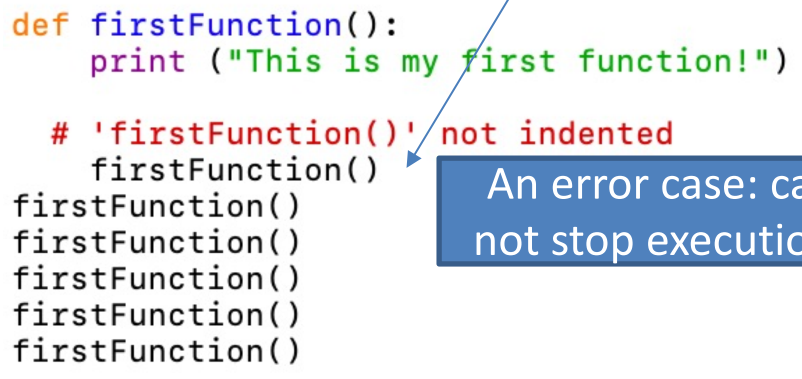
Calling (*using*) functions

- After we have *defined* a function, we will probably want to use it! We call a function to execute it. When a function is called:
    1. The interpreter jumps to the function and executes statements in the block
    2. The interpreter jumps back to part of the program that called the function

       – *This is known as a function return*

To access (use) the function we defined on the previous slide, we type the following (after the function definition, and **not indented**!)

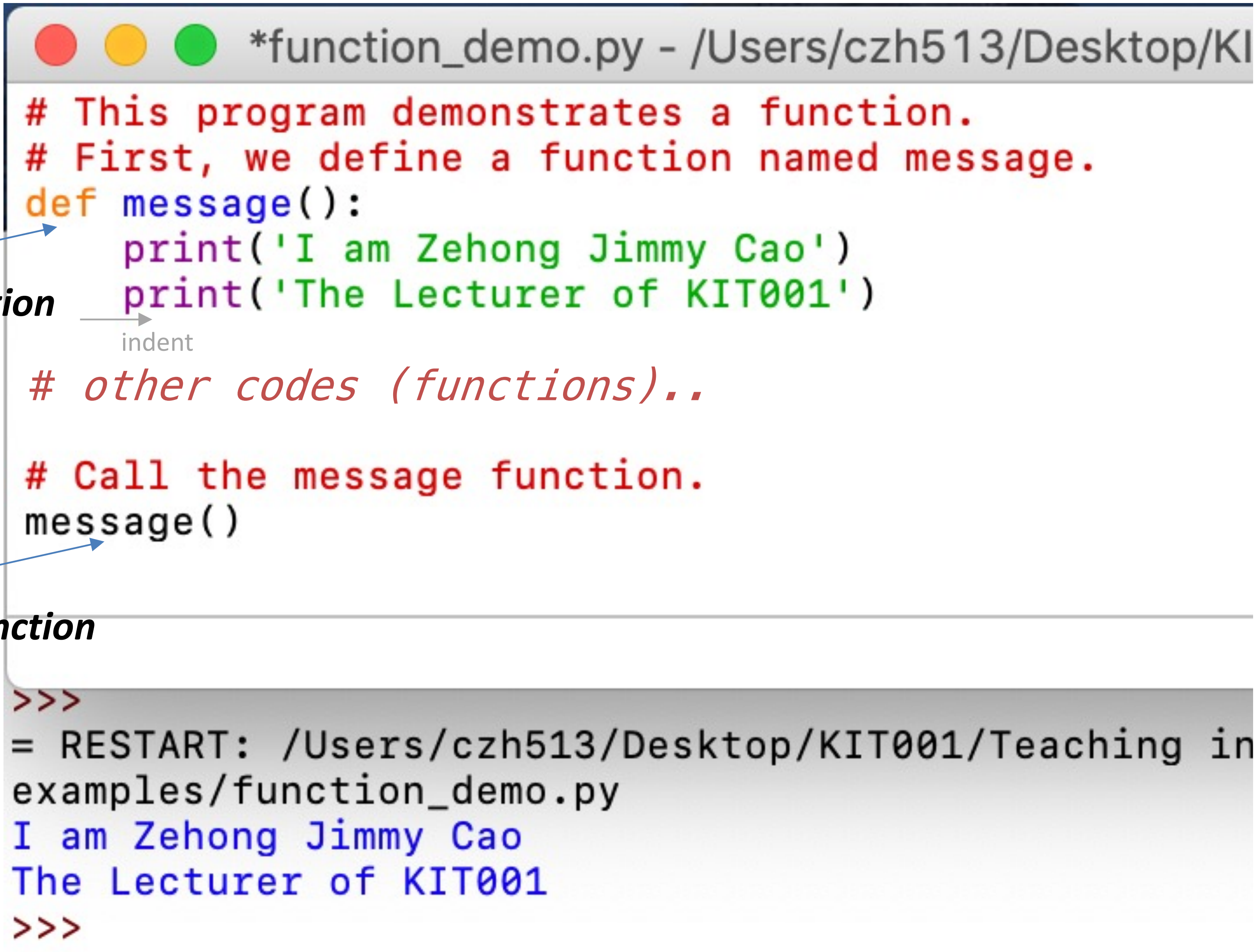
`firstFunction()`



```
def firstFunction():
    print ("This is my first function!")

  # 'firstFunction()' not indented
    firstFunction()
firstFunction()
firstFunction()
firstFunction()
firstFunction()
firstFunction()
```

An error case: can not stop execution!

Accessing (using) Functions



```
# This program demonstrates a function.
# First, we define a function named message.
def message():
    print('I am Zehong Jimmy Cao')
    print('The Lecturer of KIT001')

# other codes (functions)..

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

Define the function

Use (call) the function

Output:

```
>>>
= RESTART: /Users/czh513/Desktop/KIT001/Teaching in
examples/function_demo.py
I am Zehong Jimmy Cao
The Lecturer of KIT001
>>>
```

Defining and calling a function

- `main` **function**: this is usually called when the program starts
  - Calls other functions when they are needed
  - Defines the *mainline logic* of the program

  - Your programs up to now have simply started to execute statements when they are found. Other languages require a main function to be defined and then called. It makes it obvious where the program starts.

Defining and calling a function

Define what to do when the program starts →

call other functions if needed

```
def main():
    print ("This is where the program starts!")
    otherFunction()
```

Define other functions →

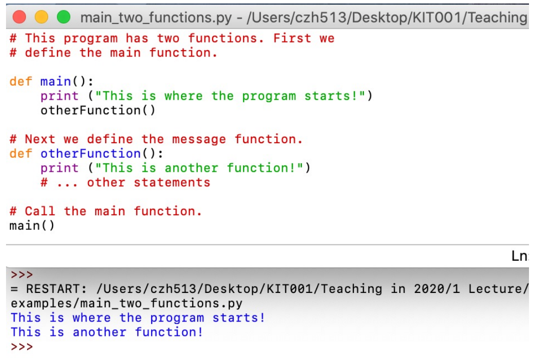
```
def otherFunction():
    print ("This is another function!")
    # ... other statements
```

Start (call/use) the main function →

```
main()
```

Execution starts here!

```
Output:
This is where the program starts!
This is another function!
```



```
# This program has two functions. First we
# define the main function.

def main():
    print ("This is where the program starts!")
    otherFunction()

# Next we define the message function.
def otherFunction():
    print ("This is another function!")
    # ... other statements

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

```
>>>
= RESTART: /Users/czh513/Desktop/KIT001/Teaching in 2020/1 Lecture/
examples/main_two_functions.py
This is where the program starts!
This is another function!
>>>
```

main_two_functions.py - /Users/czh513/Desktop/KIT001/Teaching
Ln

Sending information to functions

- Functions should be flexible and allow you to alter their behavior (for example, their output or return value) each time we use them by providing them with different data.
- The **data** we give to a function are known as **arguments** (also known as ***parameters, inputs***).
- Basically an argument / parameter / input makes it possible for you to send data to the function so that the function can use it when performing a task.

Understanding arguments / parameters / inputs

Task: we are naming the argument the function expects to given to be called greeting here.

Define the function:

```
def hello(greeting):
    print(greeting)
```

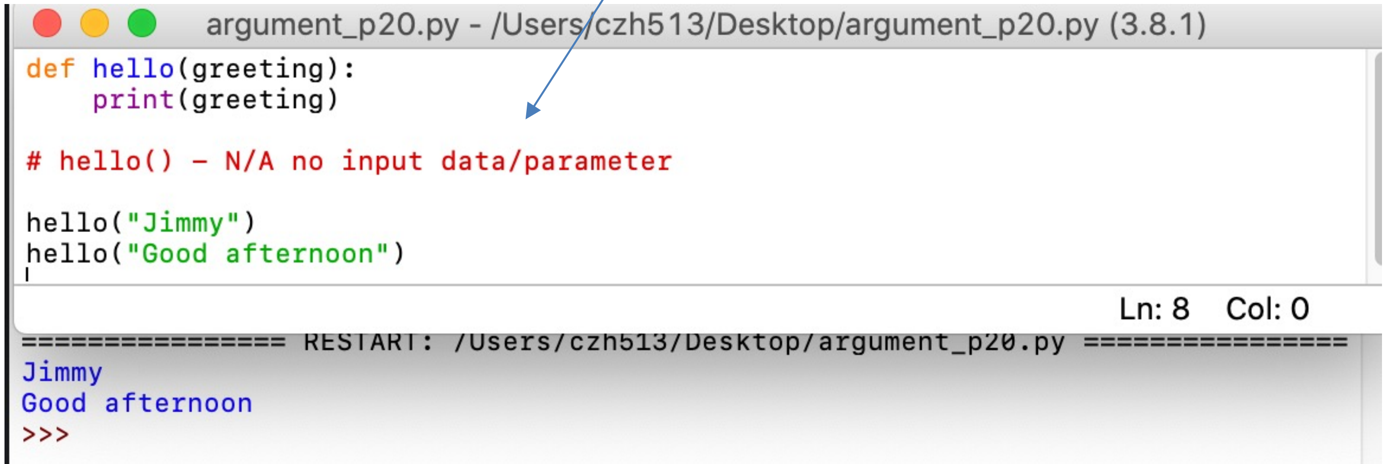
- *Think of it as a variable called greeting that gets assigned some data (by the caller) when the function is called.*
- *Note Python doesn't specify what **type** of data is placed in greeting..*

Sending required arguments

Call the function:

```
hello()

Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: hello() missing 1 required positional argument: 'greeting'
```

We have got an error, as the **`hello`** function *definition* on the previous slide says it expects to be given a value for the **`greeting`** *parameter*. The call here didn't provide any data, so **`greeting`** is undefined!



argument_p20.py - /Users/czh513/Desktop/argument_p20.py (3.8.1)

```
def hello(greeting):
    print(greeting)

# hello() - N/A no input data/parameter

hello("Jimmy")
hello("Good afternoon")
```

Ln: 8   Col: 0

```
================ RESTART: /Users/czh513/Desktop/argument_p20.py ================
Jimmy
Good afternoon
>>>
```

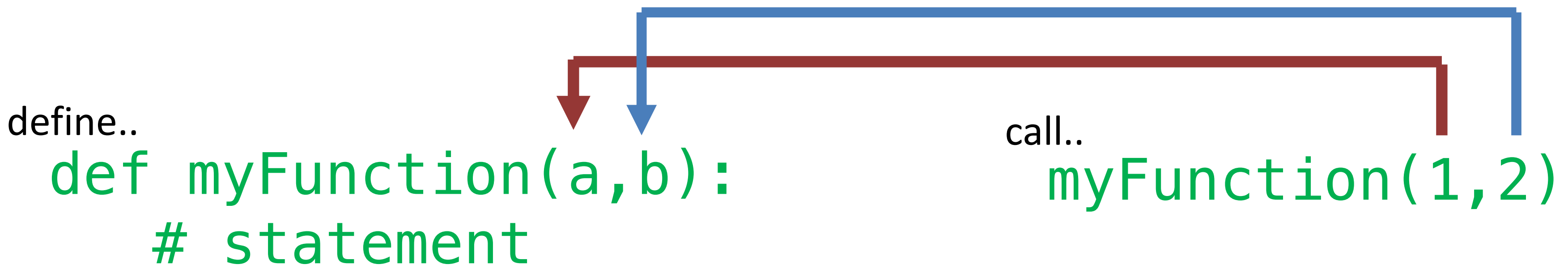
Sending information to functions

*Note Python doesn't specify what **type** of data is placed in greeting in the previous slide..*

```
hello("This is an interesting function...")     #Type: String
Output: This is an interesting function...

hello(True)    #Type: Boolean
Output: True

hello(5+5)     #Type: Integer
Output: 10
```

Passing multiple arguments

- Python allows writing functions that accept multiple arguments
  - A *parameter list* replaces the single parameter
    - The parameter list items are separated by commas
- Arguments are passed *by* ***position*** to the corresponding parameters
  - The first parameter receives value of the first argument, the second parameter receives value of the second argument, etc.



define..

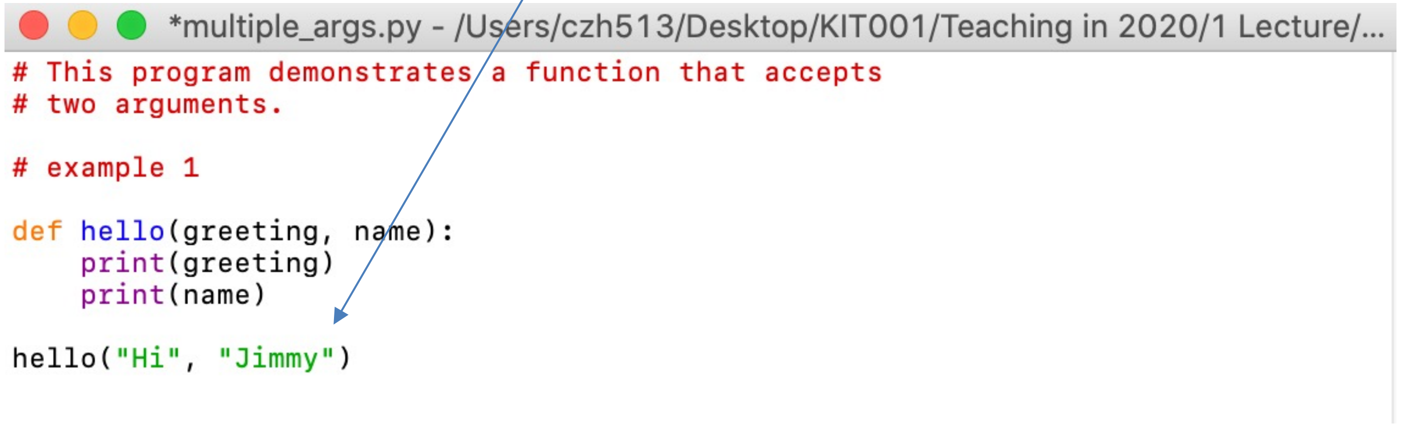
```
def myFunction(a,b):
    # statement
```

call..

```
myFunction(1,2)
```

Making changes to parameters

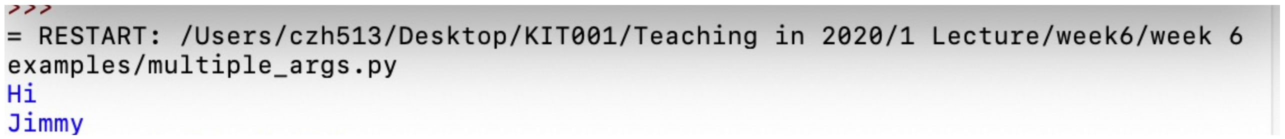
- Changes made to a parameter value *within* the function do not affect the argument in the calling code
  - This is known as *pass by value*
  - It provides a way for unidirectional (one-way) communication between one function and another function
    - Calling function can communicate with called function



*multiple_args.py - /Users/czh513/Desktop/KIT001/Teaching in 2020/1 Lecture/...

```python
# This program demonstrates a function that accepts
# two arguments.

# example 1

def hello(greeting, name):
    print(greeting)
    print(name)

hello("Hi", "Jimmy")
```

Output



```
>>>
= RESTART: /Users/czh513/Desktop/KIT001/Teaching in 2020/1 Lecture/week6/week 6
examples/multiple_args.py
Hi
Jimmy
```

Making changes to parameters

```
# example 2

def main():
    print('The sum of 12 and 45 is')
    print(show_sum(12, 45))

# The show_sum function accepts two arguments
# and displays their sum.
def show_sum(num1, num2):
    result = num1 + num2
    return result

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

Arguments are passed *by* ***position*** to the corresponding parameters

Output

```
The sum of 12 and 45 is
57
```

```
# example 3
def appendFlag(target, value):
    target += value
    return target

t=appendFlag('KIT','001')
print(t)
```

Output

```
KIT001
```

Sending more than one argument

Define the function:

```
def addIt(value1, value2):
   print(value1,"+",value2,"=",value1+value2)
```

Use (call) the function:

```
addIt(3,4)
```

```
Output:3 + 4 = 7
```

Note: when we **use** **`addIt`** we have to give it **two** arguments as the definition shows it expects **two** values (which it names as the variables *`value1`* and *`value2`*).

*The result of calling (using) the function above says "go and run the code for **`addIt`**, and give the variable `value1` a value of 3, give the variable `value2` a value of 4.*

Giving the function arguments a default value

- So far, our functions have required that one or more values are supplied as arguments / parameters / inputs, when we use the function (if the function definition says it needs values).

- Sometimes a function can use **default values** for parameters,
  - meaning when we use the function, we don't have to provide some of the data.

- **Default values** make the function easier to use and less likely to cause errors if an input hasn't been supplied.

Functions with default values

*This **assignment** here only occurs if no value for greeting was provided*

Define the function:

```
def hello2(greeting = "No value supplied!"):
    print(greeting)
```

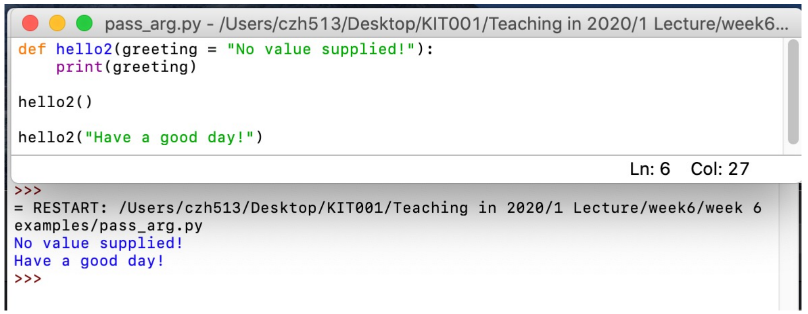
Use (call) the function:

We didn't provide any data to the function, so the default value is used inside the function

```
hello2()
```
Output: `No value supplied!`

```
hello2("Have a good day!")
```
Output: `Have a good day!`

Here we are overriding the default value of the greeting parameter


pass_arg.py - /Users/czh513/Desktop/KIT001/Teaching in 2020/1 Lecture/week6...

```
def hello2(greeting = "No value supplied!"):
    print(greeting)

hello2()

hello2("Have a good day!")
```

Ln: 6   Col: 27

```
>>>
= RESTART: /Users/czh513/Desktop/KIT001/Teaching in 2020/1 Lecture/week6/week 6
examples/pass_arg.py
No value supplied!
Have a good day!
>>>
```

Two types of functions

Now you know,

- **void function**: group of statements within a program for performing components of a specific task – no data is returned to the caller
  - Call the function when you need to perform the task

- **value-returning function**: like a void function, but it **returns** a value to the caller
  - The value is returned to the part of program that called the function (when function finishes executing)

How to use value-returning functions

- A value-returning function can be useful in specific situations
  - Example: have a function to prompt the user for input and return the user's input to the caller
- **How to use the returned value**:
  - **Assign it to a variable** or use it as an argument in another function

Examples:

```
1 name = input("Enter your name")
```

1 Assign the returned value from the `input` function to a variable

```
2 number = int(input("Enter a number:"))
```

2 Use the returned value from the `input` function as an argument to the `int` function (covert all input numbers to the integer type)

Function example: returning strings

- You can write functions that **return strings**
- For example:

```
def get_name():
    # Get the user's name.
    name = input('Enter your name: ')
    # Return the name.
    return name
```

The function returns whatever data the user has entered via the input function.

A more complex function might verify what the user has entered, and (repeatedly) prompt them to enter correctly validated data before it returns.

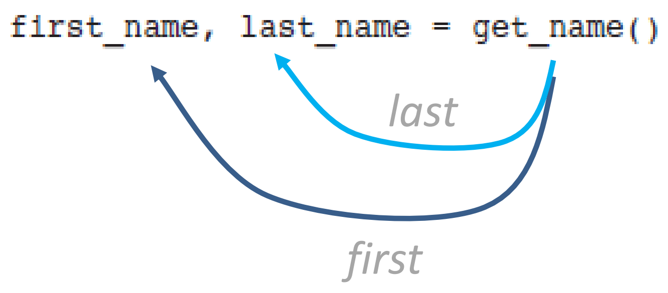
Returning multiple values

- In Python, a function can even **return** **multiple values**
  - The values are specified after the `return` statement, separated by commas
    - Format: ***`return expression1,expression2`*** *etc.*
  - When you call such a function in an assignment statement, you need separate variables on the left side of the = operator to receive **each** returned value (in order)

1. Define the function:

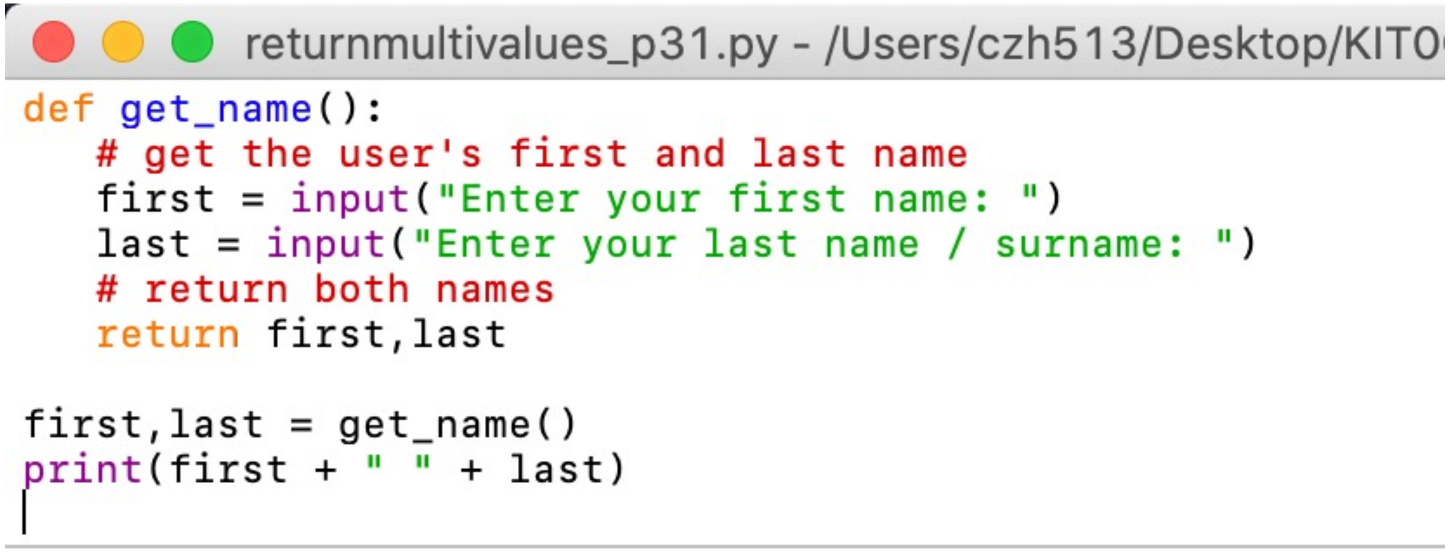
```
def get_name():
    # Get the user's first and last names.
    first = input('Enter your first name: ')
    last = input('Enter your last name: ')

    # Return both names.
    return first, last
```

2. Call the function:



```
first_name, last_name = get_name()
```

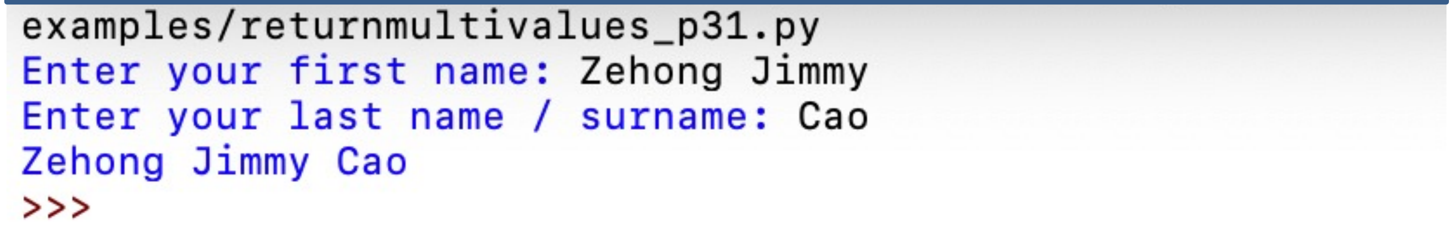
Returning multiple values



returnmultivalues_p31.py - /Users/czh513/Desktop/KITO

```python
def get_name():
    # get the user's first and last name
    first = input("Enter your first name: ")
    last = input("Enter your last name / surname: ")
    # return both names
    return first,last

first,last = get_name()
print(first + " " + last)
```

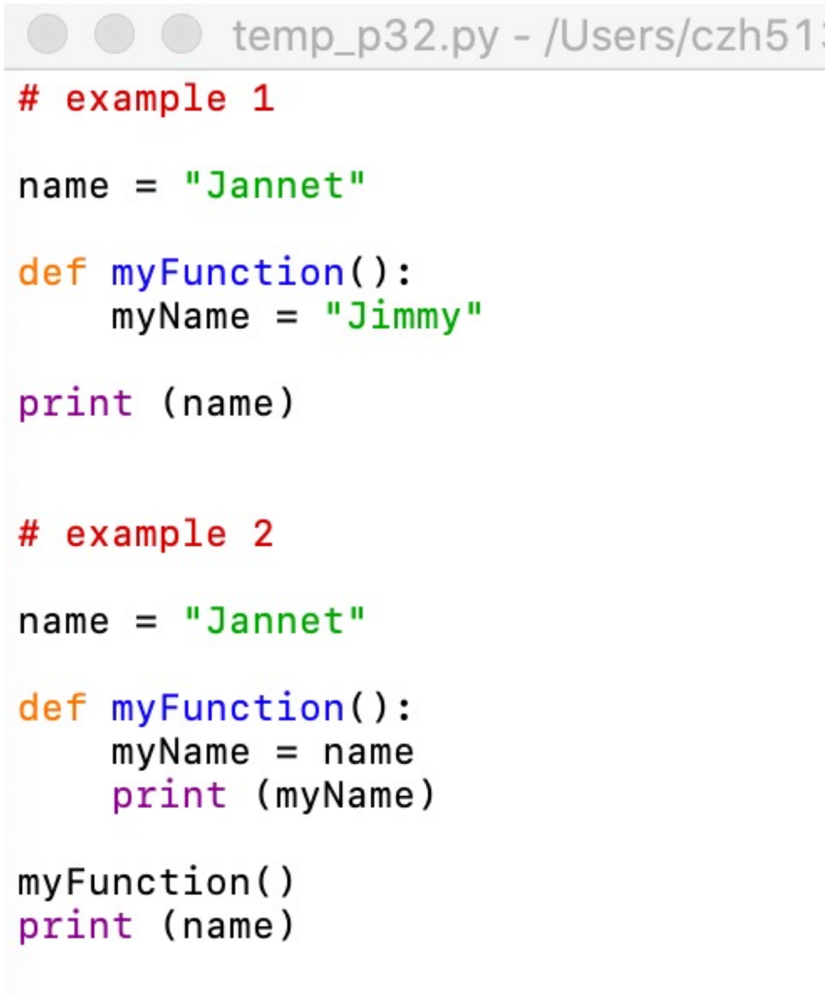
Output



```
examples/returnmultivalues_p31.py
Enter your first name: Zehong Jimmy
Enter your last name / surname: Cao
Zehong Jimmy Cao
>>>
```

More examples

1. Pass by values

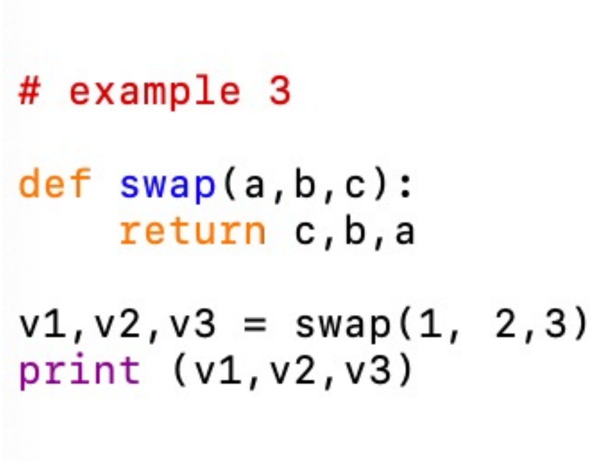


temp_p32.py - /Users/czh51

```
# example 1

name = "Jannet"

def myFunction():
    myName = "Jimmy"

print (name)


# example 2

name = "Jannet"

def myFunction():
    myName = name
    print (myName)

myFunction()
print (name)
```

What is the output?

2. Return values



```
# example 3

def swap(a,b,c):
    return c,b,a

v1,v2,v3 = swap(1, 2,3)
print (v1,v2,v3)
```

More examples

3. Pass to a main function

```python
# example 4 – an argument being passed to a main function.

def main():
    value = 5
    show_double(value)

# The show_double function accepts an argument
# and displays double its value.
def show_double(number):
    result = number * 2
    print(result)

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

```python
# example 5 – pass to a main function.

def main():
    # Get the user's age.
    first_age = int(input('Enter your age: '))

    # Get the user's best friend's age.
    second_age = int(input("Enter your best friend's age: "))

    # Get the sum of both ages.
    total = sum(first_age, second_age)

    # Display the total age.
    print('Together you are', total, 'years old.')

# The sum function accepts two numeric arguments and
# returns the sum of those arguments.
def sum(num1, num2):
    result = num1 + num2
    return result

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

What is the output?

Questions?