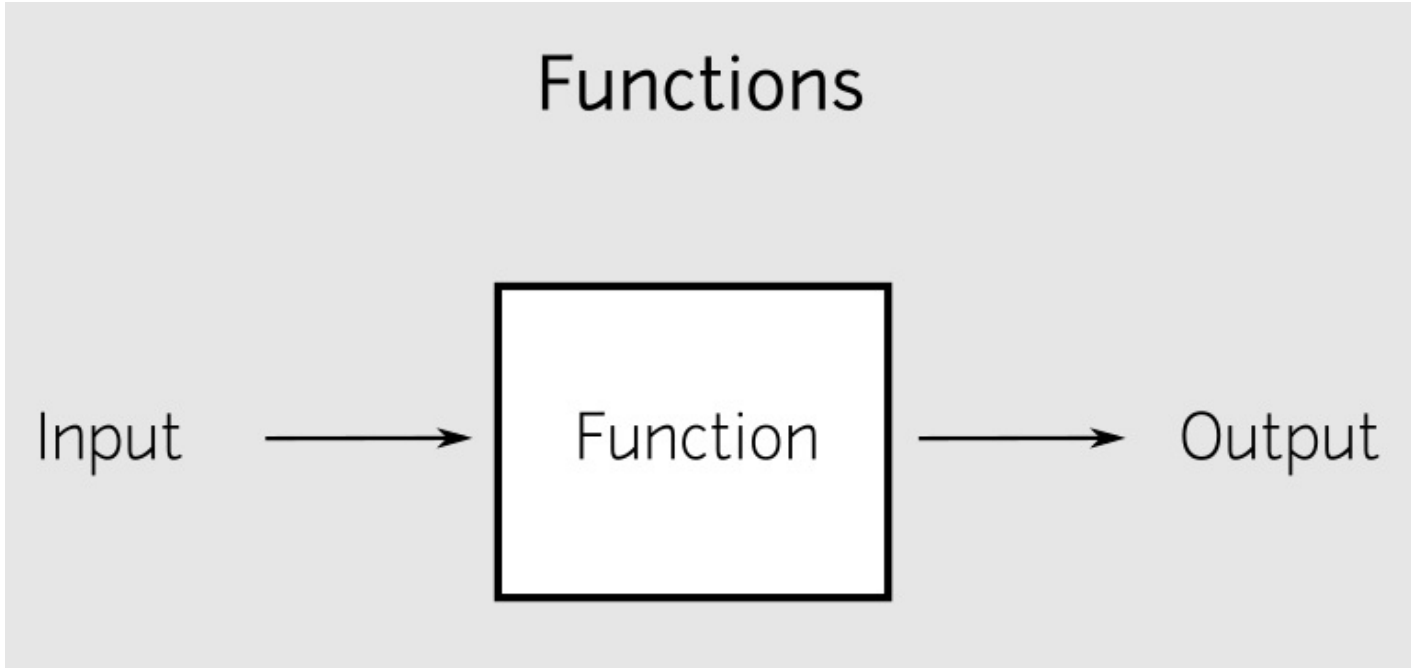
Module 5

Functions

Instructor: Jonny C.H Yu

Function: The Basics

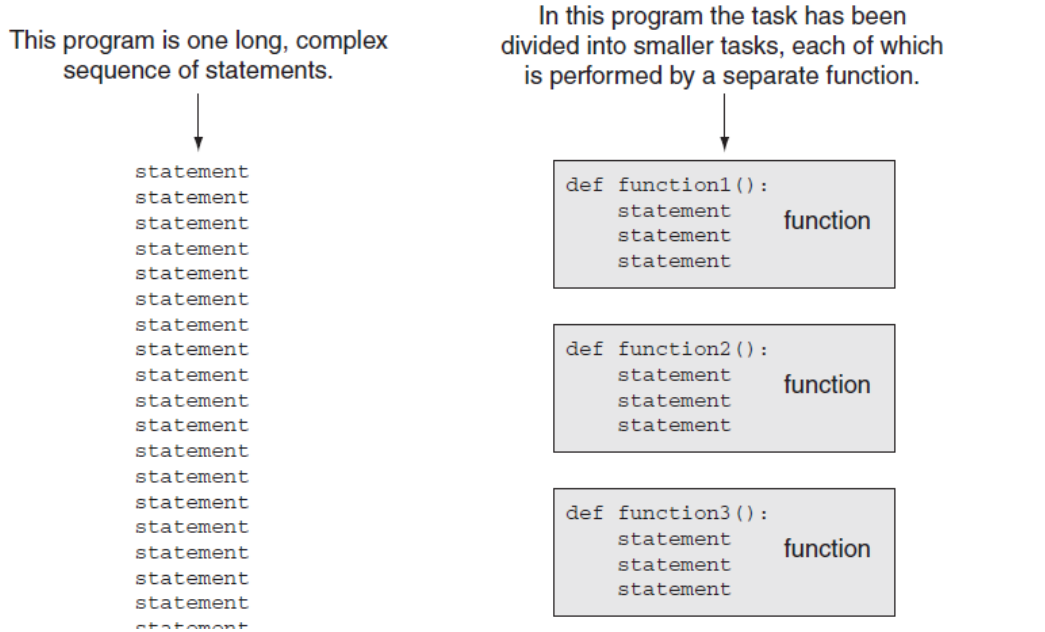


A function is like a box. It can take input and return output. It can be passed around and reused.

Introduction to Functions

- Function: group of statements within a program that perform as specific task.
  - Usually, one task of a large program.
    - Functions can be executed in order to perform overall program task.
  - Known as *divide and conquer* approach.

Figure 5-1 Using functions to divide and conquer a large task



Benefits of Modularizing a Program with Functions

- The benefits of using functions include:
  - 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

Defining and Calling a Function

- Function definition: specifies what function does

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

- Function name convention
  - be lowercase
  - have_an_underscore_between_words
  - not start with numbers
  - not override built-ins
  - not be a keyword

Example

```
[1]: def square(number):
         """Calculate the square of number."""
         return number ** 2

[2]: square(7)

[2]: 49

[3]: square(2.5)

[3]: 6.25
```

- Calling `square` with … because the exponentiation operator (`**`) works …

Defining a Custom …

- Definition begins with … eses and a colon (`:`).
- By convention functio… s underscores should separate each word.
- Required parentheses …
- Empty parentheses m…
- The indented lines after the colon (`:`) are the fu… k
  - Consists of an optional docstring followed … ts that perform the function's task.

Specifying a Custom Function's Docstring

- *Style Guide for Python Code*: First line in a function's block should be a docstring that briefly explains the function's purpose.

Returning a Result to a Function's Caller

- Function calls also can be embedded in expressions:

Let us try it! Download Codes_Module05.zip, upzip and run M5_ExampleSqaure.ipynb

Recap:
Main Parts of a Function

- the `def` keyword
- a function name (`have_underscore_between_words`)
- function parameters between parentheses
- a colon (:)
- indentation
  - *docstring*
    logic
    return statement

Recap: Function Basics

- You can do two things with a function
  - **Define** it, with zero or more parameters
  - **Call** it, and get zero or more results
- Define a function with def

```
def function_name(a, b, c):
        statement
        statement
```

- Call a function with parentheses

```
function_name(d, e, f)
```

- The same principles apply to any function, including built-in functions.

Exercise

- Start from M5_C2F.ipynb and define convert_to_F function that converts Celsius to Fahrenheit.

Exercise (Ans)

- Start from M5_C2F.ipynb and define convert_to_F function that converts Celsius to Fahrenheit.

In [1]:
```
def convert_to_F(celsius):
    """Convert Celisus to Fahrenheit"""
    fahrenheit = (9 / 5) * celsius + 32
    return fahrenheit
```

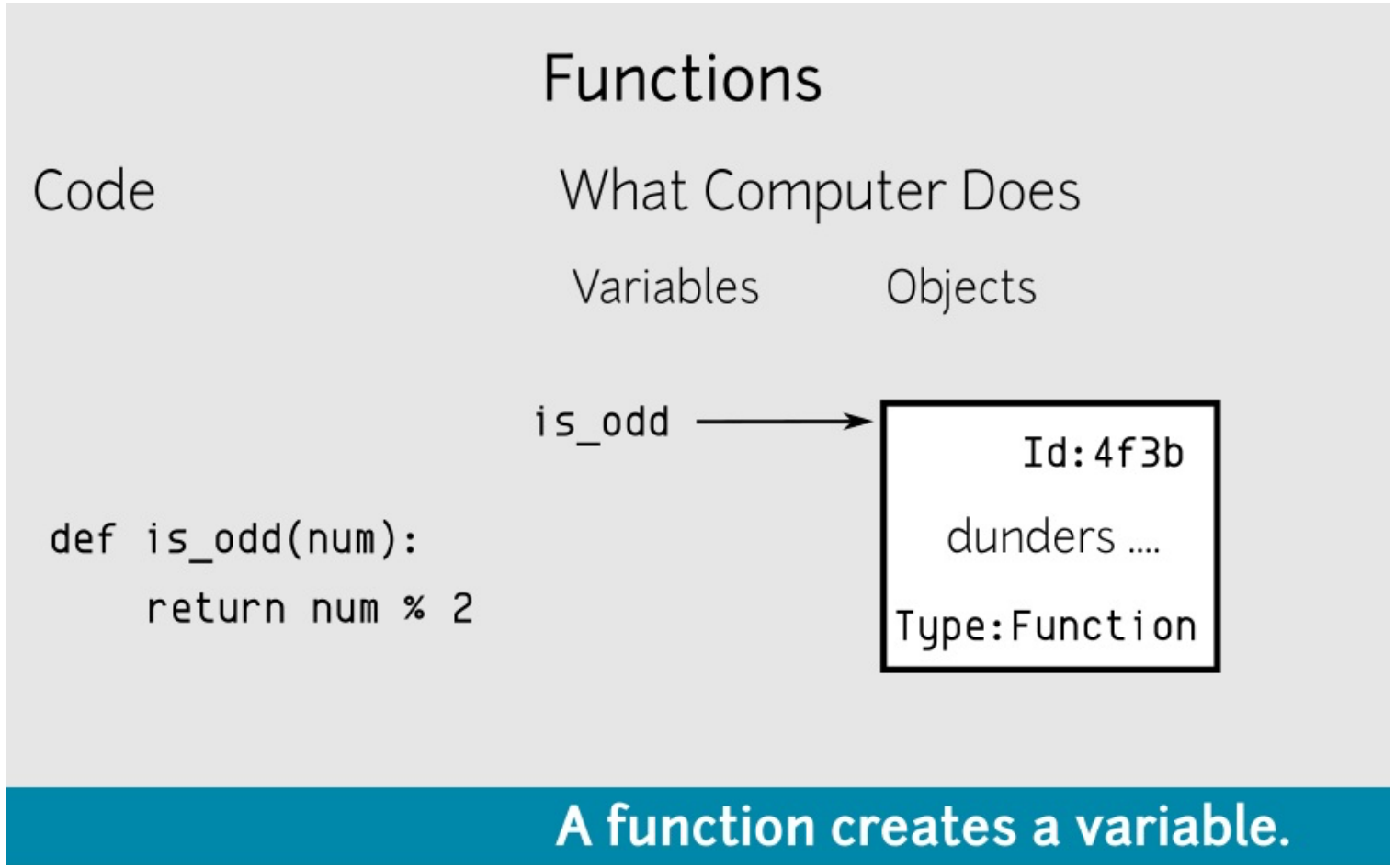
In [2]:
```
name = input('What is your name:')
print('Hello', name)
celisus = float(input('Please enter temperature in Celsius:'))
print('The temperature in C is', format(celisus,'.2f'))
print('The temperature in F is', format(convert_to_F(celisus),'.2f'))
```

```
What is your name:David
Hello David
Please enter temperature in Celsius:22.5
The temperature in C is 22.50
The temperature in F is 72.50
```

In [3]:
```
convert_to_F?
```

```
Signature: convert_to_F(celsius)
Docstring: Convert Celisus to Fahrenheit
File:      c:\users\user\desktop\研究資料\課程\ml_coure_2022\python_module\codes_module05\<ipython-input-1-ddea99d7ca6e>
Type:      function
```

- Concept: creation of a function and function object
- Concept: scope
- Passing multiple arguments



This illustrates the creation of a function. Note that Python creates a new function object, then points a variable to it using the name of the function.

Scope

- Python looks for variables in various places. We call these places scopes.
- When looking for a variable, Python will look in the following locations, in this order:
  - Local scope: variables defined inside of functions.
  - Global scope: variables defined at the global level.
  - Built-in scope: variables predefined in Python.

[1]:
```
x = 2 #Global
def scope_demo():
    y = 3
    print('Local:', y)
    print('Global:', x)
    print('Built-in', dir)
```

[2]:
```
scope_demo()
```

```
Local: 3
Global: 2
Built-in <built-in function dir>
```

What is the output?

[ ]:
```
y = 2 #Global
def scope_demo():
    y = 3
    print('y =', y)
```

[ ]:
```
scope_demo()
```

Passing Multiple Arguments

- Argument: piece of data that is sent into a function

- Python allows writing a function that accepts multiple arguments
  - Parameter list replaces single parameter
    - Parameter list items separated by comma

- Arguments are passed *by position* to corresponding parameters
  - First parameter receives value of first argument, second parameter receives value of second argument, etc.

Example: Function with Multiple Parameters

- `maximum` function that determines and returns the largest of three values.

```
[1]: def maximum(value1, value2, value3):
         """Return the maximum of three values."""
         max_value = value1
         if value2 > max_value:
             max_value = value2
         if value3 > max_value:
             max_value = value3
         return max_value
```

```
[2]: maximum(12, 27, 36)
```

```
[2]: 36
```

```
[3]: maximum(12.3, 45.6, 9.7)
```

```
[3]: 45.6
```


```
[4]: maximum('yellow', 'red', 'orange')
```

```
[4]: 'yellow'
```

- We may call maximum with mixed types, such as `int` s and `float` s.

```
[5]: maximum(13.5, -3, 7)
```

M5_Multiple_Parameters.ipynb

Example: Function with Multiple Parameters

Start from `M5_SumRange.ipynb` and define `sum_range(start, stop, step)` function that returns the sum of a given range . The convention follows three-argument `range` function.

```
[ ]:
```

```
[ ]: start = int(input('Enter the value to start:'))
     stop = int(input('Enter the value to stop:'))
     step = int(input('Enter the increment:'))
```

```
[ ]: print('The sum from', start, 'to', stop, 'with increment', step, 'is',
           sum_range(start, stop, step))
```

Enter the value to start: 1
Enter the value to stop: 11
Enter the increment: 1
The sum from 1 to 11 with increment 1 is 55

Enter the value to start: 10
Enter the value to stop: 0
Enter the increment: -1
The sum from 10 to 0 with increment -1 is 55

Example: Function with Multiple Parameters

Start from `M5_SumRange.ipynb` and define `sum_range(start, stop, step)` function that returns the sum of a given range . The convention follows three-argument `range` function.

```
[ ]: def sum_range(start, stop, step):
         """sum from start to stop with step increment"""
         total = 0
         for i in range(start, stop, step):
             total += i
         return total
```

```
[ ]: start = int(input('Enter the value to start:'))
     stop = int(input('Enter the value to stop:'))
     step = int(input('Enter the increment:'))
```

```
[ ]: print('The sum from', start, 'to', stop, 'with increment', step, 'is',
           sum_range(start, stop, step))
```

Enter the value to start: 1
Enter the value to stop: 11
Enter the increment: 1
The sum from 1 to 11 with increment 1 is 55

Enter the value to start: 10
Enter the value to stop: 0
Enter the increment: -1
The sum from 10 to 0 with increment -1 is 55

Parameters:

A Deeper Look

- Default parameters
- Keyword arguments
- Arbitrary argument lists

Return: A Deeper Look

Default Parameters

- You can specify that a parameter has a default value.
- When calling the function, if you omit the argument for a parameter with a default parameter value, the default value for that parameter is automatically passed.

```
[1]: def rectangle_area(length=2, width=3):
         """Return a rectangle's area."""
         return length * width
```

- Specify a default parameter value by following a parameter's name with an `=` and a value.

```
[2]: rectangle_area()
```

```
[2]: 6
```

```
[3]: rectangle_area(10)
```

```
[3]: 30
```

```
[4]: rectangle_area(10, 5)
```

```
[4]: 50
```

Default Parameters (Cont’)

- Default parameters must be declared after non-default parameters.

[1]:
```
def rectangle_area(length, width=3):
    """Return a rectangle's area."""
    return length * width
```

[2]:
```
rectangle_area(10, 5)
```

[2]:
```
50
```

[3]:
```
rectangle_area(10)
```

[3]:
```
30
```

[4]:
```
rectangle_area()
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-4-392616f2ec38> in <module>
----> 1 rectangle_area()

TypeError: rectangle_area() missing 1 required positional argument: 'length'
```

Keyword Arguments

- When calling functions, you can use keyword arguments to pass arguments in any order.

```
[ ]: def rectangle_area(length, width):
         """Return a rectangle's area."""
         return length * width
```

- Each keyword *argument in a call* has the form *parametername=value*.
- Order of keyword arguments does not matter.

```
[ ]: rectangle_area(width=5, length=10)
```

Arbitrary Argument Lists

- Functions with **arbitrary argument lists**, such as built-in functions min and max, can receive *any* number of arguments.

- Functions with **arbitrary argument lists**, such as built-in functions `min` and `max`, can receive *any* number of arguments.
- Function `min`'s documentation states that `min` has two *required* parameters (named `arg1` and `arg2`) and an optional third parameter of the form `*args`, indicating that the function can receive any number of additional arguments.
- The `*` before the parameter name tells Python to pack any remaining arguments into a tuple that's passed to the `args` parameter.

Defining a Function with

- `average` function that can re

```
[ ]: def average(*args):
         return sum(args) / len(a
```

- The `*args` parameter must b

```
[ ]: average(5, 10)
```

```
[ ]: average(5, 10, 15)
```

```
[ ]: average(5, 10, 15, 20)
```

M5_ArbitraryArg.ipynb

Returning Multiple Values

- In Python, a function can return multiple values
  - Specified after the `return` statement separated by commas
    - Format:

`return` *`expression1, expression2, etc.`*

```
[1]: 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]: first_name, last_name = get_name()
```

```
Enter your first name:  David
Enter your last name:  Chen
```

```
[3]: print('Hello:', first_name, last_name)
```

```
Hello: David Chen
```

When you call such a function in an assignment statement, you need a separate variable on the left side of the = operator to receive each returned value

Summary

- This module covered:
  - The syntax for defining and calling a function
  - Use of local variables and their scope
  - Behavior of passing multiple arguments to functions

To be continued…