PROGRAMMING FOR BUSINESS COMPUTING
商管程式設計

Functions and fruitful functions

Hsin-Min Lu
盧信銘
台大資管系

Functions

- A function is a named sequence of statements that performs a computation.
- To use a function, you need to:
  - Define the function, and
  - Call the function by its name
  - You can pass input variables (i.e., arguments) to the function.

- ☼

More About Functions

- Python has many useful built-in functions

```
>>> #example functions
>>> abs(-3.2)
3.2
>>> import math
>>> math.sqrt(3.2)
1.7888543819998317
```

- The correct verb for using a function is “call” or “invoke.” (呼叫、執行) ☼

Let's Start with a Song

- Print the Lyrics of "Five Little Ducks"
-
```
#Five little ducks
print("Five little ducks went out one day")
print("Over the hills and far away")
print("Mother duck said quack quack quack")
print("But only four little ducks came back")
print()
print("Four little ducks went out one day")
print("Over the hills and far away")
print("Mother duck said quack quack quack")
print("But only three little ducks came back")
print()
print("Three little ducks went out one day")
print("Over the hills and far away")
print("Mother duck said quack quack quack")
print("But only two little ducks came back")
```

Output of the Small Program

```
Five little ducks went out one day
Over the hills and far away
Mother duck said quack quack quack
But only four little ducks came back

Four little ducks went out one day
Over the hills and far away
Mother duck said quack quack quack
But only three little ducks came back

Three little ducks went out one day
Over the hills and far away
Mother duck said quack quack quack
But only two little ducks came back
```

How can we do better?

- Notice that there are a lot of duplications…
- The second and third lines have repeated for three times.
- …

```
print("Over the hills and far away")
print("Mother duck said quack quack quack")
```

…

- We can define the function for these two lines.
-

```
def over_mother():
    print("Over the hills and far away")
    print("Mother duck said quack quack quack")
```

- Now the program become shorter…

Five Little Ducks, Version 2

```
#Five little ducks, v2
def over_mother():
    print("Over the hills and far away")
    print("Mother duck said quack quack quack")

print("Five little ducks went out one day")
over_mother()
print("But only four little ducks came back")
print()
print("Four little ducks went out one day")
over_mother()
print("But only three little ducks came back")
print()
print("Three little ducks went out one day")
over_mother()
print("But only two little ducks came back")
```

How Far Can We Go?

- Still, the first line of each part is very similar.
  - `Five little ducks went out one day`
  - `Four little ducks went out one day`
  - `Three little ducks went out one day`

- We can define a function that output the three similar lines:
- ```
  def n_ducks(num):
      print("%s little ducks went out one day" % num)
  ```

- Do the same for the last line:
- ```
  def only_n(num):
      print("But only %s little ducks came back" % num)
  ```

Arguments and Parameters

- When you define a function, you can allow the function to take input variables.
- The “place” to receive input variables are “parameters.”
- For example,
- ```
  def n_ducks(num):
      print("%s little ducks went out one day" % num)
  ```

- “num” is a parameter that can receive input variables.
- We assign the value of “num” by passing a “argument” to it. For example,
- `n_ducks("Five")`
- `"Five"` is a argument (input value).
- The variable num will be assign with the value “Five.”
  ```

Five Little Ducks, Version 3

```
#Five little ducks, v3
def over_mother():
    print("Over the hills and far away")
    print("Mother duck said quack quack quack")
def n_ducks(num):
    print("%s little ducks went out one day" % num)
def only_n(num):
    print("But only %s little ducks came back" % num)

n_ducks("Five")
over_mother()
only_n("Four")
print()
n_ducks("Four")
over_mother()
only_n("Three")
print()
n_ducks("Three")
over_mother()
only_n("Two")
```

And More…

- We can further define a new function that combines the three functions.
- We need two parameters, for the first and last lines.

- ```
  def sing_unit(num1, num2):
      n_ducks(num1)
      over_mother()
      only_n(num2)
  ```
- Now we can print out the lyrics by:
- ```
  sing_unit("Five", "Four")
  print()
  sing_unit("Four", "Three")
  print()
  sing_unit("Three", "Two")
  ```

What Happens When You Call a Function?

- At the point of calling, the original program suspended execution.
- The parameters of the function are assigned to the values of arguments.
- Execute the body of the function.
- Control return to the original point just after where the function was called.

Calling a Function

```
sing_unit("Five", "Four")
```

num1: “Five”, num2: “Four”

```
def sing_unit(num1, num2):
    n_ducks(num1)
    over_mother()
    only_n(num2)
```

num: “Five”

```
def n_ducks(num):
    print("%s little ducks went out one day" % num)

def over_mother():
    print("Over the hills and far away")
    print("Mother duck said quack quack quack")
def only_n(num):
    print("But only %s little ducks came back" % num)
```

Defining Functions

Function definition begins with “def.”

Function name and its parameters.

```
def get_result(var1, var2):
    """Compute final result."""
    line1
    line2
    return total_counter
```

Provide documentation about this function.

Colon.

The indentation matters…
First line with less indentation is considered to be outside of the function definition.

The keyword ‘return’ indicates the value to be sent back to the caller. ☼

Indentation (縮排) can be spaces or tabs.
Need to be consistent!

☼

Default Arguments

- A function can have multiple parameters.
- You can assign default values to some of these parameters.
- ```
  def print_msg(item, ndays=30):
      print("Please return %s in %d days" % (item, ndays))

  print_msg("items")
  print_msg("items", 50)
  ```
- Output:

```
>>> print_msg("items")
Please return items in 30 days
>>> print_msg("items", 50)
Please return items in 50 days
```
```

Calling with Parameter Names

- You can use parameter names when calling a function.
- When parameter names are used, the order of arguments does not matter.

- ```
  def print_msg(item, ndays=30):
      print("Please return %s in %d days" % (item, ndays))
  ```

```
>>> print_msg(ndays=25, item="books")
Please return books in 25 days
```
```

Scope of Variables and Parameters

- Parameters and variables defined within a function are local (區域變數).
- Local variables only exist inside the function.
- Variables defined on the outer part of a py file are called global variables (全域變數). You can access the global variables within a function.
- The only way for a function to see a variable from another function is for that variable to be passed as a parameter.

Local and Global Variables

- Consider the scenario that we are creating a function named “give_total” that compute the total amount due for a product.
- “give_total” takes only one input, n, which is the number of product purchased.
- Assume that the unit price of the product is $1.2.
-
```
def give_total(n):
    unitp = 1.2
    total = unitp * n
    return total
```

```
>>> give_total(3)
3.5999999999999996
>>> give_total(10)
12.0
```

Local and Global Variables (Cont'd)

- We cannot access unitp outsize of the function:

```
>>> unitp
Traceback (most recent call last):
  File "<input>", line 1, in <module>
NameError: name 'unitp' is not defined
```

- Because: unitp is a local variable that live inside the "give_total" function.
- How about we define a global variable of the same name?
- Can we overwrite the local unitp?
  - No, you cannot!

```
>>> unitp=5
>>> give_total(1)
1.2
```

Local and Global Variables (Cont'd)

- On the other hand, you can have read only access of global inside a function.
- Assume that we have a global variable weekend that tell us whether today is weekend.

```
def give_total(n):
    print("Weekend=%d" % weekend)
    unitp = 1.2
    total = unitp * n
    return total


weekend = True
m=give_total(5)
```

- Execution result: `Weekend=1`

Local and Global Variables (Cont'd)

- If you need to change the value of weekend inside a function, you need to use the "global" keyword to declare that this is a global variable.

- Otherwise, outside world will not see the changes.

Example: Changing Global Variable

```
#give_total, v2
def give_total(n):
    unitp = 1.2
    weekend = False
    print("New Weekend=%d" % weekend)
    total = unitp * n
    return total

weekend = True
print("Outside, before calling: Weekend=%d" % weekend)
m1=give_total(5)
print("Outside: Weekend=%d" % weekend)
```

- Result:

```
Outside, before calling: Weekend=1
New Weekend=0
Outside: Weekend=1
```

Example: The “global” keyword

```
def give_total(n):
    global weekend
    print("Weekend=%d" % weekend)
    unitp = 1.2
    weekend = False
    print("New Weekend=%d" % weekend)
    total = unitp * n
    return total
```

```
>>> weekend = True
>>> m1=give_total(5)
Weekend=1
New Weekend=0
>>> print("Weekend (outside)=%d" % weekend)
Weekend (outside)=0
```

Return Values of Functions

- Function can provide output variables to its caller.
- Act just like functions in mathematics.
- You can “chain” the output to the input.
- Think about the give_total function we encountered.
-
```
def give_total(n):
    unitp = 1.2
    total = unitp * n
    return total
```
- This function returns the total amount due for a purchase.

Return Values of Functions (Cont'd.)

- ```
  def give_total(n):
      unitp = 1.2
      total = unitp * n
      return total
  ```
- When `return` are reached, Python leaves the function and gives the control back to the caller of the function.
- Moreover, Python sends back the values provided in the `return` statement as the execution results. ☼
  ```

Additional Execution Example

```
>>> print(give_total(1))
1.2
>>> print(give_total(3) + give_total(7))
12.0
>>> n=6
>>> print(give_total(n))
7.199999999999999
>>> m=give_total(n)
>>> print(m)
7.199999999999999
```

Returning Many Values

- Compute the minimum and maximum of four input variables.
- ```
  def min_max(x1, x2, x3, x4):
      r1 = min(x1, x2, x3, x4)
      r2 = max(x1, x2, x3, x4)
      return r1, r2

  a1=input("Provide input value a1:")
  a2=input("Provide input value a2:")
  a3=input("Provide input value a3:")
  a4=input("Provide input value a4:")

  out1, out2 = min_max (a1, a2, a3, a4)
  print("Minimal =", out1)
  print("Maximal =", out2)
  ```
- Here out1 gets the first value and out2 gets the second.
  ```

Let's Try This Out!

- First attempt:

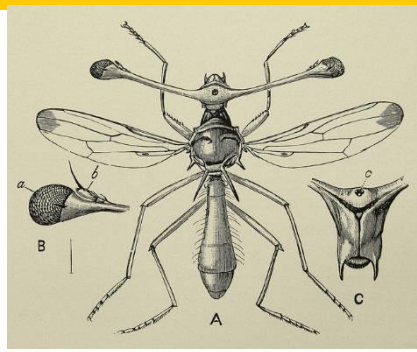
```
Provide input value a1:9
Provide input value a2:5
Provide input value a3:2
Provide input value a4:7
Minimal = 2
Maximal = 9
```

- Second attempt:

```
Provide input value a1:55
Provide input value a2:2
Provide input value a3:9
Provide input value a4:1111
Minimal = 1111
Maximal = 9
```

- Wait! Something is wrong… 哪裡怪?

☼



Happy Debugging…

- Run the program in interactive mode.
- In notepad++, Press F5 (or “Run”→”Run”), and enter the following commands:
- python3 -u -i "$(FULL_CURRENT_PATH)"
- In cmd, use this:
  python3 -u -i “your_program.py”

```
Provide input value a1:55
Provide input value a2:2
Provide input value a3:9
Provide input value a4:1111
Minimal = 1111
Maximal = 9
>>> a1
'55'
>>> a2
'2'
>>> a3
'9'
>>> a3
'9'
>>> type(a1)
<class 'str'>
>>> type(a3)
<class 'str'>
```

- Cause: a1, a2, a3, a4 are strings.
- The rule to compare strings is different from floats.

☼

How to Fix the Bug?

- Convert to float!

```
def min_max(x1, x2, x3, x4):
    r1 = min(x1, x2, x3, x4)
    r2 = max(x1, x2, x3, x4)
    return r1, r2

a1=input("Provide input value a1:")
a2=input("Provide input value a2:")
a3=input("Provide input value a3:")
a4=input("Provide input value a4:")

out1, out2 = min_max (float(a1), float(a2), float(a3), float(a4))

print("Minimal =", out1)
print("Maximal =", out2)
```

Now We are Good!

- Consider the previous example:

```
Provide input value a1:55
Provide input value a2:2
Provide input value a3:9
Provide input value a4:1111
Minimal = 2.0
Maximal = 1111.0
```

- Additional testing:

```
Provide input value a1:52
Provide input value a2:-3
Provide input value a3:8.2
Provide input value a4:99.4
Minimal = -3.0
Maximal = 99.4
```

Function Return Values

- Question: What will happen if your function does not have the "`return`" keyword?
- Answer: Python will still execute your function until the end.
- All Python functions return a value.
- If you did not provide the return value, Python will hand back a special object, denoted `None`.

- A common bug is writing a value-returning function and omitting the `return`! ☼

Example Code

```
def myabs(x):
    if x< 0:
        return -x
    if x> 0:
        return x

print(myabs(-3.5))
print(myabs(0))
```

What will be the result of the function calls?

Example Code (Cont'd.)

- `myabs(-3.5)` gives 3.5
- `myabs(0)` gives None!
- What is going on?
- myabs does not handle the case x=0 !!!
  - How do we fix it?

```
def myabs(x):
    if x< 0:
        return -x
    if x>= 0:
        return x

print(myabs(-3.5))
print(myabs(0))
```

Arguments and Return Values

- We use return values to send information back to the caller of a function.
- Question: Can we send back information using arguments (the inputs of functions)?
  - It depends.
- Different programming languages have different designs.
- Some languages such as C or C++ can use the so call “Call by Reference” approach to send back information using arguments.
- Another approach is “Call by Value.”
  - Make copies of all input arguments before passing them into a function.

Python's Call by Assignment

- The detailed mechanisms are too complicated for beginners.
- We will focus on the consequence of this design.
- Simply put, it depends on whether the argument is immutable or mutable.
- First, we need to know that there are two types of objects in Python: immutable and mutable.
- Example of immutable objects: string, int, float
  - Also: decimal, complex, bool, tuple, range, frozenset, bytes
- Example of mutable objects: list
  - Also: dict, set, bytearray, user-defined classes.

Mutable and Immutable Objects

- Lists are *mutable*, meaning they can be changed. Strings can **not** be changed.

```
>>> myList = [34, 26, 15, 10]
>>> myList[2]
15
>>> myList[2] = 0
>>> myList
[34, 26, 0, 10]
>>> myString = "Hello World"
>>> myString[2]
'l'
>>> myString[2] = "p"

Traceback (most recent call last):
  File "<pyshell#16>", line 1, in -toplevel-
    myString[2] = "p"
TypeError: object doesn't support item assignment
```

Passing in Immutable Objects

- Outside world cannot see changes made to immutable objects inside the function.
  - Similar to “call by value.”
- ```
  def swap1(x1, x2):
      tmp=x1
      x1=x2
      x2=tmp
      print("Inside swap1: x1=", x1, "x2=", x2)

  x1, x2 = 100, 5
  print("Outside: x1=", x1, "x2=", x2)
  swap1(x1, x2)
  print("Outside: x1=", x1, "x2=", x2)
  ```
- Output:

```
Outside: x1= 100 x2= 5
Inside swap1: x1= 5 x2= 100
Outside: x1= 100 x2= 5
```
```

Passing in Mutable Object

- Outside world can see changes made to mutable objects inside the function.
  - Similar to “call by reference.”
-
```
def swap2(xlist):
    tmp=xlist[0]
    xlist[0] = xlist[1]
    xlist[1] = tmp
    print("Inside swap2: xlist=", xlist)

xlist = [100, 5]
print("Outside: xlist=", xlist)
swap2(xlist)
print("Outside: xlist=", xlist)
```

- Output:

```
Outside: xlist= [100, 5]
Inside swap2: xlist= [5, 100]
Outside: xlist= [5, 100]
```

Recursion for factorial function

- In mathematics, the factorial of a non-negative integer n, denoted by n!, is the product of all positive integers less than or equal to n.
  - For example, 5!=5x4x3x2x1=120
- Factorial can be defined recursively:
  - $0! = 1$
  - $n! = n \times (n-1)!$
- Now, how do we create our own factorial function?

```
def factorial(n):
    if n == 0:
        return 1
    else:
        recurse = factorial(n-1)
        result = n * recurse
        return result
```

Example from Section 6.5 of Think Python, Version 2.2.17

Recursion: factorial(3)

```
def factorial(n):
    if n == 0:
        return 1
    else:
        recurse = factorial(n-1)
        result = n * recurse
        return result
```

- When we run “factorial(3)”
  - Since 3 is not 0, call factorial(2)
  - In factorial(2): since 2 is not 0, call factorial(1)
  - In factorial(1): since 1 is not 0, call factorial(0)
  - In factorial(0): return 0

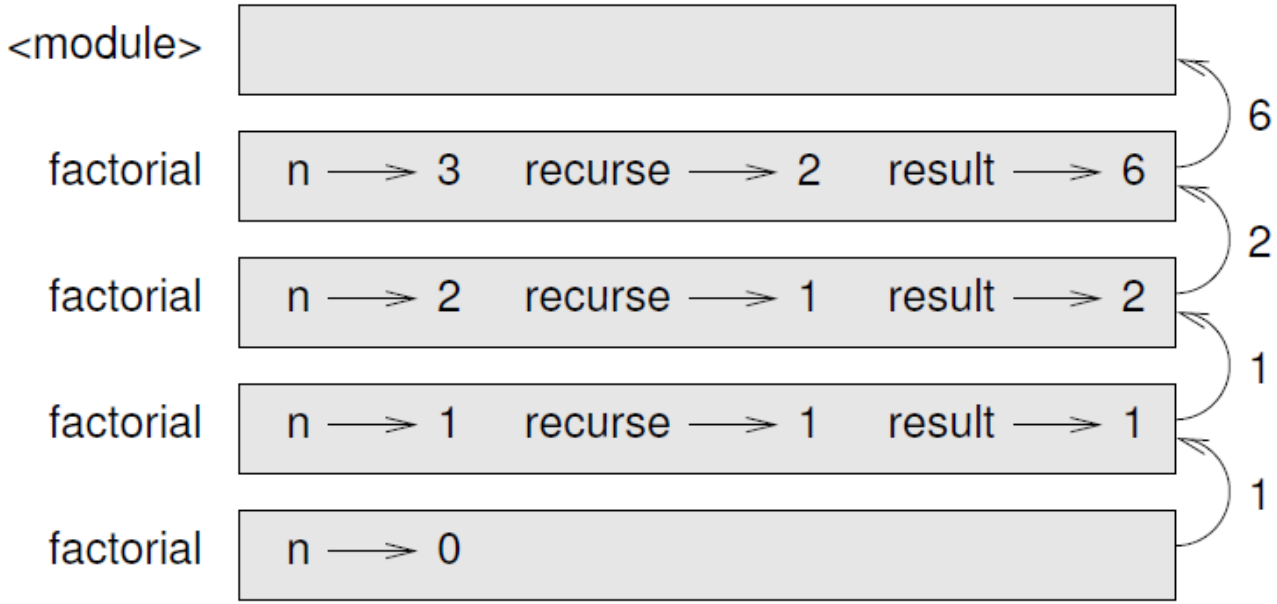


Figure 6.1 of Think Python

Handling Illegal Inputs

- What will happen if we call factorial(1.5)?
  - RuntimeError: maximum recursion depth exceeded in cmp
- Why? Because we encounter an infinite recursion because n will never be 0.
- Also negative values should not be allowed.
- Need to check for valid input ➔ Create guardian (門神)

```
def factorial(n):
    if not isinstance(n, int):
        print('Factorial is only defined for integers.')
        return None
    elif n<0:
        print('Factorial is not defined for negative values.')
        return None
    elif n==0:
        return 1
    else:
        return n * factorial(n-1)
```

☼

Now it's working fine

```
>>> factorial('Qoo')
Factorial is only defined for integers.
>>> factorial(-3)
Factorial is not defined for negative values.
>>> factorial(3.2)
Factorial is only defined for integers.
>>> factorial(4)
24
```

```
def factorial(n):
    if not isinstance(n, int):
        print('Factorial is only defined for integers.')
        return None
    elif n<0:
        print('Factorial is not defined for negative values.
        return None
    elif n==0:
        return 1
    else:
        return n * factorial(n-1)
```

Debugging

- It is common to encounter problems when developing your program.
- Using functions allow you to focus on a few functions that are causing problems.
- Common possible causes:
  - The input is wrong. Invalid arguments
  - The logic of the function is wrong
  - The output of the function is used in the wrong way
- How do we pin down the causes?
  - Print out status
    - Also used extensively when developing new programs
    - Scaffolding ➔ remove these redundant print command afterward.
  - Using built-in debugging function (pdb). ☼

Debugging: Print out status

```
def factorial(n):
    space = ' ' * (4*n)
    print(space, 'factorial', n)
    if not isinstance(n, int):
        print('Factorial is only defined for integers.')
        return None
    elif n<0:
        print('Factorial is not defined for negative values.')
        return None
    elif n==0:
        print(space, 'returning 1')
        return 1
    else:
        result = n * factorial(n-1)
        print(space, 'returning', result)
        return result
```

Debugging: Print out status

```
>>> factorial(4)
                    factorial 4
                factorial 3
            factorial 2
        factorial 1
 factorial 0
 returning 1
        returning 1
            returning 2
                returning 6
                    returning 24
```

THANK YOU!

Questions?