Chapter 4: Functions I

Stephen Huang

February 15, 2023

Functions

- We divide our discussion of functions into two parts to start using functions early.

- Part 2 will follow later:
  - Scope rules
  - Default argument
  - Function as an argument
  - Recursive functions (briefly)
  - Lambda functions (briefly)

Contents

1. Why do we need functions?

- It is common for us to do some identical (or very similar) operations several times in a program.
  - Find the largest number in a list of numbers,
  - Swap two numbers,
  - Sort an array/list,
  - Compute the average of a list of numbers
- It is better to make it into a function so we do not have to duplicate the code.
- Defined once, used many times.

Why?

- Functions allow us to reuse a section of code more than once. If you have to write it twice, make it a function.
  - This is the <u>original</u> reason to have functions.

- Functions also allow us to group codes into logical units. So, we may write a function even if we only use it once.
  - This may be more important than reusing the code.

Where to put the functions?

- You can put function definitions almost anywhere.
- But a function must be defined before being used.
- Typically, we group all the functions at the top of the program. Good programming practice.

Type of Functions

- For Python 3.6, there are 68 built-in functions, such as abs(), max(), and min(). Check with python.org for a complete list.
- Python provides many existing library functions for us to use.
  - More on this later.
- A user can define functions for his use. They are called user-defined functions.

Functions vs. Procedures

- There are two slightly different forms for a function definition, depending on whether the function returns a value.
- Functions with no return value are not like mathematical functions, and in some programming languages, they are called subroutines or procedures.
- Python uses function only.

Return

- If the function does return a value, then the function body must contain 1 or more return statements followed by the value to be returned.
  - The return statement is typically at the end of the function, but it can occur anywhere in the function, and more than 1 return can occur.

- Functions that do not return a value do not need a return statement.
  - When execution reaches the end of such a function, it automatically returns to the location where the function was called.

None

- If a function doesn't explicitly return data and is used in an expression, the function evaluates to the value None.

- Although the print() produces output, it does not explicitly return anything.

```
>>> x = print("Hello")
Hello
>>> type(x)
<class 'NoneType'>
>>>
```

Example

```
def test(level):
    print("    "*level, "Entering test() at level ", level)
    foo(level+1)
    print("    "*level, "Leaving  test()", level)

def foo(level):
    print("    "*level, "Entering foo() at level ", level)
    bar(level+1)
    print("    "*level, "Leaving  foo() at level ", level)

def bar(level):
    print("    "*level, "Entering bar() at level ", level)
    print("    "*level, "Leaving  bar() at level ", level)
    return (1)

test(1)
```

Levels

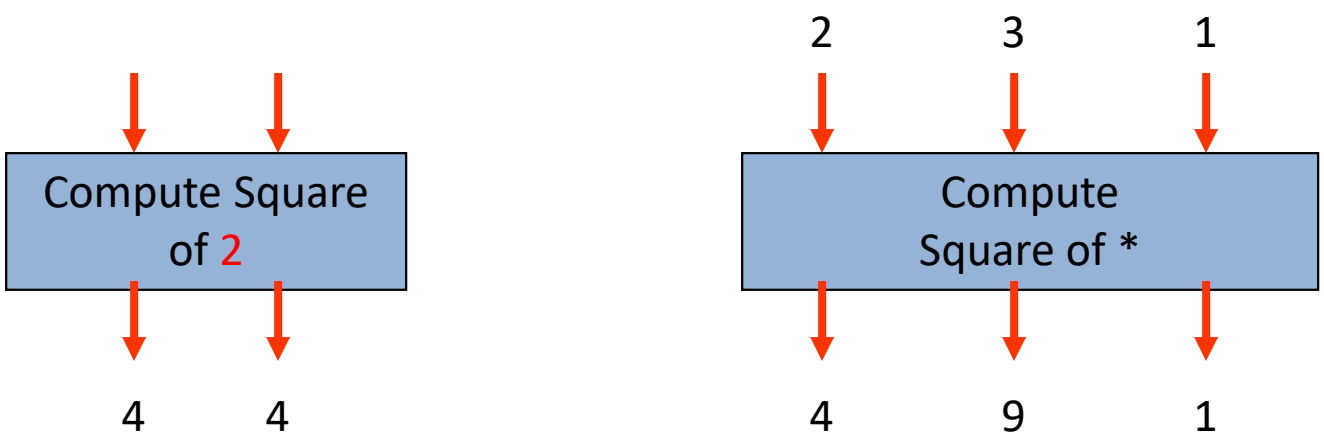
Run function-001

```
        Entering test() at level 1
            Entering foo() at level 2
                Entering bar() at level 3
                Leaving  bar() at level 3
            Leaving  foo() at level 2
        Leaving  test() 1

Process finished with exit code 0
```

Parameters

- Functions are not very useful if it is not allowed to act on different pieces of data at different times.
- It doesn't make sense to write two functions to sort two integer arrays. It will be much better to do it with only one.
- Thus, we have to pass some data to the function so that it can work on that piece of the data.



Parameters

- Parameters are the mechanism for conveying the data to a function that needs to perform its task.
- The parameter list is a list of parameters separated by commas between a pair of parentheses.
- The list can also be empty in which case the parentheses after the function name are still required, e.g., `f( )`.
- There will be more on parameters in a later lecture.

Examples

Python 3.6.0 Shell

File Edit Shell Debug Options Window Help

```
>>> abs(-12.34)
12.34
>>>
>>> def double(num):
        return 2*num

>>> double (35)
70
>>>
>>> double (3.56)
7.12
>>> |
```

Ln: 95 Col: 4

Main function

- All the code outside the functions collectively is called the main function even if we did not define it.
  - See the test(), foo(), bar() example before.
- It is okay to define a function called main() but it is just like any other function one defines.
  - In some programming languages, the execution of the programs starts with the main(). This is NOT the case for Python.

2. Defining Functions

- The keyword def introduces a function definition.
- It is followed by a function name and a parameter list.
- A simplified form:

```
def <func_name> (<para_list>):
    <Statement(s)>
```

- The first statement of the function body can optionally be a string literal; this string literal is the function's documentation string or docstring.

Definition

- The parentheses following a function's name are mandatory even though the function does not have any parameter.
- The parentheses enclose a list of formal parameters which are identifiers separated by commas.
- The actual values assigned to these parameters are established when the function is called. They are called actual parameters or arguments.

Example

```python
def test(level):
    print("    "*level, "Entering test() at level ", level)
    print("    "*level, "Leaving  test() at level ", level)

def foo(level):
    print("    "*level, "Entering foo()  at level ", level)
    print("    "*level, "Leaving  foo()  at level ", level)

def bar(level):
    print("    "*level, "Entering bar()  at level ", level)
    print("    "*level, "Leaving  bar()  at level ", level)
    return (1)

test(1)
foo(1)
bar(1)
```

Run function-002

```
    Entering test() at level  1
    Leaving  test() at level  1
    Entering foo()  at level  1
    Leaving  foo()  at level  1
    Entering bar()  at level  1
    Leaving  bar()  at level  1
```

Example

```python
def test(level):
    print("    "*level, "Entering test() at level ", level)
    if level<5:
        test(level+1)
    print("    "*level, "Leaving  test() at level ", level)

test(1)
#
```

Run function-003

```
Entering test() at level  1
    Entering test() at level  2
        Entering test() at level  3
            Entering test() at level  4
                Entering test() at level  5
                Leaving  test() at level  5
            Leaving  test() at level  4
        Leaving  test() at level  3
    Leaving  test() at level  2
Leaving  test() at level  1
```

3. Using functions

- A function is not executed when it is defined.
- To execute (or call, or invoke) a function, one writes the function's name followed by parentheses with the requisite number of (actual) parameters.
- The order of the actual parameters must match that of the formal parameters.
  - There are exceptions.

Polymorphism

- In Python, functions polymorphism is possible as we don't specify the argument types while creating functions. (dynamically typed)
  - The behavior of a function may vary depending upon the arguments passed to it.
  - The same function can accept arguments of different object types.
  - If the objects find a matching interface, the function can process them.

Using functions

- When a function has more than a few numeric arguments, it is easy to forget what they are, or where they should be in the list.
- In that case, it is often a good idea to include the names of the parameters in the argument list when calling the function.

```
polygon(bob, n=7, length=70)
```

- These are called keyword arguments because they include the parameter names as "keywords."

Examples

```
def greeting(name):
    print("Hello ", name, ".", sep='')

greeting("Stephen")
greeting("Robert")
greeting("everyone")

def getName():
    return (input("Enter a name: "))

name1 = getName()
greeting(name1)
```

Execution

- Before the function's body is executed, the actual parameters are assigned to the corresponding formal parameters.
- A copy of the value is given to the formal parameter. Pass-by-Value.
- Think of a formal parameter as a local variable initialized with a value given in the actual parameter.
- The actual parameter can be an expression.

Composition of function calls

- If a function **`first`** returns a value of type X and,
- If a function **`second`** takes a parameter of type X,
- We can make consecutive function calls like

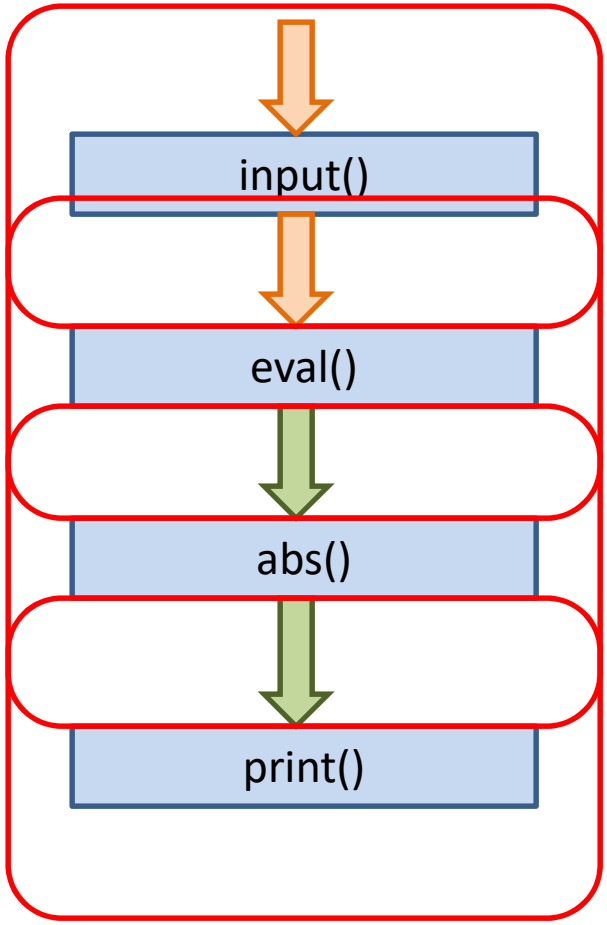
```
y = second(first(value))
print(second(first(value)))
```

instead of

```
x = first(value)
y = second(x)
```

Composition of functions

```
print(abs(eval(input("Enter a value: "))))
```



Example: BMI

- We are going to show a series of scripts that computes BMI.
- You will be able to see the "generalization" of the code.
- We are making the code more useful and more structured. The BMI is a small program, so it may not sound that important to make it more structured. Think big.

Version 1

```
weight = float(input(
         "Enter weight [pounds]: "))
height = float(input(
         "Enter height [inches]: "))
bmi = 703*weight/(height*height)
print("Your body mass index is:", bmi)
```

Version 2

```
def cal_bmi():
    weight = float(input(
            "Enter weight [pounds]:"))
    height = float(input(
            "Enter height [inches]:"))
    bmi = 703*weight/(height*height)
    print("Your body mass index is:", bmi)

bmi()
```

Version 3

```
def getNum(valueType, unit):
    return(eval(input("Enter "+valueType+ \
            " in "+unit+": ")))

def cal_bmi():
    weight = getNum("weight", "pounds")
    height = getNum("height", "inches")
    bmi = 703*weight/(height*height)
    print("Your body mass index is:", bmi)

cal_bmi()
```

Version 4

```
def cal_bmi(w, h):
    return 703*w/(h*h)

def getNum(valueType, unit):
    return(eval(input("Enter "+valueType+ \
            " in "+unit+": ")))

weight = getNum("weight", "pounds")
height = getNum("height", "inches")
bmi = cal_bmi(weight, height)
print("Your body mass index is:", bmi)
```

Version 5

```
def cal_bmi(w, h):
    return 703*w/(h*h)

def getNum(valueType, unit):
    return (eval(input("Enter "+valueType+ \
           " in "+unit+": ")))

def main():
    weight = getNum("weight", "pounds")
    height = getNum("height", "inches")
    print("Your BMI is:", cal_bmi(weight,height))

main()
```

4. Library Functions

- Python included some of the most commonly used functions as "built-in" functions.

```
dir(__builtins__)
```

- Python distributions include an extensive standard library that you must import before using it.
  - Math
  - Regular Expressions
- We can easily create a module that contain a collection of functions.

Import

- The contents of an entire module can be imported using any of the following statements:
  - `import <module>`
  - `import <module> as <id>`
  - `from <module> import *`

Python Built-in Functions

		Built-in Functions		
abs()	dict()	help()	min()	setattr()
all()	dir()	hex()	next()	slice()
any()	divmod()	id()	object()	sorted()
ascii()	enumerate()	input()	oct()	staticmethod()
bin()	eval()	int()	open()	str()
bool()	exec()	isinstance()	ord()	sum()
bytearray()	filter()	issubclass()	pow()	super()
bytes()	float()	iter()	print()	tuple()
callable()	format()	len()	property()	type()
chr()	frozenset()	list()	range()	vars()
classmethod()	getattr()	locals()	repr()	zip()
compile()	globals()	map()	reversed()	__import__()
complex()	hasattr()	max()	round()	
delattr()	hash()	memoryview()	set()	

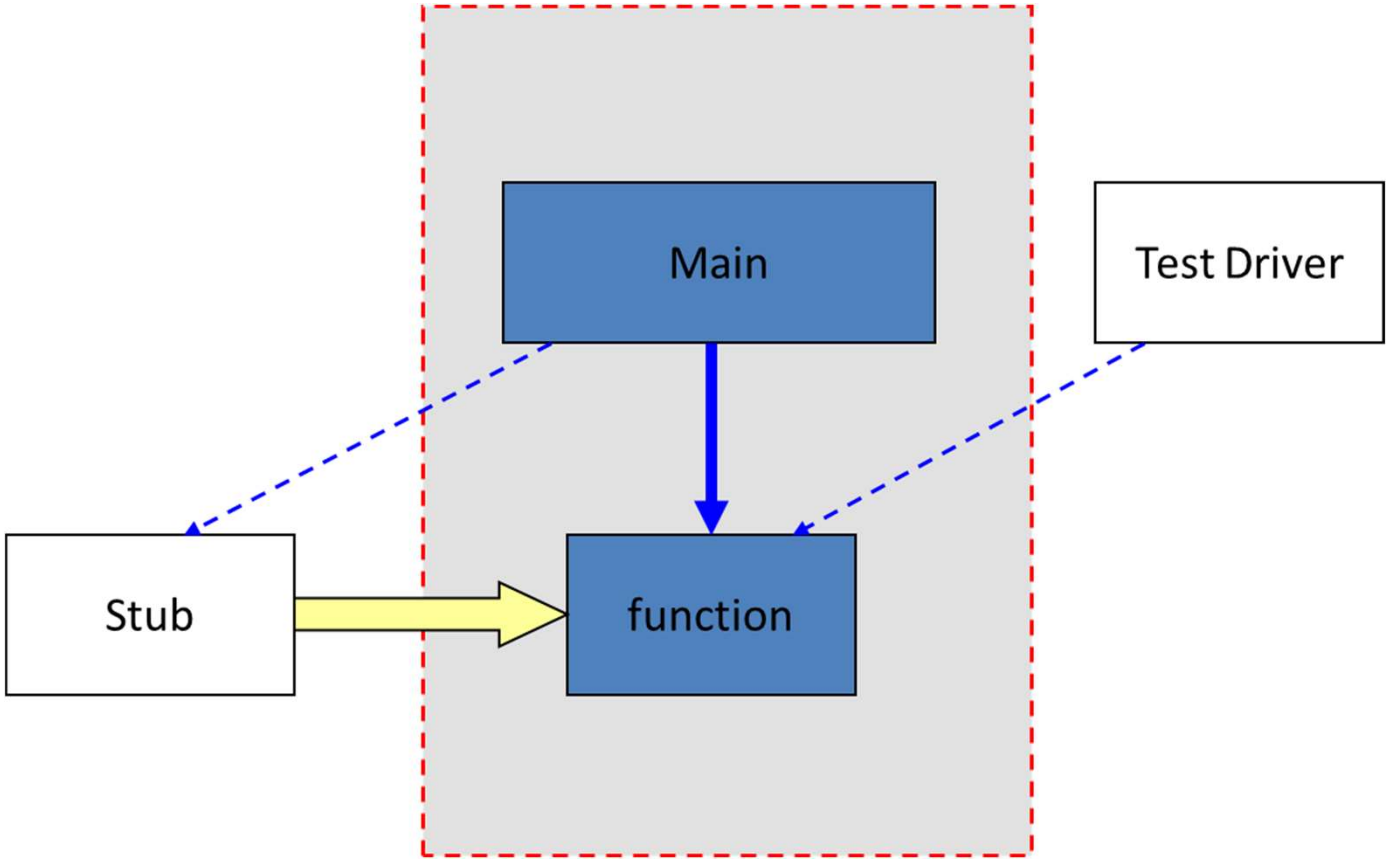
References

- List of Python standard library functions: https://docs.python.org/3/library

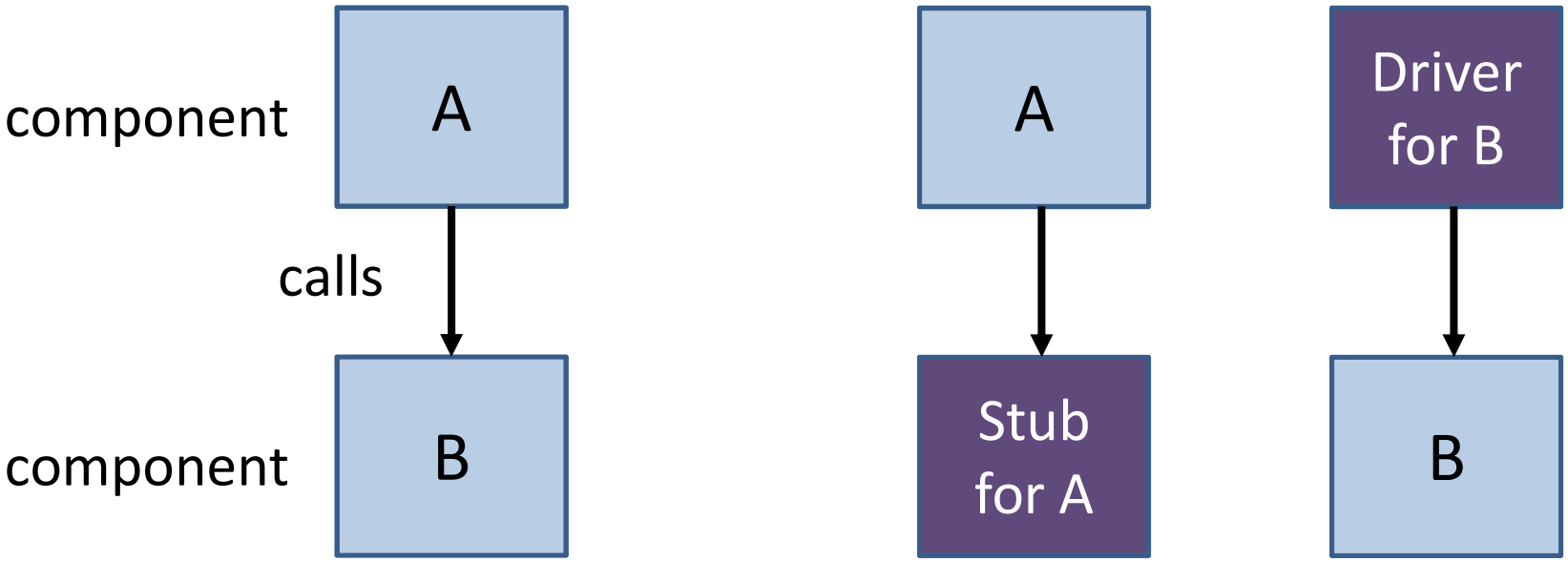
5. Stubs & Drivers

- How do you develop a program with multiple functions?
- Software Engineering.
- It may not be clear why it is important to use these functions for simple programs.
- Stubs & Drivers is one suggested way of developing a program.
- This section is only a very brief introduction.

Stub and Driver



Stubs & Drivers



Which one should we write and test first?
A or B?