浙江理工大学
ZHEJIANG SCI-TECH UNIVERSITY

7. Function (1)

《Python programming》 / Lecturer：Zhiyi Luo (罗志一)

School of Computer Science and Technology
计算机科学与技术学院

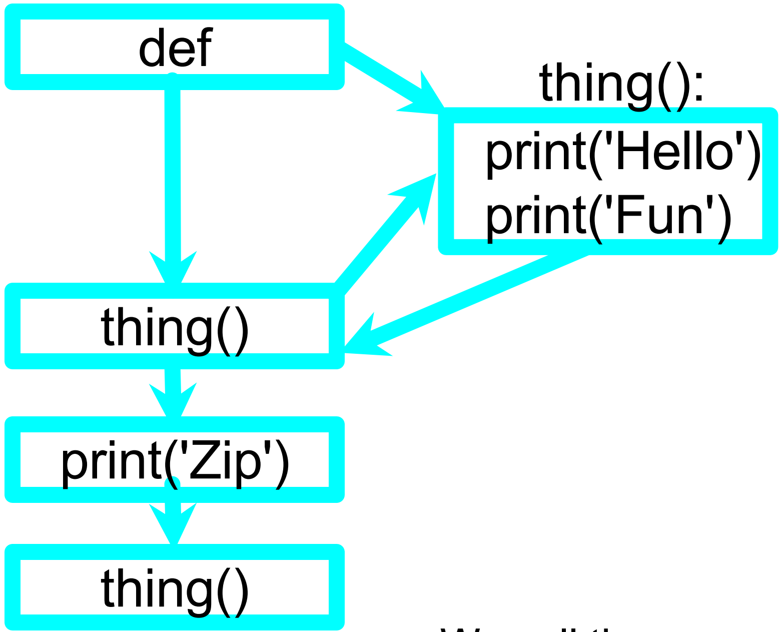
Python Functions

- In Python, the function is a block of code defined with a name. We use functions whenever we need to perform the same task multiple times without writing the same code again.
- Consider a scenario where we need to do some action/task many times. We can define that action only once using a function and call that function whenever required to do the same activity.
- Function improves efficiency and reduces errors because of the reusablility of a code. Once we create a function, we can call it anywhere and anytime. The benefit of using a function is reusability and modularity.

Function types

- There are two kinds of functions in Python
  - Built-in functions that are provided as part of Python - print(), input(), type(), float(), int() ...
  - Functions that we define ourselves and then use
- We treat the built-in function names as “new” reserved words (i.e. we avoid them as variable names)

Stored (and reused) Steps



Program:

```
def thing():
    print('Hello')
    print('Fun')

thing()
print('Zip')
thing()
```

Output:

```
Hello
Fun
Zip
Hello
Fun
```

We call these reusable pieces of code “functions”

Function Definition

In Python a function is some reusable code that takes arguments(s) as input, does some computation, and then returns a result or results

We define a function using the def reserved word

We call/invoke the function by using the function name, parentheses, and arguments in an expression

Creating a Function

- Using the following steps to define a function in Python.
  - Use the def keyword with the function name to define a function.
  - Next, pass the number of parameters as per your requirement.
  - Next, define the function body with a block of code. This block of code is nothing but the action you wanted to perform.
- In Python, no need to specify curly braces for the function body. The only indentation is essential to separate code blocks. Otherwise, you will get an error

syntax

```
def function_name(parameter1, parameter2):
    # function body
    # write some action
    return value
```

Creating a Function

- In Python, no need to specify curly braces for the function body. The only indentation is essential to separate code blocks. Otherwise, you will get an error
  - function_name
  - parameters
  - function_body
  - return value

syntax

```
def function_name(parameter1, parameter2):
    # function body
    # write some action
    return value
```

Arguments

- An argument is a value we pass into the function as its input when we call the function
- We use arguments so we can direct the function to do different kinds of work when we call it at different times
- We put the arguments in parentheses after the name of the function

```
big = max('Hello world')
```

Argument

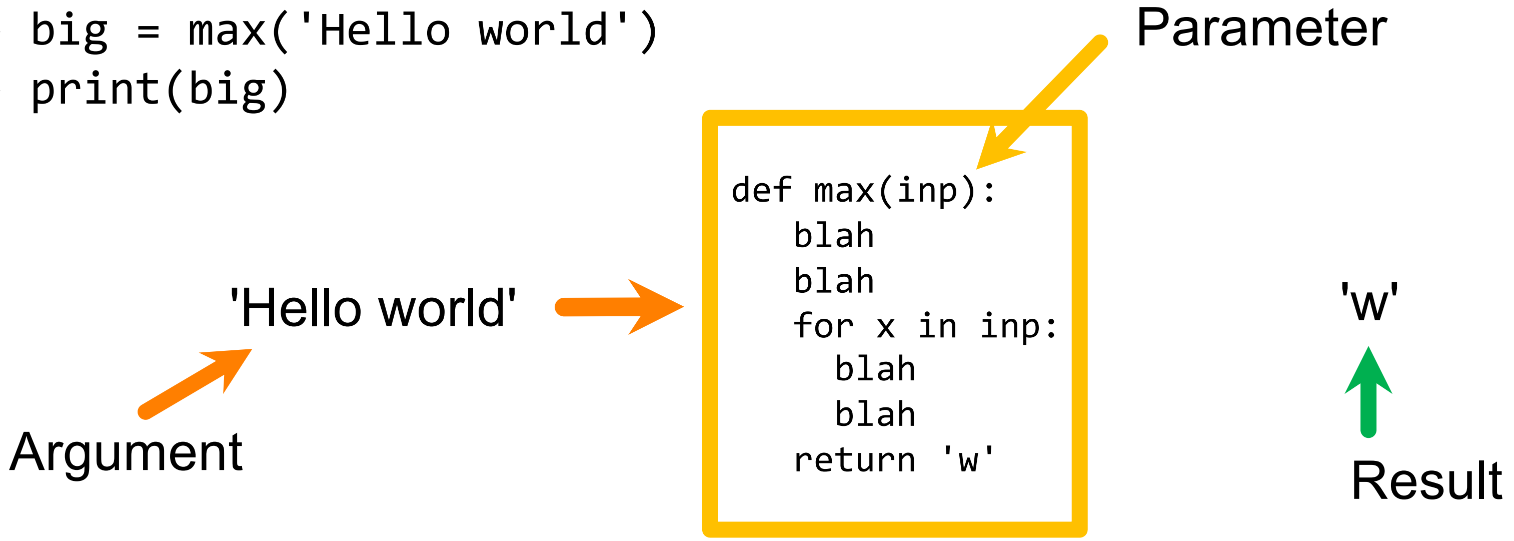
Parameters

A parameter is a variable which we use in the function definition. It is a “handle” that allows the code in the function to access the arguments for a particular function invocation.

```
>>> def greet(lang):
...     if lang == 'es':
...        print('Hola')
...     elif lang == 'fr':
...        print('Bonjour')
...     else:
...        print('Hello')
...
>>> greet('en')
Hello
>>> greet('es')
Hola
>>> greet('fr')
Bonjour
>>>
```

Arguments, Parameters, and Results

```
>>> big = max('Hello world')
>>> print(big)
w
```



```
def max(inp):
    blah
    blah
    for x in inp:
       blah
       blah
    return 'w'
```

Multiple Parameters / Arguments

We can define more than one parameter in the function definition

We simply add more arguments when we call the function

We match the number and order of arguments and parameters

```
def addtwo(a, b):
    added = a + b
    return added

x = addtwo(3, 5)
print(x)
```

```
8
```

Arguments

- In Python, there are four types of arguments allowed.
  - Positional arguments
  - Keyword arguments
  - Default arguments
  - Variable-length arguments

Positional Arguments

- Positional arguments are arguments that are pass to function in **proper positional order**. That is, the 1st positional argument needs to be 1st when the function is called. The 2nd positional argument needs to be 2nd when the function is called, etc.

```
def subtract(a, b):
    print(a - b)

add(50, 10)
add(10, 50)
```

```
add(105, 561, 4)  # error
```

TypeError: add() takes 2 positional arguments but 3 were given

Keyword Arguments

- A keyword argument is an argument value, passed to function preceded by the variable name and an equals sign.
- In keyword arguments order of argument is not matter, but the number of arguments must match. Otherwise, we will get an error.

```python
def message(name, surname):
    print("Hello", name, surname)

message(name="John", surname="Wilson")
message(surname="Ault", name="Kelly")
```

Keyword Arguments

- While using keyword and positional argument simultaneously, we need to pass first arguments as positional arguments and then keyword arguments. Otherwise, we will get SyntaxError.

```
def message(first_nm, last_nm):
    print("Hello..!", first_nm, last_nm)

# correct use
message("John", "Wilson")
message("John", last_nm="Wilson")

# Error
# SyntaxError: positional argument follows keyword argument
message(first_nm="John", "Wilson")
```

Default Arguments

- Default arguments take the default value during the function call if we do not pass them. We can assign a default value to an argument in function definition using the =assignment operator.

```python
# function with default argument
def message(name="Guest"):
    print("Hello", name)

# calling function with argument
message("John")

# calling function without argument
message()
```

Default Arguments

```
>>> def f(x=1, y):
...     print(x, y)
...
  File "<stdin>", line 1
SyntaxError: non-default argument follows default argument
```

Variable-length Arguments

- In Python, sometimes, there is a situation where we need to pass multiple numbers of arguments to the function. Such types of arguments are called **variable-length arguments**. We can declare a variable-length argument with the * **(asterisk)** symbol.
- We can pass any number of arguments to this function. Internally all these values are represented in the form of a **tuple**.

```
def function_name(*var):
    # function body
    # write some action
    return value
```

syntax

Variable-length Arguments

```python
def addition(*numbers):
    total = 0
    for no in numbers:
        total = total + no
    print("Sum is:", total)

# 0 arguments
addition()
# 5 arguments
addition(10, 5, 2, 5, 4)
# 3 arguments
addition(78, 7, 2.5)
```

Return Values

- Often a function will take its arguments, do some computation, and return a value to be used as the value of the function call in the calling expression. The return keyword is used for this.

```
def greet():
    return "Hello"

print(greet(), "Glenn")
print(greet(), "Sally")
```

```
Hello Glenn
Hello Sally
```

Return Values

A “fruitful” function is one that produces a result (or return value)

The return statement ends the function execution and “sends back” the result of the function

```
>>> def greet(lang):
...     if lang == 'es':
...         return 'Hola'
...     elif lang == 'fr':
...         return 'Bonjour'
...     else:
...         return 'Hello'
...
>>> print(greet('en'),'Glenn')
Hello Glenn
>>> print(greet('es'),'Sally')
Hola Sally
>>> print(greet('fr'),'Michael')
Bonjour Michael
>>>
```

Return Values

- The return value is nothing but a outcome of function.
  - The return statement ends the function execution.
  - For a function, it is not mandatory to return a value.
  - If a return statement is used without any expression, then the None is returned.
  - The return statement should be inside of the function block.

None

- None is a special Python value that holds a place when there is nothing to say.

```
>>> thing = None
>>> if thing is None:
...     print(“It’s nothing”)
... else:
...     print(“It’s something”)
...
It’s nothing
```

Return Multiple Values

- You can also return multiple values from a function. Use the return statement by separating each expression by a comma.

绑定（Python）

Python绑定指向同一个对象，可以看作别名；没有类型限制

```
a = [1]
b = a
a[0] = 2
b[0] # 2
```

重新绑定与解除绑定

已经绑定的符号可以重新绑定到新的值，绑定之后与原来的值的关系解除：

```
a = [1,2]
b = a
a = [2,3]
b # [1,2]
a[0] = 3
b[0] # 1
```

可以用del语句强制解除某个绑定：

```
a = [1,2]
del a
```

特殊绑定

特殊绑定除了语法以外，与普通绑定完全等价

```
def myfunc(a,b):
    ...

myfunc_alias = myfunc  # 可以用于绑定其他名称
myfunc_alias(1,2)
myfunc = 3            # 可以重新绑定
del myfunc            # 可以解除绑定
```

复合语句

- 多个语句组成的具有特殊功能的结构，称为复合语句
  - if ... elif ... else ...
  - while ... else ...
  - for ... else ...
  - with ...
  - def ...
  - class ...
  - try ... except ... else ... finally ...
- 复合语句内部包含的语句范围以缩进标示，因此不需要表示结束的语法
- 有多个子句的复合语句，每个子句的缩进应当相同

函数定义

```
def func(a, b):
    ...
    return ...

func(1, 2)
func(var1, var2)
return_value = func(3, 4)
```

函数定义

- 函数定义将新定义的函数对象绑定到选定的名称上
- 函数是一种复合语句：
  - 函数内部的语句在调用时执行，每次调用执行一次
- 函数有额外的名称：参数
  - 参数名称对应的值，在调用时，由调用者进行绑定
- 函数可以有返回值，返回值是调用函数的表达式的值
  - 用return语句返回值
  - 没有执行return语句就返回了，或者return后面没有跟表达式的情况下，返回值是None

```
def add(a, b):
    return a + b
```

参数绑定

- 调用方有两种绑定参数的方式：
  - 按顺序绑定：从左到右指定的每个参数，绑定到参数列表中从左到右对应的名称上
  - 按名称绑定：使用 name = value的方式，直接指定要绑定的参数
- 两种方式可以混用，但不可以同时指定同一个参数
- 参数在函数内可以重新绑定到其他值，解开与调用方传入值的关系

```
def  func(a, b):
     ...
func(1, 2)
func(a = 1,  b = 2)
func(1,  b = 2)
func(1,  2,  a = 3)  #  出错
```

默认参数

- 定义函数时，可以指定参数的默认值，如果调用方未绑定该参数，则使用默认值
    - 可以理解为：在定义函数时已经绑定了该参数的值，如果调用时重新指定则重新绑定该参数到新的值，否则使用旧的绑定
    - 服从绑定的一般规则：所有调用中默认参数绑定的是相同对象
    - 默认参数的绑定在定义时进行

警惕：绑定默认值到可变类型时需要格外注意

```
def func(v, a = []):
    a.append(v)
    return a
a = func(1)          # [1]
b = func(1)          # [1,1]
a  #  [1,1]
```

- 默认值必须从后向前指定，不能在有默认值的参数后加没有默认值的参数

可变参数

- 可变参数是特殊的参数绑定方式

```
def var_func(a, b, *more_args, **more_kw_args):
    ...
var_func(1,2,3,4, test = 5)  # a = 1, b = 2, more_args = (3,4),
                                                # more_kwargs = {'test': 5}
```

- 如果定义了*args形式的参数，调用时，所有没有匹配到其他参数的多余的位置参数，构成元组，绑定到args
- 如果定义了**kwargs形式的参数，调用时，所有没有匹配到其他参数的多余的命名参数，构成字典，绑定到kwargs

可变参数的相反操作：解构绑定参数

调用时可以使用元组和字典提供参数

```
def func(a,b):
    ...
c = (1,2)
func(*c) # 等效于：func(1,2)
d = {'a': 1, 'b': 2}
func(**d) # 等效于：func(a = 1, b = 2)
```

Exercise

- Write a program create function func1() to accept a variable length of arguments and print their values.

Exercise

- Define a function, find the largest item from a given list.

Exercise

- Understanding the following code:

```
>>> def f(**kw):
...     print([k for k in kw])
...
>>> f(**{'a':1, 'b':2})
['a', 'b']
```