SI100B Introduction to Information Science and Technology
Python Programming

张海鹏 Haipeng Zhang
School of Information Science and Technology
ShanghaiTech University

Learning Objectives

- Functions
- Scope

Reusable codes

Calc:
Do the following operations for two numbers and print results:
+, -, *, /

```
def calc(a, b):
        sum = a + b
        subtraction = a – b
        multiply = a * b
        division = a / b
        print(“sum is: ”, sum)
        print(“subtraction is: ”, subtraction)
        print(“multiplication is: ”, multiply)
        print(“division is: ”, division)
```

- >>> calc(5, 7)
- >>> calc(3.0, 4)
- >>> calc(1.2, 2.5)

Function

- A function is a block of organized, reusable code that is used to perform a group of related actions.

- Functions provide better modularity for your application and a high degree of code reusing.

Python functions

- There are two kinds of functions in Python
  - Built-in functions that are provided as part of Python
    - input(), type(), float(), int(), …
  - Functions that we define by ourselves

Built-in functions

- Math functions
  - Python has a math module that provides most of the familiar mathematical functions.
  - Before use all these functions, we have to import them: >>> import math
  - To call one of the functions, we have to specify the name of the module and the name of the function, separated by a dot. This format is also called **dot notation**.
    - >>>decibel = math.log10(17.0)
    - >>>angle = 1.5
    - >>>sine = math.sin(angle)

Many other math functions

Function name	Description
`abs(`**value**`)`	absolute value
`ceil(`**value**`)`	rounds up
`cos(`**value**`)`	cosine, in radians
`degrees(`**value**`)`	convert radians to degrees
`floor(`**value**`)`	rounds down
`log(`**value**`, `**base**`)`	logarithm in any base
`log10(`**value**`)`	logarithm, base 10
`max(`**value1**`, `**value2, ...**`)`	larger of two (or more) values
`min(`**value1**`, `**value2, ...**`)`	smaller of two (or more) values
`radians(`**value**`)`	convert degrees to radians
`round(`**value**`)`	nearest whole number
`sin(`**value**`)`	sine, in radians
`sqrt(`**value**`)`	square root
`tan(`**value**`)`	tangent

Constant	Description
`e`	2.7182818...
`pi`	3.1415926...

Defining our own functions

- Simple rules to define a function
  1. Function blocks begin with the keyword **def** followed by the function name, parentheses (), and a colon :
  2. Any parameters or arguments should be placed within these parentheses.
  3. The code block within every function is indented.

```
def function_name (parameters):
    statement 1 in the function body
    statement 2 in the function body
    …
    statement n in the function body
```

Building our own functions

- Defining a function doesn't mean we execute the body of the function
- test.py file:

```
x = 5
print('Hello')

def print_oks():
    print(”OK 1“)
    print(’OK 2’)

print('Yo')
x = x + 2
print(x)
```

- python test.py

```
Hello
Yo
7
```

Calling a function

- Once we have defined a function, we can call (or invoke) it as many times as we like
- This is a store and reuse pattern
- test.py file:

```
x = 5                           (1)
print('Hello')                  (2)

def print_oks():                (5)
    print(”OK 1”)               (6)
    print(’OK 2’)               (7)

print('World')                  (3)
print_oks()                     (4)
x = x + 2                       (8)
print(x)                        (9)
```

- python test.py

```
Hello
World
OK 1
OK 2
7
```

Arguments

- An argument/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

The return statement

- 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.
- A return statement with no arguments is the same as return none.

```
def greet():
    return("Hello")      (2) (4)

print(greet(), "Glenn") (1)
print(greet(), "Sally") (3)
```

```
Hello Glenn
Hello Sally
```

The return statement

- 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
>>>
```

```
What greet does:

1)Receives the argument value

2)If the lang is es, then
return Hola, otherwise,

2.1)If the lang is fr, then
return Bonjour, otherwise
return Hello
```

Type annotation for arguments and return values

Types of parameters type

```
def GetLarger(x:int, y:int) -> int:
    if x>=y:
        return x
    else:
        return y
```

return type

Does it work?

Type annotation

```
def GetLarger(x:int, y:int) -> int:
    if x>=y:
        return x
    else:
        return y
print(GetLarger(1,2))
print(GetLarger(1,2.0))
```

Output

```
2
2.0
```

To increase readability, no real usage

Default values of arguments

- Parameters can have default values,

  def funcname(p1 = v1, p2 = v2,…, pn = vn):

- All the parameters on the right-hand side of the parameter with default value must have default values

Default values of arguments

```
def GetLarger(x = 1, y = 2):
    if x>=y:
        return x
    else:
        return y
```

```
def GetLarger(x:int, y:int=2) -> int:
    if x>=y:
        return x
    else:
        return y
```

No error

Default values of arguments

```
def GetLarger(x = 1, y):
    if x>=y:
        return x
    else:
        return y
```

```
def GetLarger(x:int=1, y:int) -> int:
    if x>=y:
        return x
    else:
        return y
```

SyntaxError: non-default argument follows default argument

Function call

```
def funcname(p1,p2,…,pn):
    body
```

Actual arguments in function call can be passed by

- **positional arguments**: arguments are passed in the order of parameters

```
funcname(v1,v2,…,vn)
```

- **keyword arguments**: arguments are passed by preceding the corresponding parameters without preserving their orders

```
funcname(pi=vi,pj=vj,…)
```

Function call: Example

```
def GetPow(base, exp):
    '''input: two integers base and exp
       return  base**exp'''
    return base**exp

print(GetPow(2,3))
print(GetPow(exp = 3,base = 2))
```

Output

```
8
8
```

Arbitrary number of parameters

Receiving arguments as a tuple:

funcname(v_1,…,v_n, *para, x_1,…,x_m)

- zero or more normal arguments may occur for *para
- para is used as a tuple
- Parameters x_1,…,x_m after *para should be used as keyword arguments

Receiving parameters as a tuple

```
>>> def demo(*p):
    print(p)

>>> demo(1,2,3)
(1, 2, 3)
>>> demo(1,2)
(1, 2)
>>> demo(1,2,3,4,5,6,7)
(1, 2, 3, 4, 5, 6, 7)
>>> demo()
()
```

Receiving parameters as a tuple

```
>>> def demo(*p,v):
    print(p,v)

>>> demo(1,2,3,v=4)
(1, 2, 3) 4
>>> demo(1,2,3)
Traceback (most recent call last):
  File "<pyshell#9>", line 1, in <module>
    demo(1,2,3)
TypeError: demo() missing 1 required keyword-
only argument: 'v'
>>>
```

Type error

Arbitrary number of parameters

Another way to define a function with arbitrary number of parameters, receiving parameters as a dict:

funcname(v_1,…,v_n, **para)

- zero or more keyword arguments may occur for **para
- para is used as a dictionary dict
- **para should be the last parameter
- v_n could be arbitrary argument list

Receiving parameters as a dict

```
>>> def demo(**p):
    print(p)

>>> demo(x=1,y=2)
{'x': 1, 'y': 2}
>>> demo(x=1,y=2,z=3)
{'x': 1, 'y': 2, 'z': 3}
>>>
```

keyword arguments

Dict

Receiving parameters as a dict

```
>>> def demo(**p,v):
    print(p)

SyntaxError: invalid syntax
>>>
```

SyntaxError

```
>>> def demo(**p,v=1):
    print(p)

SyntaxError: invalid syntax
>>>
```

Receiving parameters as tuple/dict

```
>>> def demo(x,y,*t,**d):
    print(x,y,t,d)

>>> demo(1,2,3,4,a=5,b=6)
1 2 (3, 4) {'a': 5, 'b': 6}
>>>
```

Anonymous functions

- Anonymous functions are not declared in the standard manner by using the **def** keyword.

- Use the **lambda** keyword to create small anonymous functions.

```
lambda : expression
lambda arg1, arg2, ..., argN: expression
```

Anonymous function rules

- Lambda forms can take any number of arguments but return just one value in the form of an expression. They can not contain commands or multiple expressions.

```
>>>func = lambda : 5+3
>>>func
8

>>>func = lambda a, b: a+b
>>>sum = func(5,3)
>>>print "sum is: ", sum
sum is: 8
```

```
>>>func = lambda a,b: a+b; a-b
```

Nested Functions

Functions can be defined in a function body

```
def maker(n):
    def action(x):
        return x ** n
    return action
f = maker(2)
print(f)
print(f(3))
```

Function action is defined in the function maker

Output

```
<function maker.<locals>.action at 0x00C6B8E8>
9
>>>
```

Learning Objectives

- Functions
- Scope

Scope

- A scope defines the visibility of a name (variable, function, etc.) within a block
- If a local name is defined in a block, its scope includes that block
- If the definition occurs in a function block, the scope extends to any blocks contained within the defining one, unless a contained block introduces a different binding for the name
- When a name is used in a code block, it is resolved using the nearest enclosing scope
- When a name is not found at all, a NameError exception is raised
- Scopes are either non-overlapped or nested

Scope

```
x = "global"
def foo():
    x = "non-local"
    def bar():
        x = "local"
        print(x)
    print(x)
    bar()
print(x)
foo()
```

Scopes:

- global
- foo
- bar

The scope of x

- global x: global - foo
- non-local x: foo - bar
- local x: bar

Output

```
global
non-local
local
>>>
```

Scope

```
x = "global"
def foo():
    x = "non-local"
    def bar():
        print(x)
    print(x)
    bar()
print(x)
foo()
```


Scopes:

- global
- foo
- bar

The scope of x

- global x: global - foo
- non-local x: foo - bar

Output

```
global
non-local
non-local
>>>
```

- When a **name** is not found in the current block, it is resolved using the **nearest enclosing scope**

Scope

```
x = "global"
def foo():
    def bar():
        x = "local"
        print(x)
    print(x)
    bar()
print(x)
foo()
```

Scopes:

- global
- foo
- bar

The scope of x

- global x: global – foo
- local x: bar

Output

```
global
global
local
>>>
```

Scope

```
def foo():
    def bar():
        print(x)
        x = "local"
    print(x)
    bar()
print(x)
foo()
```

Scopes:

- global
- foo
- bar

The scope of x

- local x: bar

When a name is not found at all, a NameError exception is raised

Scope

```
x = "global"
def foo():
    x = "non-local-foo"
    def bar():
        x = "local-bar"
        print(x)
    print(x)
    bar()
def baz():
    x = "local-baz"
    print(x)
print(x)
foo()
baz()
bar()
```

Scopes:

- global
- foo
- bar
- baz

The scope of x

- global x: global
- non-local-foo x: foo
- local-bar x: bar
- local-baz x: baz

Scope foo and baz are non-overlapped

Scope

```
x = "global"
def foo():
    x = "non-local-foo"
    def bar():
        x = "local-bar"
        print(x)
    print(x)
    bar()
def baz():
    x = "local-baz"
    print(x)
print(x)
foo()
baz()
bar()
```

Output

```
global
non-local-foo
local-bar
local-baz
Traceback (most recent call last):
  File "D:\Test\fib.py", line 16, in
<module>
    bar()
NameError: name 'bar' is not
defined
```

Global Statements

- If the global statement global x occurs within a block, all uses of the name x specified in the statement refer to the binding of that name in the global namespace
- The global statement has the same scope as a name binding operation in the same block
- If the nearest enclosing scope for a free variable contains a global statement, the free variable is treated as a global
- If no global x exists, then NameError

Global Statements: Example

```
x = "global"
def foo():
    x = "non-local"
    def bar():
        global x
        print(x)
    print(x)
    bar()
print(x)
foo()
```

Output

```
global
non-local
global
>>>
```

Read global x in local block

Global Statements: Example

```
x = "global"
def foo():
    x = "non-local"
    def bar():
        global x
        x = "newglobal"
        print(x)
    print(x)
    bar()
print(x)
foo()
print(x)
```

Output

```
global
non-local
new-global
new-global
>>>
```

Rebinding global x in local block

Global Statements: Example

```
x = "global"
def foo():
    x = "non-local"
    def bar():
        def baz():
            print(x)
        baz()
    global x
    print(x)
    bar()
print(x)
foo()
```

Output

```
global
non-local
global
>>>
```

Read global x in nearest enclosing scope, even global is declared after baz()

Nonlocal Statements

- The nonlocal statement nonlocal x causes corresponding name x to refer to previously bound variables in the nearest enclosing function scope (excluding global scope)
- SyntaxError is raised at compile time if the given name does not exist in any enclosing function scope

Nonlocal Statements: Example

```
x = "global"
def foo():
    x = "non-local"
    def bar():
        x = "local"
        def baz():
            nonlocal x
            print(x)
        baz()
    bar()
foo()
```

Output

```
local
>>>
```

x to refer to previously bound variables in the nearest enclosing function scope

Nonlocal Statements: Example

```
x = "global"
def foo():
    x = "non-local"
    def bar():
        def baz():
            nonlocal x
            print(x)
        baz()
    bar()
foo()
```

Output

```
non-local
>>>
```

x to refer to previously bound variables in the nearest enclosing function scope

Nonlocal Statements: Example

```
x = "global"
def foo():
    def bar():
        def baz():
            nonlocal x
            print(x)
        baz()
    bar()
foo()
```

Output

SyntaxError: no binding for nonlocal 'x' found (<string>, line 5)

x cannot refer to previously bound variables in the global scope

Readings (recommended)

- The Python Tutorial
  - 4.7.1 – 4.7.2 in 4. More Control Flow Tools
  - 6. Modules

Recap

- Functions
- Scopes