More on Functions

Computer Science & Engineering Department

The Chinese University of Hong Kong

Content

- LEGB & Scope
- Revisit function parameters
- Arbitrary number of parameters

Identifiers in Program

- Names of variables, functions, modules can collide with others – same name used unintentionally (Python allows this)
- Managed using name spaces
- Encapsulation of names through levels of abstraction
- Three levels of encapsulation
  - LEGB rule for simple variables
  - Qualified names
  - modules

LEGB

- **L**: local
- **E**: Enclosing function definitions
- **G**: Global
- **B**: built-in functions
- When Python is looking for meaning attached to a name, it search the scope in the order: *Local, Enclosing, Global, Built-in*

```
>>> x = 42
>>> def afun():
...  x = 12
...  print (x)
>>> afun()
12
>>> print (x)
42
```

Only creating local var

Enclosing

- Occurs when one function is defined inside another
- Each function definition creates a new scope for variables at that level

```
>>> def a(x):
...     def b(y):
...         print (x + y)
...     y = 11
...     b(3)
...     print (y)
...
>>> a(4)
7
11
```

Name Scope

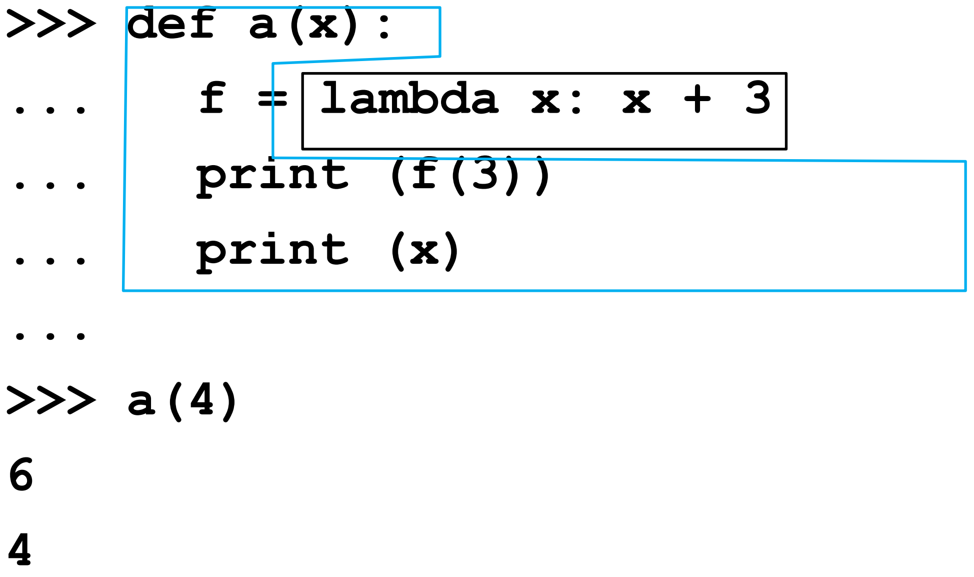
- Names defined outside functions have *global* scope
- Any local names will shadow the global (same name)
- All values & names destroyed after return

```
>>> x=4
>>> def scopetest(a):
...     return x + a
...
>>> print (scopetest(3))
 7
>>>
```

```
x=4
>>> def scopeTest(a):
...     x=7
...     return x + a
...
>>> print (scopeTest(3))
10
```

Scopes

- Described as a series of nested boxes
- To find a match for a given variable, the boxes are examined from inside out until the name is found
- Lambda create their own local scope
- Thus distinct from surrounding function scope



```
>>> def a(x):
...     f = lambda x: x + 3
...     print (f(3))
...     print (x)
...
>>> a(4)
6
4
```

Using globals

- To have assignment access on global variables, use global statement

```
>>> b = 2
>>> def scopeTest (a):
...  global b
...  b = 4
...  print ('inside func, b is ', b)
...
>>> a = 1
>>> scopeTest(a)
inside func, b is  4
>>> print ('after func, b is ', b)
after func, b is  4
```

Block Scope

- Python has no block scope like C or Java in loop

```
>>> for i in range(5):
...     print (i) if i%2 == 0 else print (-i)
... print (i)
0
-1
2
-3
4
4
```

This is not an error

Built-in functions

- Functions that are initially part of any Python program e.g. open, zip, etc
- Can be overridden in a different scope
- For example, a programmer can define his/her own open.
- However it will prevent access to the standard function i.e. file open

dir function

- dir can be used to access a list of names in current scope
- Get global scope in topmost level

```
>>> z = 12
>>> def dirTest():
...           x = 34
...           print (dir())
>>> print (dir())
['__builtins__', '__doc__', '__name__', '__package__',
'dirTest', 'pywin', 'z']
>>> dirTest()
['x']
```

dir function

- Can accept an argument
- Return scope of the object

```
>>> dir (dirTest)
['__call__', '__class__', '__closure__', '__code__', '__defaults__', '__delattr__',
'__dict__', '__doc__', '__format__', '__get__', '__getattribute__', '__globals__',
'__hash__', '__init__', '__module__', '__name__', '__new__', '__reduce__',
'__reduce_ex__', '__repr__', '__setattr__', '__sizeof__', '__str__', '__subclasshook__',
'func_closure', 'func_code', 'func_defaults', 'func_dict', 'func_doc', 'func_globals',
'func_name']
>>> import math
>>> dir(math)
['__doc__', '__name__', '__package__', 'acos', 'acosh', 'asin', 'asinh', 'atan', 'atan2',
'atanh', 'ceil', 'copysign', 'cos', 'cosh', 'degrees', 'e', 'exp', 'fabs', 'factorial', 'floor',
'fmod', 'frexp', 'fsum', 'hypot', 'isinf', 'isnan', 'ldexp', 'log', 'log10', 'log1p', 'modf', 'pi',
'pow', 'radians', 'sin', 'sinh', 'sqrt', 'tan', 'tanh', 'trunc']
```

Revision

```python
def foo(x, y):
    print("{} {}" .format(x, y))

x = 3
y = 2

foo(x, y)

foo(y, x)

```

Positional Arguments

- Arguments and parameters are matched by positions (not by names).

Revision

```python
def foo(x, y):
    print("{} {}" .format(x, y))

foo(x=3, y=2)

foo(y=2, x=3)

foo(3, y=2)
```

```
3 2
3 2
3 2
```

Keyword arguments

- Arguments in function call can also be in the form of keyword (match by names)
- Must use keyword argument for all others once used for the first of them

Revision

```
def foo(x, y=2):
    print("{} {}" .format(x, y))

foo(x=3)

foo(3)

foo(x=2, y=3)

foo(2, 3)
```

```
3 2
3 2
2 3
2 3
```

Default arguments

- an argument that assumes a default value if a value is not provided in the function call for that argument
- Default argument must appear later than those without default value i.e. (x=3, y) is not allowed

Function Parameters Revisited

```
def describe_person(first_name, last_name, age=None, favorite_language=None,
died=None):
    print("First name: {:s}".format( first_name.title()))
    print("Last name: {:s}".format( last_name.title()))
    if age:
        print("Age: {:d}".format(age))
    if favorite_language:
        print("Favorite language: {:s}".format( favorite_language))
    if died:
        print("Died at: {:d}".format( died))
    print("\n")

describe_person('ken', 'thompson', age=70)
describe_person('adele', 'goldberg', age=68, favorite_language='Smalltalk')
describe_person('dennis', 'ritchie', favorite_language='C', died=2011)
```

Function Parameters Revisited

```
First name: Ken
Last name: Thompson
Age: 70


First name: Adele
Last name: Goldberg
Age: 68
Favorite language: Smalltalk


First name: Dennis
Last name: Ritchie
Favorite language: C
Died at: 2011
```

Arbitrary Number of Arguments

- Python functions can be flexible to handle different situation by combination of keyword arguments and default value
- But we can do even better
- Consider a function to add up two numbers

```python
# This function adds two numbers together, and prints the sum
def adder(num_1, num_2):
    sum = num_1 + num_2
    print("The sum of your numbers is {:d}.".format(sum))

# Let's add some numbers.
adder(1, 2)
adder(-1, 2)
adder(1, -2)
```

```
The sum of your numbers is 3.
The sum of your numbers is 1.
The sum of your numbers is -1.
```

Arbitrary Number of Arguments

- If we pass three arguments to it, the function will has problem

```
def adder(num_1, num_2):
    sum = num_1 + num_2
    print("The sum of your numbers is {:d}.".format(sum))

# Let's add some numbers.
adder(1, 2, 3)
```

```
TypeError         Traceback (most recent call last)
<ipython-input-1-ef5adcfdedef> in <module>
----> 7 adder(1, 2, 3)

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

Arbitrary Number of Arguments

- For end of the list of arguments with an asterisk in front of it, that argument will collect any remaining values from the calling statement into a tuple

```
def example_function(arg_1, arg_2, *arg_3):
    print('\narg_1:', arg_1)
    print('arg_2:', arg_2)
    print('arg_3:', arg_3)

example_function(1, 2)
example_function(1, 2, 3)
example_function(1, 2, 3, 4)
```

```
arg_1: 1
arg_2: 2
arg_3: ()

arg_1: 1
arg_2: 2
arg_3: (3,)

arg_1: 1
arg_2: 2
arg_3: (3, 4)
```

Arbitrary Number of Arguments

- We can use a *loop* to process these other arguments

```
def example_function(arg_1, arg_2, *arg_3):
    print('arg_1:', arg_1)
    print('arg_2:', arg_2)
    for value in arg_3:
        print('arg_3 value:', value)


example_function(1, 2, 3, 4)
example_function(1, 2, 3, 4, 5)
```

```
arg_1: 1
arg_2: 2
arg_3 value: 3
arg_3 value: 4

arg_1: 1
arg_2: 2
arg_3 value: 3
arg_3 value: 4
arg_3 value: 5
```

Arbitrary Number of Arguments

- Let's rewrite our *adder()* function

```
def adder(num_1, num_2, *nums):
    sum = num_1 + num_2
    for num in nums:
        sum = sum + num

    print("The sum of your numbers is %d." % sum)

# Let's add some numbers.
adder(3,4)
adder(1, 2, 3)
adder(1,2,3,4,5)
```

```
The sum of your numbers is 7.
The sum of your numbers is 6.
The sum of your numbers is 15.
```

Arbitrary Number of Arguments

- Accepting an arbitrary number of keyword arguments is also possible

```python
def example_function(arg_1, arg_2, **kwargs):
    print('\narg_1:', arg_1)
    print('arg_2:', arg_2)
    print('arg_3:', kwargs)

example_function('a', 'b')
example_function('a', 'b', value_3='c')
example_function('a', 'b', value_3='c', value_4='d')
```

```
arg_1: a
arg_2: b
arg_3: {}

arg_1: a
arg_2: b
arg_3: {'value_3': 'c'}

arg_1: a
arg_2: b
arg_3: {'value_3': 'c',
'value_4': 'd'}
```

Arbitrary Number of Arguments

- Keyword arguments are stored as a dictionary

```python
def example_function(arg_1, arg_2, **kwargs):
    print('\narg_1:', arg_1)
    print('arg_2:', arg_2)
    for key, value in kwargs.items():
        print('arg_3 value:', value)

example_function('a', 'b')
example_function('a', 'b', value_3='c')
example_function('a', 'b', value_3='c', value_4='d')
```

```
arg_1: a
arg_2: b

arg_1: a
arg_2: b
arg_3 value: c

arg_1: a
arg_2: b
arg_3 value: c
arg_3 value: d
```

Exercise

- Can you rewrite the adder() with the following function signature?

```
def adder(num_1, num_2, **nums):
    # your implementation goes here
```

First Example Rewritten

```
def describe_person(first_name, last_name, **kwargs):
    print("First name: %s" % first_name.title())
    print("Last name: %s" % last_name.title())
# Optional information:
    for key in kwargs:
        print("%s: %s" % (key.title(), kwargs[key]))
    print("\n")

describe_person('brian', 'kernighan', favorite_language='C', famous_book='The
C Programming Language')
describe_person('dennis', 'ritchie', favorite_language='C', died=2011,
famous_book='The C Programming Language')
describe_person('guido', 'van rossum', favorite_language='Python',
company='Dropbox')
```

Function Parameters

```
First name: Brian
Last name: Kernighan
Favorite_Language: C
Famous_Book: The C Programming Language

First name: Dennis
Last name: Ritchie
Favorite_Language: C
Died: 2011
Famous_Book: The C Programming Language

First name: Guido
Last name: Van Rossum
Favorite_Language: Python
Company: Dropbox
```

Pickle & Dictionary

- Dictionary is best to pair with Pickle

```
import pickle       # import to use
info = {'name':'Batman', 'age':82, 'weight':180}
f = open('pickle1.pyp','wb')
pickle.dump(info, f)   # store the dictionary in a file
```

- Later on, in another program

```
>>> import pickle
>>> f = open('pickle1.pyp', "rb")
>>> d = pickle.load(f)
>>> for i in d.keys():
>>>     print (f"{i}:{d[i]}",end=',')
name:Batman,age:82,weight:180
```

Appendix

Try except

- Similar to the try-catch mechanism in other programming language, python use try-except-else-finally block to handle exception

```
def divide(x, y):
    try:
        result = x // y     # Floor Division : Gives only Fractional
    except ZeroDivisionError:
        print ("Sorry ! You are dividing by zero ")
    else:
        print ("Yeah ! Your answer is :", result)

divide(3, 2)

divide(3, 0)

divide(3,'1')

>>> Yeah ! Your answer is : 1

>>> Sorry ! You are dividing by zero

> Traceback (most recent call last):
    File "COURSE/python/divide.py", line 10, in <module>
       divide(3,'1')
    File "COURSE/python/divide.py", line 3, in divide
        result = x //y
  TypeError: unsupported operand type(s) for //: 'int' and 'str'
```

Raise exception

- When the program is running at a point that the current function cannot solve the problem, we can raise an exception for caller to handle

```
def divide(x, y):
    try:
        result = x // y     # Floor Division : Gives only Fractional
    except ZeroDivisionError:
        print ("Sorry ! You are dividing by zero ")
    except Exception as e:
        raise
    else:
        print ("Yeah ! Your answer is :", result)


try:

    divide(1, 's')

except Exception as e:

    print (e)


>> unsupported operand type(s) for //: 'int' and 'str'
```

Restoring redirected Standard I/O

- After redirecting the stdout, how to restore it?

```
import sys
fout = sys.stdout = open('output.txt', 'w')
with open('error.txt', 'w') as ferr:
    try:
        print("see where this goes")
        print(5 / 4)
        print(7.0 / 0)
    except Exception as e:
        print(type(e).__name__, e, sep=": ", file=ferr)
sys.stdout = sys.__stdout__   # reset stdout to normal
fout.close()
```

In output.txt

```
see where this goes
1.25
```

In error.txt

```
ZeroDivisionError: float
division by zero
```