Lecture 3: Python Functions

IPEC Winter School 2015
B-IT

Dr. Tiansi Dong & Dr. Joachim Köhler
Python Functions

- arguments
- Global vars
- Files/streams

Function

- return obj
- Global vars
- Mutable arguments
- Files/streams
function design concepts

- Avoid using global variables
- Do not change mutable arguments (avoid side-effect)
- Functions should have a single and unified purpose
- Functions shall be small
- Avoid changing variables in other module files directly
Mutable and Immutable arguments

```python
def func(arg_str ="0", arg_list = []):
    print(arg_str, arg_list)
    arg_str += "1"
    arg_list.append("0")

for i in range(4):
    Func()
0, []
0, ['0]
0, ['0','0']
0, ['0', '0', '0']
```
Recursive functions

- Task: given a list of numbers, calculate the sum of these numbers

```python
def SumOfAllMembers(L):
    if not L:
        return 0
    else:
        return L[0] + SumOfAllMembers(L[1:])
```

SumOfAllMembers([1,2,3,4,5])
15
Recursive functions

- Thinking recursively
  - What are “natural numbers”? 
  - Can we define them recursively? 
  - In computer science, “existence” equals to “being created” 
  - Can you define rational numbers? 
  - Can you define spatial relations? 
  - What is a list? 
  - If an object can be defined recursively, so will be operations on these objects.
Recursive functions

- **Thinking recursively**

```python
def SumOfAllMembers(L):
    head, *tail = L

    return head if not tail else head + SumOfAllMembers(tail)

def SumOfList(L):
    if L:
        return SumOfAllMembers(L)
    else:
        return 0

def SumOfAllMembers(L):
    return 0 if not L else L[0] + SumOfAllMembers(L[1:]):
```

```python
return 0 if not L else L[0] + SumOfAllMembers(L[1:]):
```
Python Function are objects

```python
>>> X = SumOfAllMembers

>>> X([2,3,4,5])
14
```
Python functions are objects

- Can be assigned to a variable
- Have attributes
- Have annotations in Python 3.0
def picking_up_telephone(who):
    def calling(whom):
        def echo(message):
            print(who+'->'+whom+':'+message)
            return echo
        return calling
    return calling

Tom_picking_up_telephone=picking_up_telephone('Tom')
Tom_call_Sam = Tom_picking_up_telephone('Sam')
Tom_call_Sam('How are you?')

Tom->Sam:How are you?
Python Function are objects

- User defined attribute

  Tom_picking_up_telephone.when = 'feel alone'

- Function annotation in Python 3.0
  - Arbitrary used-defined data about a function's argument and result
  - Coded in def header line
  - Arbitrary expressions associated with arguments and return value

```python
def func(a, b, c):
    return a + b + c

def func(a: 'number', b: 'number', c: 'number') -> int:
    return a + b + c

func.__annotations__

{'a': 'number', 'c': 'number', 'b': 'number', 'return': <class 'int'>}
Python Function are objects

Tom_picking_up_telephones.__name__
calling

Tom_call_Sam.__name__
echo

Tom_call_Sam.__code__
<code object echo at 0x02972D40, file "<pyshell#42>", line 3>
dir(Tom_call_Sam)
['__annotations__', '__call__', '__class__', '__closure__', '__code__', '__defaults__', '__delattr__', '__dict__', '__doc__', '__eq__', '__format__', '__ge__', '__get__', '__getattribute__', '__globals__', '__gt__', '__hash__', '__init__', '__kwdefaults__', '__le__', '__lt__', '__module__', '__name__', '__ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr__', '__setattr__', '__sizeof__', '__str__', '__subclasshook__']
Lambda function

- Lambda function is an anonymous function
- Lambda function provides a uniformed method to define functions

```python
lambda arg1, arg2, .. argn: f(arg1, ..., argn)
```

```python
increase = lambda a: a + 1
```

- Lambda function is an expression, not a statement
  - Can appear where `def` statement can not appear
- Lambda body is a single expression, not a block of statements
Functional programming

- Functional: a function which takes functions as inputs
- Mapping functions over sequences
  ```python
  >>> lst = [1, 2, 3, 4, 5, 6]
  >>> list(map(lambda x: x+1, lst))
  [2, 3, 4, 5, 6, 7]
  ```
- Filters and reduces functions
  ```python
  >>> list(filter(lambda x: x>3, lst))
  [4, 5, 6, 7]
  >>> from functools import reduce
  >>> reduce(lambda x, y: x+y, lst)
  21
  ```
- **How is the `reduce` function implemented?**

```python
>>> def my_reduce(func, seq):
    result = seq[0]
    for nxt in seq[1:]:
        result = func(result, nxt)
    return result

>>> my_reduce((lambda x,y: x+y), lst)
21
```
Iterations and Comprehensions

- Iteration over sequences is common and pervasive
  >>> for x in [1,2,4]: print(x, end='*')
  1*2*4*

- File iterator
  >>> f = open('scripts\script0.py')
  >>> f.readline()
  'this file has only one line'
  >>> f.readline()
  ''
  >>> f.readline()
  ''
**Iterations and Comprehensions**

- **File iterator**

  ```python
  >>> f = open('scripts\script0.py')
  >>> f.__next__()
  'this file has only one line'
  >>> f.__next__()
  Traceback (most recent call last):
      File "<pyshell#35>", line 1, in <module>
        f.__next__()
  StopIteration
  ```
Iterations and Comprehensions

- File iterator

```python
>>> for line in open('scripts\script0.py'):
    print(line)
'this file has only one line'

>>> for line in open('scripts\script0.py').readlines():
    print(line)
'this file has only one line'

    ##not a good solution, when the file is too big

>>> f = open('scripts\script0.py')

>>> while True:
    line = f.readline()
    if not line: break
    print(line)

    ##slow than for loop, for iterators run at C speed inside Python, while loop runs at Python virtual machine
```
Iterations and Comprehensions

- Manual Iteration: `iter()` and `next()`
  - X is an iterable object
  - `iter(X)` initializes an iterator object
  - `next(X)` returns the next object

```python
>>> f = open('scripts\script0.py')
>>> iter(f) is f
True

>>> next(f)
'this file has only one line'
```
>>> next(f)

Traceback (most recent call last):
  File "<pyshell#35>", line 1, in <module>
    f.__next__()
StopIteration

>>> Lst = [1,2,3]
>>> I = iter(L)
>>> next(I)
1
>>> Lst = [1,2,3]
>>> I = iter(L)
>>> while True:
        try:
            X = next(I)
        except StopIteration:
            break
        print(X, end='')
1 2 3
List Comprehensions

- List comprehension is a very common place for iteration
  
  ```python
  >>> Lst = [1, 2, 3]
  >>> for index in range(len(lst)):
  ...     Lst[index] += 10
  
  >>> Lst0 = [x + 10 for x in Lst]
  ```

- Its syntax is inspired from the set construction notation in set theory:
  - A new set S is constructed such that each of its member e is such and such
  - Lst0 is constructed such that each element x is 10 greater than elements in Lst
List Comprehensions

- Extended list comprehension

```python
>>> Lst0 = [x + 10 for x in Lst if x < 3]
```

- Some built-in operations on lists (try)
  - `sum([2,3,4,5,6])`
  - `any(['asdf', '', '34', 41])`
  - `all(['asdf', '', '34', 41])`
  - `max([23,2,2,3,4,5])`
  - `min([23,2,2,3,4,5])`
  - `zip('abc','xyz')`
  - `zip('abc','xyzwsu')`
List Comprehension as functional tool

- Compare:

```python
>>> [x for x in range(5) if x % 2 == 0]
[0,2,4]
>>> list(filter((lambda x:x % 2 == 0),range(5)))
[0,2,4]
```

- Which one do you like? Why?

- More examples

```python
>>> [(x,y) for x in range(5) if x % 2 == 0 for y in range(5) if y % 2 == 1]
[(0, 1), (0, 3), (2, 1), (2, 3), (4, 1), (4, 3)]
```
Generator function

• A simple generator function

```python
>>> def gensquares(N):
    for I in range(N):
        yield I*I

>>> for I in gensquares(5):
    print(I, end=' ')  # 0 1 4 9 16
```

• `yield` vs. `return`
  - With `yield` the process is suspended and will be resumed when next value generation is called.
  - With `return` the process is exited.
Generator function

• To continue the example

```python
gensquares(N):
    for I in range(N):
        yield i*i
```

```python
>>> X = gensquares(2)
>>> next(X)
0
>>> next(X)
1
```
Generator function

```python
>>> next(X)
Traceback (most recent call last):
  File "<pyshell#18>", line 1, in <module>
    next(X)
StopIteration
```
Generator function

- To continue the example

```python
>>> def gensquares0(N):
    for I in range(N):
        return i*i

>>> X = gensquares0(2)
0
```
Generator function

```python
>>> for i in gensquares(3):
    print(i)

Traceback (most recent call last):
  File "<pyshell#26>", line 1, in <module>
    for i in gensquares(3):
TypeError: 'int' object is not iterable
```
Generator expression

- Syntactically similar to comprehension – only that they are enclosed in parentheses

```python
>>> (x*x for x in range(4))
<generator object <genexpr> at 0x02B96490>

>>> [x*x for x in range(4)]
[0, 1, 4, 9]

>>> g = (x*x for x in range(4))
>>> next(g)
0
```
Coding functionals

- How can the `map` functional implemented?
  
  ```python
  >>> list(map(pow, [1,2,3], [2,3,4,5]))
  [1, 8, 81]
  >>> [x ** y for (x,y) in zip([1,2,3],[2,3,4,5])]
  [1, 8, 81]
  >>> def map0(func, *seqs):
      res = []
      for args in zip(*seqs):
          res.append(func(*args))
      return res
  >>> map0(pow, [1,2,3],[2,3,4,5])
  [1, 8, 81]
  ```
How can the `zip` function implemented?

```python
def myzip(*seqs):
    seqs = [list(S) for S in seqs]
    res = []
    while all(seqs):
        res.append(tuple(S.pop(0) for S in seqs))
    return res
```
Coding functionals

- >>> list(zip0('abc', '1234'))
  [('a', '1'), ('b', '2'), ('c', '3'), (None, '4')]

- >>> def myzip0(*seqs, pad=None):
    seqs = [list(S) for S in seqs]
    res = []
    while any(seqs):
        res.append(tuple((S.pop(0) if S else pad) for S in seqs))
    return res
• # comments
  - In-file documentation

• dir function
  - List of attributes dir(quit)

• Docstring: __doc__
  - In-file documentation attached to objects
  - Strings at the tops of modules, functions, or classes
    - print(sys.__doc__)

• help function
  - Interactive help for objects