01-functions2

February 3, 2017

1 Args are "passed by object", and an object may be returned

- args are bound to objects references
- mutable objects can be changed
- new objects created can be returned
- a single object can be returned
  - multiple values can be returned in a list, dict, set, etc
- function body defines a 'namespace'
  - args and variables defined by assignment in the function body are 'local' to the function

```
In [1]: # scoping example
        # function can reference global value of 'g'
        # 2nd arg, a list, is modified
        # outer value of 'm' is not changed by function

        x = [3,5,7]
        m = 20
        g = 30

        def foo(m, x2):
            # can see g
            print('g',g)
            # created a new local, ignores outer 'm'
            m = 55
            x2[0] = 'mod'

        foo(8, x)
        [m, x]
```

```
g 30
```

```
Out[1]: [20, ['mod', 5, 7]]
```

```
In [2]: # what's going on here????
```

```
        g = 55
        def foo():
            print(g)
            g = 22
        foo()
```

```
        ---------------------------------------------------------------------------------

        UnboundLocalError                           Traceback (most recent call last)

        <ipython-input-2-75eae2fea8ba> in <module>()
          5     print(g)
          6     g = 22
    ----> 7 foo()


        <ipython-input-2-75eae2fea8ba> in foo()
          3 g = 55
          4 def foo():
    ----> 5     print(g)
          6     g = 22
          7 foo()


        UnboundLocalError: local variable 'g' referenced before assignment
```

```
In [ ]: # above may seem weird...well it is
        # the print is looking at the local 'g', not the global one
        # the function body is scanned for assignments, it sees the 'g',
        # treats it as a local, then executes the body, and at 'print(g)'
        # time, the local 'g' is still undefined
```

2 global

- using global is usually a very bad idea
- but, handy for debugging and interactive use
- can see values of function locals w/o prints or debugger

```
In [3]: def foo():
            global x,y,z
            x = 5
            y = x + 20
            z = x - y + x**2
            return(x - y + z//2)
```

```
In [4]: foo()
```

```
Out[4]: -18
```

```
In [5]: [x,y,z]
```

```
Out[5]: [5, 25, 5]
```

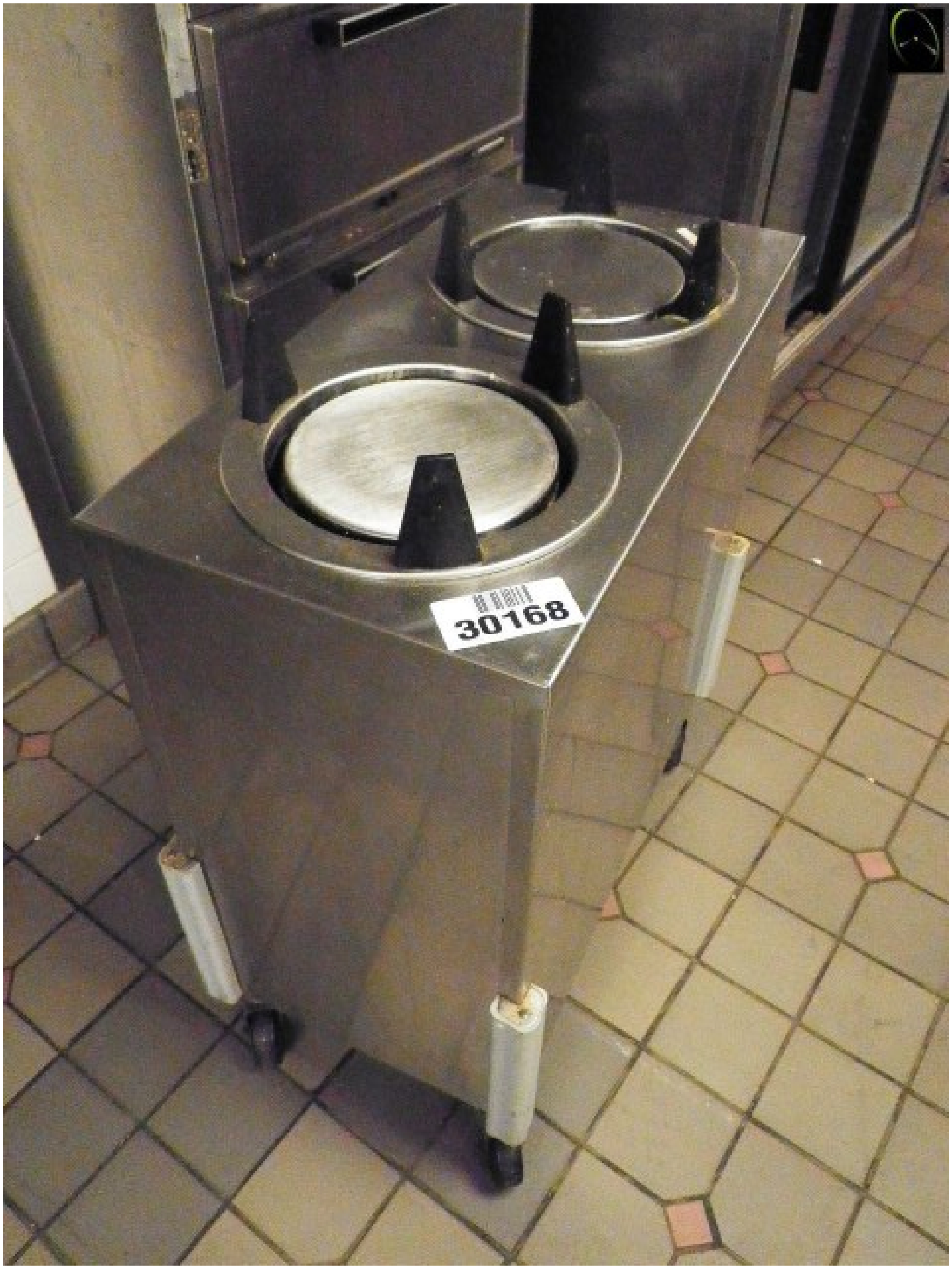
3 Stacks

- a ‘stack’ basically has two operations
  - ‘push’ something onto the stack
  - ‘pop’ something off the stack
  - think of a ‘spring loaded dish rack’

```
In [6]: from IPython.display import Image

        Image('http://images.rasmuscatalog.com/M20217%20Former%20Bank/30168.jpg')
```

```
Out[6]:
```


30168

4 Call stack

- holds runtime info for function calls
- important for understanding recursion, generators, and error handling
- each time a function is called, a new ‘stack frame’ is ‘pushed’ onto the call stack
- each time a function returns, its stack frame is ‘popped’ from the call stack
- nothing special about recursive calls
- demo using spyder

5 lambda

- ‘lambda’ defines anonymous functions(function doesn’t get a name)
- ‘def’ is a statement, ‘lambda’ is an expression, so lambda can go places def can’t
- lambda body is a single expression, so can not be as complex as a lambda
- mainly intended for simple things
- type name is ‘function’

```
In [8]: # z holds a reference to the lambda object defined on the right

        z = lambda x : x + 5
        [z(33), type(z)]

Out[8]: [38, function]
```

```
In [9]: # call each lambda

        [f(10) for f in lams]


        ---------------------------------------------------------------------------

        NameError                                 Traceback (most recent call last)

        <ipython-input-9-2954b04e289d> in <module>()
          1 # call each lambda
          2
    ----> 3 [f(10) for f in lams]


        NameError: name 'lams' is not defined
```

```
In [ ]: # 'map' takes a function and a list as args
        # the function is applied to each element of the list,
        # and the values returned by the function are collected
        # into a new list
        # map is lazy
        def add2(n):
            return n + 2

        list(map(add2, [1,4,3,7]))

In [10]: # with a lambda, can directly pass function as an arg
         # without first setting a name with def -
         # less clutter

         list(map(lambda x : x + 2, [1,4,3,7]))

Out[10]: [3, 6, 5, 9]
```

6 Example: circlePoints

```
In [11]: # first attempt used for loop with accumulation var

         import math

         def circlePoints(n, radius):
             ans = []
             for j in range(n):
                 ang = j * 2 * math.pi / n
                 ans.append([radius * math.cos(ang), radius * math.sin(ang)])
             return ans

In [ ]: circlePoints(4,1)

In [ ]: # use a comprehension and a lambda

        def circlePoints2(n, radius):
            lam = lambda ang: [radius * math.cos(ang), radius * math.sin(ang)]
            return [lam(j*2*math.pi/n) for j in range(n)]

In [ ]: circlePoints2(4,1)

In [ ]: # two lines

        def circlePoints3(n, radius):
            return [(lambda ang: [radius * math.cos(ang), radius * math.sin(ang)])

In [ ]: circlePoints3(4,1)
```

7 Multiple value return

- strictly speaking, a function returns at most one object
- can return easily return multiple values by returning a 'collection' object, like a list
- unpacking can be convenient

```
In [12]: # return one list with two values

         def makePoint(x, y):
             return [x,y]

         makePoint(5,8)

Out[12]: [5, 8]
```

```
In [13]: # unpack

         x , y = makePoint(3,4)

         [x, y]

Out[13]: [3, 4]
```

8 Mutable args can be modified

- So, can return vals w/o return statement

```
In [1]: l = [1,2,3]

        def foo(l):
            l[1] = 55

        foo(l)
        l

Out[1]: [1, 55, 3]
```

9 Function overloading

- Python does not have 'overloaded' functions, like C/C++/Java
- in those languages, can do

```
void foo(float f) { // do float thing }
void foo(string s) ( // do string thing }
```

- no argument types in Python, can't tell the two foo's apart, so no overloading in python
- but, can do something similiar with run time typing

```
In [15]: def foo(arg):
             if isinstance(arg, (int, float)):
                 print('do number thing')
             if isinstance(arg, str):
                 print('do string thing')
```

```
         foo(34.4)
         foo(234)
         foo('')
         foo(dict())

do number thing
do number thing
do string thing
```

10 Function definitions can specify complex argument processing

- Sort of a pattern matching scheme - many possibilities
- Downside - makes function calls more expensive
- Two arg types
  - positional
  - keyword
- Args can be matched or collected

```
In [16]: # three required positional args

         def a3(a,b,c):
             return(a,b,c)

         a3(1,2,3)

Out[16]: (1, 2, 3)

In [17]: # only two args is an error
         # all three must be matched

         a3(1,2)


        ------------------------------------------------------------------------------------

        TypeError                                 Traceback (most recent call last)

        <ipython-input-17-8f4c9c5a78c7> in <module>()
          2 # all three must be matched
          3
    ----> 4 a3(1,2)


        TypeError: a3() missing 1 required positional argument: 'c'
```

```
In [18]: # by using 'keyword args' (a=2), can supply the args in arbitrary order

         [a3(1,2,3), a3(1, c=2, b=3), a3(c=5, a=2, b=8)]

Out[18]: [(1, 2, 3), (1, 3, 2), (2, 8, 5)]
```

```
In [19]: # can give args default values

         def a3(a, b, c=22):
             return([a,b,c])

         [a3(2,3,4), a3(2,3), a3(b=3,a=2), a3(b=3,c=9,a=2)]

Out[19]: [[2, 3, 4], [2, 3, 22], [2, 3, 22], [2, 3, 9]]
```

```
In [20]: # b must get a value

         a3(c=5, a=3)


        ---------------------------------------------------------------------------

        TypeError                                 Traceback (most recent call last)

        <ipython-input-20-7116748d3c99> in <module>()
          1 # b must get a value
          2
    ----> 3 a3(c=5, a=3)


        TypeError: a3() missing 1 required positional argument: 'b'
```

```
In [21]: # can pick up any number of 'unclaimed' positional and keyword args
         # *pos is a tuple
         # **kws is a dictionary
         # all positional args must come before keyword args

         def pk(a, b, c=5, *pos, **kws):
             return([a, b, c, pos, kws])

         pk(1,2,3,4,5,6, foo=5, bar=9)

Out[21]: [1, 2, 3, (4, 5, 6), {'bar': 9, 'foo': 5}]
```

11 For clarity, can force args to be specified with keywords

- args following a '*' must be keywords

```
In [22]: def foo(*, a, b):
             return 2*a + 3*b

          foo(3,5)
```

```
        ------------------------------------------------------------------------


        TypeError                                 Traceback (most recent call last)

        <ipython-input-22-47e1662101bc> in <module>()
          2     return 2*a + 3*b
          3
    ----> 4 foo(3,5)


        TypeError: foo() takes 0 positional arguments but 2 were given
```

```
In [23]: foo(a=4, b=8)

Out[23]: 32
```

12 Example: print function has keyword args

```
In [24]: print(1,2,3,4)

1 2 3 4
```

```
In [25]: print(1,2,3,4, sep='--')

1--2--3--4
```

```
In [26]: # finish print with 3 new lines, instead of 1

         print(1,2,3,4,end='\n\n\n')

1 2 3 4
```

13 Example: discriminate on number of args

- in C++/Java

void foo(float f) { // do one arg thing }
void foo(float f, float f2) (// do two arg thing }

```
In [27]: def onetwo(*pos):
             if 1 == len(pos):
                 a = pos[0]
                 print('one arg',a)
             else:
                 [a,b] = pos
                 print('two args', a, b)
```

```
In [28]: onetwo(1)
```

```
one arg 1
```

```
In [29]: onetwo(1,2)
```

```
two args 1 2
```

14 Function caller can manipulate how arguments are passed

```
In [30]: # '*' 'spreads' a list over the positional args

         def foo(a,b,c):
             return([a,b,c])

         l = [1,2,3]

         foo(l[0],l[1],l[2])
```

```
Out[30]: [1, 2, 3]
```

```
In [31]: foo(*[1,2,3])
```

```
Out[31]: [1, 2, 3]
```

```
In [32]: # *pos gets the range
         # '**kw' maps a dictionary into keyword args

         def bar(*pos, **kw):
             return(pos, kw)

         d = {'mudd':'compsci', 'butler':'library'}
         bar(*range(5), **d)
```

```
Out[32]: ((0, 1, 2, 3, 4), {'butler': 'library', 'mudd': 'compsci'})
```

15 Example: ‘printf’ style args

```
In [34]: def printf(controlString, *vals):
             print(controlString)
             print(vals)
             return controlString.format(*vals)

         printf('an int: {} a float: {} a string: {}', 234, 3.34, 'foo')
```

```
an int: {} a float: {} a string: {}
(234, 3.34, 'foo')
```

```
Out[34]: 'an int: 234 a float: 3.34 a string: foo'
```

16 Top level builtin functions

- doc for all the builtins

17 All builtins

- functions
- classes
- a few othre random things
- do NOT redefine any of them

```
In [35]: import builtins

         [f for f in dir(builtins) ]
```

```
Out[35]: ['ArithmeticError',
          'AssertionError',
          'AttributeError',
          'BaseException',
          'BlockingIOError',
          'BrokenPipeError',
          'BufferError',
          'BytesWarning',
          'ChildProcessError',
          'ConnectionAbortedError',
          'ConnectionError',
          'ConnectionRefusedError',
          'ConnectionResetError',
          'DeprecationWarning',
          'EOFError',
          'Ellipsis',
          'EnvironmentError',
          'Exception',
          'False',
          'FileExistsError',
          'FileNotFoundError',
          'FloatingPointError',
          'FutureWarning',
          'GeneratorExit',
          'IOError',
          'ImportError',
          'ImportWarning',
          'IndentationError',
          'IndexError',
          'InterruptedError',
          'IsADirectoryError',
          'KeyError',
          'KeyboardInterrupt',
          'LookupError',
          'MemoryError',
          'NameError',
          'None',
          'NotADirectoryError',
          'NotImplemented',
          'NotImplementedError',
          'OSError',
          'OverflowError',
          'PendingDeprecationWarning',
          'PermissionError',
          'ProcessLookupError',
          'RecursionError',
          'ReferenceError',
          'ResourceWarning',
          'RuntimeError',
          'RuntimeWarning',
          'StopAsyncIteration',
          'StopIteration',
          'SyntaxError',
          'SyntaxWarning',
          'SystemError',
          'SystemExit',
          'TabError',
          'TimeoutError',
          'True',
          'TypeError',
          'UnboundLocalError',
          'UnicodeDecodeError',
          'UnicodeEncodeError',
          'UnicodeError',
          'UnicodeTranslateError',
          'UnicodeWarning',
          'UserWarning',
          'ValueError',
          'Warning',
          'ZeroDivisionError',
          '__IPYTHON__',
          '__build_class__',
          '__debug__',
          '__doc__',
          '__import__',
          '__loader__',
          '__name__',
          '__package__',
          '__spec__',
          'abs',
          'all',
          'any',
          'ascii',
          'bin',
          'bool',
          'bytearray',
          'bytes',
          'callable',
          'chr',
          'classmethod',
          'compile',
          'complex',
          'copyright',
          'credits',
          'delattr',
          'dict',
          'dir',
          'divmod',
          'dreload',
          'enumerate',
          'eval',
          'exec',
          'filter',
          'float',
          'format',
          'frozenset',
          'get_ipython',
          'getattr',
          'globals',
          'hasattr',
          'hash',
          'help',
          'hex',
          'id',
          'input',
          'int',
          'isinstance',
          'issubclass',
          'iter',
          'len',
          'license',
          'list',
          'locals',
          'map',
          'max',
          'memoryview',
          'min',
          'next',
          'object',
          'oct',
          'open',
          'ord',
          'pow',
          'print',
          'property',
          'range',
          'repr',
          'reversed',
          'round',
          'set',
          'setattr',
          'slice',
          'sorted',
          'staticmethod',
          'str',
          'sum',
          'super',
          'tuple',
          'type',
          'vars',
          'zip']
```

18 operator module

- consists of functions that implement Python operators
- useful for functional programming
- doc

```
In [36]: import operator

         [operator.add(2,3), operator.mod(5,2), operator.concat('foo', 'bar'), oper
```

```
Out[36]: [5, 1, 'foobar', [1, 2, 3, 4, 5, 6]]
```

19 Horrible!! What is going on??

```
In [37]: def foo(x=[]):
             x.append(1)
             return(x)
```

```
In [38]: foo([2,3])
```

```
Out[38]: [2, 3, 1]
```

```
In [39]: foo([])
```

```
Out[39]: [1]
```

```
In [40]: foo()
```

```
Out[40]: [1]
```

```
In [41]: foo()
```

```
Out[41]: [1, 1]
```

```
In [42]: foo()
```

```
Out[42]: [1, 1, 1]
```

```
In [43]: foo()
```

```
Out[43]: [1, 1, 1, 1]
```

```
In [47]: # the x=[] happens at function definition time, not at invocation time
         # so a redefinition will 'reset'

         def foo(x=list()):
             x.append(1)
             return(x)

         foo()
```

```
Out[47]: [1]
```

```
In [48]: foo()
```

```
Out[48]: [1, 1]
```

```
In [49]: foo()
```

```
Out[49]: [1, 1, 1]
```