05-functions

January 27, 2017

1 def

- used to define functions
- is an executable statement, not a declaration, that can appear anywhere, even inside another function definition
- updates an existing function definition
- objects are passed as arguments
- variables in function body are 'local' to the function. they disappear when the function terminates
- like 'if', def defines a statement block, so there must be a ':' at the end of the def line, and all the statements in the function body must be indented

```
In [2]: # both prints are indented, forming a statement block

        def foo(n):
            print(n)
            print(n*n)

        foo(8)

8
64
```

```
In [3]: # redefine foo

        def foo(n):
            print(n/2)
            print(2*n)

        foo(8)

4.0
16
```

```
In [4]: # bad indenting
```

```
        def foo(n):
          print(n)
            print(n*n)


          File "<ipython-input-4-15e5a301bfe0>", line 5
        print(n*n)
        ^
    IndentationError: unexpected indent
```

2 return statement

```
In [5]: def foo():
            print('here')
            # exit foo, no explicit return val, so return 'None'
            return
            print('there')

        foo()

here
```

```
In [6]: def foo():
            print('here')
            # exit, return 234
            return 234
            print('there')

        foo()

here


Out[6]: 234
```

```
In [7]: # falling off the end of a function...

        def foo():
            print('here')
```

```
In [8]: # is equivalent to

        def foo():
            print('here')
            return None

        foo()
```

```
here
```

3 args are not typed

```
In [9]: # since args are not types,
        # foo can take any type of args that work with '*'

        def foo(a,b):
            return (a*b)
```

```
In [10]: foo(2,5)
```

```
Out[10]: 10
```

```
In [11]: foo('bar', 4)
```

```
Out[11]: 'barbarbarbar'
```

```
In [12]: foo(3+5j, 10)
```

```
Out[12]: (30+50j)
```

4 Example - palindromes

- unchanged under reverse

```
In [13]: pals = ['radar', 'level', 'larry', 'step on no pets']

         def pal(s):
             l = len(s)
             # len of half, ignoring middle if odd
             lh = l//2
             for j in range(0, lh):
                 if s[j] != s[l-j-1]:
                     return False
             return True

         for p in pals:
             print(p, pal(p))
```

```
radar True
level True
larry False
step on no pets True
```

5 Python supports recursive functions

```
In [14]: def fact(n):
             if n == 0:
                 return(1)
             else:
                 return(n * fact(n-1))

         fact(5)

Out[14]: 120
```

6 Supply a docstring(and comments) to increase readibility

- a docstring is a comment placed as the first statement in the function definition
- can use triple quotes(‘”) for multiline docstrings
- many tools(like spyder) will display the docstring automatically
- in Jupyter notebooks, type function name, then hit shift-tab
- docstring is available as a function attribute

```
In [15]: # a comment as the first line of the function
         # in triple quotes can be accessed by interactive documentation tools

         def fact(n):
             "This function recursively computes factorial"
             # termination case
             if n == 0:
                 return(1)
             else:
                 # solve a simpler problem
                 return(n * fact(n-1))

         [fact(5), fact.__doc__]

Out[15]: [120, 'This function recursively computes factorial']

In [16]: fact(4)

Out[16]: 24

In [17]: # recursive version of pal
         # checks first and last chars, then works on the middle

         def palr(s):
             # empty
             if len(s) == 0:
                 return True
             # middle when odd
```

```
            if len(s) == 1:
                return True
            if s[0] == s[-1]:
                # first and last chars are the same
                #
                return palr(s[1:-1])
            else:
                return False

        for p in pals:
            print(p, palr(p))
```

```
radar True
level True
larry False
step on no pets True
```

```
In [18]: # easier way to do pal
         # just reverse and compare

         def paleasy(s):
             return s == s[::-1]

In [19]: # pal function also works on lists

         pal([1,2,5,2,1])

Out[19]: True

In [20]: # and tuples

         pal((1,2,5,2,1))

Out[20]: True
```

7 Functions are objects

- like everything else in python, functions are just objects, with the special property that a function can be 'applied to arguments'
- functions can be
  - assigned to variables
  - passed as arguments
  - returned as values
  - held in collections

```
In [21]: # foo refers to same function object as fact
```

```
        foo = fact
        print(foo)
        print(fact)
        foo(50)

<function fact at 0x106a58d08>
<function fact at 0x106a58d08>


Out[21]: 30414093201713378043612608166064768844377641568960512000000000000

In [22]: # takes a function as 2nd arg

         def outer2(n, inner):
             return(inner(n), inner(n-1))

         outer2(4, fact)

Out[22]: (24, 6)

In [23]: # stick some functions in a list and run each of them

         def f1(n):
             return n + 1

         def f2(n):
             return n + 2

         def f3(n):
             return n + 3

         flist = [f1,f2,f3]
         flist

Out[23]: [<function __main__.f1>, <function __main__.f2>, <function __main__.f3>]

In [24]: # run the list of functions

         [f(10) for f in flist]

Out[24]: [11, 12, 13]
```

8 Can nest function definitions

```
In [25]: def outer(n):
             # nested def
             def inner(z):
                 return(z+1)
             # return two values and the inner function
```

```
                return([inner(n), inner(n-1), inner])

            [val1, val2, inner] = outer(4)

            [val1, val2, inner(4)]

Out[25]: [5, 4, 5]
```

9 inner functions can form 'closures'

- advanced technique, but can be very useful

```
In [2]: def outer(n):
            # nested def
            def inner(z):
                # inner will 'capture' the value of n
                return(z+n+1)
            return inner

        inner4 = outer(4)
        print(inner4(10))

        inner8 = outer(8)
        print(inner8(10))

15
19
```