9. Functions

Syntax

```
def name(parameter1, parameter2, . . .):
        "function_docstring"
        function_suite
        return [expression]
```

```
def print1( str ):
    "This prints a passed string into
this function"
    print(str)
    return

print1("Hello")
```

Output

Hello

docstring

```
def function():
    "This line is an optional documentation string or docstring. "

print(function.__doc__)
```

Positional parameters

```
def power(x, y):
        r = 1
        while y > 0:
                r = r * x
                y = y - 1
        return r


>>> power(3, 3)
27
>>> power(6, 4)


>>> power(4, 6)
```

```
>>> power(3)
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
TypeError: power() takes exactly 2 positional arguments (1 given)
>>>
```

Need not remember the parameter order

```
def sum(a,b,c):
    return a+b+c
(x,y,z)=(10,20,30)

print(sum(c=z,a=x,b=y))
```

Output

60

```
def interest(p,t=12,r=7.8):
    "Calculates Rate of Interest"
    return (p*t*r)/100

print(interest(20000,r=8))
```

Output

19200

Passing arguments by parameter name

- >>> power(2, 3)
- 8
- >>> power(3, 2)
- 9
- >>> power(y=2, x=3)
- 9

→ Called as keyword passing

Default Values

- def fun(arg1, arg2=default2, arg3=default3, . . .)

```
>>> def power(x, y=2):
... r = 1
... while y > 0:
        r = r * x
        y = y - 1
... return r
```

- >>> power(3, 3)
- 27
- >>> power(3)
- 9

Variable numbers of arguments

```
def maximum(*numbers):
  if len(numbers) == 0:
        return None
 else:
        max = numbers[0]
        for n in numbers[1:]:
            if n > max:
                max = n
        return max


print(maximum(10,20,34,22,11,23))
```

DEALING WITH AN INDEFINITE NUMBER OF ARGUMENTS PASSED BY KEYWORD

```
def f(x, y, **other):
    print("x=",x,", y=",y,", other.keys=",other.keys())
    print("other.values=",other.values())
    print("**other=",dict(**other))

f(1,2,a=3,b=4,c=5)
```

DEALING WITH AN INDEFINITE NUMBER OF ARGUMENTS PASSED BY KEYWORD

```
def f(x, y, **other):
    print("x=",x,", y=",y,", other.keys=",list(other.keys()))
    print("other.values=",list(other.values()))
    print("**other=",dict(**other))

f(1,2,a=3,b=4,c=5)
```

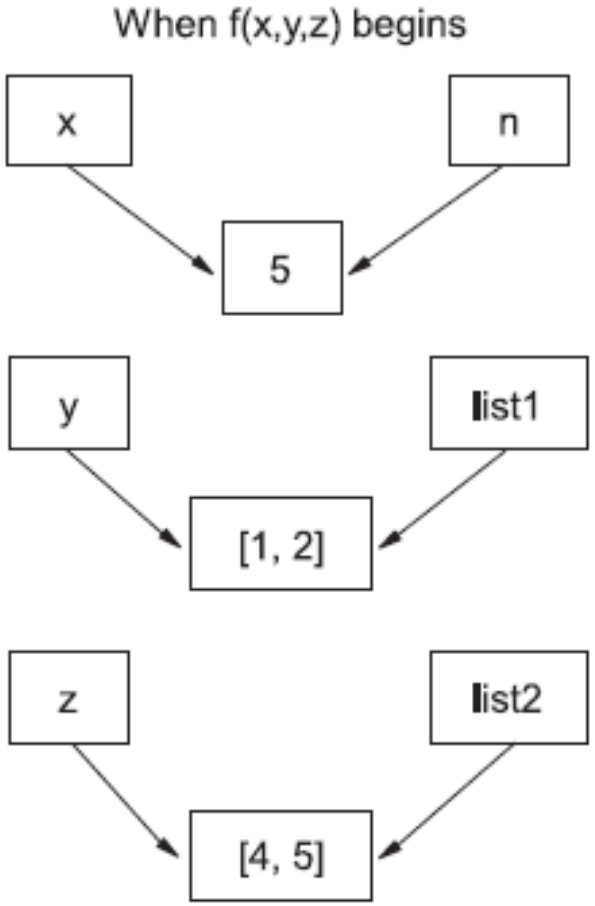
Mutable objects as arguments

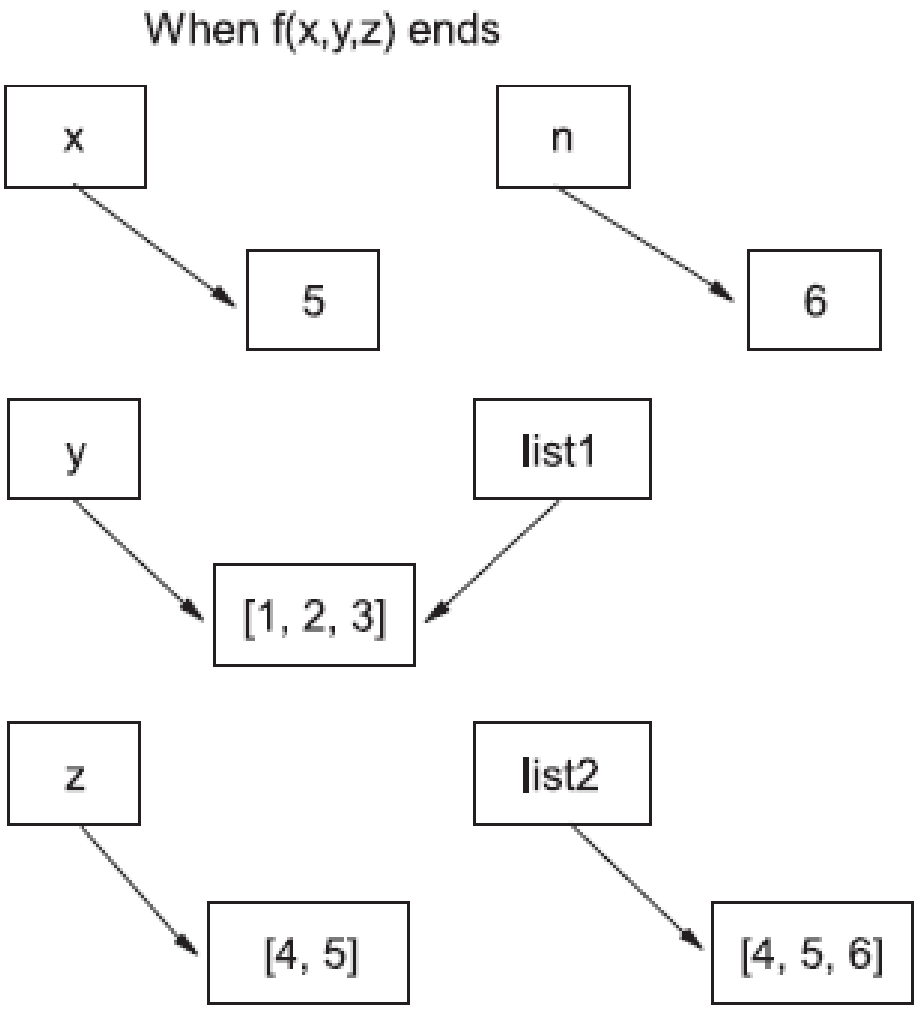
- Arguments are passed in by object reference.
- The parameter becomes a new reference to the object.
- For immutable objects (such as tuples, strings, and numbers)
  - no effect outside the function.
- For mutable object
  - any change made to the object will change what the argument is referencing outside the function.
  - Reassigning the parameter doesn't affect the argument

Mutable objects as arguments

```
def f(n, list1, list2):
        list1.append(3)
        list2 = [4, 5, 6]
        n = n + 1
```

```
x = 5
y = [1, 2]
z = [4, 5]
f(x, y, z)
```




When f(x,y,z) ends
x
5
n
6
y
list1
[1, 2, 3]
z
[4, 5]
list2
[4, 5, 6]

Local, nonlocal and global variables

- Global scope:
  - A variable defined outside all the functions.
  - Declared using global keyword
- Local scope
  - A variable defined within a function.

```
def fact(n):
"""Return the factorial of the given number."""
global a
r = 1
while n > 0:
r = r * n
n = n - 1
return r
```

Local, nonlocal and global variables

- nonlocal statement,
  - Used in nested functions
  - Neither global nor local scope
  - If we change nonlocal variable change appears in local variable
  - which causes an identifier to refer to a previously bound variable in the closest enclosing scope.

Nonlocal

Without using nonlocal

```
def main_function():
    msg = "main-function!"
    def nested_function():
        msg = "Nested Function!"
        print(msg)
    nested_function()
    print(msg)

main_function()
```

Using nonlocal

```
def main_function():
    msg = "main-function!"
    def nested_function():
        nonlocal msg
        msg = "Nested Function!"
        print(msg)
    nested_function()
    print(msg)

main_function()
```

Assigning functions to variables

```
def add(a,b):
    return a+b
def mul(a,b):
    return a*b
var1=add;print(var1(2,4))  #prints 6
var1=mul;print(var1(2,4))  #prints 8
d={'var1':add,'var2':mul}
d['var1'](2,4)
d['var2'](2,4)
```

lambda expressions

- General Form:

  var=lambda parameter1, parameter2, . . .: expression

- lambda expressions are **anonymous** little functions that you can quickly define **inline.**

- **f=lambda a,b,c:a+b+c**
- **print(f(2,3,4)) #prints 9**

lambda expressions

```
l=[1234,23,1,234]
l.sort(key=lambda item:len(str(item)))
print(l)                          #prints [1, 23, 234, 1234]
```

Generator functions

- A generator function is a special kind of function that can be used to define our own iterators.
  - Each iteration's value is returned using the yield keyword.

```
def four():
    x = 0
    while x < 4:
        print("in generator, x =", x)
        yield x
        x += 1

for i in four():
    print(i)
```

```
in generator, x = 0
0
in generator, x = 1
1
in generator, x = 2
2
in generator, x = 3
3
```

```
def four():
    x = 0
    while x < 4:
        print("in generator, x =", x)
        yield x*2
        x += 1

for i in four():
    print(i)
```

```
in generator, x = 0
0
in generator, x = 1
2
in generator, x = 2
4
in generator, x = 3
6
```