11 Functions

Scope of a variable

Consider the following code fragment:

```
x = 20
y = "years"

def myFunction(a,b,c):
    x = a + b + c
    y = a * b * c
    z = a - b - c

    return x,y,z

print(myFunction(4,5,6))
print(x,y)
```

11 Functions

Scope of a variable

Consider the following code fragment:

```
x = 20
y = "years"

def myFunction(a,b,c):
    x = a + b + c
    y = a * b * c
    z = a - b - c

    return x,y,z

print(myFunction(4,5,6))
print(x,y)
```

output:

```
(15, 120, -7)
20 years
```

11 Functions

Scope of a variable

Consider the following code fragment:

```
x = 20
y = "years"

def myFunction(a,b,c):
    x = a + b + c
    y = a * b * c
    z = a - b - c

    return x,y,z

print(myFunction(4,5,6))
print(x,y)
```

scope of local variable x

output:

```
(15, 120, -7)
20 years
```

11 Functions

Scope of a variable

Consider the following code fragment:

```
x = 20
y = "years"

def myFunction(a,b,c):
    x = a + b + c
    y = a * b * c
    z = a - b - c

    return x,y,z

print(myFunction(4,5,6))
print(x,y)
```

scope of local variable y

output:

```
(15, 120, -7)
20 years
```

11 Functions

Scope of a variable

Consider the following code fragment:

```
x = 20
y = "years"

def myFunction(a,b,c):
    x = a + b + c
    y = a * b * c
    z = a - b - c

    return x,y,z

print(myFunctio
print(x,y)
```

local variables x,y,z are released, no access to them is available

output:

```
(15, 120, -7)
20 years
```

11 Functions

Global variables

Consider a slightly modified code fragment:

```
x = 20
y = "years"

def myFunction2(a,b,c):
    global x
    x = x + a + b + c
    y = a - b

    return x

print(myFunction2(4,5,6))
print(x,y)
```

11 Functions

Global variables

Consider a slightly modified code fragment:

```
x = 20
y = "years"

def myFunction2(a,b,c):
    global x
    x = x + a + b + c
    y = a - b

    return x

print(myFunction2(4,5,6))
print(x,y)
```

output:

```
35
35 years
```

11 Functions

Global variables

Consider a slightly modified code fragment:

```
x = 20
y = "years"

def myFunction2(a,b,c):
    global x
    x = x + a + b + c
    y = a - b

    return x

print(myFunction2(4,5,6))
print(x,y)
```

global variables x and y

local variable y

output:

```
35
35 years
```

11 Functions

Namespaces

A *namespace* maps names to objects.
The Python interpreter uses *namespaces* to track all of the objects in a program.

```
number = 50
price = 2.50
discount = 30 # in percent

finalPrice = number*price
if number > 20: # apply the discount
   finalPrice = finalPrice*(100-discount)/100

print(finalPrice)
```

11 Functions

Namespaces

A *namespace* maps names to objects.
The Python interpreter uses *namespaces* to track all of the objects in a program.

global namespace:

number	50
price	
discount	
finalPrice	

```
number = 50
price = 2.50
discount = 30 # in percent

finalPrice = number*price
if number > 20: # apply the discount
   finalPrice = finalPrice*(100-discount)/100

print(finalPrice)
```

11 Functions

Namespaces

A *namespace* maps names to objects.
The Python interpreter uses *namespaces* to track all of the objects in a program.

global namespace:

number	50
price	2.50
discount	
finalPrice	

```
number = 50
price = 2.50
discount = 30 # in percent

finalPrice = number*price
if number > 20: # apply the discount
   finalPrice = finalPrice*(100-discount)/100

print(finalPrice)
```

11 Functions

Namespaces

A *namespace* maps names to objects.
The Python interpreter uses *namespaces* to track all of the objects in a program.

global namespace:

number	50
price	2.50
discount	30
finalPrice	

```
number = 50
price = 2.50
discount = 30 # in percent

finalPrice = number*price
if number > 20: # apply the discount
   finalPrice = finalPrice*(100-discount)/100

print(finalPrice)
```

11 Functions

Namespaces

A *namespace* maps names to objects.
The Python interpreter uses *namespaces* to track all of the objects in a program.

global namespace:

number	50
price	2.50
discount	30
finalPrice	125

```
number = 50
price = 2.50
discount = 30 # in percent

finalPrice = number*price
if number > 20: # apply the discount
   finalPrice = finalPrice*(100-discount)/100

print(finalPrice)
```

11 Functions

Namespaces

A *namespace* maps names to objects.
The Python interpreter uses *namespaces* to track all of the objects in a program.

global namespace:

number	50
price	2.50
discount	30
finalPrice	125

```
number = 50
price = 2.50
discount = 30 # in percent

finalPrice = number*price
if number > 20: # apply the discount
   finalPrice = finalPrice*(100-discount)/100

print(finalPrice)
```

11 Functions

Namespaces

A *namespace* maps names to objects.
The Python interpreter uses *namespaces* to track all of the objects in a program.

global namespace:

number	50
price	2.50
discount	30
finalPrice	87.50

```
number = 50
price = 2.50
discount = 30 # in percent

finalPrice = number*price
if number > 20: # apply the discount
    finalPrice = finalPrice*(100-discount)/100

print(finalPrice)
```

11 Functions

Namespaces

A *namespace* maps names to objects.
The Python interpreter uses *namespaces* to track all of the objects in a program.

In fact, a *namespace* is actually just a normal *Python dictionary* whose keys are the names and whose values are the objects.

Use the built-in functions:
`locals()` to see the current *local namespace*

`globals()` to see the current *global namespace*

11 Functions

Scopes and scope resolution

Scope is the area of code where a name is visible.
Namespaces are used to make scope work.
Each *scope*, such as *global scope* or a *local function scope*, has its own *namespace*.

There are at least three nested scopes that are active at any point in a program's execution:

- Built-in scope : contains all of the built-in names of Python, such as `int()`, `str()`, `list()`, `range()`, etc.
- Global scope : contains all globally defined names outside of any functions.
- Local scope : usually refers to scope within the currently executing function, but is the same as global scope if no function is executing.

11 Functions

Function parameters/arguments

Consider the following code fragment:

```
x = 20
y = [10,20,30]

def myFunction1(a):
    a = a + 10
    return a

def myFunction2(a):
    a.append(40)
    a.remove(20)

print(myFunction1(x))
print(x)
myFunction2(y)
print(y)
```

11 Functions

Function parameters/arguments

Consider the following code fragment:

```
x = 20
y = [10,20,30]

def myFunction1(a):
    a = a + 10
    return a

def myFunction2(a):
    a.append(40)
    a.remove(20)

print(myFunction1(x))
print(x)
myFunction2(y)
print(y)
```

output:

```
30
20
[10, 30, 40]
```

11 Functions

Function parameters/arguments

When a function modifies a parameter, whether or not that modification is seen outside the scope of the function depends on the *mutability* of the argument object:

- If the object is *immutable* (string,integer), then the modification is limited to inside the function.
- If the object is *mutable*, then in-place modification of the object can be seen outside the scope of the function.

11 Functions

In-class practice

See the handout