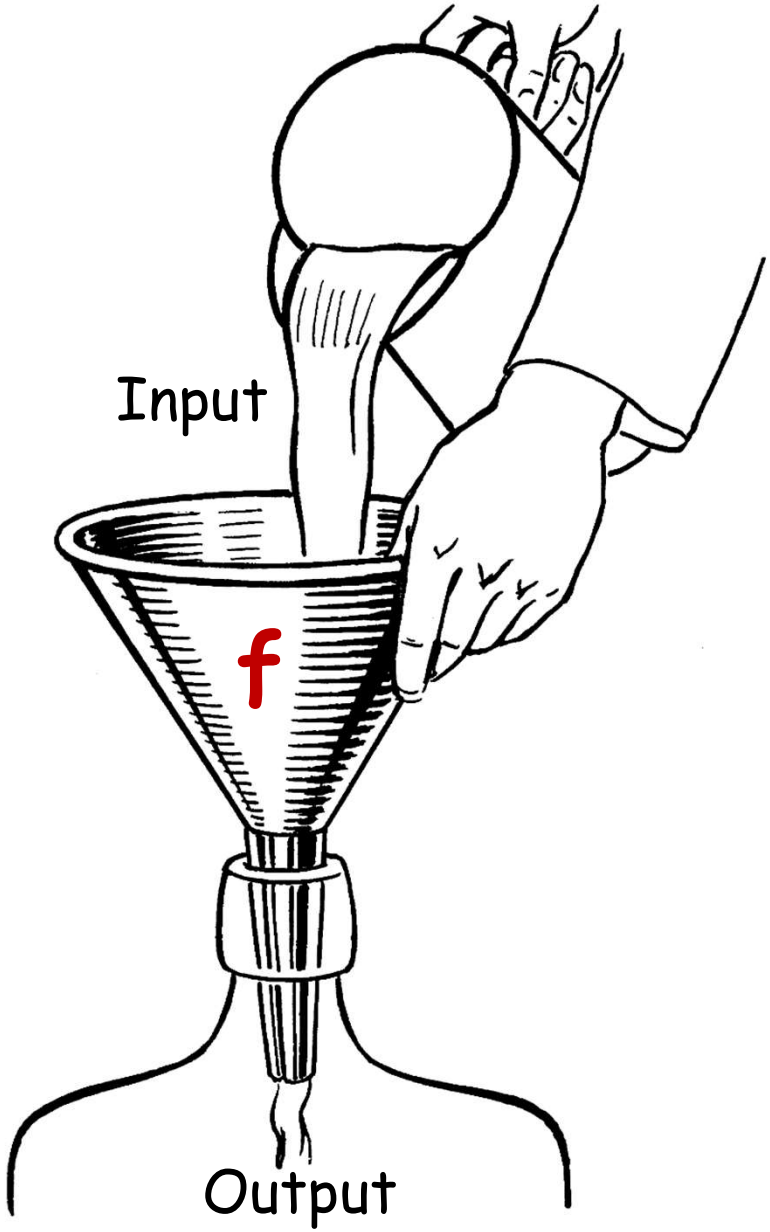
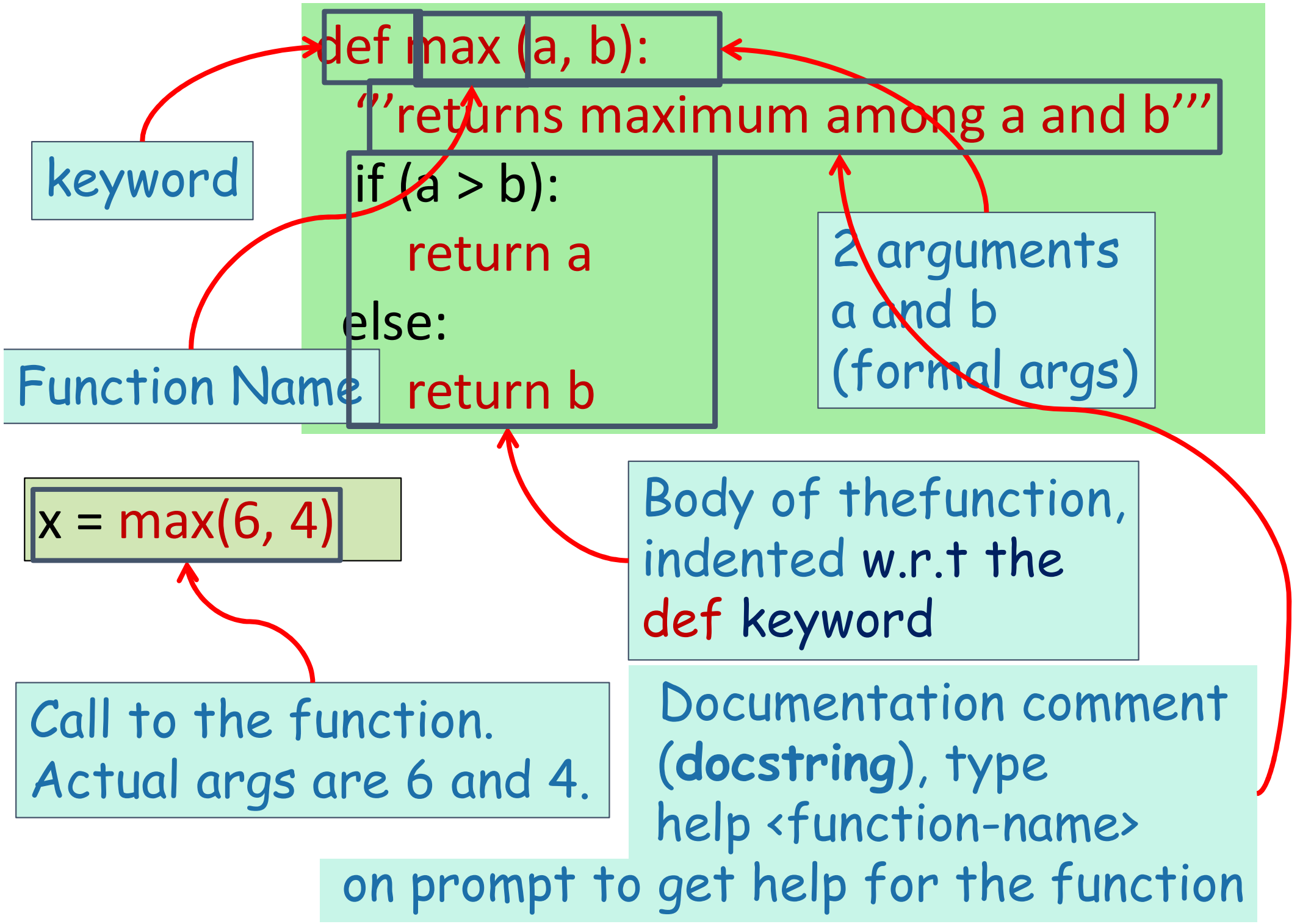
Programming using Python

f(unctions)

Parts of a function




def max (a, b):
'''returns maximum among a and b'''
if (a > b):
return a
else:
return b
keyword
Function Name
2 arguments
a and b
(formal args)
x = max(6, 4)
Body of thefunction,
indented w.r.t the
def keyword
Call to the function.
Actual args are 6 and 4.
Documentation comment
(docstring), type
help <function-name>
on prompt to get help for the function

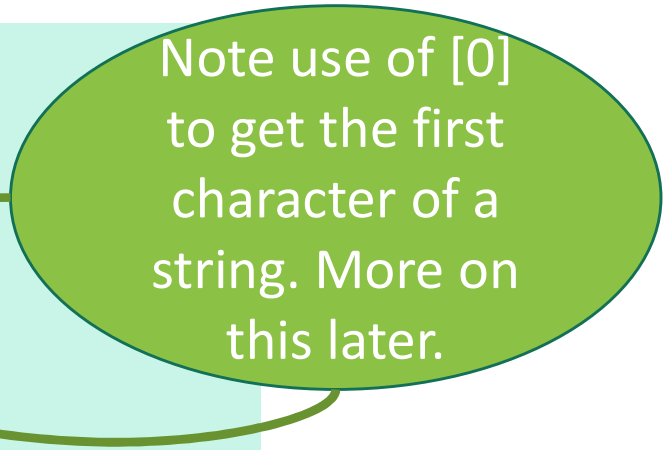
```
def max (a, b):
  '''returns maximum among a and b'''
  if (a > b):
     return a
  else:
     return b
```

```
In[3] : help(max)
Help on function max in module __main__:

max(a, b)
    returns maximum among a and b
```

Keyword Arguments

```
def printName(first, last, initials) :
    if initials:
        print (first[0] + '. ' + last[0] + '.')
    else:
        print (first, last)
```



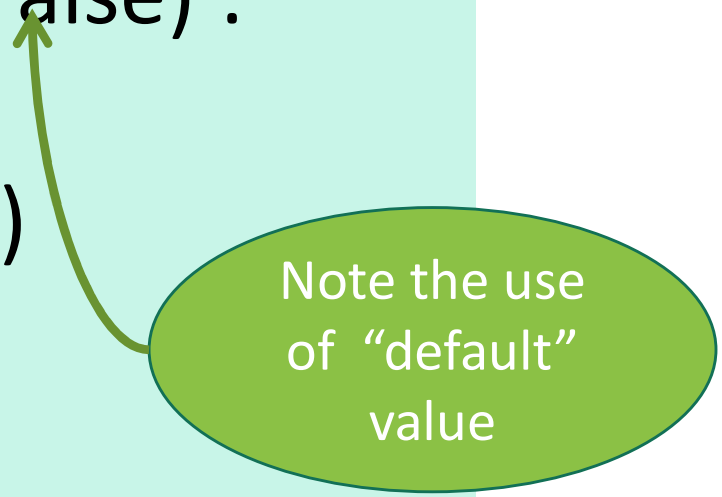
Call	Output
printName('Acads', 'Institute', False)	Acads Institute

Keyword Arguments

- Parameter passing where formal is bound to actual using formal's name
- Can mix keyword and non-keyword arguments
  - All non-keyword arguments precede keyword arguments in the call
  - Non-keyword arguments are matched by position (order is important)
  - Order of keyword arguments is not important

Default Values

```
def printName(first, last, initials=False) :
   if initials:
      print (first[0] + '. ' + last[0] + '.')
   else:
      print (first, last)
```



Call	Output
printName('Acads', 'Institute')	Acads Institute

Default Values

- Allows user to call a function with fewer arguments
- Useful when some argument has a fixed value for most of the calls
- All arguments with default values must be at the end of argument list
  - non-default argument can not follow default argument

```
# A simple Python function to check
# whether x is even or odd
def evenOdd( x ):
    if (x % 2 == 0):
        print "even"
    else:
        print "odd"

# Driver code
evenOdd(2)
evenOdd(3)
```

Pass by Reference or pass by value?

- One important thing to note is, **in Python every variable name is a reference**.

- When we pass a variable to a function, a new reference to the object is created.

- Parameter passing in Python is same as reference passing in Java.

```
# Here x is a new reference to same list lt
def myFun(x):
    x[0] = 20

# Driver Code (Note that lst is modified
# after function call.
lt = [10, 11, 12, 13, 14, 15]
myFun(lt)
print(lt)
```

Output:

```
[20, 11, 12, 13, 14, 15]
```

```
def myFun(x):
    x = [20, 30, 40]

# Driver code
lst = [10, 11, 12, 13, 14, 15]
myFun(lst);
print(lst)
```

Output: `[10, 11, 12, 13, 14, 15]`

When we pass a reference and change the received reference to something else, the connection between passed and received parameter is broken.

Globals

- Globals allow functions to communicate with each other indirectly
  - Without parameter passing/return value
- Convenient when two seemingly “far-apart” functions want to share data
  - No *direct* caller/callee relation
- If a function has to update a global, it must re-declare the global variable with global keyword.

Globals

```
PI = 3.14
def perimeter(r):
    return 2 * PI * r
def area(r):
    return PI * r * r
def update_pi():
    global PI
    PI = 3.14159
```

```
>>> print(area (100))
31400.0
>>> print(perimeter(10))
62.800000000000004
>>> update_pi()
>>> print(area(100))
31415.999999999996
>>> print(perimeter(10))
62.832
```

defines PI to be of float type with value 3.14. PI can be used across functions. Any change to PI in update_pi will be visible to all due to the use of global.

Variable number of arguments:

- We can have both normal and keyword variable number of arguments.

```
def myFun(*args):
    for arg in args:
        print (arg)

myFun(“FDP”, “on”,
“Python”, “Programming”)
```

```
Output:
FDP
on
Python
Programming
```

```
def myFun(**kwargs):
    for key, value in
kwargs.items():
        print ("%s == %s" %(key,
value))

# Driver code
myFun(first =‘santosh', mid
=‘kumar', last=‘verma’)
```

```
Output:
last == verma
mid == kumar
first == santosh
```

Anonymous functions/ Lambda Abstraction

- Anonymous function means that a function is without a name.
- **As we already know that def keyword is used to define the normal functions**
- While, the **lambda** keyword is used to create anonymous functions. A lambda function can take any number of arguments but can only have one expression.

```
cube = lambda x: x*x*x
print(cube(7))
```

```
Output: 343
```

```
x = lambda a : a + 10
print(x(5))
```

```
Output: 15
```

```
x = lambda a, b, c : a + b + c
print(x(5, 6, 2))
```

```
Output: 13
```

Note: It is as similar as inline functions in other programming language.

Recursion

- Recursion is nothing but calling a function directly or in-directly and must terminate on a base criteria.

```
def factI(n):
    '''Assumes n an int > 0
    returns n!'''
    result = 1
    while n>1:
        result = result * n
        n - = 1
    return result
```

```
def factR(n):
    '''Assumes n an int > 0
     returns n!'''
    if n ==1:
        return n
    else:
        return n*factR(n-1)
```