Arguments

- Arguments are passed by automatically assigning objects to local variable names.
- Assigning to argument names inside a function does not affect the caller.
- Changing a mutable object argument in a function may impact the caller.
- Immutable arguments are effectively passed "by value."
- Mutable arguments are effectively passed "by pointer."

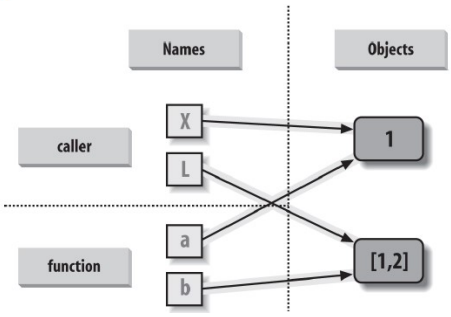
```
>>> def f(a):                  # a is assigned to (references) the passed object
        a = 99                 # Changes local variable a only

>>> b = 88
>>> f(b)                       # a and b both reference same 88 initially
>>> print(b)                   # b is not changed
88
```


Arguments

```
>>> def changer(a, b):         # Arguments assigned references to objects
        a = 2                  # Changes local name's value only
        b[0] = 'spam'          # Changes shared object in place

>>> X = 1
>>> L = [1, 2]                 # Caller:
>>> changer(X, L)              # Pass immutable and mutable objects
>>> X, L                       # X is unchanged, L is different!
(1, ['spam', 2])
```



Simulating Output Parameters and Multiple Results

- Python doesn't support what some languages label "call by reference" argument passing, we can usually simulate it by returning tuples and assigning the results back to the original argument names in the caller

```
>>> def multiple(x, y):
        x = 2                   # Changes local names only
        y = [3, 4]
        return x, y             # Return multiple new values in a tuple

>>> X = 1
>>> L = [1, 2]
>>> X, L = multiple(X, L)       # Assign results to caller's names
>>> X, L
(2, [3, 4])
```

3

Special Argument-Matching Modes

- *Positionals: matched from left to right*
- *Keywords: matched by argument name*
- *Defaults: specify values for optional arguments that aren't passed*
- *Varargs collecting: collect arbitrarily many positional or keyword arguments*
- *Varargs unpacking: pass arbitrarily many positional or keyword arguments*
- *Keyword-only arguments: arguments that must be passed by name*

4

Argument Matching Syntax

Table 18-1. Function argument-matching forms

Syntax	Location	Interpretation
`func(value)`	Caller	Normal argument: matched by position
`func(name=value)`	Caller	Keyword argument: matched by name
`func(*iterable)`	Caller	Pass all objects in *`iterable`* as individual positional arguments
`func(**dict)`	Caller	Pass all key/value pairs in *`dict`* as individual keyword arguments
`def func(name)`	Function	Normal argument: matches any passed value by position or name
`def func(name=value)`	Function	Default argument value, if not passed in the call
`def func(*name)`	Function	Matches and collects remaining positional arguments in a tuple
`def func(**name)`	Function	Matches and collects remaining keyword arguments in a dictionary
`def func(*other, name)`	Function	Arguments that must be passed by keyword only in calls (3.X)
`def func(*, name=value)`	Function	Arguments that must be passed by keyword only in calls (3.X)

5

Argument Matching Syntax

- In both the call and header, the **args form must appear last if present
- The steps that Python internally carries out to match arguments before assignment can roughly be described as follows
  1. Assign nonkeyword arguments by position.
  2. Assign keyword arguments by matching names.
  3. Assign extra nonkeyword arguments to *name tuple.
  4. Assign extra keyword arguments to **name dictionary.
  5. Assign default values to unassigned arguments in header.

6

Keyword and Default Examples

```
>>> def f(a, b, c): print(a, b, c)

>>> f(1, 2, 3)
1 2 3
>>> f(c=3, b=2, a=1)
1 2 3


>>> def f(a, b=2, c=3): print(a, b, c)          # a required, b and c optional
>>> f(1)                                        # Use defaults
1 2 3
>>> f(a=1)
1 2 3

>>> f(1, 4)                                     # Override defaults
1 4 3
>>> f(1, 4, 5)
1 4 5
```


Combining keywords and defaults

- Notice again that when keyword arguments are used in the call, the order in which the arguments are listed doesn't matter; Python matches by name, not by position

```
def func(spam, eggs, toast=0, ham=0):    # First 2 required
    print((spam, eggs, toast, ham))

func(1, 2)                               # Output: (1, 2, 0, 0)
func(1, ham=1, eggs=0)                   # Output: (1, 0, 0, 1)
func(spam=1, eggs=0)                     # Output: (1, 0, 0, 0)
func(toast=1, eggs=2, spam=3)            # Output: (3, 2, 1, 0)
func(1, 2, 3, 4)                         # Output: (1, 2, 3, 4)
```

Arbitrary Arguments Examples

- Python collects all the positional arguments into a new *tuple* and assigns the variable args to that tuple

```
>>> def f(*args): print(args)
>>> f()
()
>>> f(1)
(1,)
>>> f(1, 2, 3, 4)
(1, 2, 3, 4)
```

- The ** feature is similar, but it only works for *keyword* arguments—it collects them into a new *dictionary*, which can then be processed with normal dictionary tools.

```
>>> def f(**args): print(args)

>>> f()
{}
>>> f(a=1, b=2)
{'a': 1, 'b': 2}
```

9

Arbitrary Arguments Examples

```
>>> def f(a, *pargs, **kargs): print(a, pargs, kargs)

>>> f(1, 2, 3, x=1, y=2)
1 (2, 3) {'y': 2, 'x': 1}
```

- Calls: Unpacking arguments

```
>>> def func(a, b, c, d): print(a, b, c, d)

>>> args = (1, 2)
>>> args += (3, 4)
>>> func(*args)                          # Same as func(1, 2, 3, 4)
1 2 3 4


>>> args = {'a': 1, 'b': 2, 'c': 3}
>>> args['d'] = 4
>>> func(**args)                         # Same as func(a=1, b=2, c=3, d=4)
1 2 3 4
```

10