FUNCTIONS:

PARAMETER PASSING

Overview:

- describe role of mutability in parameter binding/passing

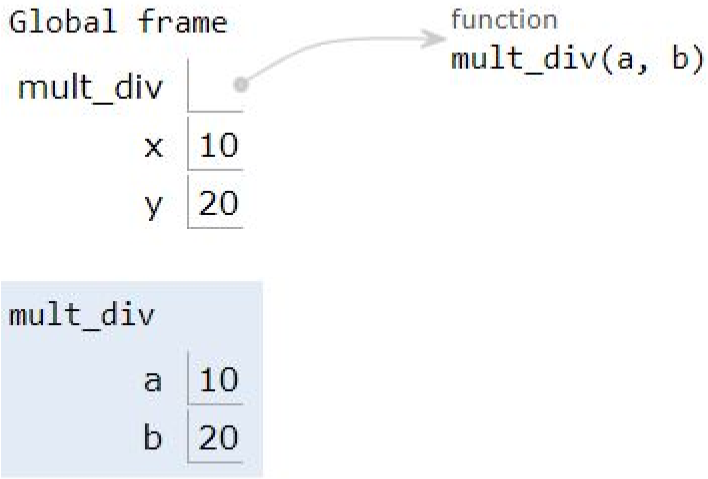
Parameter Passing

- parameters are input values passed to functions
- several method available
- passing parameters in Python is different from other languages

Parameters by Position

```
def mult_div(a, b):
    """ multiply & divide two numbers """
    result = a * b, a / b
    return result

x = 10; y = 20
result = mult_div(x, y)
```

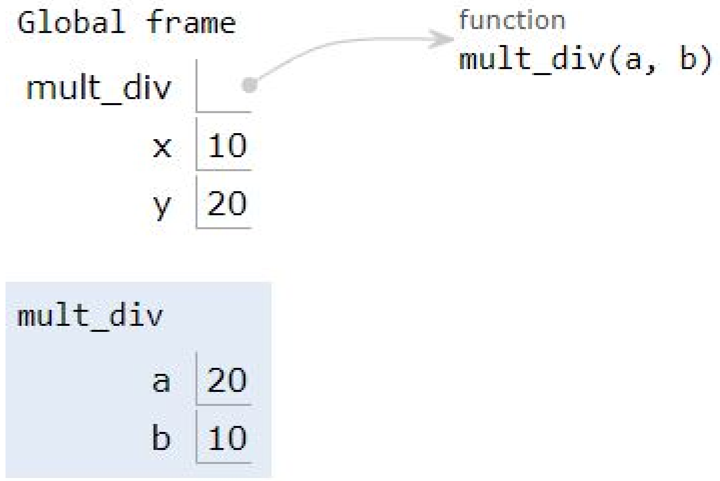


- parameters bound by position (default)

Parameters by Keyword

```
def mult_div(a, b):
    """ multiply & divide two numbers """
    result = a * b, a / b
    return result

x = 10; y = 20
result = mult_div(b = x, a = y)
```



Global frame

mult_div

x 10

y 20

function

mult_div(a, b)

mult_div

a 20

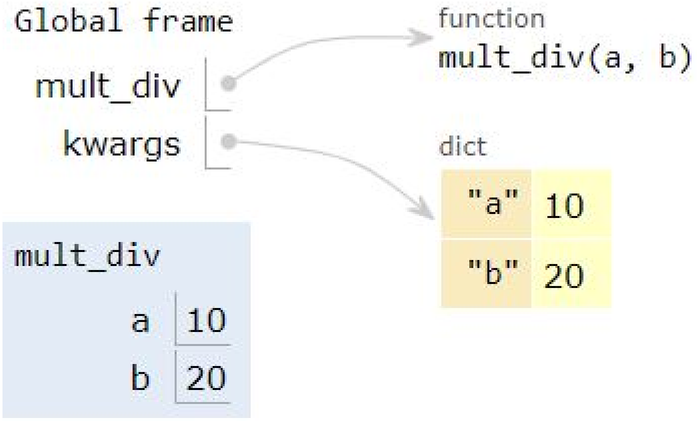
b 10

- parameters bound by keywords

Parameters by Dictionary

```
def mult_div(a, b):
    """ multiply & divide two numbers """
    result = a * b, a / b
    return result

kwargs = {'a': 10, 'b': 20}
x, y = mult_div(**kwargs)
```

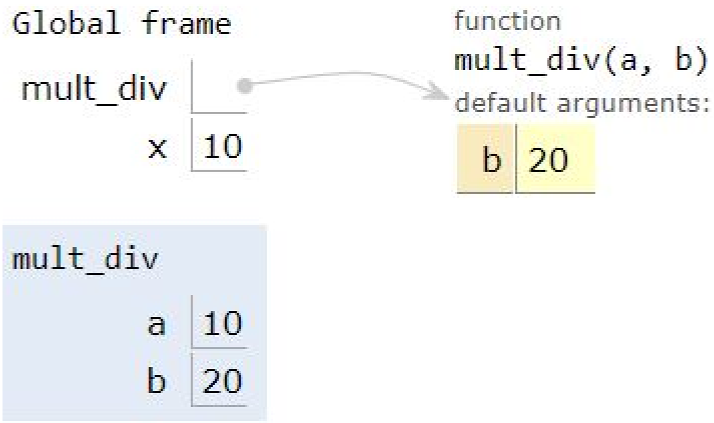


- syntax: function(**dict)

Optional Parameters

```
def mult_div(a, b = 20):
    """ multiply & divide two numbers """
    result = a * b, a / b
    return result

x = 10
result = mult_div(x)
```



- bound upon creation

Function Signatures

```
def mult_div(a, b = 20):
    """ multiply & divide two numbers """
    result = a * b, a / b
    return result

x = 10; y = 20
kwargs = {’a’: 10, ’b’: 20}

results = mult_div(x)
results = mult_div(x, y)
results = mult_div(b = y, a = x)
results = mult_div(**kwargs)
```

- same name $\mapsto$ same function

Exercise(s):

- show three ways to pass parameters to function $f(a, d, n)$ that returns a list of first n values in arithmetic progression $A(a, d)$

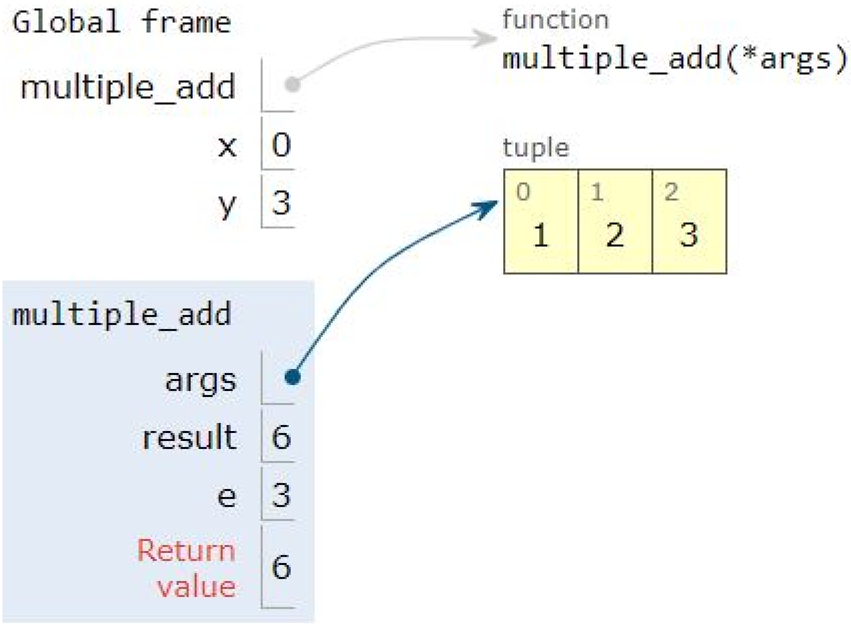
Exercise(s):

- show three ways to pass parameters to function $g(b, q, n)$ that returns a list of first n values in geometric progression $G(b, q)$

Variable Positional Args

```
def multiple_add(*args):
    result = 0
    for e in args:
        result = result + e
    return result

x = multiple_add();
y = multiple_add(1, 2)
z = multiple_add(1, 2, 3)
```



Parameter Passing

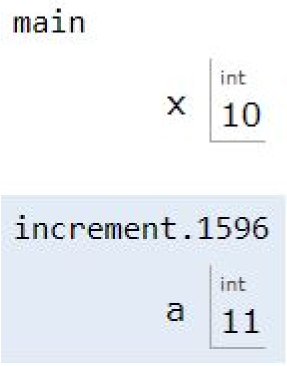
- *call by value*:
  1. parameters aee copied
  2. changes to parameters are **not visible**
- *call by reference*:
  1. pointers to parameters are passed
  2. changes to parameters are **visible**
- Python: *call by assignment*

Call by *value* in C/C++

```
int main() {

void increment(int a) {
     a = a + 1;       }

int x = 10;
inc_value(x);
printf("%d", x);     }
```



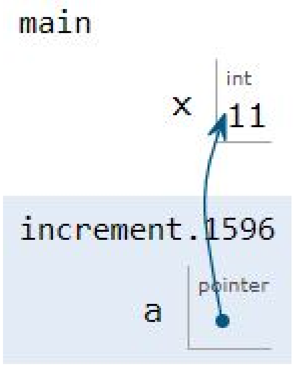
- changes are not visible

Call by *reference* in C/C++

```
int main() {

void increment(int* a) {
     *a = *a + 1;     }

int x = 10;
increment(&x);
printf("%d", x);     }
```

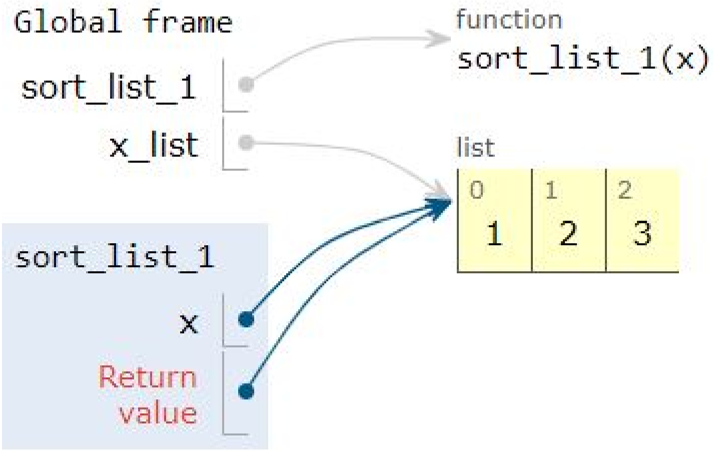


- changes are visible

Mutable Parameter

```
def sort_list_1(x):
    x.sort()
        return x

x_list = [3, 1, 2]
x_list = sort_list_1(x_list)
```

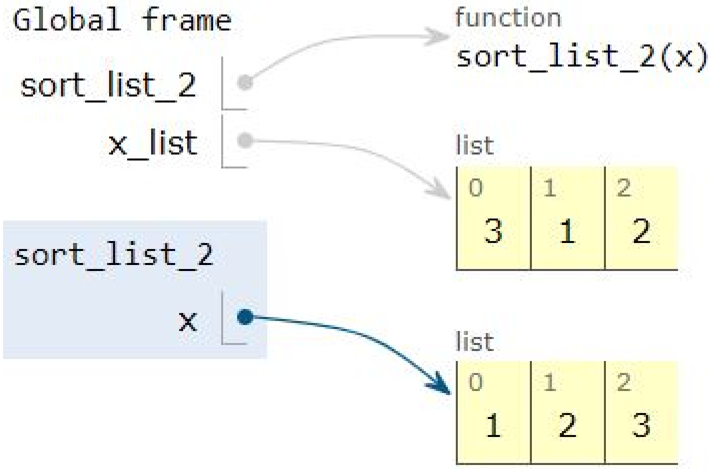


- ”in-place” ↦ by *reference*

Mutable Parameter

```
def sort_list_2(x):
    x = sorted(x)
        return x

x_list = [3,1,2]
x_list = sort_list_2(x_list)
```

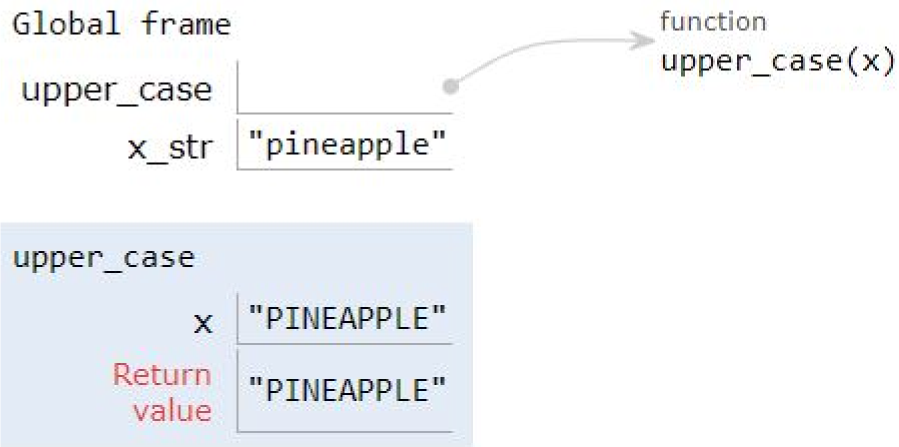


- new object ↦ by *value*

Immutable Parameter

```
def upper_case(x):
    x = x.upper()
    return x

x_str = 'pineapple'
y_str = upper_case(x_str)
```



Global frame

upper_case

x_str "pineapple"

function
upper_case(x)

upper_case

x "PINEAPPLE"

Return value "PINEAPPLE"

- cannot modify ”in-place”
- new object ↦ by *value*

Exercise(s):

- write a function *check_arith* that takes a list of values and determines if this list is an arithmetic progression.

Exercise(s):

- write a function *arith_1*() that takes *x_list* of values. If it is an arithmetic progression, it adds next value to *x_list*.

```
x_list = [1,3,5,7]
# input is OK, add 9
x_list = [1,3,5,7,9]

x_list = [1,3,5,17]
# input is not OK
x_list = [1,3,5,17]
```

Exercise(s):

- write a function *arith_2*() that takes *x_list* of values. If it is an arithmetic progression, it returns a *y_list* from *x_list* and the next value.

```
x_list = [1,3,5,7]
# input is OK
y_list = [1,3,5,7,9]

x_list = [1,3,5,17]
# input is not OK
y_list = [1,3,5,17]
```

Exercise(s):

- write a function *check_geom* that takes a list of values and determines if this list is a geometric progression.

Exercise(s):

- write a function *geom_1*() that takes *z_list* of values. If it is a geometric progression, it adds next value to *z_list*.

```
z_list = [1,3,9,27]
# input is OK, add 81
z_list = [1,3,9,27,81]

z_list = [1,3,30,50]
# input is not OK
z_list = [1,3,30,50]
```

Exercise(s):

- write a function *geom_2*() that takes *z_list* of values. If it is a geometric progression, it returns a *w_list* from *z_list* and the next value.

```
z_list = [1,3,9,27]
# input is OK, add 81
w_list = [1,3,9,27,81]

z_list = [1,3,30,50]
# input is not OK
w_list = [1,3,30,50]
```