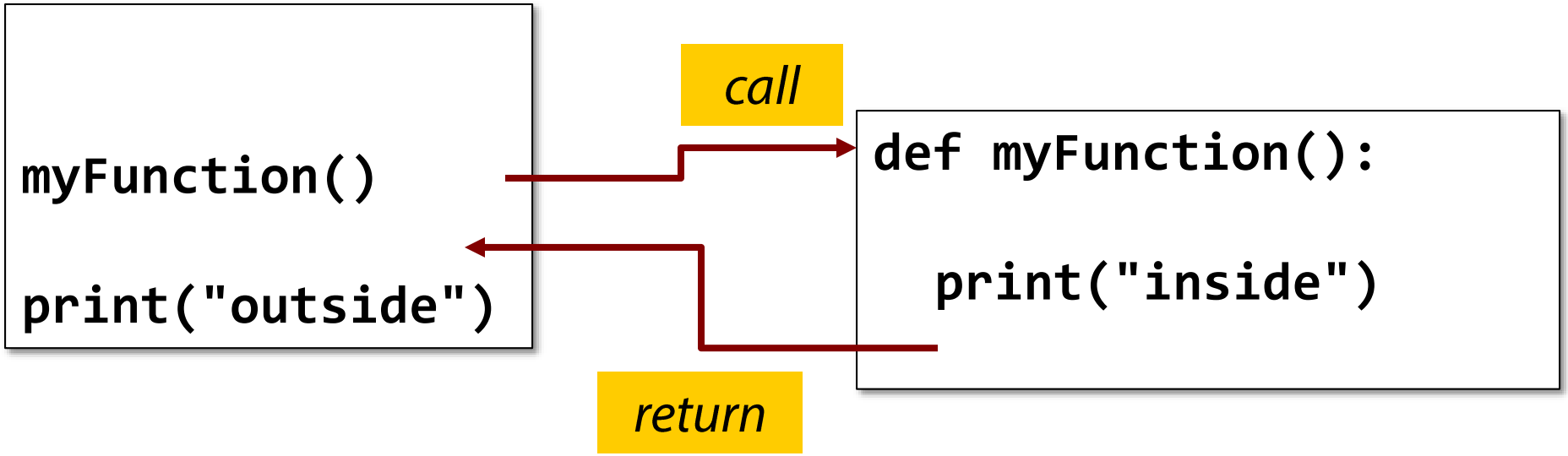
ITP 115 – Programming in Python

Functions

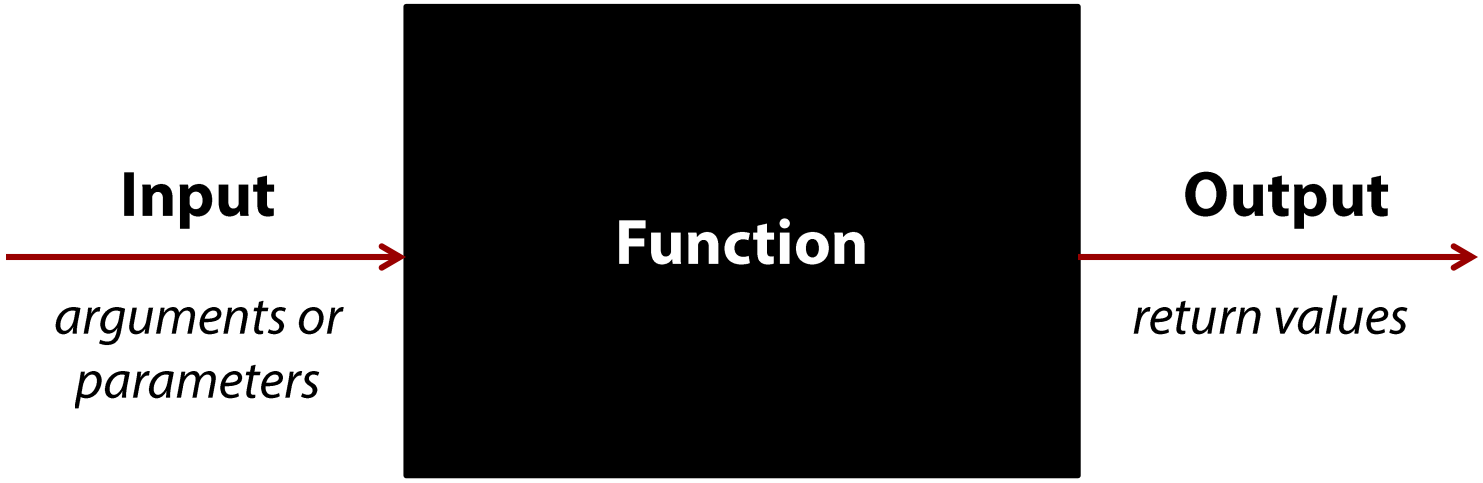
Summarizing functions

- Functions tell Python, go off and perform a task and then return control to your program
- Allow you to break up your code into manageable, bite-sized chunks
- Programs with functions can be easier to create and work with
- Check last week's slides for the exact syntax

Flow of Control with Functions



Functions with Input and Output



We left off here…

```
def main():
    number = int(input("Enter a number: "))
    result = square(number)
    print("The square of", number, "is", result)

def square(x):
    return x * x

main()
```

Comments

- Every function needs a function header

- A comment with 5 lines…
  - Function name
  - Function input
  - Function output
  - Function side effect (i.e. what is printed)
  - Function description

Review: Namespaces

weather.py

```
def func1():
  airQuality = 1

def func2():
  rain = 3
```

- `airQuality` is a **local** variable
  - Can be accessed ONLY from `func1()`
- `rain` is a **local** variable
  - Can be accessed ONLY from `func2()`

Review: Namespaces

weather.py

```
def func1():
  airQuality = 1
  print(rain)

def func2():
   rain = 3
   print(airQuality)

def main():
  func1()
  func2()
```

- Calling/running either of these functions will result in errors
- `func1()`
  - `rain` was not defined inside this function
- `func2()`
  - `airQuality` was not defined inside this function

Aside: Constants

- A **constant** is a variable that can not change

- Constants can be useful to ensure some important data never changes
  - Ex: Sales tax rate or speed of light

- Style: constants are **ALL_CAPS_WITH_UNDERSCORES**
  - Ex. `SALES_TAX_RATE` or `SPEED_OF_LIGHT`

Global Constants

- Global constants are **constants** created in the global namespace
  - This means on the far left of the file
- Global constants can be access from everywhere in your program (e.g. inside functions)
- Global constants can not change their values once they are assigned

Constants and Namespaces

weather.py

```
AVG_TEMP = 87

def func1():
  airQuality = 1

def func2():
  rain = 3
```

- `AVG_TEMP` is a **global** constant
  - Can be accessed from within any function
- `airQuality` is a **local** variable
  - Can be accessed ONLY from `func1()`
- `rain` is a **local** variable
  - Can be accessed ONLY from `func2()`

Constants and Namespaces

weather.py

```
AVG_TEMP = 87

def func1():
  airQuality = 1
  print(AVG_TEMP)

def func2():
   rain = 3
   print(AVG_TEMP)

def main():
   func1()
   func2()
```

- **func1()** has access to…
  - **airQuality** which is **local**
  - **AVG_TEMP** which is **global**
- **func2()** has access to…
  - **rain** which is **local**
  - **AVG_TEMP** which is **global**

Discussion of Variable References

- A variable does not *actually* store the data you assign to it
- A variable instead stores a **reference** to the computer's memory where the data is stored
- The **reference** exists in the current namespace

When you pass variable to a function

- Immutable variables are **not** affected by any changes made within the function

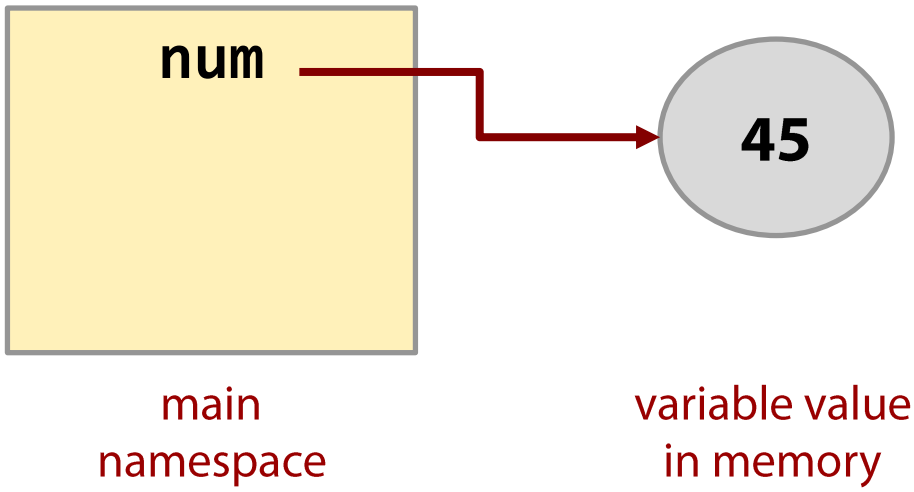
- Mutable variables **may** be affected by changes in the function
  - Modifying operations *([ ], **append**, **del**, etc.)* **do** affect the original variable
  - Assignment (=) will **not** affect the original variables

Discussion of Variable References

- When you see

  ```
  num = 45
  ```

  you should imagine…



Immutable Objects

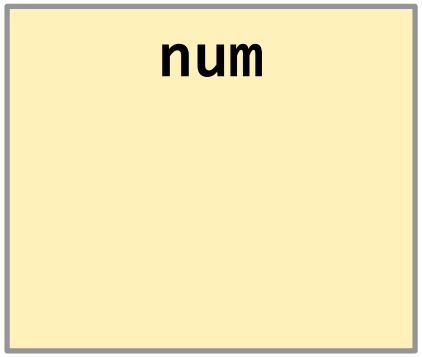
- Immutable objects can't be changed
- When you re-assign an immutable object, it creates a new one

- Immutable objects are
  - *strings*
  - *ints*
  - *floats*
  - *tuples*

Immutable Objects

- What happens in the following?

```
num = 45
num = 81
```



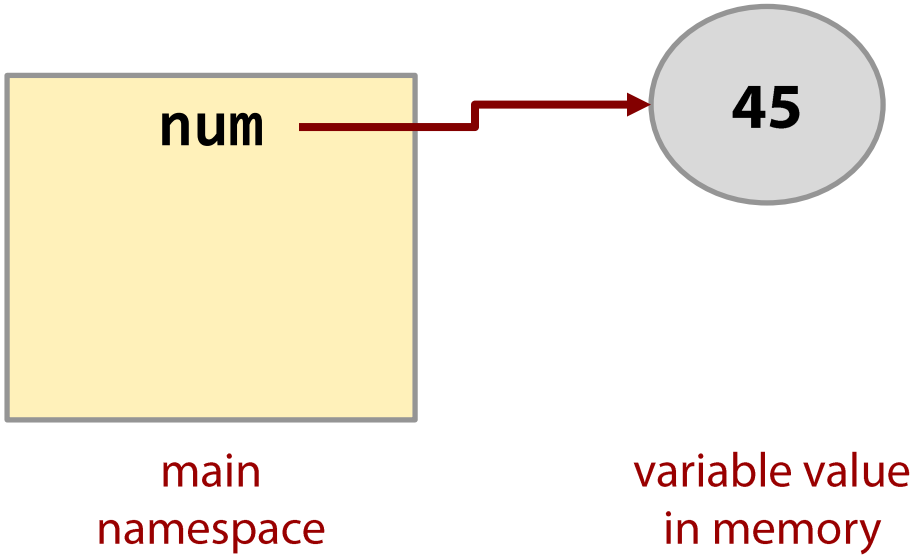
main namespace

variable value in memory

Immutable Objects

- What happens in the following?

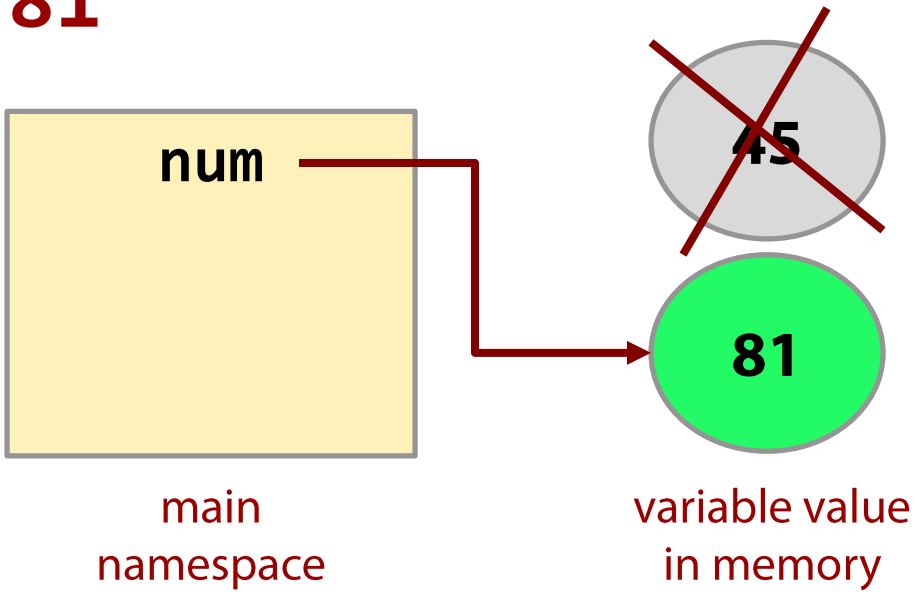
```
num = 45
num = 81
```



Immutable Objects

- What happens in the following?

```
num = 45
num = 81
```

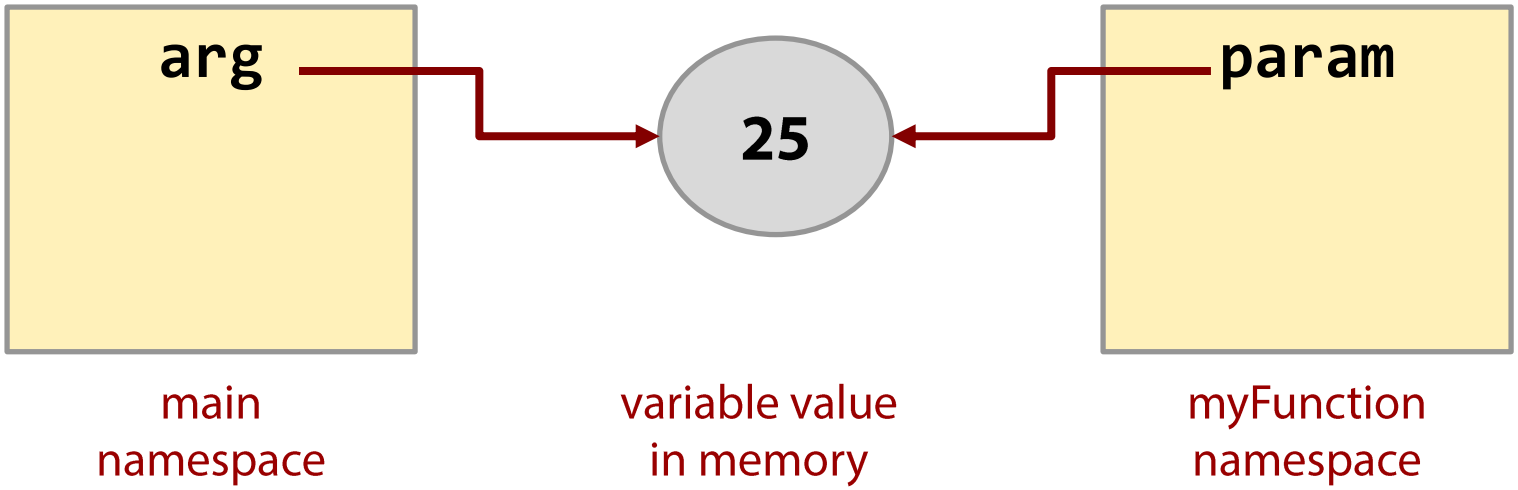


Passing Immutable Objects

- What happens we pass immutable objects to a function?
    - A *copy* of the reference is made
        - But both the original and the copy point to same data in memory
    - Since the object is immutable, any changes to its value *inside* the function **will not affect** the original variable

```
arg = 25
myFunction(arg)
print(arg)
```

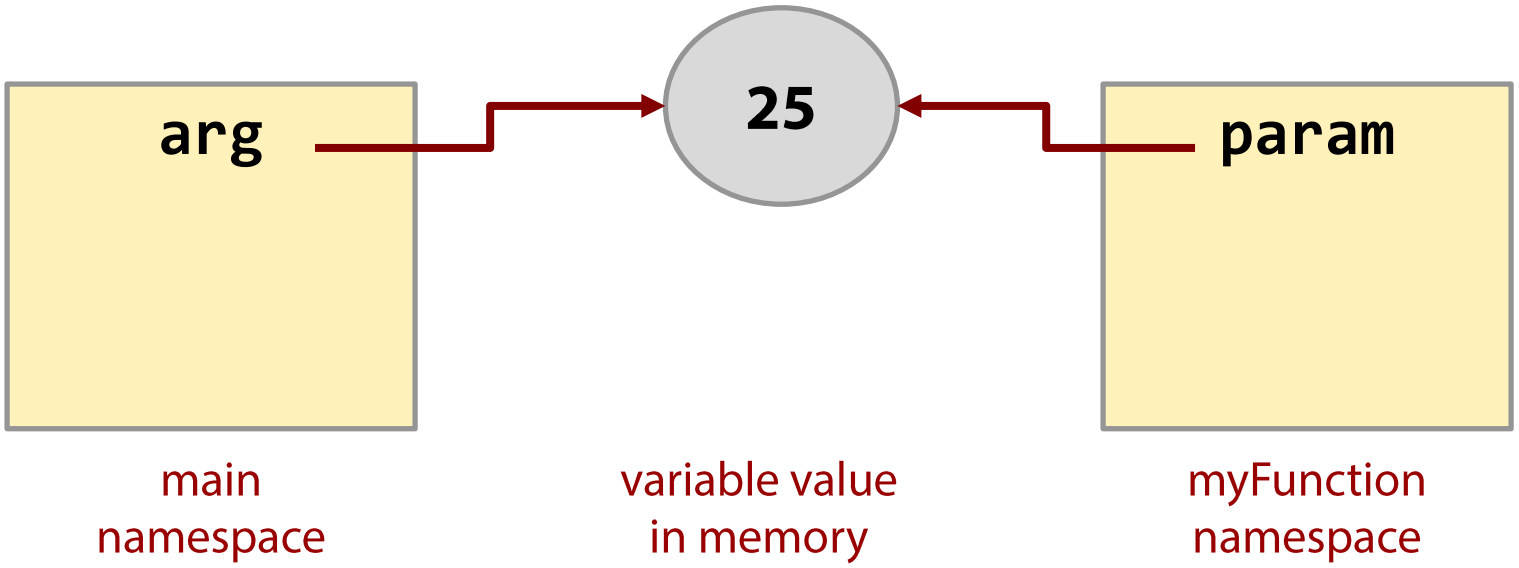
```
def myFunction(param):
    print(param)
```



What if an immutable object changes in a function?

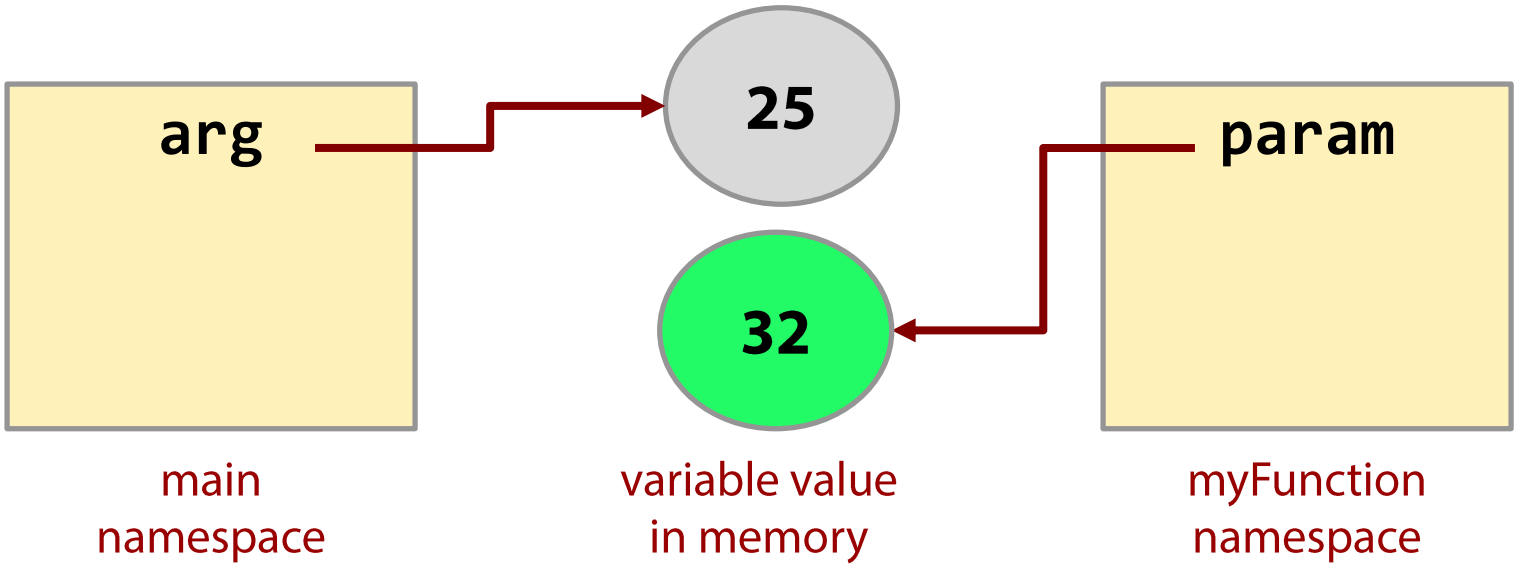
```
arg = 25
myFunction(arg)
print(arg)
```

```
def myFunction(param):
    param = 32
    print(param)
```



```
arg = 25
myFunction(arg)
print(arg)
```

```
def myFunction(param):
    param = 32
    print(param)
```



Variable References

Mutable Objects

- Mutable objects **can** be changed
  - **`append, del, remove, []`**

- However, if you re-assign a mutable object, it **still creates a new one**

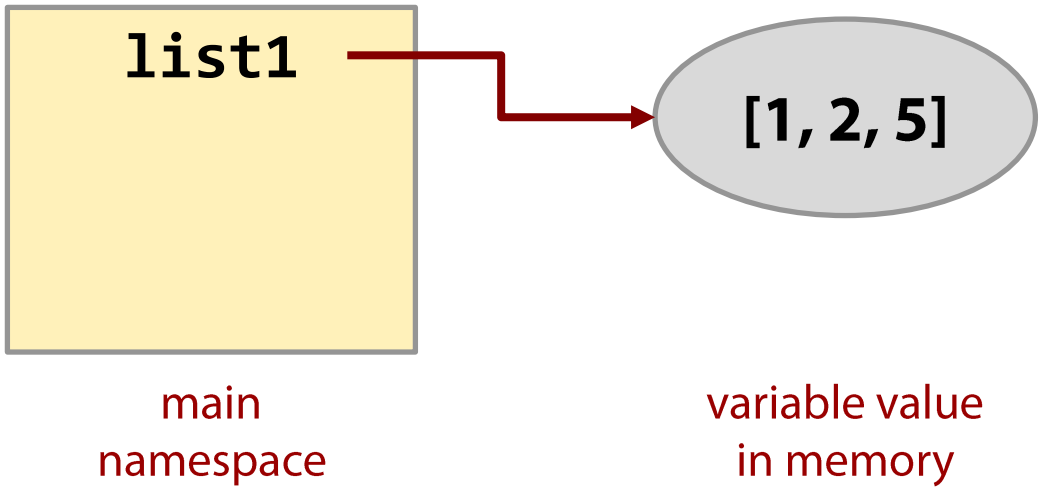
- Mutable objects
  - *lists*
  - *dictionaries (later)*

Mutable Objects

- When you see

  ```
  list1 = [1, 2, 5]
  ```

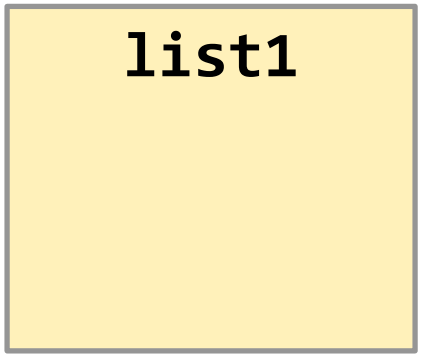
  you should imagine…



Mutable Objects

- What happens in the following?

```
list1 = [1, 2, 5]
list1[0] = 9
```

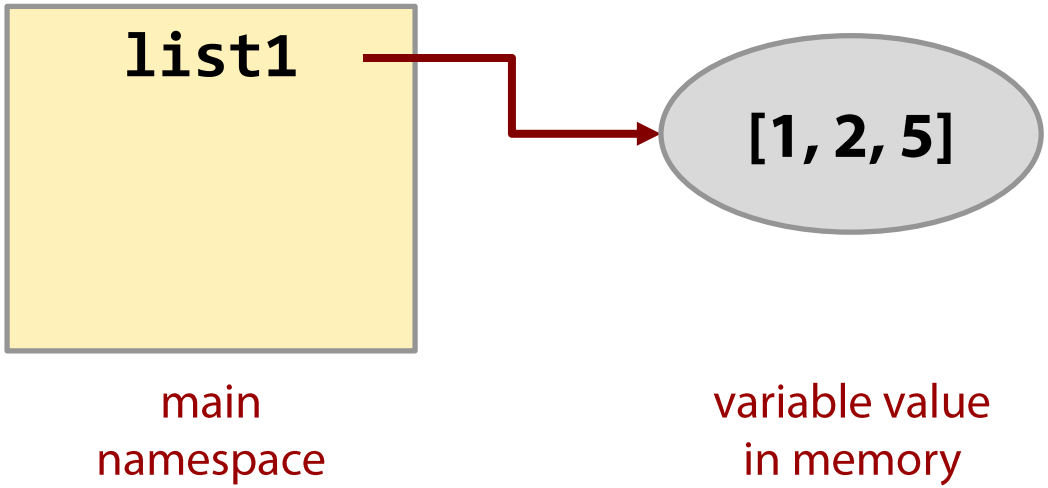


main
namespace

Mutable Objects

- What happens in the following?

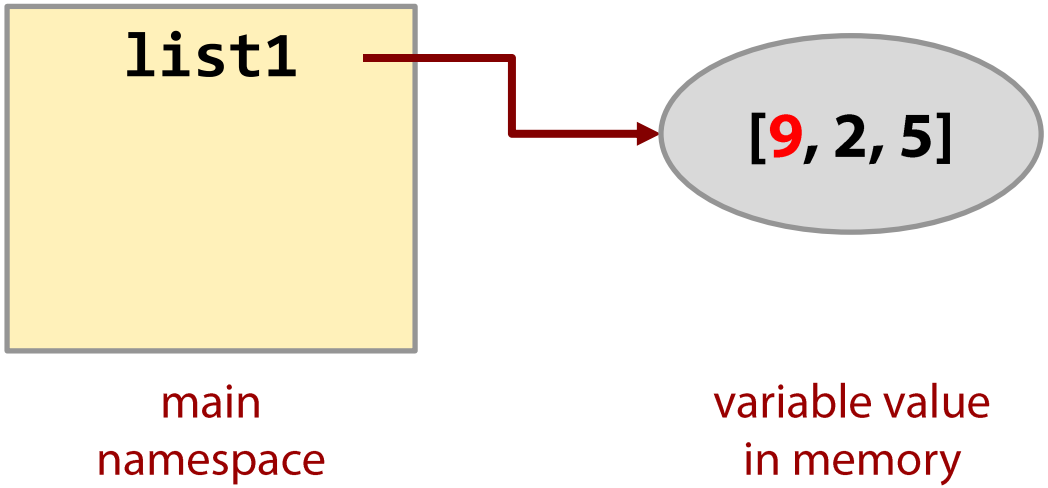
```
list1 = [1, 2, 5]
list1[0] = 9
```



Mutable Objects

- What happens in the following?

```
list1 = [1, 2, 5]
list1[0] = 9
```

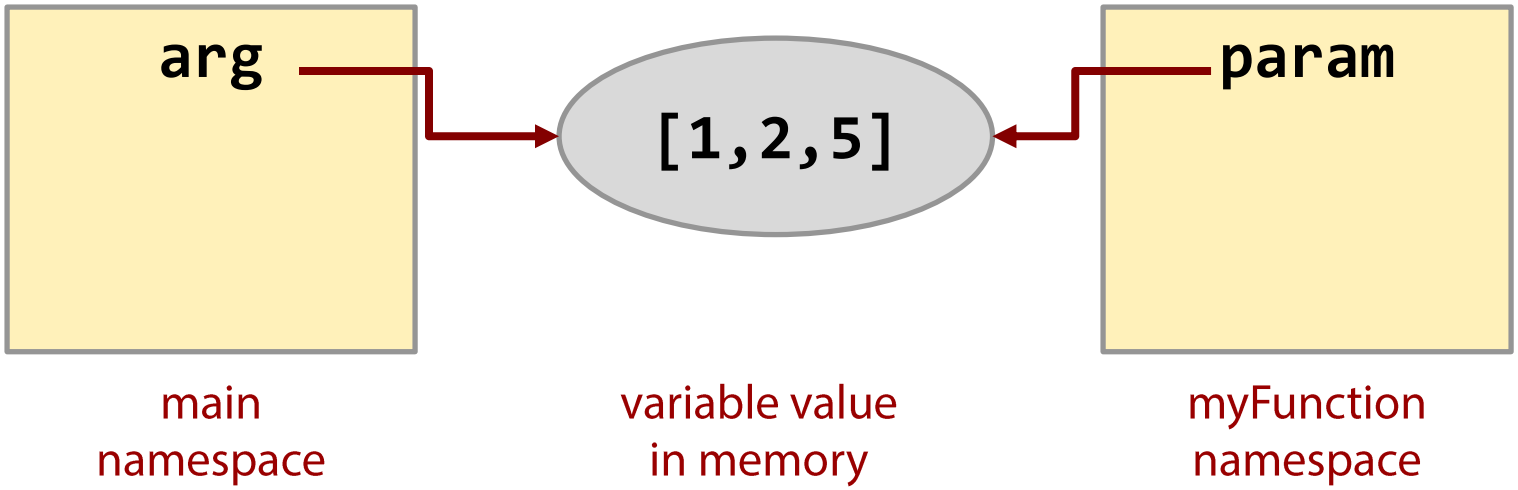


Passing Mutable Objects

- What happens we pass mutable objects to a function?
    - A *copy* of the reference is made
        - But the original and the copy point to same data in memory
    - However, since object is mutable, any changes to its value *inside* the function **will affect** the original variable

```
arg = [1,2,5]
myFunction(arg)
print(arg)
```

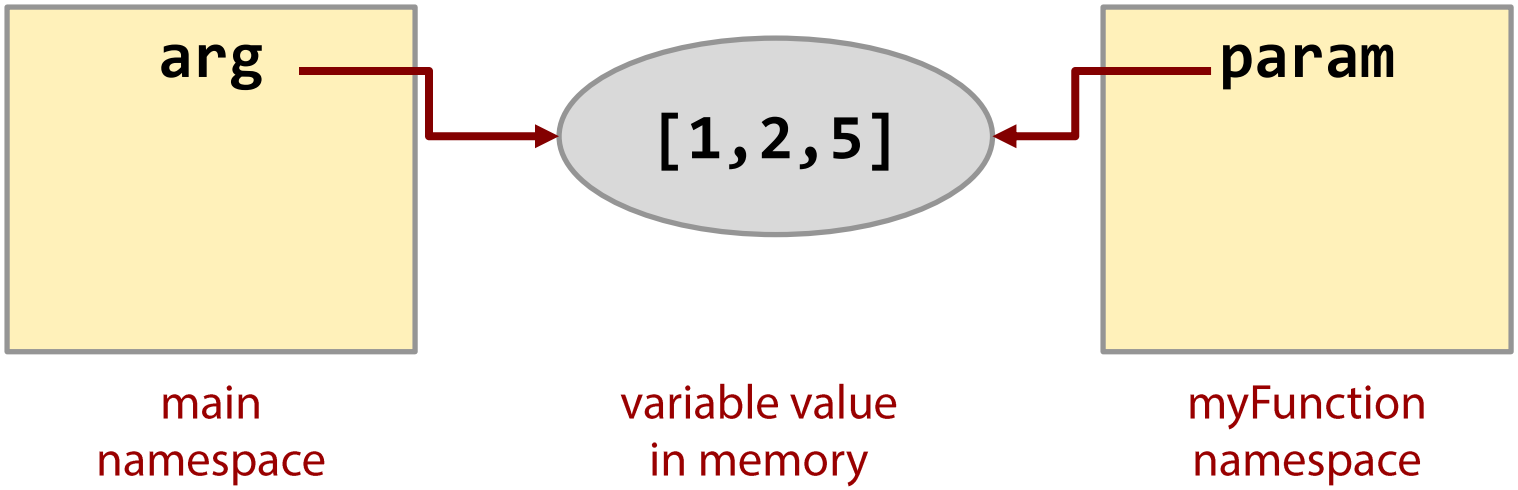
```
def myFunction(param):
    print(param)
```



What if a mutable object changes in a function?

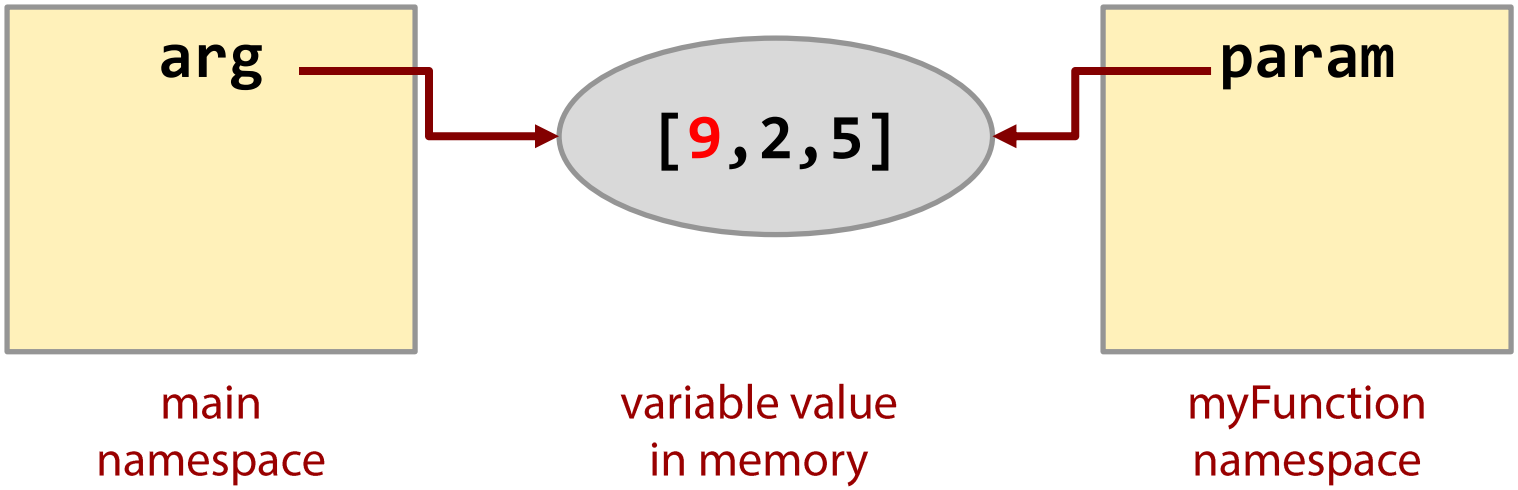
```
arg = [1,2,5]
myFunction(arg)
print(arg)
```

```
def myFunction(param):
    param[0] = 9
    print(param)
```



```
arg = [1,2,5]
myFunction(arg)
print(arg)
```

```
def myFunction(param):
    param[0] = 9
    print(param)
```



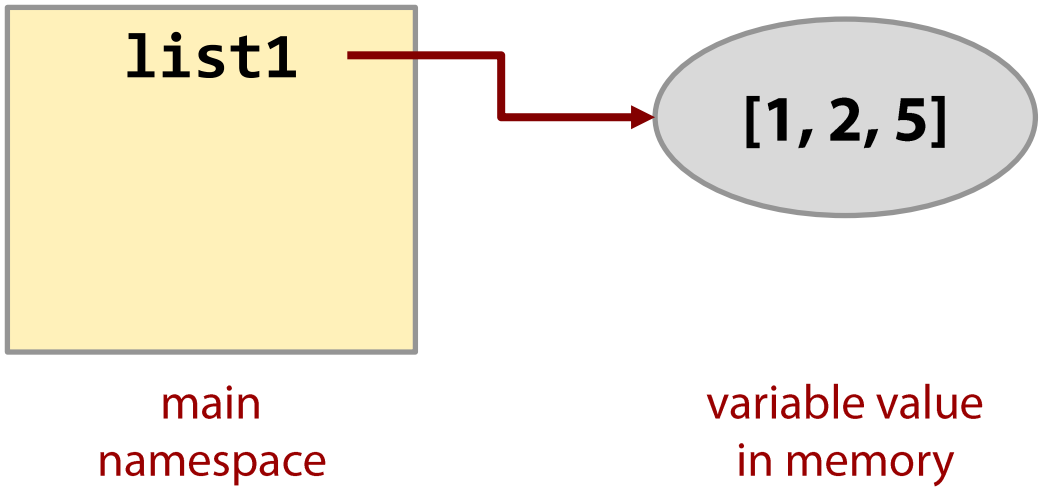
But wait!

Mutable Objects

- When you see

  ```
  list1 = [1, 2, 5]
  ```

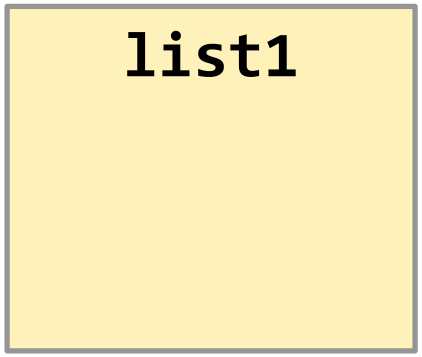
  you should imagine…



Mutable Objects

- What happens in the following?

```
list1 = [1, 2, 5]
list1 = [4, 3]
```

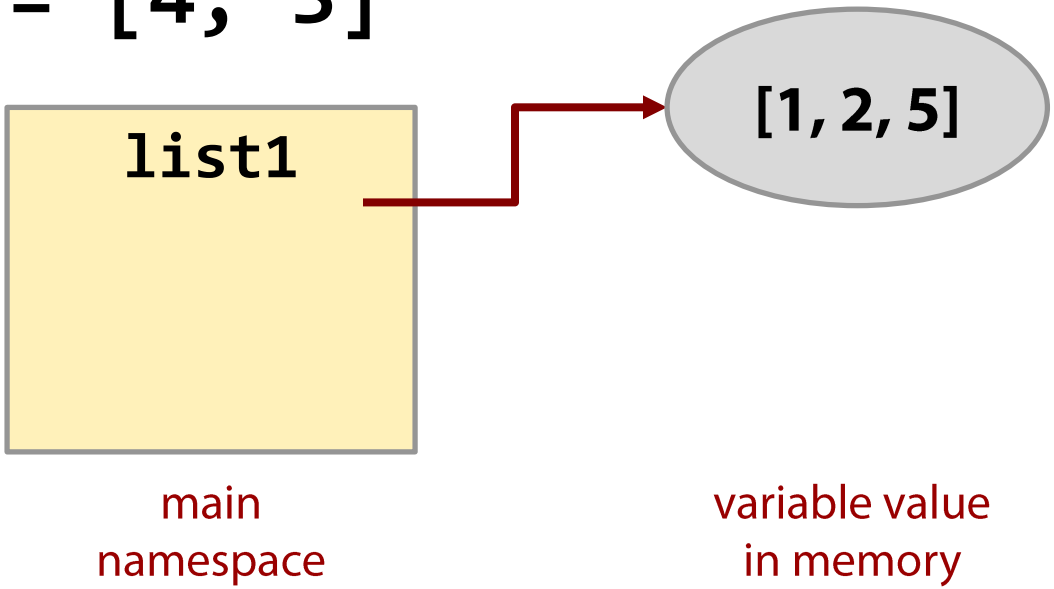


main
namespace

Mutable Objects

- What happens in the following?

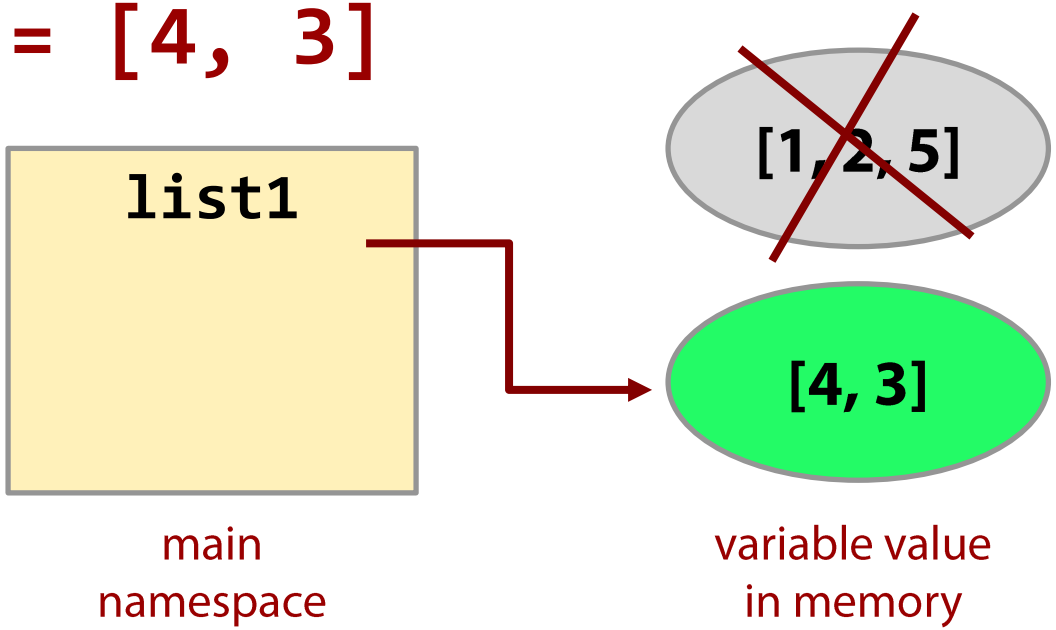
```
list1 = [1, 2, 5]
list1 = [4, 3]
```



Mutable Objects

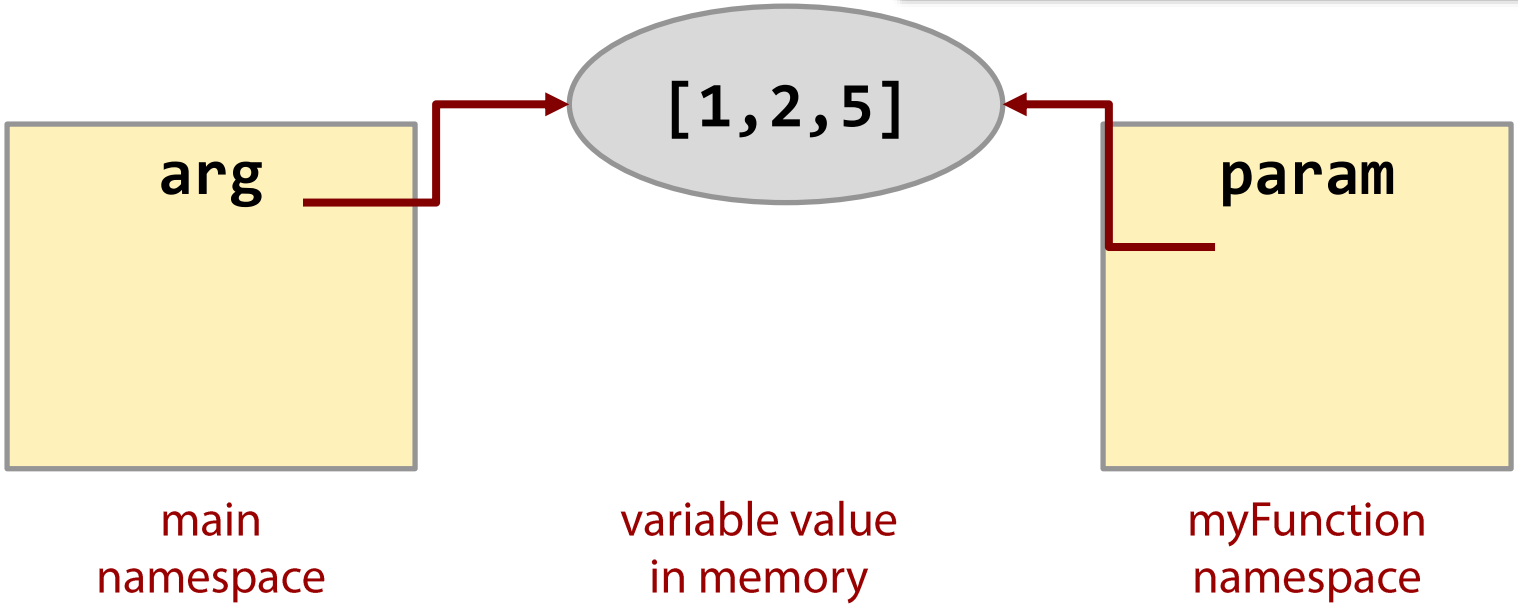
- What happens in the following?

```
list1 = [1, 2, 5]
list1 = [4, 3]
```



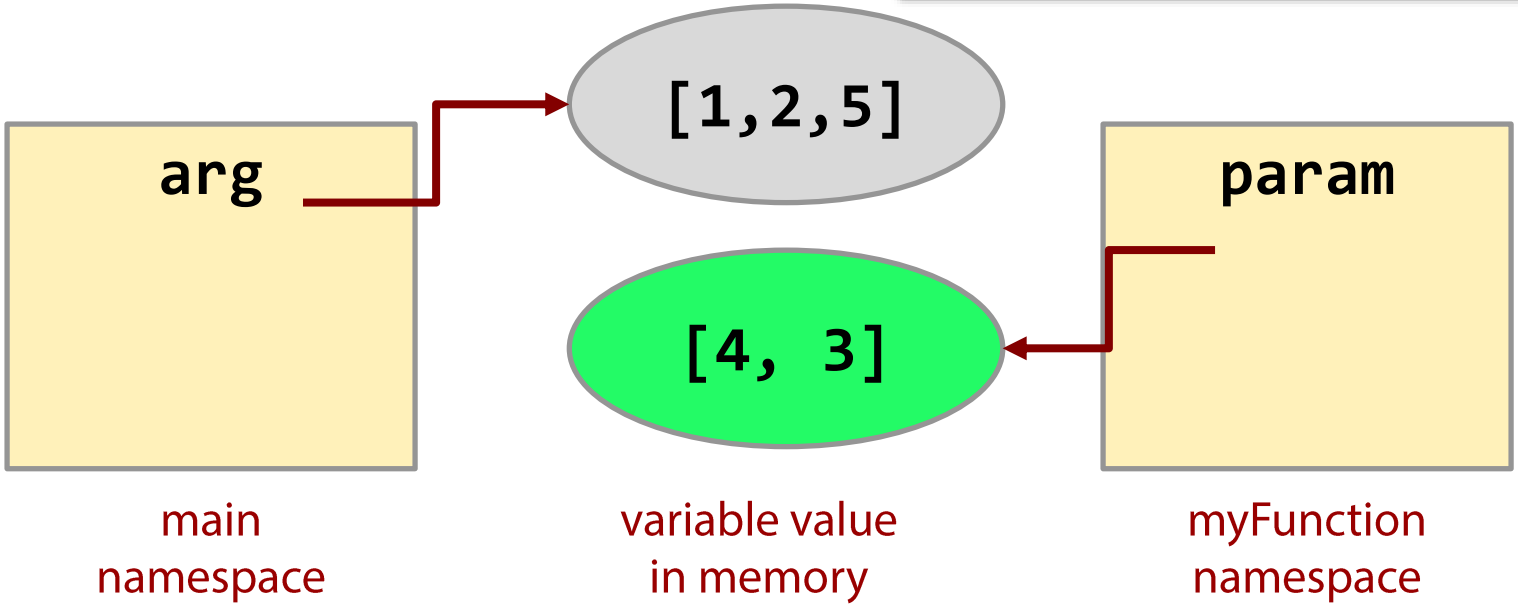
```
arg = [1,2,5]
myFunction(arg)
print(arg)
```

```
def myFunction(param):
    param = [4, 3]
    print(param)
```



```
arg = [1,2,5]
myFunction(arg)
print(arg)
```

```
def myFunction(param):
    param = [4, 3]
    print(param)
```



Aren't List Mutable?

- Every time you use assignment, Python **creates a new variable** (mutable or immutable)

```
num = 45
num = 81
```

```
list1 = [1, 2, 5]
list1 = [4, 3]
```

- However, with mutable objects, you can modify them without creating a new variable

```
list1 = [1, 2, 5]
list1[0] = 9
```

```
list2 = [4, 7]
list2.append(8)
```

When you pass variable to a function

- Immutable variables are **not** affected by any changes made within the function

- Mutable variables **may** be affected by changes in the function
  - Modifying operations *(**[ ]**, **append**, **del**, etc.)* **do** affect the original variable
  - Assignment (=) will **not** affect the original variables

- *End lecture*