

# Python Q&A

YAO ZHAO

```
if __name__ == "__main__"
```

- ▶ Every module in Python has a special attribute called `__name__`. The value of `__name__` attribute is set to `"__main__"` when module is run as main program. Otherwise, the value of `__name__` is set to contain the name of the module.
- ▶ We use `if __name__ == "__main__"` block to prevent (certain) code from being run when the module is imported.

# Pass by value or pass by reference in Python?

```
def Change(var):  
    var[0] = 'Changed'
```

```
def Original(var):  
    var = 'Changed'
```

```
variable = ['Original']  
Original(variable)  
print(variable)
```

```
Change(variable)  
print(variable)
```

What's the output?

# Pass by value or pass by reference in Python?

```
def Change(var):  
    var[0] = 'Changed'
```

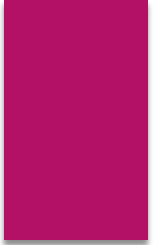
```
def Original(var):  
    var = 'Changed'
```

```
variable = ['Original']  
Original(variable)  
print(variable)
```

```
Change(variable)  
print(variable)
```

output

```
['Original']  
['Changed']
```



```
def Change(var):
```

```
    var[0] = 'Changed' #change the content that the reference var points to. Because var  
                        # and variable refer to the same object, any changes to the object  
                        # are reflected in both places
```

```
def Original(var):
```

```
    var = 'Changed' # reassign the reference var to a different string object 'Changed',  
                    # but the reference variable is separate and does not change.
```

```
variable = ['Original']
```

```
Original(variable)
```

```
#variable is a reference to the string object 'Original'
```

```
#When call Original, will create a second reference var to  
#the object 'Original'
```

```
print(variable)
```

```
Change(variable)
```

```
print(variable)
```



It is neither pass-by-value or pass-by-reference..

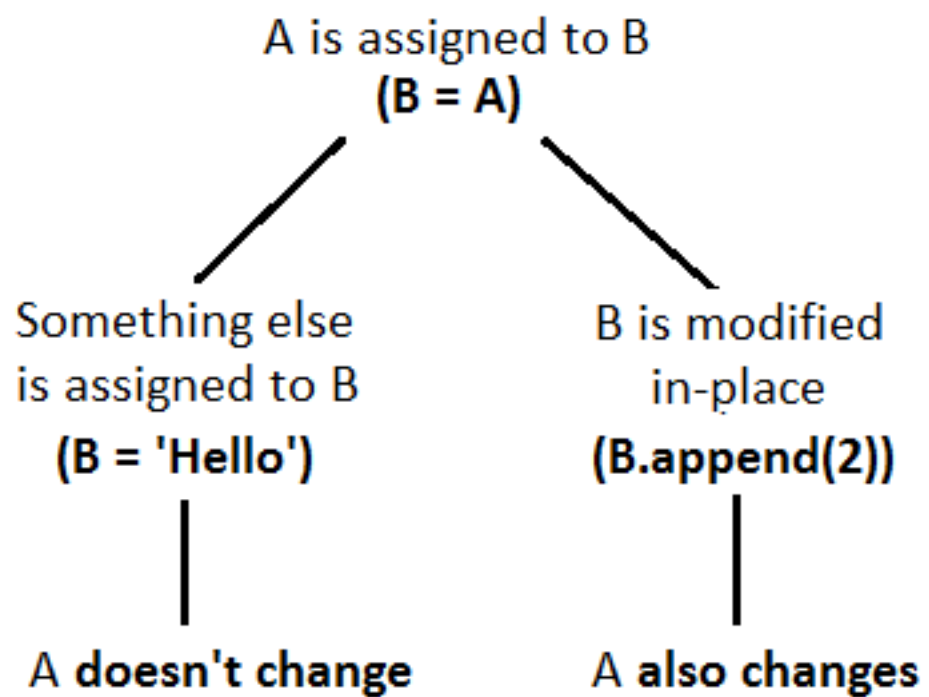
it is "**call-by-object**" or "**call by sharing**". Or, if you prefer, "**call by object reference**".

<http://effbot.org/zone/call-by-object.htm>

"...variables [names] are *not* objects; they cannot be denoted by other variables or referred to by objects."

In the example, when the Original method is called--a **namespace** is created for it; and **var** becomes a name, within that namespace, for the string object 'Original'. That object then has a name **in two namespaces**. Next, var = 'Changed' binds **var** to a new string object, and thus the method's namespace forgets about 'Original'. Finally, that namespace is forgotten, and the string 'Changed' along with it.





# `*args, **kwargs`

we can pass a variable number of arguments to a function using special symbols. There are two special symbols:

- 1) `*args` (Non-Keyword Arguments)
- 2) `**kwargs` (Keyword Arguments)



## **\*args**

The special syntax *\*args* in function definitions in python is used to pass a variable number of arguments to a function. It is used to pass a non-key worded, variable-length argument list.

- The syntax is to use the symbol *\** to take in a variable number of arguments; by convention, it is often used with the word *args*.

- What *\*args* allows you to do is take in more arguments than the number of formal arguments that you previously defined. With *\*args*, any number of extra arguments can be tacked on to your current formal parameters (including zero extra arguments).

- For example : we want to make a multiply function that takes any number of arguments and able to multiply them all together. It can be done using *\*args*.

- Using the *\**, the variable that we associate with the *\** becomes an iterable meaning you can do things like iterate over it, run some higher-order functions such as *map* and *filter*, etc.

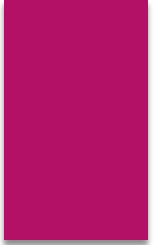
```
# *args for variable number of arguments
```

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

```
myFun('Hello,', 'Welcome', 'to', 'AI')
```



```
Hello,  
Welcome  
to  
AI
```



```
# *args with first extra arguments
def myFun(arg1, *argv):
    print ("First argument :", arg1)
    for arg in argv:
        print("Next argument through *argv :", arg)

myFun('Hello,', 'Welcome', 'to', 'AI')
```



```
First argument : Hello,
Next argument through *argv : Welcome
Next argument through *argv : to
Next argument through *argv :AI
```

## **\*\*kwargs**

The special syntax **\*\*kwargs** in function definitions in python is used to pass a keyworded, variable-length argument list. We use the name **kwargs** with the double star. The reason is because the double star allows us to pass through keyword arguments (and any number of them).

- A keyword argument is where you provide a name to the variable as you pass it into the function.
- One can think of the *kwargs* as being a dictionary that maps each keyword to the value that we pass alongside it. That is why when we iterate over the *kwargs* there doesn't seem to be any order in which they were printed out.

```
# **kwargs for variable number of keyword arguments
```

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

```
#Driver code
```

```
myFun(first='CS', mid='303', last='Lab2')
```



```
first == CS  
mid == 303  
last == Lab2
```

# \*\*kwargs for variable number of keyword arguments with  
# one extra argument.

```
def myFun(arg1, **kwargs):  
    for key, value in kwargs.items():  
        print("%s == %s" %(key, value))
```

#Driver code  
myFun('Hi', first='CS', mid='303', last='Lab2')



```
first == CS  
mid == 303  
last == Lab2
```

```
def myFun(arg1, arg2, arg3):  
    print("arg1:", arg1)  
    print("arg2:", arg2)  
    print("arg3:", arg3)  
# Now we can use *args or **kwargs to  
# pass arguments to this function :  
args = ("CS", "303", "Lab2")  
myFun(*args)  
  
kwargs = {"arg1" : "CS", "arg2" : "303", "arg3" : "Lab2"}  
myFun(**kwargs)
```

output

```
arg1: CS  
arg2: 303  
arg3: Lab2  
arg1: CS  
arg2: 303  
arg3: Lab2
```

```
def myFun(*args,**kwargs):  
    print("args: ", args)  
    print("kwargs: ", kwargs)
```

```
# Now we can use both *args ,**kwargs  
# to pass arguments to this function :  
myFun('CS','303','Lab2',first="CS",mid="303",last="Lab2")
```



```
args: ('CS', '303', 'Lab2')  
kwargs: {'first': 'CS', 'mid': '303', 'last': 'Lab2'}
```



# For in Python **vs** List Comprehension

- Store the items of iterable string 'python' to a list.

For:

```
list_letters = []  
  
for letter in 'python':  
    list_letters.append(letter)  
  
print(list_letters)
```

List Comprehension(列表推导式) :

```
list_letters = [ letter for letter in 'python' ]  
print(list_letters)
```

# Simple List Comprehension Examples

Get cubic numbers of 0,1,2,3:

```
[x**3 for x in range(0,4)]
```

Get all even numbers less than 50:

```
[x for x in range(0,50) if x%2==0 ]
```

Get all vowels in string 'python':

```
[x for x in 'python' if x in ['A','E','I','O','U']]
```

# Loop in Python **vs** List Comprehension

```
for (set of values to iterate):  
    if (conditional filtering):  
        output_expression()
```

**VS**

```
[ output_expression() for(set of values to iterate) if(conditional filtering) ]
```

# Advantages of List Comprehension

- ▶ Short codes
- ▶ Execute faster: List Comprehensions are 35% faster than FOR loop and 45% faster than map function.

LC is a simple but powerful technique and can help you accomplish a variety of tasks with ease. Things to keep in mind:

- ▶ LC will **always return a result**, whether you use the result or not.
- ▶ The iteration and conditional expressions can **be nested with multiple instances**. Even the overall LC can be nested inside another LC.
- ▶ **Multiple variables** can be iterated and **manipulated at same time**.

# More List Comprehension Examples

- ▶ Take two list of same length as input and return a dictionary with one as keys and other as values.

```
dic = {}  
for i in range(len(keys)):  
    dic[keys[i]] = values[i]
```

```
{ keys[i] : values[i] for i in range(len(keys)) }
```

- ▶ Create a matrix containing the different powers of 'number' variable.

```
[[i**p for p in range(2,7) ] for i in range(1,6) ]]
```

# \_ in python

- ▶ **Can represent the values that you don't care**

\_ is to represent the value that you don't care or will not be used later in the program. If you apply a linter like Flake8 to your program, you will get an error from the linter ([F841](#)) if you have a variable name assigned but never used. Assigning variables that you don't care to \_ can solve this problem.

- ▶ **Represent the last expression in the interpreter**

The special identifier \_ is used in the interactive interpreter to store the result of the last evaluation. It is stored in the builtin module.

- ▶ **Visual separator for digit grouping purposes**

From Python 3.6, underscore \_ can also be used as a visual separator for digit grouping purposes. As stated in [PEP515](#), it works for integers, floating-point, and complex number literals.



## Can represent the values that you don't care

```
# Example 1:uses _ to represent the index of each element in a list
for _ in range(5):
    print("I don't care about index")
```

```
# Example 2:only care about year, month and day from the tuple
year, month, day, _, _, _ = (2020, 7, 10, 12, 10, 59)
# year=2020, month=7, day=10
# so can assign _ to the rest (hour, minute, second).
# But if print out _, only get the last expression which is 59
print(_) # 59
```

```
# Example 3:supports extended iterable unpacking
# You can use *_ to represent multiple values.
# Here _ actually represents a list of values that we want to ignore.
year, *_ , second = (2020, 7, 10, 12, 10, 59) # year=2020
print(_) # [7, 10, 12, 10]
```

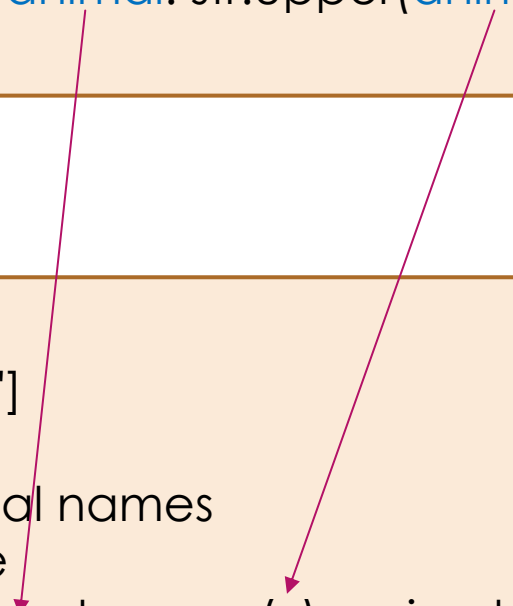
output

```
I don't care about index
I don't care about index
I don't care about index
I don't care about index
I don't care about index
59
[7, 10, 12, 10]
```

## Lambda function also supports \_

```
animals = ['dog', 'cat', 'parrot', 'rabbit']  
  
# here we intend to change all animal names  
# to upper case and return the same  
uppered_animals = list(map(lambda animal: str.upper(animal), animals))
```

```
animals = ['dog', 'cat', 'parrot', 'rabbit']  
  
# here we intend to change all animal names  
# to upper case and return the same  
uppered_animals = list(map(lambda _: str.upper(_), animals))
```

Two red arrows originate from the word 'animal' in the first code block and point to the underscore '\_' in the second code block, illustrating the replacement of a variable name with the underscore symbol in a lambda function parameter.

## Visual separator for digit grouping purposes

```
integer = 1_000  
amount = 1_000_000.1  
binary = 0b_0100_1110  
hex = 0xCAFE_F00D  
print(integer)  
print(amount)  
print(binary)  
print(hex)
```

output

```
1000  
1000000.1  
78  
3405705229
```

## Represent the last expression in the interpreter

```
(base) zhaoyao@zhaoyao pythonProject2 % python
Python 3.9.7 | packaged by conda-forge | (default, Sep  2 2021, 17:58:46)
[Clang 11.1.0 ] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> _
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name '_' is not defined
>>> 1_1
11
>>> _+1
12
>>> █
```

# The import system in Python

► <https://docs.python.org/3/reference/import.html>