\*args and \*\*kwargs

# What are \*args and \*kwargs in Python?

\*args 🡪 stands for arguments that you pass into a function

Arguments look like this:

def addargs(a, b):  
 return a + b  
  
print(addargs(5, 5))

\*kwargs 🡪 stands for keyword arguments that you pass into a function

Keyword Arguments look like this:

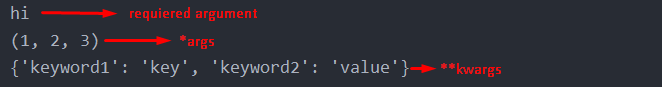
def addkwargs(a=5, b=12):  
 return a + b  
  
print(addkwargs()) # Result : 17

We use args and kwargs when we don’t know exactly how many arguments or keyword arguemnts our function will use

# Using \*args and \*kwargs

# \*args and \*\*kwargs example  
  
def foo(required, \*args, \*\*kwargs):  
 print(required)  
 if args:  
 print(args)  
 if kwargs:  
 print(kwargs)  
  
  
foo('hi', 1, 2, 3, keyword1='key', keyword2='value')

Result:



The require argument returned a string because we passed in a string

\*args returned a tuple

\*kwargs returned a dictionary

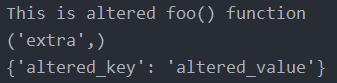
# Other uses with functions

We can use this technique to alter the functionality of another function without actually modifying it’s source code. Take not this is not really good to do unless you have experience.

def alteredfoo(new\_requirment, \*args, \*\*kwargs):  
 kwargs['altered\_key'] = 'altered\_value'  
 new\_args = args + ('extra', ) # New variable because tuples are immutable  
 foo(new\_requirment, \*new\_args, \*\*kwargs)

alteredfoo('This is altered foo() function')

Result:



Here we modified our original foo() function with the function alteredfoo(). As you can we modified kwargs to a custom dictionary, we assigned a new variable called new\_args because tuple are immutable and we can’t modify them.

And we added \*new\_args as an argument because \*args and \*kwargs are just naming convetions in python , you can use whatever you want but don’t do it unless you neceseraly have to.

Be careful this can create a maitanance nightmare.

# Using args and kwargs with classes

class Car():  
 def \_\_init\_\_(self, mileage, color):  
 self.mileage = mileage  
 self.color = color  
  
class RedCar(Car):  
 def \_\_init\_\_(self, \*args, \*\*kwargs):  
 super().\_\_init\_\_(args, kwargs)  
 self.color = 'red'  
  
my\_car = Car(120456, 'blue')  
my\_red\_car = RedCar(101234)  
  
print(my\_car.mileage, my\_car.color)  
print(my\_red\_car.mileage, my\_red\_car.color)

# Use it with decorators

This is a really good feature to use with decorators

def my\_timer(orig\_func):  
 import time # Import time module  
 from time import sleep  
  
 @wraps(orig\_func)  
 def wrapper(\*args, \*\*kwargs):  
 """We make a variable called to\_be\_timed otherwise we have to run the function  
 and that would print hello + {msg} again."""  
 t1 = time.time() # Get's T1  
 sleep(0.2)  
 to\_be\_timed = orig\_func(\*args, \*\*kwargs)  
 t2 = time.time() - t1 # t2 - t1 gives us the actual runtime of our function  
 print(f'{orig\_func.\_\_name\_\_} ran in : {t2} sec')  
 return to\_be\_timed # We return our newly made variable