*args and **kwargs

Work with Python long enough, and eventually you will encounter `*args` and `**kwargs`. These strange terms show up as parameters in function definitions. What do they do? Let's review a simple function:

```
In [3]:  def myfunc(a,b):
             return sum((a,b))*.05

         myfunc(40, 60)

Out[3]:  5.0
```

This function returns 5% of the sum of **a** and **b**. In this example, **a** and **b** are *positional* arguments; that is, 40 is assigned to **a** because it is the first argument, and 60 to **b**. Notice also that to work with multiple positional arguments in the `sum()` function we had to pass them in as a tuple.

What if we want to work with more than two numbers? One way would be to assign a *lot* of parameters, and give each one a default value.

```
In [17]:  def myfunc(a=1,b=0,c=0,d=0,e=0):
              print('a= ', a, 'b= ', b, 'c=',c, 'd=',d, 'e=',e)
              return a + b + c + d + e

          myfunc(4,b=2, d=7)

          a=  4 b=  2 c= 0 d= 7 e= 0

Out[17]:  13
```

Obviously this is not a very efficient solution, and that's where `*args` comes in.

*args

When a function parameter starts with an asterisk, it allows for an *arbitrary number* of arguments, and the function takes them in as a tuple of values. Rewriting the above function:

```
In [19]:  def myfunc(*args):
              print(args)
              return sum(args)*.05

          myfunc(40,60,20, 4)

          (40, 60, 20, 4)

Out[19]:  6.2
```

Notice how passing the keyword "args" into the `sum()` function did the same thing as a tuple of arguments.

It is worth noting that the word "args" is itself arbitrary - any word will do so long as it's preceded by an asterisk. To demonstrate this:

```
In [20]: def myfunc(*spam):
             return sum(spam)*.05

         myfunc(40,60,20)

Out[20]: 6.0
```

**kwargs

Similarly, Python offers a way to handle arbitrary numbers of *keyworded* arguments. Instead of creating a tuple of values, `**kwargs` builds a dictionary of key/value pairs. For example:

```
In [5]: def myfunc(**kwargs):
            if 'fruit' in kwargs:
                print(f"My favorite fruit is {kwargs['fruit']}")  # review String Fo
        rmatting and f-strings if this syntax is unfamiliar
            else:
                print("I don't like fruit")

        myfunc(fruit='pineapple')

        My favorite fruit is pineapple
```

```
In [6]: myfunc()

        I don't like fruit
```

*args and **kwargs combined

You can pass `*args` and `**kwargs` into the same function, but `*args` have to appear before `**kwargs`

```
In [7]: def myfunc(*args, **kwargs):
            if 'fruit' and 'juice' in kwargs:
                print(f"I like {' and '.join(args)} and my favorite fruit is {kwargs
        ['fruit']}")
                print(f"May I have some {kwargs['juice']} juice?")
            else:
                pass

        myfunc('eggs','spam',fruit='cherries',juice='orange')

        I like eggs and spam and my favorite fruit is cherries
        May I have some orange juice?
```

Placing keyworded arguments ahead of positional arguments raises an exception:

```
In [8]: myfunc(fruit='cherries',juice='orange','eggs','spam')

          File "<ipython-input-8-fc6ff65addcc>", line 1
            myfunc(fruit='cherries',juice='orange','eggs','spam')
                                                  ^
        SyntaxError: positional argument follows keyword argument
```

As with "args", you can use any name you'd like for keyworded arguments - "kwargs" is just a popular convention.

That's it! Now you should understand how `*args` and `**kwargs` provide the flexibilty to work with arbitrary numbers of arguments!

Memory:
|-----------------------------|
|main: | | a = 2, | | func1 | |-----------------------------| input =
|func1: | | a = 5 | | b = a + 1 | |-----------------------------|

In [35]:
```
a = 2

def func1():
    global a
    b = a + 1
    a = b + 1
    print('in func1:',a, b)
    return a+b

func1()

print(a)
```

```
in func1: 4 3
4
```

In [ ]:

In [ ]:

In [20]:
```
a = 2
def func1():


print(func1(), a)
```

```
Im here
5 1
```

In [ ]:

In [ ]:

In [ ]:

In [ ]: