In Python, we can pass a variable number of arguements to a function using special symbols.

There are two special symbols for passing arguments:

1. *args (Non-keyword arguments)
2. **kwargs (Keyword Arguments)

1.) *args

The special syntax *args in function defintions in Python is used to pass a variable number of arguements 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 allow 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 ba 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 ble 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.

In [1]:
```
def myfunc(a,b):
    #Returns 5% of the sum of a and b
    return sum((a,b)) * 0.05
```

In [2]:
```
myfunc(40,60)
```

Out[2]:
```
5.0
```

The function above 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 becausr 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 paramters, and give each one default value

In [3]:
```
def myfunc(a=0,b=0,c=0,d=0,e=0):
    return sum((a,b,c,d,e)) * .05
myfunc(40,60,20)
```

Out[3]:
```
6.0
```

In [4]:
```
myfunc (40,60,100,100)
```

Out[4]:
```
15.0
```

Loading [MathJax]/extensions/Safe.js

```
In [6]:  myfunc (40,60,100,100,3,5)
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
~\AppData\Local\Temp/ipykernel_12024/611888793.py in <module>
----> 1 myfunc (40,60,100,100,3,5)

TypeError: myfunc() takes from 0 to 5 positional arguments but 6 were given
```

Once we exceed the set amount of arguements, we get the error pictured above.

When we a function paramters 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 [7]:  def myfunc(*args):
             return sum(args)*0.05
```

```
In [8]:  myfunc(40,60)

Out[8]:  5.0
```

```
In [9]:  myfunc(40,60,100)

Out[9]:  10.0
```

```
In [10]: myfunc(40,60,100,1)

Out[10]: 10.05
```

```
In [11]: myfunc(40,60,100,1,34)

Out[11]: 11.75
```

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 its preceded by an asterisk.

To demonstrate this:

```
In [13]: def myfunc(*spam):
             return sum(spam)*.05

         myfunc(40,60,100,1,34)

Out[13]: 11.75
```

```
In [14]: def myfunc(*args):
             for item in args:
                 print(item)
```

Loading [MathJax]/extensions/Safe.js

In [15]:
```python
myfunc(40,60,100,1,34)
```
```
40
60
100
1
34
```

**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 doesnt seem to be any order in which they were printed out

Similarly, Python offers a way to handle arbitrary numbers of keyworded arguments.

Instead of creating a tuple of values, **kwargs build a dictionary of key/value pairs.

For example:

In [4]:
```python
def myfunc(**kwargs):
    if 'fruit' in kwargs:
        print(f"My favorite fruit is {kwargs['fruit']}")
    else:
        print("I don't like fruit")

myfunc(fruit='pineapple')
```
```
My favorite fruit is pineapple
```

In [5]:
```python
myfunc()
```
```
I don't like fruit
```

Notice, Python will not complain with the following:

In [6]:
```python
myfunc(fruit='apple' ,veggie = 'lettuce')
```
```
My favorite fruit is apple
```

This is the power of arbitrary arguments such as **arg and kwargs.

In [7]:
```python
def myfunc(**kwargs):
    print(kwargs)
    if 'fruit' in kwargs:
        print(f"My favorite fruit is {kwargs['fruit']}")
    else:
        print("I don't like fruit")

myfunc(fruit='pineapple')
```

```
{'fruit': 'pineapple'}
My favorite fruit is pineapple
```

In [8]:
```
myfunc(fruit='apple' ,veggie = 'lettuce')
```

```
{'fruit': 'apple', 'veggie': 'lettuce'}
My favorite fruit is apple
```

Pictured above we see kwargs returned a dictionary, unlike args which returned a tuple

In [ ]:
```

```

In [ ]:
```

```

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

In [9]:
```
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 postional arguments raises an exception:

In [10]:
```
myfunc(fruit='cherries' ,juice='orange''eggs','spam')
```

```
  File "C:\Users\Keegz\AppData\Local\Temp/ipykernel_9992/892951554.py", line 1
    myfunc(fruit='cherries' ,juice='orange''eggs','spam')
                                                         ^
SyntaxError: positional argument follows keyword argument
```

Notice above, as with "args", you can use any name you would like for keyworded arguments - "kwargs" is just a poular convetnion

In [ ]:
```

```