ARGUMENT TYPES

There are several **types of arguments** that can be passed on to a function:

- **Positional** arguments are passed in the order they were defined in the function
- **Keyword** arguments are passed in any order by using the argument's name
- **Default** arguments pass a preset value if nothing is passed in the function call
- ***args** arguments pass any number of positional arguments as tuples
- ****kwargs** arguments pass any number of keyword arguments as dictionaries

ARGUMENT TYPES

Positional arguments are passed in the order they were defined in the function

```
def concatenator(string1, string2):
    return string1 + ' ' + string2
```

```
concatenator('Hello', 'World!')
```

```
'Hello World!'
```

The first value passed in the function will be string1, and the second will be string2

```
concatenator('World!', 'Hello')
```

```
'World! Hello'
```

Therefore, changing the order of the inputs changes the output

ARGUMENT TYPES

Keyword arguments are passed in any order by using the argument's name

```
def concatenator(string1, string2):
    return string1 + ' ' + string2
```

```
concatenator('Hello', 'World!')
```

```
'Hello World!'
```

```
concatenator(string2='World!', string1='Hello')
```

```
'Hello World!'
```

By specifying the value to pass for each argument, the order no longer matters

```
concatenator(string2='World!', 'Hello')
```

```
SyntaxError: positional argument follows keyword argument
```

*Keyword arguments **cannot** be followed by positional arguments*

```
concatenator('Hello', string2='World!')
```

```
'Hello World!'
```

*Positional arguments **can** be followed by keyword arguments*

(the first argument is typically reserved for primary input data)

ARGUMENT TYPES

Default arguments pass a preset value if nothing is passed in the function call

```python
def concatenator(string1, string2='World!'):
    return string1 + ' ' + string2
```

Assign a default value by using '=' when defining the function

```python
concatenator('Hola')
```

```
'Hola World!'
```

Since a single argument was passed, the second argument defaults to 'World!'

```python
concatenator('Hola', 'Mundo!')
```

```
'Hola Mundo!'
```

By specifying a second argument, the default value is no longer used

```python
def concatenator(string1='Hello', string2):
    return string1 + ' ' + string2
```

```
SyntaxError: non-default argument follows default argument
```

Default arguments must come after arguments without default values

ARGUMENT TYPES

***args** arguments pass any number of positional arguments as tuples

```
def concatenator(*args):
    new_string = ''
    for arg in args:
        new_string += (arg + ' ')
    return new_string.rstrip()
```

```
concatenator('Hello', 'world!', 'How', 'are', 'you?')
```

```
'Hello world! How are you?'
```

Using '' before the argument name allows users to enter any number of strings for the function to concatenate*

Since the arguments are passed as a tuple, we can loop through them or unpack them

```
def concatenator(*words):
    new_string = ''
    for word in words:
        new_string += (word + ' ')
    return new_string.rstrip()
```

```
concatenator('Hello', 'world!')
```

```
'Hello world!'
```

It's not necessary to use 'args' as long as the asterisk is there

Here we're using 'words' as the argument name, and only passing through two words

ARGUMENT TYPES

****kwargs** arguments pass any number of keyword arguments as dictionaries

```
def concatenator(**words):
    new_string = ''
    for word in words.values():
        new_string += (word + ' ')
    return new_string.rstrip()
```

```
concatenator(a='Hello', b ='there!',
             c="What's", d='up?')
```

```
"Hello there! What's up?"
```

*Using '** before the argument name allows users to enter any number of keyword arguments for the function to concatenate*

Note that since the arguments are passed as dictionaries, you need to use the .values() method to loop through them

PRO TIP: Use **kwargs arguments to unpack dictionaries and pass them as keyword arguments

```
def exponentiator(constant, base, exponent):
    return constant * (base**exponent)
```

```
param_dict = {'constant': 2, 'base': 3, 'exponent': 2}

exponentiator(**param_dict)
```

```
18
```

The exponentiator function has three arguments: constant, base, and exponent

Note that the dictionary keys in 'param_dict' match the argument names for the function

*By using '** to pass the dictionary to the function, the dictionary is unpacked, and the value for each key is mapped to the corresponding argument*