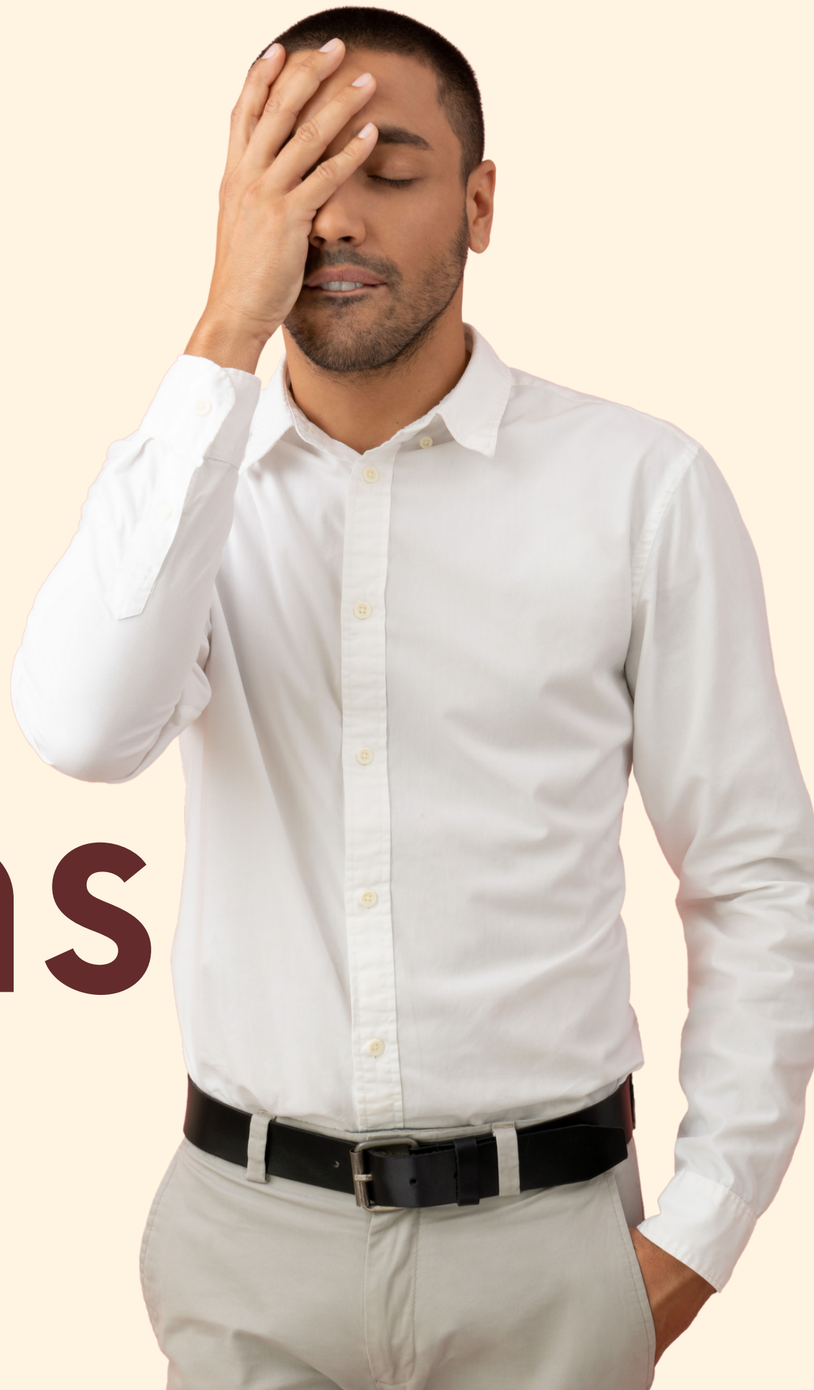
Returning To Functions



*args

We can use the wildcard or * notation to write functions that accept **any number of arguments**

```python
def average(*args):
    total = 0
    for arg in args:
        total += arg
    return total/len(args)
```

Gathers all remaining arguments into a tuple.

```
def average(*args):
    total = 0
    for arg in args:
        total += arg
    return total/len(args)
```

Pass as many arguments as we want! 5 args in this case:

```
def average(*args):
    total = 0
    for arg in args:
        total += arg
    return total/len(args)
```

```
average(1,2,3,4,5)
3.0
```

```python
def average(*args):
    total = 0
    for arg in args:
        total += arg
    return total/len(args)
```

```python
average(1,2,3,4,5)
3.0
```

```python
average(10,1)
5.5
```

2 arguments in this example

Name this parameter whatever you want. args is common but NOT required

```
def average(*nums):
    total = 0
    for arg in nums:
        total += arg
    return total/len(nums)
```

```
average(1,2,3,4,5)
3.0
```

```
average(10,1)
5.5
```

**kwargs

We can use the ** notation to write functions that accept **any number of keyword arguments**

```python
def print_ages(**kwargs):
    for k,v in kwargs.items():
        print(f"{k} is {v} years old")
```

Gathers all keyword arguments into a dictionary

```
def print_ages(**kwargs):
    for k,v in kwargs.items():
        print(f"{k} is {v} years old")
```

```python
def print_ages(**kwargs):
    for k,v in kwargs.items():
        print(f"{k} is {v} years old")
```

```python
print_ages(max=67,sue=59,kim=14)
max is 67 years old
sue is 59 years old
kim is 14 years old
```

name this whatever you want.
It's just a parameter!

```
def print_ages(**ages):
    for k,v in ages.items():
        print(f"{k} is {v} years old")
```

```
print_ages(max=67,sue=59,kim=14)
max is 67 years old
sue is 59 years old
kim is 14 years old
```

Order Matters

(parameters | *args | default parameters | **kwargs)

When defining functions, the order of parameters matters!

An Annoying Gotcha

With mutable default arguments

```python
def add_twice(val, lst=[]):
    lst.append(val)
    lst.append(val)
    return lst
```

add_twice expects a value and a list to be passed in. It appends the value to the list twice and returns the list.

add_twice expects a value and a list to be passed in. It appends the value to the list twice and returns the list.

```
def add_twice(val, lst=[]):
    lst.append(val)
    lst.append(val)
    return lst
```

```
add_twice('hi', [1,2,3])
[1, 2, 3, 'hi', 'hi']
```

add_twice expects a value and a list to be passed in. It appends the value to the list twice and returns the list.

```
def add_twice(val, lst=[]):
    lst.append(val)
    lst.append(val)
    return lst
```

```
add_twice('hi', [1,2,3])
[1, 2, 3, 'hi', 'hi']
```

```
add_twice('lol', ['ha'])
['ha', 'lol', 'lol']
```

```python
def add_twice(val, lst=[]):
    lst.append(val)
    lst.append(val)
    return lst
```

If no list is passed in, we've added a default value of []

```python
def add_twice(val, lst=[]):
    lst.append(val)
    lst.append(val)
    return lst
```

it seems to be working just fine...

```python
add_twice('yay')
['yay', 'yay']
```

```python
def add_twice(val, lst=[]):
    lst.append(val)
    lst.append(val)
    return lst
```

```python
add_twice('yay')
['yay', 'yay']
```

```python
add_twice('boo')
['yay', 'yay', 'boo', 'boo']
```

what??
what's going on?

the default value is being updated each time it's used!

The Fix

```
def add_twice(val, lst=None):
    if lst is None:
        lst = []
    lst.append(val)
    lst.append(val)
    return lst
```

give lst a default value of None

```python
def add_twice(val, lst=None):
    if lst is None:
        lst = []
    lst.append(val)
    lst.append(val)
    return lst
```

The Fix

Inside the function check to see if lst is None.

If so, set it to an empty list!

The Fix

```
def add_twice(val, lst=None):
    if lst is None:
        lst = []
    lst.append(val)
    lst.append(val)
    return lst
```

```
add_twice('yay')
['yay', 'yay']

add_twice('boo')
['boo', 'boo']
```

It works!

Argument Unpacking

Turning sequences into separate args

```python
def average(*args):
    total = 0
    for arg in args:
        total += arg
    return total/len(args)
```

This function accepts any number of arguments and returns their average

```python
average(1,2,3,4,5)
3.0
```

```python
average(10,1)
5.5
```

```python
def average(*args):
    total = 0
    for arg in args:
        total += arg
    return total/len(args)
```

```python
nums = [7,4,9,2,11,2,3,4]
average(nums)
TypeError
```

We can't pass a list of values. The function expects individual arguments, not a single collection of numbers

```python
def average(*args):
    total = 0
    for arg in args:
        total += arg
    return total/len(args)
```

```python
nums = [7,4,9,2,11,2,3,4]
average(nums)
TypeError
```

```python
average(*nums)
5.25
```

Instead, we can "unpack" the list into individual args using an asterisk