FUNCTIONS PART II

Objectives

- Use the * and ** operator as parameters to a function and outside of a function
- Leverage dictionary and tuple unpacking to create more flexible functions
- Understand what a lambda is and how they are used
- Explain what closure is and how it works in Python
- Use built in functions to sort, reverse and calculate aggregate information

`*args`

A special operator we can pass to functions

Gathers remaining arguments as a **tuple**

This is just a parameter - you can call it whatever you want!

Example

```
def sum_all_values(*args):
    total = 0
    for val in args:
        total += val

    return total

sum_all_values(1, 2, 3) # 6

sum_all_values(1, 2, 3, 4, 5) # 15
```

Another Example

```
def ensure_correct_info(*args):
    if "Colt" in args and "Steele" in args:
        return "Welcome back Colt!"

    return "Not sure who you are..."

ensure_correct_info() # Not sure who you are...

ensure_correct_info(1, True, "Steele", "Colt")
```

The order *does not* matter!

**kwargs

A special operator we can pass to functions

Gathers remaining keyword arguments as a dictionary

This is just a parameter - you can call it whatever you want!

Example

```python
def favorite_colors(**kwargs):
    for key, value in kwargs.items():
        print(f"{key}'s favorite color is {value}")

favorite_colors(rusty='green', colt='blue')

# rusty's favorite color is green
# colt's favorite color is blue
```

Another Example

```
def special_greeting(**kwargs):
    if "Colt" in kwargs and kwargs["Colt"] == "special":
        return "You get a special greeting Colt!"
    elif "Colt" in kwargs:
        return f"{kwargs["Colt"]} Colt!"

    return "Not sure who this is..."

special_greeting(Colt='Hello') # Hello Colt!
special_greeting(Bob='hello') # Not sure who this is...
special_greeting(Colt='special') # You get a special greeting Colt!
```

Parameter Ordering

1. parameters
2. *args
3. default parameters
4. **kwargs

Combined Example

```
def display_info(a, b, *args, instructor="Colt", **kwargs):
  return [a, b, args, instructor, kwargs]

display_info(1, 2, 3, last_name="Steele", job="Instructor")

[1, 2, (3,), 'Colt', {'job': 'Instructor', 'last_name': 'Steele'}]
```

What's going on with with that **(3,)** ?

When you have a tuple with one item - Python needs to distinguish between parenthesis and a **tuple**!

Using * as an Argument: Argument Unpacking

We can use * as an argument to a function to "unpack" values

```
def sum_all_values(*args):
    # there's a built in sum function - we'll see more later!
    return sum(args)

sum_all_values([1, 2, 3, 4]) # nope...
sum_all_values((1, 2, 3, 4)) # this does not work either...

sum_all_values(*[1, 2, 3, 4]) # 10
sum_all_values(*(1, 2, 3, 4)) # 10
```

Using ** as an Argument: Dictionary Unpacking

We can use ** as an argument to a function to "unpack" dictionary values into keyword arguments

```
def display_names(first, second):
    return f"{first} says hello to {second}"

names = {"first": "Colt", "second": "Rusty"}

display_names(names) # nope..

display_names(**names) "Colt says hello to Rusty"
```

Example with **
as an Argument

```
def display_names(first, second):
    return f"{first} says hello to {second}"

names = {"first": "Colt", "second": "Rusty"}

display_names(names) # nope..

display_names(**names) "Colt says hello to Rusty"
```

YOUR TURN

Lambdas

Normal functions have names...

```
def first_function():
    return 'Hello!'

first_function() # 'Hello!'

first_function.__name__ # first_function'
```

But lambdas are **anonymous** functions!

```
first_lambda = lambda x: x + 5

first_lambda(10) # 15

first_lambda.__name__ # '<lambda>'
```

Lambda Syntax

lambda parameters : body of function

```
add_values = lambda x, y: x + y

multiply_values = lambda x, y: x + y

add_values(10, 20) # 30

multiply_values(10, 20) # 200
```

map

A standard function that accepts at least two arguments, a function and an "iterable"

iterable - something that can be iterated over (lists, strings, dictionaries, sets, tuples)

runs the lambda for each value in the iterable and returns a map object which can be converted into another data structure

```
l = [1, 2, 3, 4]

doubles = list(map(lambda x: x * 2, l))

evens # [2, 4, 6, 8]
```

map in Action

```
l = [1,2,3,4]

doubles = list(map(lambda x: x*2, l))

evens # [2,4,6,8]
```

```
names = [
    {'first':'Rusty', 'last': 'Steele'},
    {'first':'Colt', 'last': 'Steele', },
    {'first':'Blue', 'last': 'Steele', }
]

first_names = list(map(lambda x: x['first'], names))

first_names # ['Rusty', 'Colt', 'Blue']
```

filter

- There is a lambda for each value in the iterable.
- Returns filter object which can be converted into other iterables
- The object contains only the values that return true to the lambda

```python
l = [1,2,3,4]

evens = list(filter(lambda x: x % 2 == 0, l))

evens # [2,4]
```

Combining *filter* and *map*

Given this list of names:

```
names = ['Lassie', 'Colt', 'Rusty']
```

Return a new list with the string
"Your instructor is " + each value in the array,
but only if the value is less than 5 characters

```
list(map(lambda name: f"Your instructor is {name}",
     filter(lambda value: len(value) < 5, names)))

# ['Your instructor is Colt']
```

What about
List Comprehension?

Given this list of names:

```
names = ['Lassie', 'Colt', 'Rusty']
```

Return a new list with the string:

"Your instructor is " + each value in the array,

but only if the value is less than 5 characters

```
[f"Your instructor is {name}" for name in names if len(name) < 5]
```

reduce

runs a *function* of two arguments cumulatively to the items of *iterable*, from left to right, which reduces the iterable to a single value

```
from functools import reduce

l = [1,2,3,4]

product = reduce(lambda x, y: x * y, l)

l = [1,2,3,4]

total = reduce(lambda x, y: x + y, l, 10)
```

You will not be using reduce frequently so it's good to know it exists, but you will not find yourself using it since we have a better option in most cases

reduce or List Comprehension?

For almost all problems especially at this stage, use list comprehension - you will see it far more in the wild

```
from functools import reduce

l = [1,2,3,4]

product = reduce(lambda x, y: x * y, l)
```

YOUR TURN

Closures

Accessing variables defined in outer functions after they have returned!

- private variables
- not using global variables

Example

Let's imagine we want a counter variable
and would like to keep track of it

"Public" Counter

```python
count = 0

def counter():
    global count
    count += 1
    return count
```

This works, but **anyone** can change *count*!

"Private" Counter

```
def counter():
    count = 0
    count += 1
    return count
```

No one can change *count* directly, but it keeps getting redefined!

Closures using *nonlocal*

```python
def counter():
    count = 0
    def inner():
        nonlocal count
        count += 1
        return count
    return inner
```

Here we're making a variable count inside the counter function, which can only be accessed by counter and inner.

Once we return inner, we can still remember count through closure!

Closures using Objects

```
def counter():
    counter.count = 0
    def inner():
        counter.count += 1
        return counter.count
    return inner
```

Here we're making a property on the *counter* function which can only be accessed by *counter* and *inner*.

Once we return *inner*, we can still remember the *count* property through closure!

Partial Application with Closures

```
def outer(a):
    def inner(b):
        return a+b
    return inner

result = outer(10)

result(20) # 30
```

When you are just using (not modifying) a variable through closure, you don't need to use *nonlocal* or objects!

You will see this pattern again when you learn about decorators!

YOUR TURN

Built-in Functions

`all`

Return **True** if all elements of the *iterable* are truthy (or if the iterable is empty)

```
all([0,1,2,3]) # False

all([char for char in 'eio' if char in 'aeiou'])

all([num for num in [4,2,10,6,8] if num % 2 == 0]) # True
```

any

Return **True** if any element of the *iterable* is truthy. If the iterable is empty, return False.

```
any([0, 1, 2, 3]) # True

any([val for val in [1,2,3] if val > 2]) # True

any([val for val in [1,2,3] if val > 5]) # False
```

sorted

Returns a new sorted list from the items in *iterable*

```
# sorted (works on anything that is iterable)

more_numbers = [6,1,8,2]
sorted(more_numbers) # [1, 2, 6, 8]
print(more_numbers) # [6, 1, 8, 2]
```

reversed

Return a reverse iterator.

```
more_numbers = [6, 1, 8, 2]
reversed(more_numbers) # <list_reverseiterator at 0x1049f7da0>
print(list(reversed(more_numbers))) # [2, 8, 1, 6]
```

Use slices or .reverse!

YOUR TURN

max

Return the largest item in an iterable or the largest of two or more arguments.

```
# max (strings, dicts with same keys)

max([3,4,1,2]) # 4
max((1,2,3,4)) # 4
max('awesome') # 'w'
max({1:'a', 3:'c', 2:'b'}) # 3
```

min

Return the smallest item in an iterable or the smallest of two or more arguments.

```
# min (strings, dicts with same keys)

min([3,4,1,2]) # 1
min((1,2,3,4)) # 1
min('awesome') # 'a'
min({1:'a', 3:'c', 2:'b'}) # 1
```

len

Return the length (the number of items) of an object. The argument may be a sequence (such as a string, tuple, list, or range) or a collection (such as a dictionary, set)

```
len('awesome') # 7
len((1,2,3,4)) # 4
len([1,2,3,4]) # 4
len(range(0,10) # 10

len({1,2,3,4}) # 4
len({'a':1, 'b':2, 'c':2} # 3
```

abs

Return the absolute value of a number. The argument may be an integer or a floating point number.

```
abs(-5) # 5
abs(5)  # 5
```

sum

- Takes an iterable and an optional *start*.
- Returns the sum of *start* and the items of an *iterable* from left to right and returns the total.
- *start* defaults to 0

```
sum([1,2,3,4]) # 10

sum([1,2,3,4], -10) # 0
```

round

Return *number* rounded to *ndigits* precision after the decimal point. If *ndigits* is omitted or is *None*, it returns the nearest integer to its input.

```
round(10.2) # 10
round(1.212121, 2) # 1.21
```

zip

- Make an iterator that aggregates elements from each of the iterables.
- Returns an iterator of tuples, where the *i*-th tuple contains the *i*-th element from each of the argument sequences or iterables.
- The iterator stops when the shortest input iterable is exhausted.

```
first_zip = zip([1,2,3], [4,5,6])

list(first_zip) # [(1, 4), (2, 5), (3, 6)]

dict(first_zip) # {1: 4, 2: 5, 3: 6}
```

zip

```
five_by_two = [(0, 1), (1, 2), (2, 3), (3, 4), (4, 5)]

list(zip(*five_by_two))

[(0, 1, 2, 3, 4), (1, 2, 3, 4, 5)]
```

Very common when working with more complex data structures!

YOUR
TURN

Recap

- *args is useful for accepting a variable number of arguments
- **kwargs is useful when accepting a variable number of keyword arguments
- you can use * to unpack argument values
- you can use ** to unpack dictionary values
- closures are very useful for private variables
- lambdas are annonymous functions that are useful with map, filter and reduce
- map is useful for transforming lists into different lists of the same size
- filter is useful for transforming lists into lists of different sizes
- Python has quite a few built in functions - make sure to spend the time learning them!